NAIST-IS-DT0261025

**博士論文**

# レガシーソフトウェアにおける<br>暗黙的コード制約の形式化と潜在フォールトの検出

松村　知子

2004年6月24日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

松村　知子

審査委員：　松本 健一 教授
小山 正樹 教授
関 浩之 教授
飯田 元 助教授

# レガシーソフトウェアにおける<br>暗黙的コード制約の形式化と潜在フォールトの検出*

松村　知子

**内容梗概**

修正や機能追加が長年繰り返されてきた大規模ソフトウェア（レガシーソフトウェア）において，保守の障害となる多数の「暗黙的コード制約」の発生，及び，それに伴うフォールトの混入が問題となっている．暗黙的コード制約とは，開発・保守の過程で付随的に発生する明文化されていないコーディング上の制約であり，違反するとフォールトが混入する場合がある．一方，長期にわたる保守の過程では作業者が入れ替わることが多く，暗黙的コード制約を全作業者に周知徹底させることは困難である．制約の存在に気づいていない作業者は制約違反をたびたびおかし，同種のフォールトが繰り返し混入することとなる．

本論文では，レガシーソフトウェアを対象として，暗黙的コード制約の形式化の方法，及び，形式化されたコード制約を使ってフォールトを自動的に検出するシステムを提案する．提案方法は，まず，熟練した保守作業者が，暗黙的コード制約を過去のフォールト報告書から抽出し，その中からフォールトの原因となる典型的なコードパターンとして形式化可能なものを，パターン記述言語により記述する（これを「制約違反パターン」と呼ぶ）．検出システムは，プログラムの構文解析によって作成される属性付き構文木と制約違反パターンとのパターンマッチングを行い，制約に違反するコードの検出を行う．

あるレガシーソフトウェアを調査した結果，保守工程で報告された故障の 32.7%は暗黙的コード制約に違反したために混入したものであり，保守開始時に存在した暗黙的コード制約のうち 39 個を確認できた．提案するパターン記述方法では、30 個の制約に違反するコードパターンを記述できた．さらに，試作したマッチングシステムによる実験の結果，パターンに一致する箇所が対象ソフトウェア中に 772 箇所検出され，そのうちの 152 箇所にフォールトが存在したことが分かった．またそのうちの 111 箇所が未報告のフォールトであり，提案方法が潜在フォールトの検出に有効であることを確認した．

*奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DT0261025, 2004 年 6 月 24 日.

**キーワード**

レガシーソフトウェア，ソフトウェア保守，コーディング規則，パターンマッチング，フォールト検出

# Formulating Implicit Code Constraints and Detecting Potential Faults in Legacy Software *

Tomoko Matsumura

**Abstract**

In the field of legacy software maintenance, there arise a large number of "implicit code constraints", which are usually undocumented. Violating such constraints causes injection of a new fault. As maintainers move between companies and job assignments in the long maintenance process, some maintainers who are unaware of such constraints often violate them, and the same kind of faults is repeatedly introduced.

This paper proposes a method for formulating the implicit code constraints and automatically detecting code fragments by using the formulated implicit code constraints. In the method, an expert maintainer firstly investigates the causes of failures that have been described in the past fault reports; and identifies all the implicit code constraints that lie behind the faults. Then, code patterns violating the constraints (which we call "violation patterns") are formally described in a pattern description language. The detecting system searches code fragments matching the patterns with the attributed syntax trees transformed from source codes by the parser.

The result of a case study with large legacy software showed that 32.7% of failures, which have been reported during a maintenance process, were due to the violation of implicit code constraints, and there were 39 constraints at the start point of the maintenance process. 30 of these constraints could be described in the pattern description language. Moreover, 152 faults existed in 772 code fragments detected by the prototype matching system, and 111 faults among them had not been reported in the fault reports. It shows that the method we proposed is effective in detecting potential faults.

**Keywords:**

*Doctor's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT0261025, June 24, 2004.

legacy software, software maintenance, coding rules, pattern matching, fault detection

# 研究業績

## 学術論文誌

1. 松村知子，門田暁人，松本健一: 潜在コーディング規則違反を原因とするフォールトの検出支援方法の提案，情報処理学会論文誌，Vol.44，No.4，pp.1070–1082，2003.4.

## 国際会議

1. Tomoko Matsumura, Akito Monden and Ken-ichi Matsumoto: The detection of faulty code violating implicit coding rules, *Proceedings of 2002 International Symposium on Empirical Software Engineering* (ISESE 2002), pp.173–182, December, 2002.

2. Tomoko Matsumura, Akito Monden and Ken-ichi Matsumoto: A method for detecting faulty code violating implicit coding rules, *Proceedings of 5th International Workshop on Principles of Software Evolution* (IWPSE2002), pp15–21, May, 2002.

## 研究会・シンポジウム

1. 松村知子，門田暁人，松本健一: 潜在コーディング規則に基づくバグ検出方法の提案，ソフトウェアシンポジウム 2002，pp.105–114, 2002.6.

2. 松村知子，門田暁人，松本健一: バグ報告データを用いたプログラムコード検証方法の提案，電子情報通信学会技術研究報告, SS2001-34，Vol.101，No.628，pp.1–8, 2002.1.

# 目 次

# 図 目 次

# 表 目 次

# 1. はじめに

## 1.1 研究背景

今日，大規模レガシーソフトウェアの保守コストの低減は，多くの企業にとって重要な課題となっている [35][43]．レガシーソフトウェアとは，長期にわたって保守が続けられてきた遺産的なソフトウェアのことであり，30 年以上保守されているものも存在する [6]．長期にわたる保守の過程では，ソフトウェア全体の設計の見直しが行われないままに局所的な変更（機能変更，追加，及び修正）が繰り返され，その結果，Code Decay[7]（コードの劣化：ソフトウェアの構造が複雑になることを指す）が起こる．劣化したソフトウェアは理解が困難であり，変更を加える際に意図しないバグ（フォールト）が混入しやすい．そのため，影響波及解析，回帰テスト，デバッグなどにより多くのコストを必要とする [25]．また，コード劣化はソフトウェアに変更を加える度に進行するため，保守コストが年々増加することとなる．

レガシーソフトウェアにおいて保守コストを増大させる他の要因としては，プログラムコードの複雑化以外にも,「標準となる開発技術がない」「古い言語で開発されている」「役に立つドキュメントがない」「テスト手法が確立されていない」などが挙げられている [5]．さらに，長期にわたる保守期間に開発者が入れ替わったり，ドキュメントの更新・修正が適正に行われなかったりすることが，プログラムコードの理解などの保守作業を困難にしている．

一方で，レガシーソフトウェアはその信頼性の高さから，新しいシステムへの移行が進まず，高コストな保守を繰り返しながら運用し続けられる場合が多い．金融や行政分野で運用されている業務系システムでは，レガシーソフトウェアは膨大な情報資産を持ち，安定した運用実績がある．それに対して，新たなシステムの導入には，システム立ち上げ時の不安定さが伴う．社会基盤として重要なシステムの不具合やシステムダウンは社会的，経済的影響が大きいため，新システムの導入には慎重にならざるを得ない．

組み込み系のシステムにおいても，過去に開発され商品に組み込まれているソフトウェアが再利用され，長年にわたって使い続けられるケースは多い．組込み系システムでは，携帯電話や家電製品のように類似した商品のシリーズ開発が多いため，ソフトウェアの再利用が頻繁に行われてきた．再利用されるソフトウェアも，対象となる製品の仕様やハードウェアの変更に伴い，変更が加えられることが多く，プログラムは複雑化している．しかし，開発期間が短縮し開発者が流動的になってきた今日の開発形態では，システムを新たに構築し，従来のシステムと同じ信頼性を確保することは容易ではない．

複雑なプログラムにはよりフォールトが混入しやすいという視点から，プログラムの複雑度を定量的に計測し，複雑度をソフトウェア保守の現場に応用する試みも行われている [21][13][7][3]

[33][35]．これらの研究では，複雑度を様々な尺度により計測し，フォールトが混入する可能性が高いモジュールを特定することによって，重点的なコードレビューや回帰テストをするモジュールを絞込むことに用いることを目的としている．また，計測された複雑度は，部分的な再設計や再構築，リファクタリングを行う指標として用いることもできる．さらに，コンポーネントやモジュールの過去のフォールト情報から，計測された複雑度とフォールトの含み易さ（fault-proneness）を関連付けることによって，個々のシステムに応じたフォールトの存在を予測する研究も行われている．これらの研究によって，大規模なレガシーソフトウェアの保守における信頼性保持のための作業効率があがり，保守コストの低減が図れると考えられている．

しかし，実際の保守作業において，定量的に計測された複雑度を用いることは，あまり一般的に普及していない．これらの研究ではモジュール単位やコンポーネント単位での複雑度が数値として示されるが，故障の現象やフォールトとなるコードなど，保守作業者が対応するべき具体的な内容を示すことができず，プログラム変更時にフォールトの混入を検出するための手段やフォールト混入の回避方法など，保守作業者が必要とする情報を提供することもできない．そのため，保守作業者が保守現場において定量的に計測された複雑度を有効に活用するのは，困難である．

ソフトウェアの保守作業の現場では，対象のシステムの開発・保守に長年携わってきたシステムを熟知する保守作業者の経験が重視されるが，それはこのような保守作業者が，コードレビューやテストにおいて他の保守作業者が気付かないようなフォールトや故障を検出することが多いためである．長年にわたって対象となるシステムの開発・保守に携わってきた作業者には，システムの弱点に関する知識が蓄積されている．一般的に，システムに関する仕様や設計などは書類化され，すべての関係者に周知されているため，経験に頼る必要はない．書類化されない知識としては，開発中に直面した問題に関する情報や問題を回避するために設計書上明記されずに追加された制御構造などがある．このような知識を用いて，経験者はプログラムコードからフォールトを検出したり，危険なコードを特定したりすることができる．さらに，そこにはフォールトの回避方法や発生する故障の現象，テスト方法など付随する様々な知識をもっているため，効率的に保守を行うことができるのである．

ソフトウェア開発の現場において，経験知を活用するための方法として「ナレッジマネージメント」や「事例ベース推論」を導入する実践研究が行われている [23][16]．これらはソフトウェア開発現場だけではなく，広くビジネス一般で用いられている方法で，これまで業務の経験の過程で獲得され経験者の記憶にのみ存在する暗黙知を有効活用する手法として注目されている．ナレッジマネージメントとは「データとして表わされる単なる情報から，ビジネスに役立つ知識をいかにして得るかという管理プロセスのこと」で,「事例ベース推論」は「知識ベース支援哲学を過去の経験を用いた人間の推論のシュミレーションと組み合わせた技術，すなわち頭の中で過去

に起きた同じような状況を検索し，それらの状況から得られた経験を再利用すること」と言われている [16]．ソフトウェア開発へ導入するための研究としては，主に類似システムの開発における要求仕様や設計時の問題に対する解決を支援することを目的としており，キーワードによる検索をベースにしている．しかし，これらの研究では，解決するべき問題が明確で，それを特定するキーワードを選択できることが必要となる．そのため，存在も内容も明確でないプログラムコードレベルでのフォールトの検出には適用できない．

## 1.2 研究概要

本論文では，レガシーソフトウェアにおいて「プログラムコードが劣化する」というマクロ的な現象を,「保守の妨げとなる暗黙的コード制約が発生する」というより保守作業者寄りの視点で捉える．本論文では，暗黙的コード制約をレガシーソフトウェアから抽出し，制約に違反するコードパターンを形式化するとともに，形式化されたパターンを用いてフォールトを検出することを目的とする [30][29][31][28] [32].

複雑化した大規模なレガシーシフトウェアには，対象システムに特有で，開発担当者のみがその存在を認識し，ドキュメントにも明記されていないようなコーディング上の制約,「暗黙的コード制約」が多数存在する．それらを認識していない保守担当者が違反するコーディングをすることは，フォールトを混入する原因になる．例えば,「大域変数 A に特定の情報を示す値を代入しないで関数 B を処理すると，ある機能が動作しない」「関数 X の呼出し直後に関数 Y を呼び出すと，ある操作をしたとき異常が発生する」といったような制約は，開発当初から仕様として定められ明記されているものではなく，開発・保守の過程で付随的に発生したもので，システムを熟知している開発・保守担当者のみによって，暗黙的に守られている．しかし，このような制約を知らない開発者が違反したコーディングをすることで，フォールトを混入させてしまう場合がある．さらに，保守作業者が入れ替わることが多い大規模レガシーソフトウェアでは，経験的に得られた暗黙的知識は流出してしまうことが多い [34] ため，同じ制約違反によるフォールト混入が繰り返される危険がある．

提案するフォールト検出方法では，暗黙的コード制約を過去に作成されたフォールトに関する文書（本論文では,「フォールト報告データ」と呼ぶ）を調査することによって抽出した．ここでいうフォールト報告データとは，故障発生状況，故障が発生した原因（フォールト）のソースコード上の位置，及び，フォールトの除去方法等についての情報である．抽出された制約から，制約に違反している部分のソースコードをパターン化し，形式化する（本論文では，これを「制約違反パターン」と呼ぶ）．抽出した制約違反パターンの形式的な記述には，文献 [41] で提案されているパターン記述言語を拡張したものを用いる．このように形式化された制約違反パターンを用い

て，プログラムコードから一致するコードを検出するため，プログラムの構文解析によって作成される属性付き構文木と制約違反パターンとのパターンマッチングを行う．検出された一致コードついては，暗黙的コード制約の抽出元となったフォールト情報を用いて，フォールトの有無をチェックできる．

提案する方法の有効性を検証するために，本論文では，実際に現在も保守・運用が行われている組込み系システムを用いて，暗黙的コード制約を抽出し，フォールトを検出する実験を行う．その結果，暗黙的コード制約に違反するコードの検出方法としての提案方法の性能と，実際のシステム上に存在する潜在フォールトの検出方法としての有用性を実証する．

## 1.3 論文構成

2 章では，レガシーソフトウェアの保守における信頼性の問題をプログラムコードの複雑化の視点から明らかにし，本論文の目的について記述する．まず，レガシーソフトウェアの保守における信頼性の維持の重要性について，業務系，組込み系のシステムの運用現場の現状から説明する．次に，信頼性の保持を困難にしているプログラムの複雑化について，その直接的原因となる具体的なプログラムコードの変更状況や，そのような変更が行われる原因としての保守作業者や保守作業を取り巻く状況の問題を論じる．最後に，信頼性を維持しながら複雑化したプログラムコードの保守を行うために，過去のフォールト情報を用いた潜在フォールト検出を目的とした本研究のアプローチについて説明する．

3 章では，本論文で扱う暗黙的コード制約の輪郭を明らかにし，その形式化の方法を提案する．まず，暗黙的コード制約の特徴や問題点，制約の発生状況を，実際のレガシーソフトウェアから抽出した制約に基づいて説明し，暗黙的コード制約の輪郭を明確にする．次に，実際のレガシーソフトウェアにおけるデータを用いて暗黙的コード制約と発生フォールトの関係を示し，さらに暗黙的コード制約違反によるフォールトへの現状の保守現場での対応方法について説明する．最後に，制約違反パターンの形式的記述方法について，本研究で適用する形式的記述言語（文献 [41] 参照) について概要を述べ，より柔軟なプログラムコードパターンを記述するための拡張方法を提案する．

4 章では，形式化された制約違反パターンを用いたフォールトの検出方法を提案する．フォールトを含む可能性があるコードを，形式的に記述された制約違反パターンとプログラムコードから自動的に検出する「制約違反検出システム」を中心として，制約違反検出システムを適用したフォールト検出方法の全体の手順について説明する．さらに，ここでは，制約違反検出システムの保守プロセスにおける２つの適用方法についても提案する．

5 章では，提案方法の有効性の評価のため行ったケーススタディの結果について報告し，その考

察を行う．ケーススタディには，現在も運用・保守が行われている組込み系レガシーソフトウェアの１サブシステムを用いた．１年半にわたり蓄積されたフォールト報告データとソースコードを対象とし，C 言語に対応した制約違反検出システムのプロトタイプを開発して，制約違反パターンに一致するコードの自動検出を実施した．検出された一致コードに対してフォールトの有無を判定してフォールトコードを取り出し，さらにフォールトコードからフォールト報告データに未報告のフォールトを取り出すことで，提案方法が潜在フォールトの検出に有効であることを確認する．

6 章では，関連研究について記述する．

最後に，7 章で，本論文の全体のまとめと考察を行う．

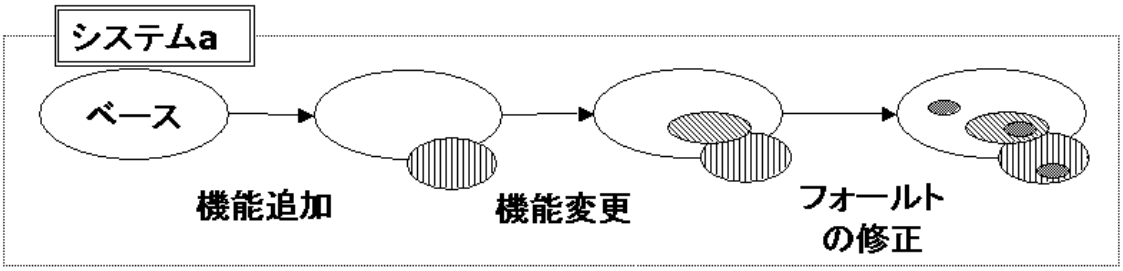


図 1 保守によるシステムの発展

# 2. レガシーソフトウェアの保守と信頼性

## 2.1 ソフトウェアの保守

ソフトウェアの保守とは,「フォールト修正や性能向上，環境変化に対する適応のために引渡し（運用開始）後にシステムを修正する工程」とある [2]．保守作業の内容として，以下のような分類例がある [13].

- Perfective Maintenance ：

  機能を変更せずにシステムを改善する（高機能のハードウェアへの変更・保守を容易にする開発環境の導入など）

- Adaptive Maintenance ：

  プログラム環境の変化に対応するデータの変更・機能追加や機能変更

- Corrective Maintenance ：

  フォールト，すなわちプログラムの間違いや故障の原因の修正

図1 に示すように，機能追加や機能変更 (Adaptive Maintenance) やフォールトの修正 (Crrective Maintenance) によって，システムは部分的に変更されていくことが多い.

一方，図 2 のような発展をする組み込み系のシステムでは，ソフトウェアの保守の範囲が運用中のシステムへの変更のみでなく，再利用されるシステムへ拡張する．組み込みシステムとは，各種の機器に組み込まれてその制御を行うコンピュータシステムのことを言う [47]．最近では，携帯電話や家電製品のマイコン制御機能の普及などにより，組み込みシステムの適用範囲は大幅に拡大している．組み込みシステムは，一旦製品に組み込まれて出荷されてしまうと，大きな問題が無い限りソフトウェアを書き換えることは無い．しかし，図 2 のようにソフトウェアが再利用

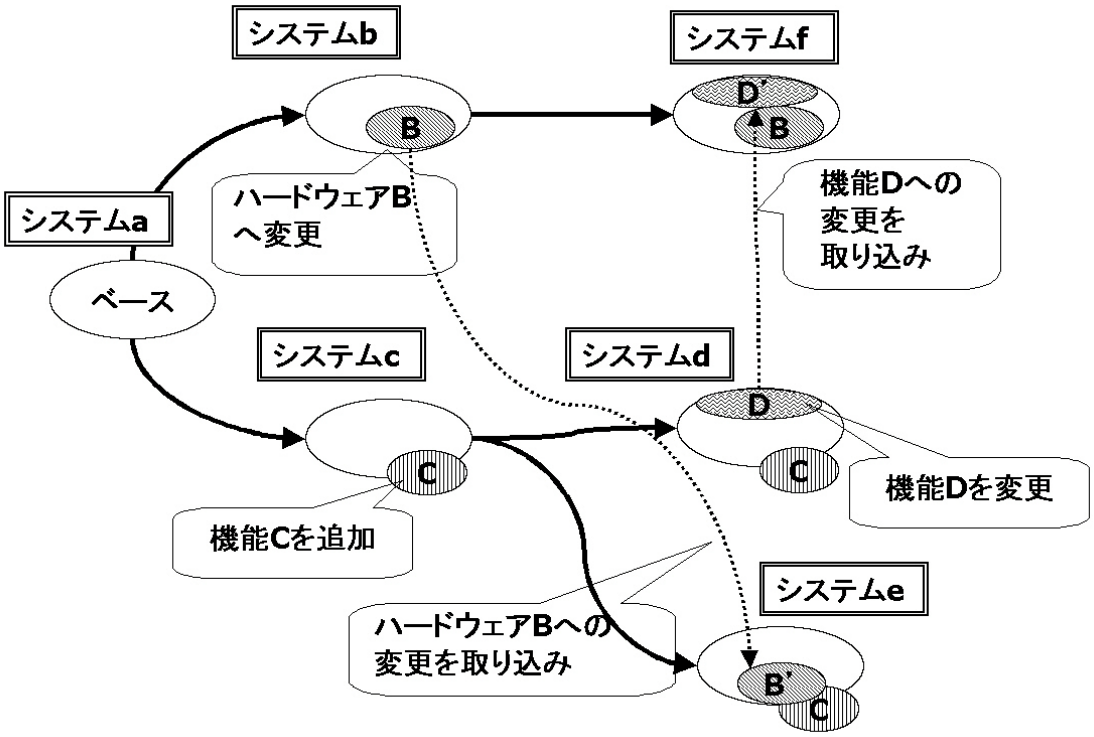


図 2 組込み系システムの発展

される場合，ソフトウェアを新製品のハードウェアに適用するための変更や新機能の追加に伴うI/F の追加といった，通常のソフトウェア保守と同じような作業が発生する．図 2 では，システム a 向けに開発されたソフトウェアを，新製品にあわせてハードウェア変更に対応したシステム b とバージョンアップによる機能追加をしたシステム c に再利用している．システム c からは，機能 D を変更したシステム d が開発される．さらに，システム c に対してハードウェアを変更のためシステム b の変更を取り込んだり，システム b に対して機能 D の変更部分をシステム d から取り込むといったケースもある．このような作業はすべて，新製品向けの新システムの開発の側面とともに，システム a のソフトウェアに対する保守という側面を持つ．

上記のような保守が長年行われながら運用・再利用されてきたソフトウェアはレガシーソフトウェアと呼ばれているが、レガシーソフトウェアの保守においては，信頼性の保持が重要な課題である．金融や行政関係の業務系のレガシーソフトウェアの多くは，現在も社会基盤の重要な役割を担って運用中であり，その不具合やシステム停止は社会的影響が大きい．さらに膨大な情報資産を持ち，長年にわたる保守により高い信頼性を保持している．そのため，レガシーソフトウェアの置き換えは困難であり，その保守による信頼性の低下は保守作業者が最も恐れるところである．

最近では，組込みシステムにおいても高い信頼性の保持が，大きな課題となってきている [22]．図 2 のように再利用されるソフトウェアで，重大な欠陥が発見された場合，その修正は，それを再利用しているすべてのシステムが対象となる．その商品の出荷数によっては，機器の回収・ソフトウェアの書き換えには，膨大なコストを必要とする [47]．しかし，組み込みシステムの大規模化とその開発サイクルの短期化は顕著で，図 2 のようなシステム開発（もしくは保守）は，複数の開発・保守グループによって並行して行われたり，各システムにおいて異なる規約や設計思想で変更されたりすることが多い．同じベースのソフトウェアに対する変更や変更内容の取り込み・再利用であっても，取り込み元・取り込み先のシステムを正確に理解しないで取り込みを行うと，取り込み先で元のシステムにはなかった故障が発生したり，既存の機能の劣化が発生したりする．そのため，再利用を行うソフトウェアの保守での信頼性の維持は，新規開発されるソフトウェアの保守より困難である．

## 2.2 プログラムコードの複雑化とその原因

レガシーソフトウェアの保守性，すなわちプログラムを修正することの容易さ [2] を悪化させている原因の 1 つに，プログラムコードの複雑化があげられる．長期間にわたってプログラムコードへの変更や追加が繰り返されることで，プログラム構造が図 3 のように複雑化する．この現象は Code Decay（コードの劣化）とも呼ばれ，Eick 等は，コード劣化を「プログラムコードの変更がより困難になること」と定義した [7]．プログラムコードの複雑化は，モジュール性（独立した

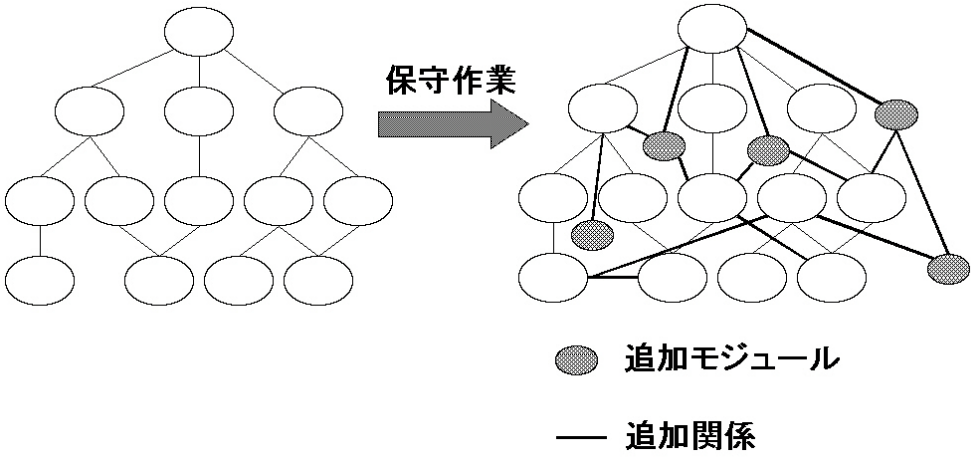


図 3 プログラム構造の複雑化

コンポーネントで構成されたシステム又はコンピュータプログラムのレベル [2]）の悪化の 1 つの原因であり，モジュール性はモジュール凝集度（単一のソフトウェアモジュールによって実現されるタスクが他のモジュールと関係している度合い [2]），モジュール間結合度（ソフトウェアモジュール間の相互依存度 [2]）によって計測される．モジュール凝集度が低い，あるいはモジュール間結合度が高いソフトウェアはコードが劣化している，と考えられる．

プログラムコードの複雑化の直接的な原因は，対症療法や不完全な構造理解に基づくプログラムコードの変更であると考えられる．対症療法とは，設計上のフォールトや機能追加・修正に伴う設計の変更やリファクタリングを行わず，場当たり的な対応で問題を回避するコーディングを行うことである．この場合，問題となっている症状にのみターゲットを絞るため，他の機能への影響は少ないが，一方で部分的に限定された制御構造が組み込まれることで，システム構造が複雑化する．また，プログラムコードの構造を完全に理解しないでコードを変更すると，そのコードが従来の設計構造を逸脱して構造を複雑化させてしまう．

このようなプログラム変更が行われる主な原因としては，保守作業への時間制約や保守作業者のレベルがあげられる．再設計やリファクタリングを行うことによって，プログラムコードの複雑化を防止することは可能であるが，既存の機能への影響もあり，設計・製造・テストを含めて膨大な時間的人的コストがかかるため，実施が困難な場合が多い．また，技術的レベルの高い開発者は，新規開発を好む傾向にあり，保守作業者は入れ替わりが激しい [4]．レベルの低い技術者の場合，既存のソースコードを理解したり，ソースコードからそのプログラムの設計構造などを理解するための抽象化能力が低いと言われている [46]．そのため，既存のソースコードの構造や

設計を逸脱するような変更を加えてしまったり，従来の設計に基づいて複雑化しない変更方法があるにもかかわらず，不必要な制御構造を追加してしまう．設計構造を逸脱するような変更や不適切な設計の制御構造の追加は，適正に原則に則った設計構造を期待してプログラムコードを解析する熟練技術者にとって理解しにくいコードを発生させてしまう [46]．さらに，このような技術的レベルの問題を補完するドキュメントが存在しない，開発・保守体制が複雑であるために重要な知識の共有を計ることが困難である，といった問題が，プログラムの複雑化を間接的に促進する．

## 2.3 プログラムコードの複雑化による問題点

複雑化したプログラムコードの保守における問題の１つに，機能追加やフォールト修正による新しい故障発生や機能の劣化（デグレーション），つまり変更前は正常に動作していた機能での故障発生がある．故障 (failure) とは，ユーザが期待するシステムの動作に対して正常に動作しないことやシステムが停止する等のシステム稼動中に発生する問題現象である．故障は，プログラムコードの変更によって混入する正しくないコード，すなわちフォールトの混入によって発生する．モジュール間の結合度が高いと，プログラム変更によって保守作業者が予期しない本来の変更対象ではないモジュールやサブシステムに影響して，機能劣化の危険性が高くなる．さらに，複雑化したプログラムコードを正確に理解することが困難なため，不適切な設計による変更が行われ，１つの変更に対して多数のフォールトが混入することもある．例えば，追加したモジュールがシステム全体で汎用的に使用されているリソースを別の用途で使用する等の原因で，多くの既存機能において故障が発生してしまうことがある．

保守作業中に混入するフォールトによって発生する故障は，早期の段階，すなわちコーディングやテスト工程などリリース前の段階で発見されないと，故障対応にかかるコストは増大するが，故障の早期発見は決して容易ではない．モジュール間の予期しない結合関係による故障は，変更対象以外のタスクやモジュールで発生したり故障としての発生頻度が低かったりすることも多く，通常の機能テスト・システムテストでの検出やデバッグが困難である [48]．コードレビューでフォールトを検出するには，モジュール間の結合関係をすべて把握し，変更が適切か，もしくは影響が波及するすべてのモジュールで問題がないかを確認しなければならないが，大規模で複雑化したソフトウェアの場合，手作業でコードレビューを完璧に行うのは困難で，見落としが発生し易い．

保守作業中に混入したフォールトが早期に発見されずにリリースされると，故障発生に対するプログラム修正・リリースが行われるが，これが頻繁に繰り返されると，さらにコードが劣化し，フォールト混入の危険が増大する，といった悪循環に陥る．

一方，故障を伴うフォールトが混入しないとしても，複雑化したプログラムコードに関わる保

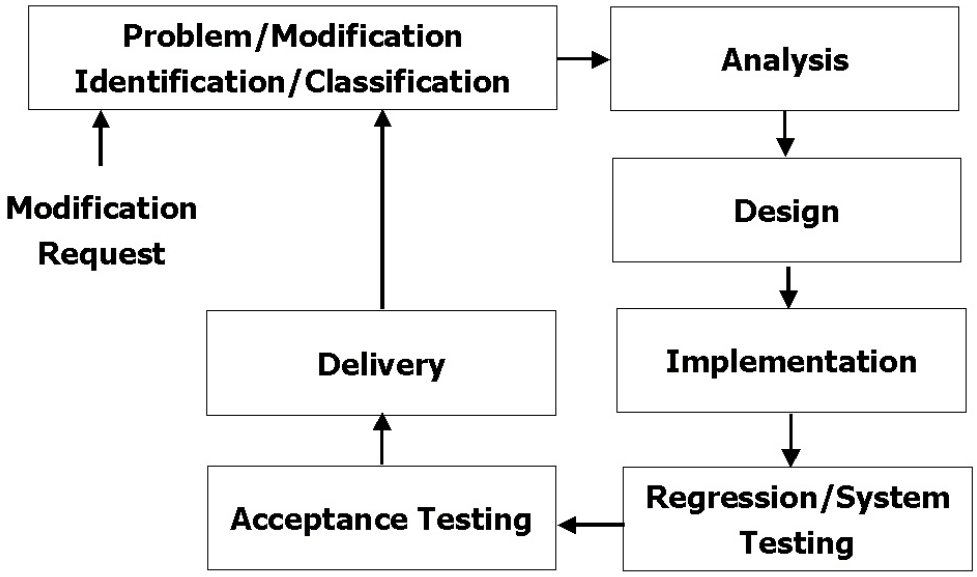


図 4 The IEEE Maintenance Process

守作業自体が，困難かつ時間的コストがかかる，と Lehman 等は言っている [25]．保守プロセスは IEEE によって図 4 のように規定されている [6] が，Analysis，Design 工程では，プログラムの正確な理解が必要であり，さらに修正による影響範囲の解析が要求される．複雑化したプログラムのモジュール間の結合関係をすべて把握して，変更による問題が発生しないことを確認することは，大規模なソフトウェアになるとコストがかかる上，見落とす可能性が高い．また，Testing 工程では，モジュール間結合度が高いと影響範囲が拡張し，膨大な再テストが必要になる．

## 2.4 本論文で使用する用語

本章では，本論文で使用するソフトウェアの保守に関わる用語について，説明する．ここでは，主に文献 [2] の説明を引用し，必要に応じて補足説明をする．

- レガシーソフトウェア (Legacy Software)：１０年以上にわたり，新しいニーズやサービスに対応するための要求に応じて変更・追加が行われながら，現在も運用中であるソフトウェア.
- 保守（Maintenance)：フォールト修正や性能向上，環境変化に対する適応のために引渡し (運用開始) 後にシステムを修正する工程.
- 保守性（Maintainability)：システムやコンポーネントをフォールト修正や性能向上，環境

変化に対する適応のために修正することの容易さ.

- 信頼性 (Software Reliability)：一定の期間安定した状態で要求された機能を実現するシステムの能力.
- 故障 (Failure)：規定された実施要求の中で，要求された機能を実施できないシステムの状態．ユーザが期待するシステムの動作に対して正常に動作しないことやシステムが停止する等のシステム稼動中に発生する問題現象に対して用いる.
- フォールト (Fault)：プログラム中に存在する正しくないステップ・処理・データ定義.
- フォールト報告データ：システムの開発・保守中に検出されたフォールトに関して作成される report．当該フォールトによって発生する故障 (failure) の現象，プログラムコード中のフォールトに関する情報，修正方法，影響波及調査，回帰テスト方法等様々な情報が記載されることが多い．多くのプロジェクトで作成されている.
- 複雑度（Complexity)：システムやコンポーネントが理解や検証が困難な設計や実装を持つレベル.
- 仕様 [書]（Specification)：システムやコンポーネントの要求や設計，ふるまい，特徴などを完全に的確で検証可能な方法で記述した書類.
- コーディング規約（Coding Standard)：プログラムの作成 (コーディング) における統一されたアプローチを規定するために採用され、強制される義務的要求.
- レビュー（Review)：コメントや承認を得るためのプロジェクト要員，マネージャ，ユーザ，顧客に対して製品を説明する工程又はミーティング.
- 回帰テスト（Regression Tesing)：修正が意図しない影響の原因となっていないか，システムが規定された要求に従っているかを検証するシステムの選択された項目の再テスト.
- 機能劣化（Deterioration)：変更前は正常に動作していた機能が，変更によって正常に動作しなくなること．Degradation，デグレーションなどとも言われる.
- デバッグ (Debug)：プログラムのフォールトを検出し，位置を特定し，修正すること.

## 2.5 過去のフォールト情報を用いた潜在フォールトの検出

本章では，レガシーソフトウェアの保守における信頼性の問題点を，プログラムコードの複雑化という視点から明らかにした．長年にわたり変更が加えられてきたレガシーソフトウェアは，

プログラムコードが複雑化し，保守に多大なコストがかかるにも関わらず，その信頼性の高さや社会的の重要度，あるいは，組込み系のシステムにおいては，開発期間の短縮やそもそものシステムの複雑さから，再設計や新システムへの移行が進まず，使い続けられている．一方で，プログラムコードの複雑化を防止するのは容易ではない．保守作業の時間的人的コストの制約から，対症療法的な変更や局所的な制御の追加などが行われることによって，プログラムコードの複雑化は進行する．プログラムコードの複雑化は，プログラムコードの理解を困難にし，そのために保守作業者がプログラムコードの変更時に意図しないフォールトを混入させてしまう原因となりやすい．また，長期にわたるソフトウェアの保守期間中に，システムを熟知した技術者が入れ替わったり，ソフトウェアの変更に対してドキュメントの保守が正確に行われないため，対象のシステムを正確に理解するためにはソースコードしかないことが，プログラムコードの理解をさらに困難にしている．現状では，そのようなフォールトの混入を防止する有効な手段がなく，手作業でのコードレビューやテストなどの従来の手段ではフォールトを見逃すことも多い．

本論文では，複雑化し理解が困難でフォールトが混入しやすくなったソフトウェアシステムを，安全に変更するための支援方法を提案することを目的とする．システムをあまり知らない保守作業者が，膨大なプログラムコードを理解する必要なく，低コストで確実に危険なプログラムコードの混入を防止するために，対象システムを熟知した保守作業者の記憶にのみ存在し明文化されていないシステム特有の構造やプログラムコードの結合関係から，フォールト混入の原因となりやすいものを抽出し，形式化する．形式化されたこれらの知識は，自動的にプログラムコードから危険なコードを検出するために用いることができる．さらに，検出される危険コードに関連する知識，混入するフォールトによる故障の現象や修正方法，テスト方法を提示することにより，危険なプログラムコードの混入を防止することが容易なると考えた．

このような危険なプログラムコードを検出するために，複雑化したプログラムコードに混入するフォールトには反復性が見られることに着目し，過去のフォールト報告データを利用できると考えた．フォールトの反復性とは，同じようなモジュール間の結合関係やプログラム構造上の弱点・欠陥に基づくフォールトが繰り返し混入することである．プログラム構造上の問題点に起因するフォールト混入は，再設計によるプログラム構造の変更がない限り，保守作業の過程でその問題を知らない保守作業者によって繰り返される可能性がある．対象システムを熟知した保守作業者は，実際のフォールトへの修正作業を通して，このような問題点に関する知識を蓄積し，コードレビューやテストに暗黙的に活用していると考えられる．そのため，多くの開発・保守プロジェクトにおいて作成・蓄積されているフォールト報告データは，新たに混入するフォールトを検出するために利用価値があると考える．

フォールトの検出方法には，大きく分けてテストなどプログラムを実行させて検出する動的方

法とプログラムコードの解析やコードレビューなどの静的方法が考えられるが，本研究では特にコーディング工程において静的方法で検出することを目指している．一般的に，フォールトは早期に発見される方が対応コストが低い，と言われている．コーディング工程でフォールトを含むプログラムコードを直接検出することは，テスト工程で発生する故障から原因となるプログラムコードを検出するより低コストである，と考えられる．また，2.3 節で述べたとおり，本論文で検出を目指す複雑化したプログラムコードに起因するフォールトは，故障現象を再現することが困難であったり発生頻度が低かったりするため，テストなどによる動的な検出では見逃してしまう可能性も高い．フォールト報告データによって，問題のプログラム構造が明確である場合，フォールトを含む可能性があるコードをあらかじめ静的に検出しておくことは可能であり，それらのコードをチェックすることによって，実際に故障の原因となるフォールトの検出が容易になると考えている．

# 3. 暗黙的コード制約の形式化

## 3.1 暗黙的コード制約の特徴と問題点

本論文では，プログラムの複雑化という現象を保守作業者の視点から「フォールトの混入を防止するために遵守すべきコーディング上の制約の発生」と捉え，それらの制約のうち一部の保守作業者の記憶にのみ存在し，繰り返し違反が行われると予想される暗黙的な制約を「暗黙的コード制約」と呼ぶ.

暗黙的コード制約とは，一般的にプログラミング工程においてあらかじめ取り決められ，保守作業者に周知徹底されている制約と異なる．プログラミングにおけるコーディング制約には，大きく分けて２つの種類がある．１つはプログラムの理解性や保守性，あるいは移植性を向上させるための命名規則や関数仕様，ファイル形式などのコーディング上の規則で一般的に「コーディング規約」と呼ばれるものである．他方は制約に従わないコーディングをすることによって正常な機能の動作が保証されない，すなわちフォールト混入の原因となる制約のことで，ライブラリ仕様やプログラミング文法などがある．一般的にこれらのコーディング制約は，システム開発前にプロジェクト全体，あるいは全社的に取り決められ，全員が遵守することを要求される．

暗黙的コード制約は，後者の一種であるが，ライブラリ仕様やプログラミング文法などとは異なる以下の特徴をもつ.

- 開発・保守組織において予め定められたものではなく，長期にわたる保守の過程で付随的に発生するものである．その多くは，設計書や仕様書などに明記されず，一部の保守作業者のみがその存在を知っている，という状況にある．そのため，保守作業者の退職や入れ替わりによって，制約に関する知識は失われやすい.

- システムの開発・保守過程で発生するため，そのシステムに特化した制約となることが多い．つまり，一般的なプログラミング方法を定めた制約（文法やライブラリ仕様など）とは異なり，異なるシステム間で共用できないものが多い.

- 既存のコンパイラ，チェックリストおよびツールによって暗黙的コード制約に違反している部分を発見することは困難である．普及しているチェックリストは，一般的に間違いやすい問題をチェック項目とし，また，lint や purify[49] のようなチェックツールは，言語文法上の問題やメモリ操作上のフォールトなど問題の対象を限定しているため，対象ソフトウェアに特化した制約違反の抽出には利用できない.

- 暗黙的コード制約を解消するための再設計は，リスクが高くコストがかかるため，容易ではない．再設計には，ソースコードの正確な理解が必要だが，正確なドキュメントがなく

熟練した保守作業者がいないと，複雑化したコードの理解には時間がかかるばかりではなく，時には新しい制約を混入してしまったり，システムの信頼性を低下させることがある．再設計手法として Refactoring[11] があるが，上記と同様の問題が存在するため，適用は必ずしも容易ではない．

暗黙的コード制約は，開発・保守の過程で発生するため，長期にわたる保守が行われてきたレガシーソフトウェアには多数存在する．本論文では，これらの制約は信頼性の低下や保守コストの増大の 1 原因として捉えているが，その理由は以下の通りである．

- 暗黙的コード制約に違反することはフォールト混入の原因となるため，すべての制約に違反するコードを検出するためのコードレビューを行う必要がある．
- 暗黙的コード制約違反に起因する故障を検出するために，プログラムコードの変更時には暗黙的コード制約違反で発生する可能性がある故障ケースに応じた再テストが必要である．
- 暗黙的コード制約を知らない保守作業者によってプログラム変更が行われた場合，上記のようなコードレビューやテストケースに漏れが生じ，フォールトを混入させてしまう可能性が多い．
- 暗黙的コード制約は，存在する限りフォールト混入の危険は解消されないため，保守が長期化すると，制約を知らない保守作業者によって，制約違反によるフォールトが繰り返し混入する可能性がある．

## 3.2 暗黙的コード制約の発生

このような暗黙的コード制約が発生する状況の一例を図 5 図 6 に示す．ある複数の画面モジュールを持つシステムで，各画面でのリピートキー機能を追加する作業が発生し，その仕様として画面消去時にはリピートキー機能を無効にする，すなわち SetRepeatKey(OFF) を呼び出すという変更が必要になった．その仕様に対して，この作業者は画面消去共通関数 BaseFrameClear() に SetRepeatKey(OFF) を追加している．これは，時間制約や修正量を考慮した上で，この変更が最も確実でコストも低い，と判断したからである．しかし，この変更によって，リピートキー機能を有効にするためには，SetRepeatKey(ON) に続いて BaseFrameClear() を呼び出してはいけない，という制約を発生させてしまっている．

同時期に，別の作業者が図 6 のような状況で，フォールトを混入させてしまう．画面 B においてある画面からの遷移時に画面にゴミが表示される，という故障に対して，この作業者は画面 B

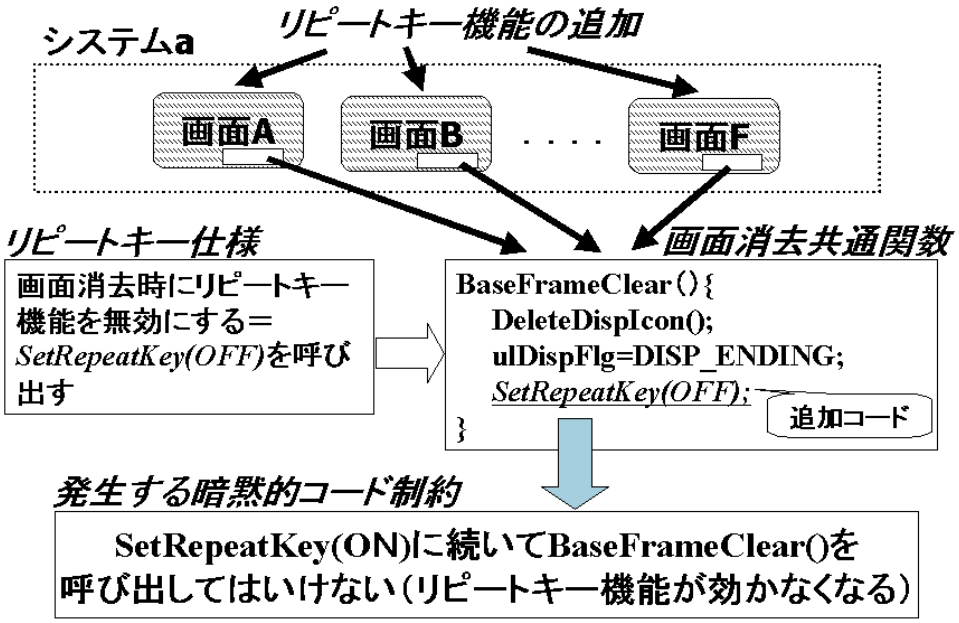


図 5 暗黙的コード制約の発生例

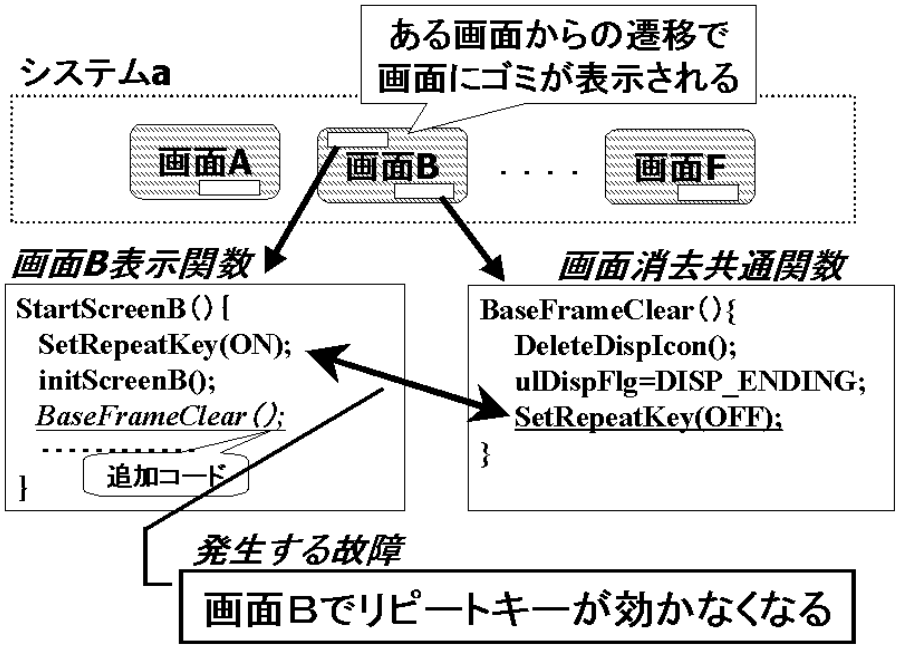


図 6 暗黙的コード制約違反による故障発生例

表示関数 StartScreenB() に BaseFrameClear() を追加している．この変更によって，制約違反が発生し，画面 B においてリピートキーが効かなくなる，という故障を発生させてしまう．

複数の保守担当者によって並行してさまざまな作業が行われる大規模なレガシーソフトウェアでは，図 5 図 6 のような状況で，コードレビューや単体テストで発見が困難なフォールトが混入することがしばしば発生する．図 5 のケースでは，設計された当初の BaseFrameClear() の仕様を逸脱する変更がなされたため，制約が発生してしまっているが，この時点ではフォールトは混入していない．一方，図 6 の保守作業者は BaseFrameClear() の本来の仕様を知っているため，この変更による影響でリピートキー機能が効かなくなるという結果を予測できず，テストやコードレビューで見逃されてしまったと考えられる．

上記のような暗黙的コード制約が混入する場面において，暗黙的コード制約が発生しないような設計をしたり，制約発見時に制約を解消するような変更を行うことは可能であるが，実際には解消されずにシステムに残留していることが多い．その理由としては，制約発見時に再設計をする時間がなかったり，１つの制約解消のために関連するプログラムが異なる組織や作業場所で保守されているなど再設計には困難な状況がある．また，保守作業者が入れ替わることで，制約を作成した作業者の意図が正確に把握できないため，再設計ができない場合もある．図 6 のケースにおいては，リピートキー機能解除と画面消去という本来独立した機能を一関数に盛り込んだという問題点を解消することなく，BaseFrameClear() と SetRepeatKey(ON) の処理順序を変えることで，問題を回避している．これは，図 5 のような制約が発生しないようにすることは，他の多くのモジュールに影響を及ぼすために再レビューや再テストが困難であったからである．

この他に，既存のレガシーシステムのフォールトを調査した結果，以下のような暗黙的コード制約の発生要因が見られた．

- 他のシステムからの取り込み

  複数システムで再利用されているプログラムコードで，あるシステムでの変更を別システムに取り込むケース．取り込み先のシステムである変数の使用方法に関してそのシステムに特有の変更を加えている場合，取り込み元のプログラムコードが前提としている変数の内容と一致しなくなり，取り込みプログラムの機能の使用に制約が必要になる．

- データ仕様変更

  従来のデータ仕様の変更に対して，プログラムでは変更されたデータ仕様に対応できる設計になっていないケース．仕様変更されたデータを扱う際に特殊な処理が必要になり，その対応を行わないと故障が発生する．

- 新機能の追加

従来のシステムにない機能を新たに追加したケース．追加した機能があるタイミングで実行された時に既存の機能と競合するシステムリソース（タイマやメモリ等）を使用していたために異常が発生してしまう．

- 関数・変数の機能の拡張

  設計時に設計者が意図していた関数や変数の仕様を逸脱して用いたり，意味を拡張するような変更を加えるケース．新しい機能やフォールト修正に対応するために既存の関数や変数の仕様を変更した場合，従来の仕様どおりにそれらの関数や変数を使用すると，意図しない動作をする．(図 5 の例など)

- 不十分な設計や局所的な修正

  新しい変数を追加する際に，その仕様を明確に規定できていなかったり，その変更に特化した明確な意味を持たない変数（フラグ）を追加したケース．新しい変数が，実際には特定の条件やタイミングでは正確にその情報を表していないため，参照範囲が限定されていたり，明確な意味を持たないフラグの追加により修正部分にのみ特有の制御構造が発生したりすることがある．このようなケースでは，次の修正時にその変数の参照・変更に注意が必要だったり，修正漏れが発生したりする可能性がある．

- 本来の設計に違反するコードの混入

  開発当初の設計に違反した使用方法で関数や変数を使用してしまうケース．設計書や仕様書が正確でない場合，プログラムコード中にある使用例を参考にして，コピーしてプログラミングをする場合がある．そのため，一旦違反したプログラムが混入してしまうと，同じ間違いが繰り返される可能性がある．偶然，１つ目の違反では故障の原因にならなかったが，別の箇所で同じ違反コーディングを行った場合に，故障の原因となる．

## 3.3 暗黙的コード制約から見たフォールトの現状

あるレガシーソフトウェアの一サブシステムを対象として，フォールトと暗黙的コード制約の件数を調査した（このソフトウェアの詳細については，5.2 節で述べる）．この調査では，テスト・運用工程で故障が発生し報告されたフォールト報告データから，故障の原因がコード上明確に指摘されているケース 165 件中 54 件が暗黙的コード制約の違反によるものであることがわかった．実際に存在する暗黙的コード制約は 45 個で，9 件の故障は原因となる制約が共通のフォールトであった．このうち，2 つの暗黙的コード制約は後日制約を解消するように再設計されたが，残り

の 43 個の暗黙的コード制約は現在も残っており，将来の保守作業でフォールト混入の原因となる可能性がある．

これらのフォールトの混入状況を詳細に見ると，明らかなレビュー漏れや複数の作業グループによる同じ制約違反によるフォールトが存在する．ある制約に関しては，調査したソフトウェアを改造した別バージョンの開発においても故障を発生させている．このように，暗黙的コード制約に基づくフォールトは，人手による検出は容易ではなく，かつ長期にわたる保守や再利用によって繰り返し混入していることが確認できた．

保守現場において，暗黙的コード制約違反に起因するフォールト (故障を発生させるコード) は，テスト工程や運用工程での故障発生時以外に，以下のような方法で検出することが可能である．

1. 故障が発生・報告された時に，その調査から制約が見つかることがある．見つかった制約に違反するコードパターンをすべてのコード中から手作業で検索し，同じような故障が発生するかどうか確認する．

2. コードレビュー時に，保守作業のため変更・追加されたコード中から過去に発生した故障原因と同じパターンのコードの存在を確認する．

1．では，発生する故障内容や原因が明確に把握された状態であるため，検出が比較的容易であり，実際に現場では暗黙的に行われている．フォールト報告データを見ると，実際に 1．のような方法で類似したフォールトを多数検出し，報告された故障と同時に修正された実績がある．しかし，大規模なレガシーソフトウェアになると，すべてのプログラムから制約に違反するパターンのコード検索を手作業で行うことは大きな負担となる．さらに，保守作業者の経験が少なかったり保守作業者がシステムをよく知らなかったりすると，見逃す可能性も高い．2．ではチェックするべきコード範囲が絞り込めるが，過去に発見されたすべての暗黙的コード制約を把握していることが必要である．コードレビューなどで，システムを熟知する保守作業者によって修正コードの危険箇所や修正による 2 次不具合（デグレーション）を指摘されることもよく見られるが，現状では保守作業者の記憶に頼って行われている．

## 3.4 制約違反パターン

暗黙的コード制約に対して，制約に違反している部分のソースコードをパターン化したものを「制約違反パターン」と呼ぶ．実際にケーススタディで抽出された暗黙的コード制約の例とそれに対する形式化前の制約違反パターンを表 1 に例示する．本研究では，この制約違反パターンに一致するコード（以降，制約違反コードと呼ぶ）を自動検出することを目的とする．

表 1　暗黙的コード制約と制約違反パターンの例

| | 暗黙的コード制約の例 | 制約違反パターンの例 |
|---|---|---|
| 1 | ある特殊キーによる割込み後，元の画面に正常に復帰するためには関数 B の呼び出し前に大域変数 A にページ番号を設定しなくてはいけない | 関数 B の呼び出し前に大域変数 A への代入命令がない |
| 2 | 関数 X は関数 Y で設定した機能を打ち消す機能を持つため，その機能が正常に動作するためには，関数 Y と関数 X を連続して呼び出してはいけない | 関数 Y の呼出し後に関数 X が呼び出される |

制約違反パターンは表 1 に示すとおり，変数の値や関数の実行順序などの動的なパターンは，静的なパターン，つまり静的解析から得られる構文要素や文字列のみのパターンに置き換える．その結果，これらの制約違反パターンはコーディング工程で検出可能であり，動的に検出するためのプログラムの実行環境を必要としない．そのため，保守作業者が任意の作業中にいつでも検出でき，また異なるシステムへの移植などで動的な環境や操作方法が変わっても，それに依存することなく検出可能である．

一方，動的なパターンを静的なパターンに置き換えることによって，コードが制約違反パターンに一致したからといって必ずしも故障の原因となることを意味しなくなる．それは，動的条件の置き換えが不完全であったり，条件が欠落するためである．本来,「故障 (=Failure)」は，システムが仕様にない，もしくは仕様に反する “状態” であった場合，発生する．例えば，表 1 の１の例では，本来検出するべき故障の状態は「ある特殊キーによる割込み後，元の画面に復帰する時に，大域変数 A にページ番号が設定されていない状態」であり，２の例では「ある機能を動作させる操作をする時に，その機能の設定が打ち消されている状態」である．たとえ，制約違反コードでも，別の処理ですでに大域変数 A にページ番号が設定されていたり（表 1-１），制約違反コードに続く処理である機能の設定が再度行われる（表 1-２）場合，故障の発生する状態にはならない．

さらに，表 1 の暗黙的コード制約から制約違反パターンへの置き換えでは，以下の条件によって故障が発生しない可能性がある．

- 「ある特殊キー押下による割り込み」が発生しない（表 1-１）
- 「元の画面」が大域変数 A を参照しない（表 1-１）
- 「ある特殊キー押下による割り込み」後「元の画面」に復帰しない（表 1-１）
- 「ある機能」の動作を期待しない（表 1-２）

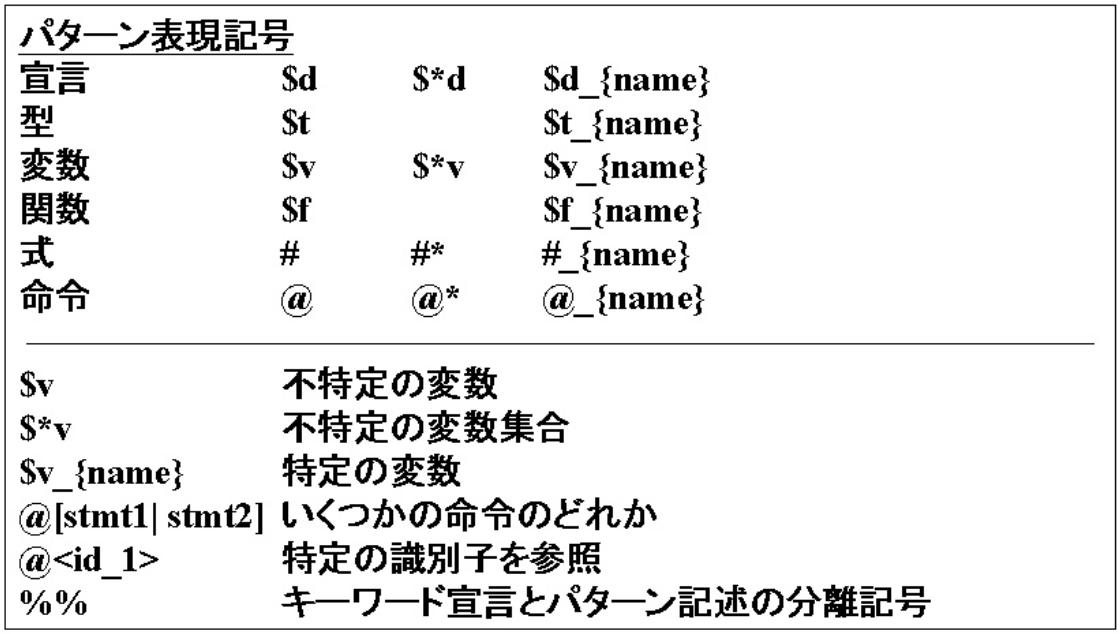


図 7 パターン表現記号一覧 (文献 [41] から)

このような動的パターンの静的パターンへの置き換えによって，制約違反コードから実際に故障を発生させるフォールトを発見するためには，さらにコード解析やテストなどの確認作業が必要になり，確認作業には「フォールト情報」が不可欠となる．フォールト情報の詳細に関しては，4.2.1 項に述べるが，表 1 の例では,「ある特殊なキーによる割り込み後元の画面に正常に復帰する」や「ある機能」という具体的な故障に関する情報がフォールト情報となる．故障を発生させるためには，非常に発生頻度の低い状態を再現させる必要があったり，特殊なデータが必要となる場合もある．そのような場合には，故障発生の条件や操作方法などの情報を提示されることによって，制約違反コードが故障発生の原因となるかどうかを確認することは容易になる．このように，制約違反パターンは，フォールト情報と関連づけられることによって，有効利用が可能なものとなる．

## 3.5 制約違反パターンの形式化

制約違反パターンの形式化は，プログラムコード中からパターンマッチングによる一致コードを検出する目的で行われる．制約違反パターンの形式的記述には，文献 [41] でのプログラムパターン記述言語を拡張して使用する．この記述言語は，コードの再利用やコード理解のためのソース

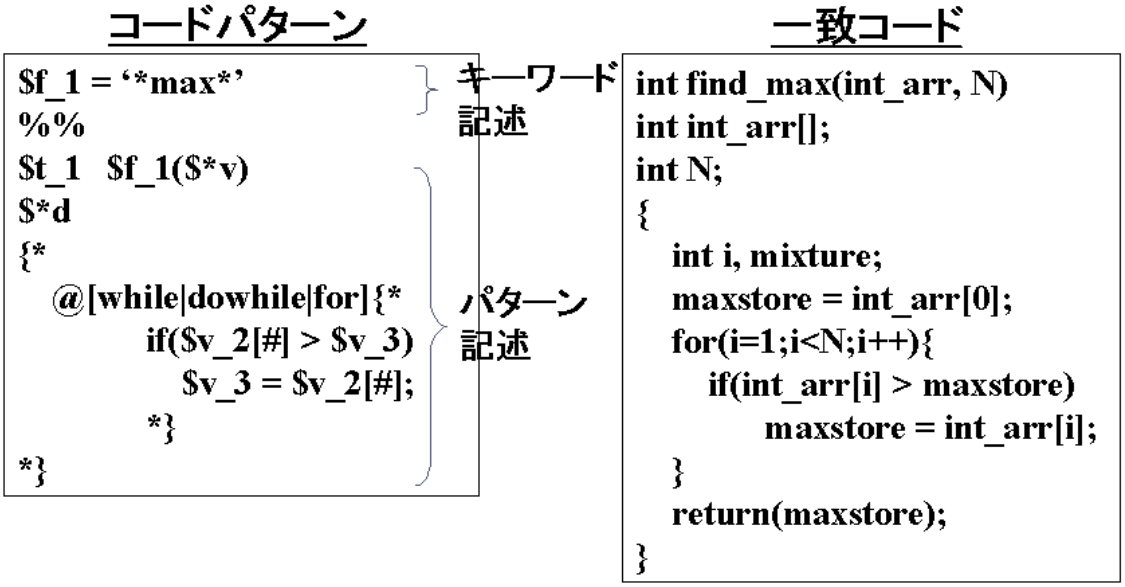


図 8 “配列から最大値を求めるアルゴリズム” のパターンの記述例 (文献 [41] から)

コード検索を目的に作成されたが，単純な文字列パターン検索ではなく「変数」「関数」などの構文要素の意味を考慮した記述が可能で，より記述者の意図を反映したパターン検索が可能である．また，記述言語がプログラミング言語に類似するため，記述が容易で拡張性も高い．実際に問い合わせを行う場合に使用する構文要素の主なものは図 7 に挙げるもので,「保守者が典型的に捜すものという認識に基づいて選択された」と文献 [41] には述べられている．さらに，もし他の構文要素の追加が必要な場合，システムの基本的なデザインを変更することなくパターン言語を容易に追加できる，と述べられている．この記述言語を用いたパターンの記述例を図 8 に示す．

今回は，プログラムパターンでも特にフォールトとなるような制約違反パターンを記述する．レガシーソフトウェアのフォールトを調査した結果，制約違反パターンを記述するために必要と判断した以下の拡張を行う．(図 9 に拡張した構文要素を示す)

1. ^ 記号の追加

   必要な処理が欠けているという制約違反パターンが多い．そのため，必要な命令が存在しない（必要な命令以外の任意のコードが存在する）というパターンの記述が必要である．図 9 例 1 の “^@” が命令の不在を表現している．

2. $vg 記号の追加

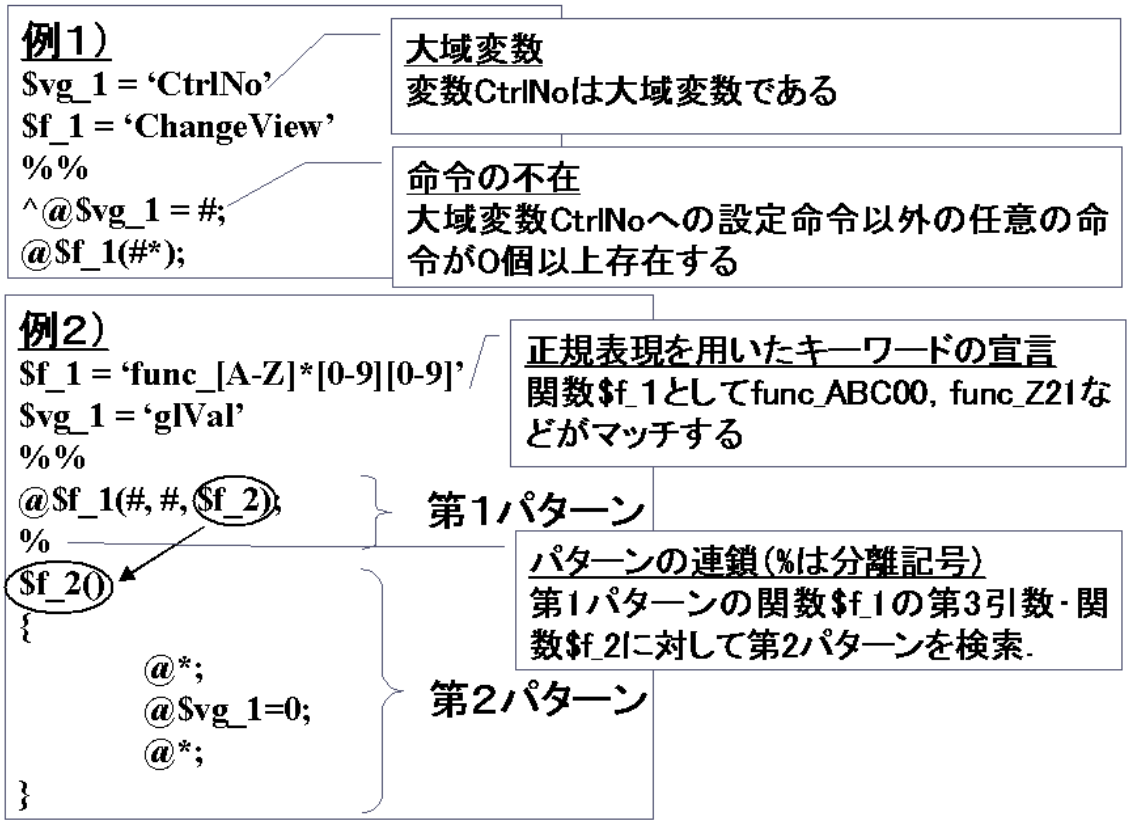


図 9 パターン記述の拡張

ローカル変数に対して大域変数が持つ特徴に起因したフォールトがある．例えば，１つの大域変数が多数のモジュールで設定・参照されている場合，ある変更が他のモジュールに影響し，フォールトの原因になる．このように，ローカル変数と大域変数の意味の違いは，フォールトに関しては大きいため，特に大域変数を示す記号が必要となる．大域変数は,「関数内に該当する変数の宣言が存在しない」というパターン表現で記述可能だが，この記号を追加することによって記述が簡易化される．基本的な使い方は$v と同じである．図 9 例 1 の 1 行目で，変数 CtrlNo が大域変数であることを表現している．

3. 正規表現の追加

   大規模なソフトウェアでは，変数・関数の命名規則などが厳格に決められている事が多く，それを指定できれば，より正確なフォールトの検出が可能になる．例えば，特定の処理に関わる関数群について，命名規則を適正に表現することによって，複数の関数を同時に指定できるとともに，将来追加される関数も対象に含めることができる．そのためには，grep などで用いられる正規表現を採用することが，有効であると考える．図 9 例 2 の 1 行目で，関数$f_1 は，“func_” に続き，任意の数の英字大文字＋数値 2 桁という命名規則に基づく関数であるということを表現している．

4. 連鎖パターンの追加

   複雑な構造を持つソフトウェアにおけるフォールトは，原因となるコードが複数の関数やファイルに分散していることが多い．そのため，複数の連鎖したパターンを検索する必要があり，そのための記号を追加する．連鎖するパターンは “%” で分離され，検索は記述順に行われる．図 9 例 2 では，まず第 1 パターンに一致するコードを検索し，一致コードの $f_2 にあたる要素をキーとして，第 2 パターンが検索される．

上記の制約違反パターンを記述言語で記述した実例を付録 B に示す．

## 3.6 提案する形式化の限界と問題点

提案する制約違反パターンの抽出と形式化の方法では，以下のような問題点が考えられる．

1. 動的パターンの欠落

   3.4 節にも述べたとおり，制約違反パターンには動的な故障発生要因となる条件を完全に正確に記述することはできない．また，制約違反パターンが完全に記述できたとしても，検出される制約違反コードが動的条件に依存して故障を発生させないケースもある．図 10 で

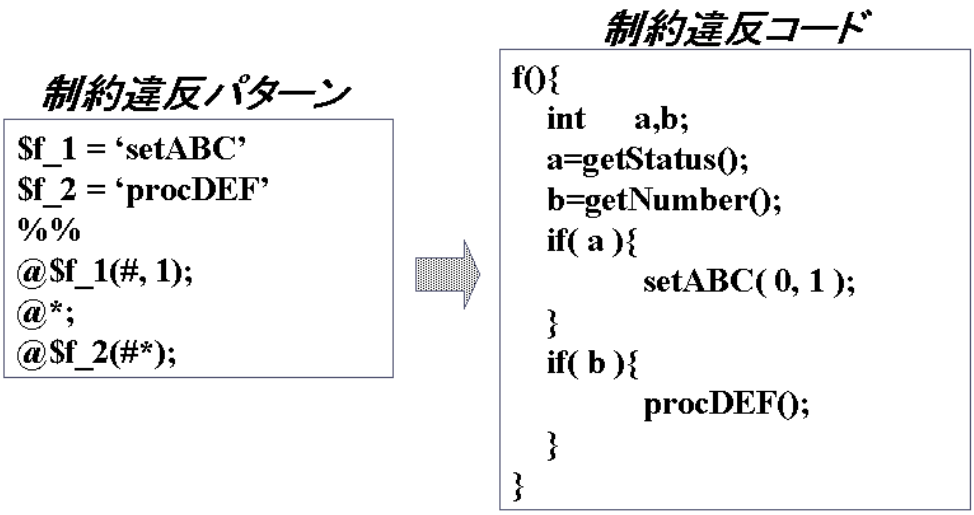


図 10 動的条件付の制約違反コード

は，関数 setABC() と関数 procDEF() が連続して呼出されることによって必ず故障が発生する制約違反パターンであるが，変数 a と変数 b の動的な値に依存するため，この制約違反コードが故障を発生させることは保証されない．この限界の中でより有効な制約違反パターンを記述するためには，対象となるシステムを正確に理解することや故障状態を詳細に把握することが求められる．現状では，この工程は熟練者による手作業に依存する．

2. 記述できない暗黙的コード制約

「関数 A は関数 B をコピーして作成されたものであり，関数 A への変更は関数 B に反映させる必要がある」，あるいは,「関数 C への追加処理には，ある条件を付加しなければならない場合がある」といったようなコード変更時の制約は，変更前と変更後の差分から問題となるコードを検出するなどの別の記述方法をとる必要がある．これらは静的なコードパターンとして記述が可能であるため，提案する記述言語を拡張することによって対応可能と考えられる．

3. フォールトではないコードの検出

3.4 節に記述した通り，動的な条件を記述できない等の制約により，フォールトではないコードが検出される可能性があり，検出されるコードが大量である場合，制約違反パターンとして実用的でない．手作業によるフォールトの有無の確認が必要であることを考慮し，検出対象となるプログラムコードの範囲を絞り込む（例えば，変更されたコードのみを対象

とする），もしくは，パターン自体に検出数を絞り込む条件を付加する等の対応が必要になる．ただし，後者の場合はフォールトを検出できなくなる危険性がある．

4. 対応済みのフォールトコードの検出

   制約違反コードが実際に故障の原因であったとしても，修正による変更で制約違反パターンに一致しなくなるわけではない．制約違反であっても，そのコードが必要で，他の方法で必要な機能が実現できないため，故障を回避するために別の変更で対応する場合もある．このようなケースでは，毎回フォールトでないコードが検出され，毎回同じ確認作業が必要になる．検出結果とチェックの情報を再利用し重複コードのチェックを排除する仕組みも考えられるが，一方システムの仕様変更などの状況の変化によりフォールトになる可能性もあるため，単純に切り捨てることはできないと思われる．

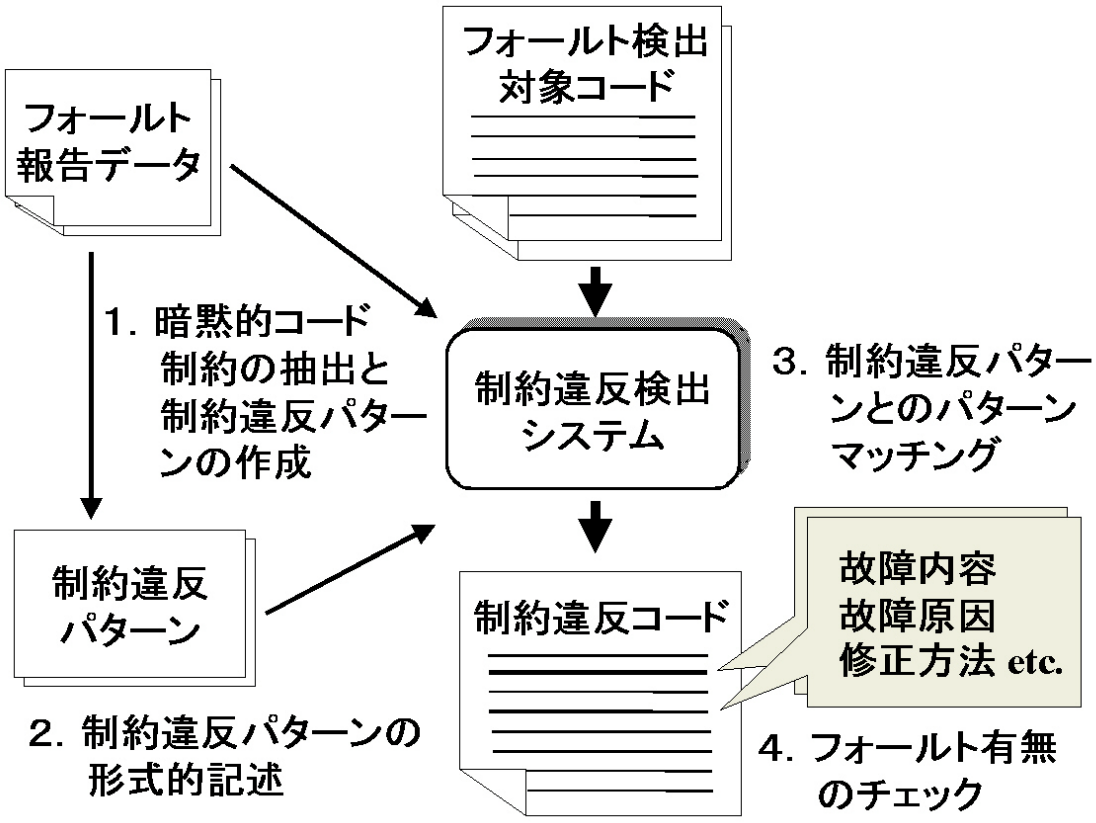


図 11 制約違反検出システムを使ったフォールト検出の手順

# 4. フォールト検出のための提案方法

## 4.1 提案するフォールト検出方法の概要

提案方法は，制約違反パターンに一致する部分のソースコード（制約違反コード）を検出・警告する．提案方法のうち，自動システム化した部分を，以降,「制約違反検出システム」と呼ぶ．提案方法によるフォールトの検出手順は，以下のとおりである（図 11 参照）．

1. 保守作業者は，過去のフォールト報告データを調査し，故障の原因となっている暗黙的コード制約を抽出する．次に，抽出された暗黙的コード制約に違反する制約違反パターンを作成する．

2. 制約違反パターンを形式的に記述し，制約違反検出システムに保存する．

3. 制約違反検出システムは，入力されたソースコードと保存された制約違反パターンでパターンマッチングを行い，一致する部分のプログラムコードを出力する．

4. 保守作業者は，制約違反検出システムから出力される制約違反コードを調査し，フォールトであるかどうか（すなわち故障の原因となりうるかどうか）を過去のフォールト情報を元に判断する．

以降，各手順の詳細について各節で説明する．

## 4.2 暗黙的コード制約の抽出と制約違反パターンの作成

### 4.2.1 フォールト報告データ

暗黙的コード制約は，フォールト報告データから抽出できる．フォールト報告データとは，プロセスデータの一つで，多くの企業では，故障発生の報告や修正・リリース時の手続きに伴って，これらの書類が作成される．今回使用したフォールト報告データには，フォールト発生報告書，不具合修正報告書，ファイル更新通知の 3 種類があるが，それぞれ故障の発生状況や原因，修正作業の内容，テスト仕様とその結果，リリース情報などの一連の情報が記載されている．(付録 A に今回ケーススタディで使用したフォールト報告データの例を添付する．）本方法でこのフォールト報告データを利用する際に重要な情報としては，故障の内容，原因，修正方法などが挙げられる（図 12 参照）．暗黙的コード制約は主に故障の原因から抽出される．しかし，その他の情報も制約違反コードが実際に故障を発生させる原因になりうるかどうかのチェックや故障の発生の確認，フォールトの修正に際して重要な情報となる．

### 4.2.2 制約の抽出とパターン作成

暗黙的コード制約を抽出する元になるフォールトの条件には，以下のようなものが考えられる．

- ソースコード上のフォールト存在個所が明確である．
- 同一原因による故障が今後も発生する可能性がある．(原因の根本的な解決を行っていない）

暗黙的コード制約からの制約違反パターンの作成には，対象ソフトウェア全体の知識や将来の変更の見通しなどが必要である．制約違反パターンの作成には，それが今後の保守においてフォールトの検出に役立つパターンであるかどうかを判定しなければならない．役立たないパターンを作成しても，実際の検出に時間がかかるばかりではなく，保守作業を増大させることになる．ま

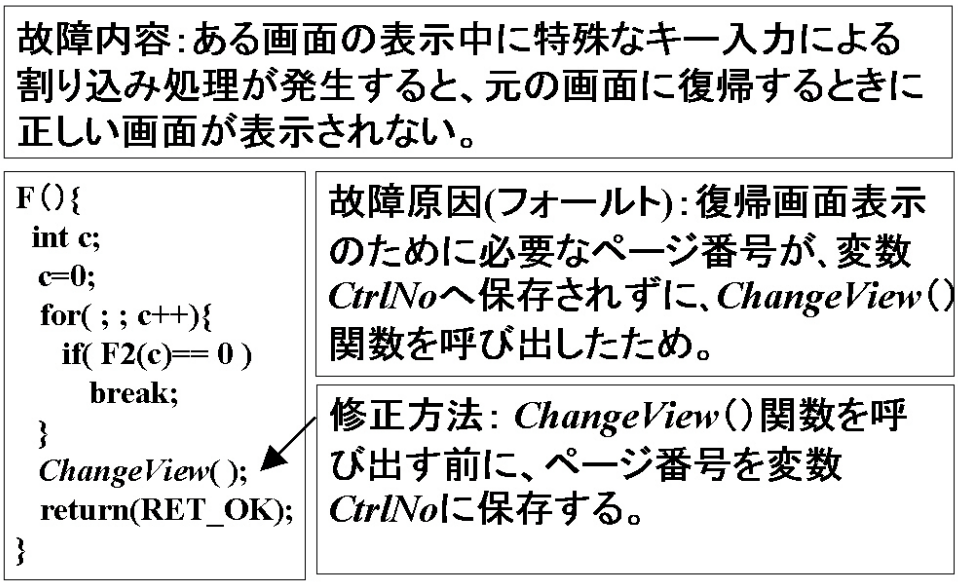


図 12 フォールト報告データの例

た，マッチング元になる制約違反パターンを正しく作成しないと，チェックが不要な箇所が多く検出されたり，逆に検出されるべき箇所が検出できないといった問題が発生する．

上記のような問題に対して，開発プロセス中に制約違反パターンの抽出を意識した活動がある程度必要になる．故障の原因解析時やコードレビュー中に将来有効と思われる制約違反パターンの有無を確認したり，抽出された制約違反パターンを分類しガイドラインを作成したりすることが有効であると考えられる．

フォールト報告データの例（図 12）から抽出される暗黙的コード制約は「ChangeView() 関数を呼び出す前には，CtrlNo へ表示中のページ番号を代入しなければならない」となり，制約違反パターンとしては「CtrlNo への代入無しに ChangeView() 関数を呼び出す」というように作成する．

このように作成された制約違反パターンは，3.5 節で説明した記述言語を用いて，形式化される．「CtrlNo への代入無しに ChangeView() 関数を呼び出す」という制約違反パターンに対する形式的記述例は，図 9 の例 1 のようになる．

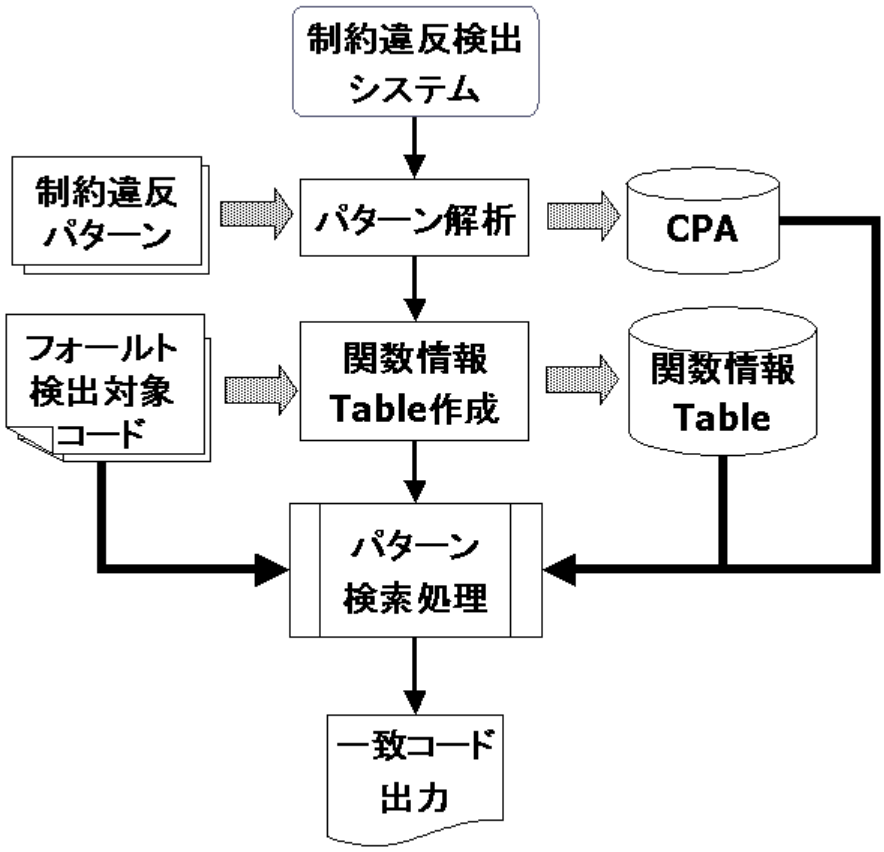


図 13 制約違反検出システム処理の流れ

### 4.3 制約違反検出システム処理の流れ

制約違反検出システムは，提案方法の自動化された処理部分で，図 13 はシステムの流れを示す．本研究では，C 言語をターゲットとした制約違反コードを検出するシステムを開発した．制約違反システムへの入力は，形式的記述言語で記述された制約違反パターンとフォールト検出対象コードである．制約違反パターンは，CPA（Code Pattern Automata）に変換される．検出対象コードからは，関数情報 Table が作成される．関数情報 Table は，パターン検索処理を効率的に行うために使用されるが，これは，以降で詳細に説明する．関数情報 Table には，構文解析により取得できる検出対象コード中に存在する関数名，関数の実体が存在するファイル名，関数の開始行番号，終了行番号を保存する．さらに，各関数から呼出される関数名と呼出し行番号を同時に取得する（図 14 参照）．

パターン検索処理は，CPA，関数情報 Table，検出対象コードを用いて行われるが，その処理

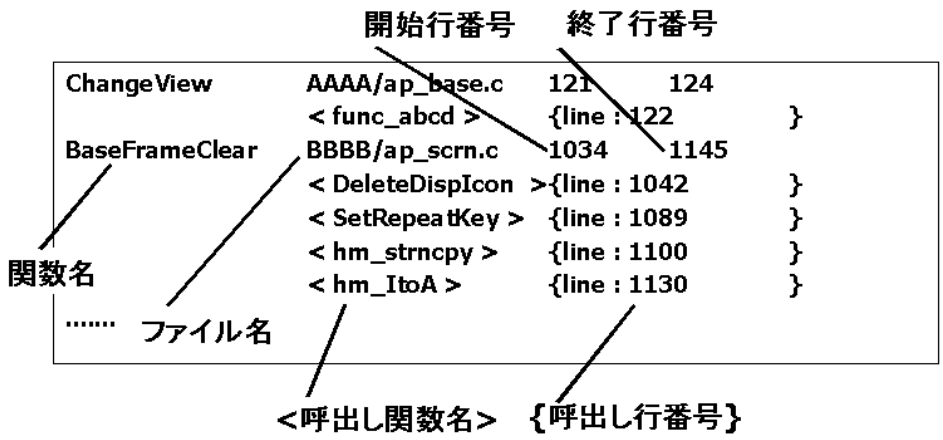


図 14 関数情報 Table

の流れを図 15 に示す．まず，制約違反パターンの最初のキーワード（キーワード記述部分で*付きのもの）を検索し，キーワードが存在するファイル名を取り出す（図 16 参照）．さらに，パターンを「命令処理型」とそれ以外に分類し，命令処理型（図 16 のパターンなど）の場合は，そのキーワードが存在する関数情報を関数情報 Table から取得する．これによって，パターンマッチングを行うプログラムコードの範囲を関数単位に絞り込むことが可能になる．命令処理型以外のパターンとしては，データテーブル型などが考えられるが，これらに関しては，ファイル単位でのマッチングを行う（図 17 の第 1 パターンなど）．このように検出対象コードを絞り込んだ上で，実際にパターンマッチングを行い，一致コードが検出できたときは，それを制約違反コードとして出力する．連鎖するパターンが存在するときは，次のパターンへのキーワードを取出し，キーワード検索からの処理を繰り返す．

本システムでは，命令処理型のパターンマッチングにおいて，マッチング対象のコードを呼出し関数のコードまで拡張し，幅優先で検索処理を行う．制約違反パターンに合致するコードは呼び出し関数間にまたがって存在する場合があるため，キーワード検索で検索対象として取り出した関数内のコードのみを検索しては，検出したい制約違反コードが確実に検出できないパターンが多い．例えば，図 17 の第 2 パターンのパターンマッチングにおいて，第 1 パターンから抽出したキーワード検索で$f_4 に一致する関数 func_y() が取り出されるが，この関数には制約違反パターンに一致するコードは存在しない．この場合，関数 func_y() から呼び出されている関数にマッチングの範囲を拡大し，呼び出された関数内に制約違反パターンに一致する処理が存在する場合，それを一致コードとして取り出すことにする．このとき，呼出し関数の情報，つまり呼出し関数名やその関数定義が存在するファイル名やファイル上の位置などを取得するために，関数

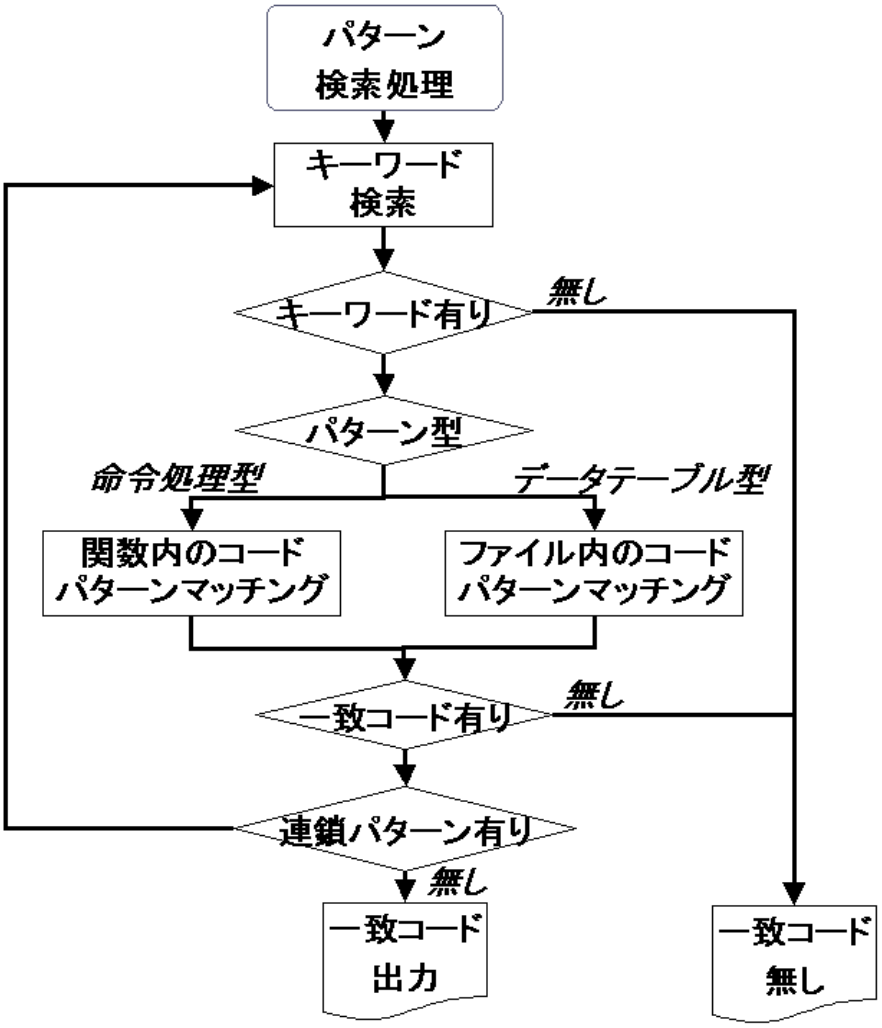


図 15　パターン検索処理の流れ

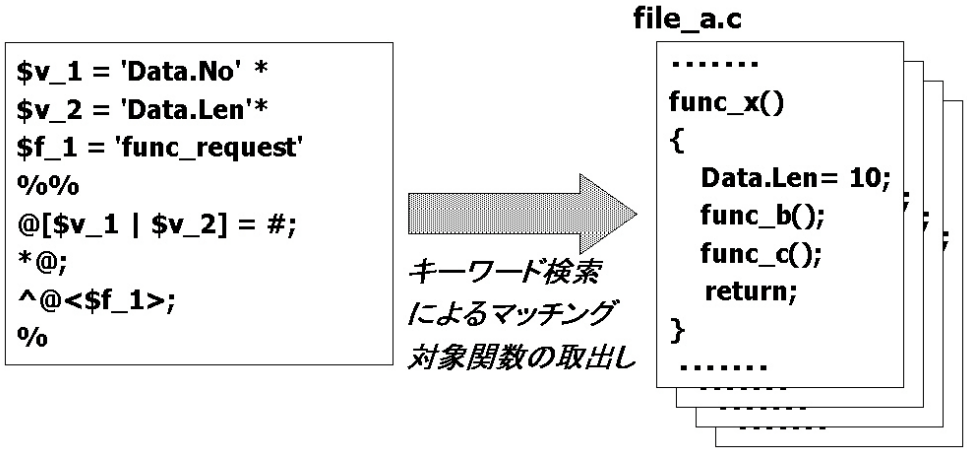


図 16　キーワード検索

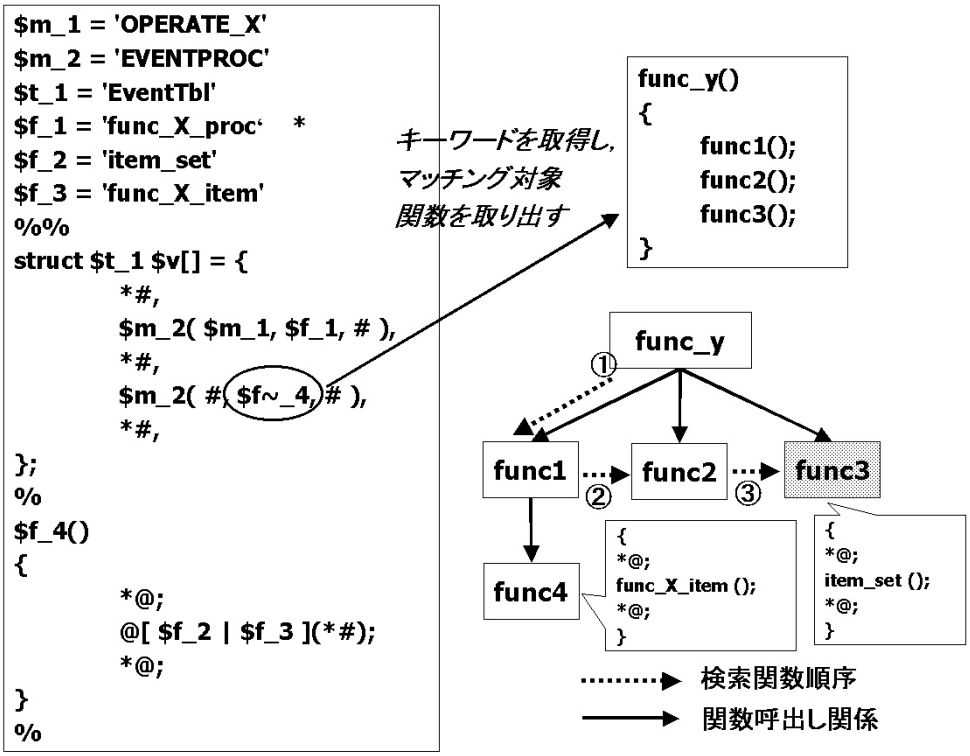


図 17　呼出し関数検索処理

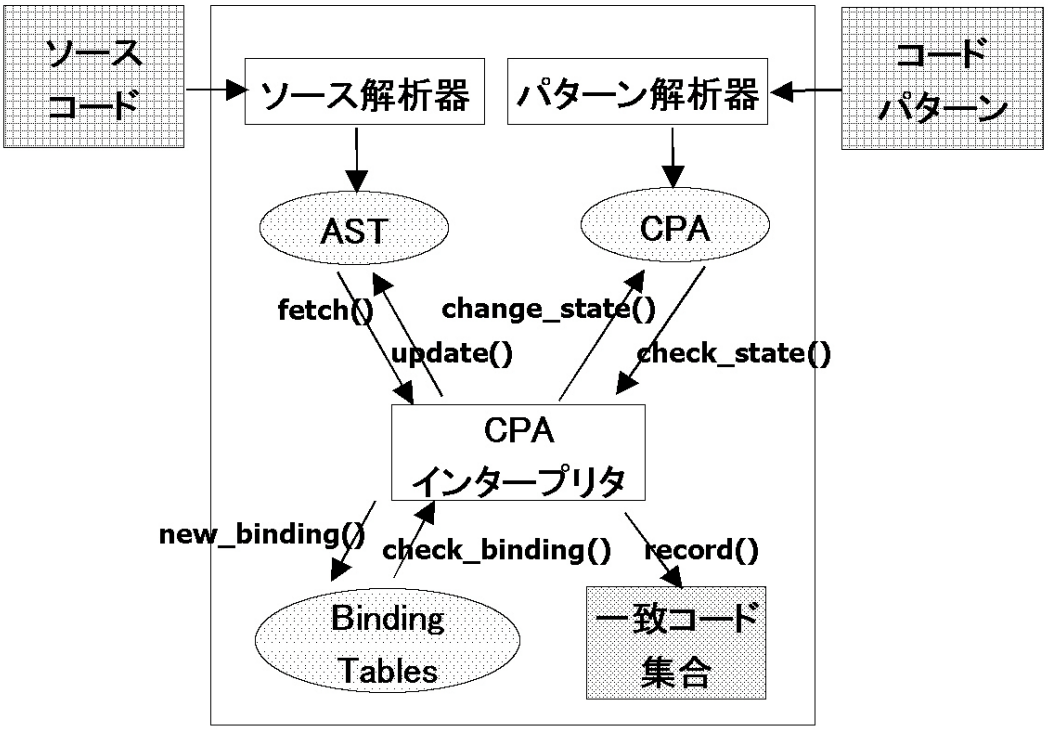


図 18 SCRUPLE システムのアーキテクチャ (文献 [41] から)

情報 Table を参照する．図 17 では，幅優先で検索した結果，関数 func_y() の呼出し関数 func3() で一致コードが見つかっている．

本システムでは，呼出し関数への検索範囲の拡張に関して，実装上の問題（メモリ不足等）により，呼出す関数の深さを 3 までに制限している．また，再帰呼び出しを行う関数や汎用的で頻繁に呼び出される関数を考慮し，一度検索した関数を再度検索しないように制御を行う．

## 4.4 コードパターンのマッチング

文献 [41] で実装されたパターンマッチングシステムのアーキテクチャは，図 18 の通りである．ソースコードはソース解析器によって AST （Attributed Syntax Trees = 属性付き構文木）に変換され，パターンはパターン解析によって CPA（Code Pattern Automata）に変換される．ソースコードの一部を AST に変換した例を図 19 に示す．また，図 9 の例 1 に示す制約違反パターンを CPA に変換したものを図 20 に示す (図中の記号の詳細は，文献 [41] を参照されたい)．CPA

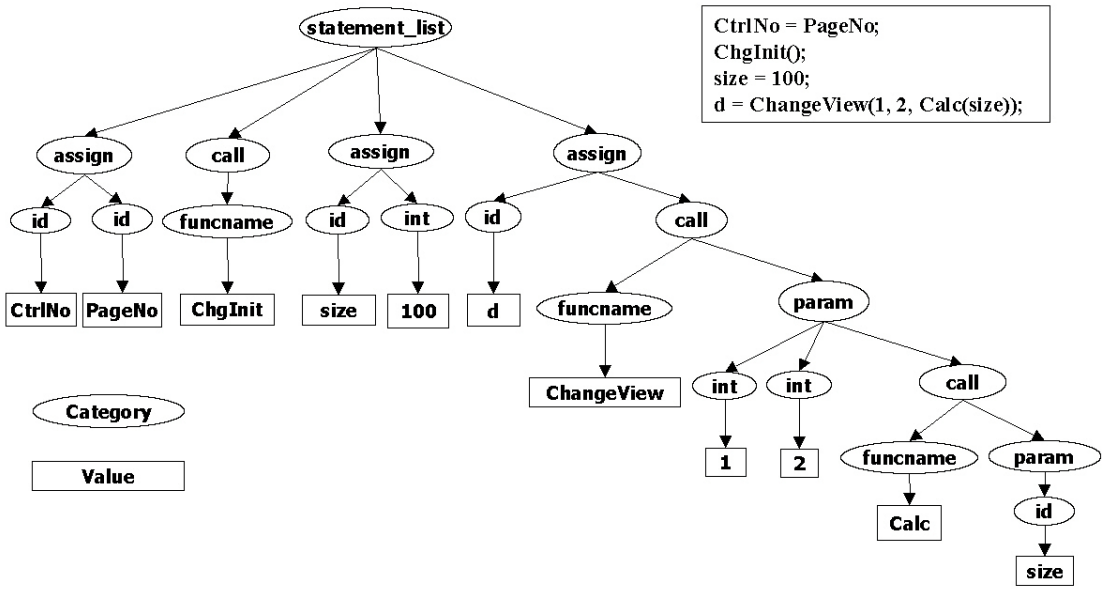


図 19 AST（Attributed Syntax Trees = 属性付き構文木）の例

インタープリタは AST 上で CPA をシミュレートし，最終状態に到達した（受理された）入力をパターンに一致コードとして保存していく．Binding Tables はパターン中の特定要素（名前つき要素：図 17 の$f_4 など）の情報を保持するために利用される．

本研究におけるパターンマッチングには，基本的にこのアーキテクチャに沿ったシステムを作成する．ただし．3.5 節のパターン記述言語の拡張に伴い，命令文中の変数が大域変数かローカル変数かを区別するために，その関数内に宣言文があるかどうかをチェックし，AST 上の各変数に大域・ローカルを示す情報を付加する．

CPA は，構文木上でのシミュレートを行うために拡張を行ったが，基本的には非決定性有限オートマトンである (CPA の詳細については，文献 [41] を参照されたい)．そのため，構文木上でシミュレートするが，図 21 のような明らかにパターン記述者の意図に沿わない一致コードを排除することはできない．このようなケースを排除し，正確に連続処理の可能性があるコードのみを検出するためには，コードパターンを文脈自由言語で記述し，プッシュダウンオートマトンを用いてマッチング処理を行う方法が考えられる．しかし，以下の理由から，文脈自由言語を用いるのは困難であると考えられる．

- 文脈自由言語によるパターン記述は，BNF（Bakus Naur Form）などの特殊な文法記述言語を用いる必要があるため，一般的なプログラマにとって容易ではない．図 21 のパターン

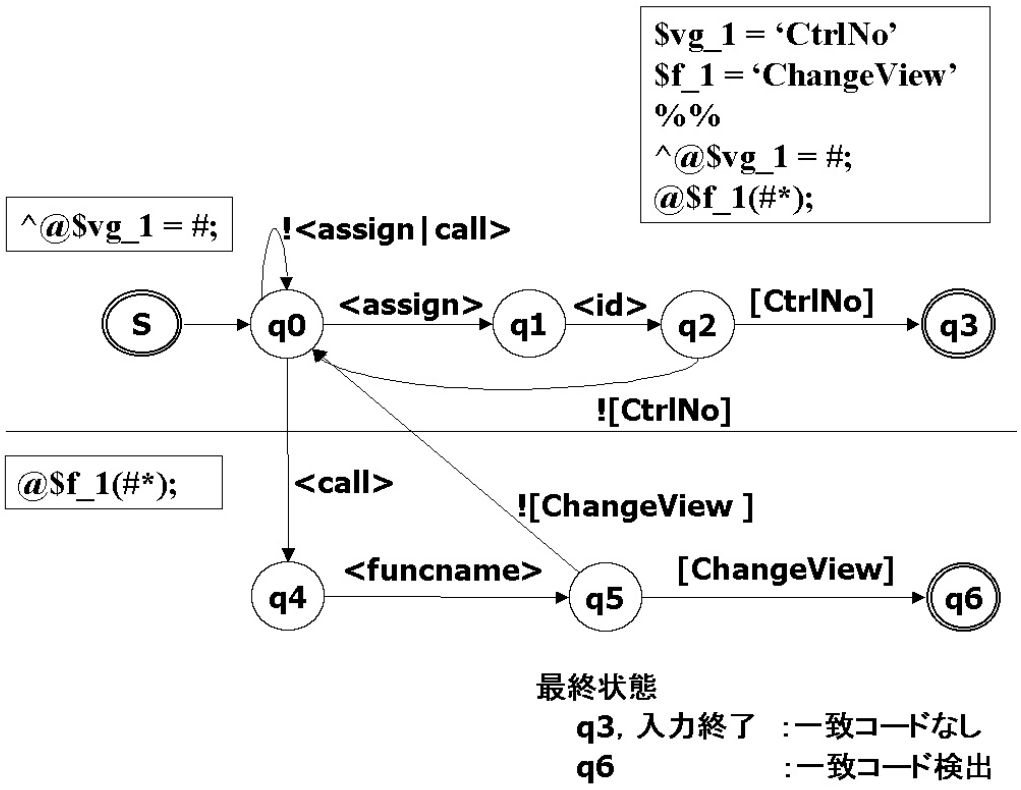


図 20 CPA の例（図 9 例 1 による）

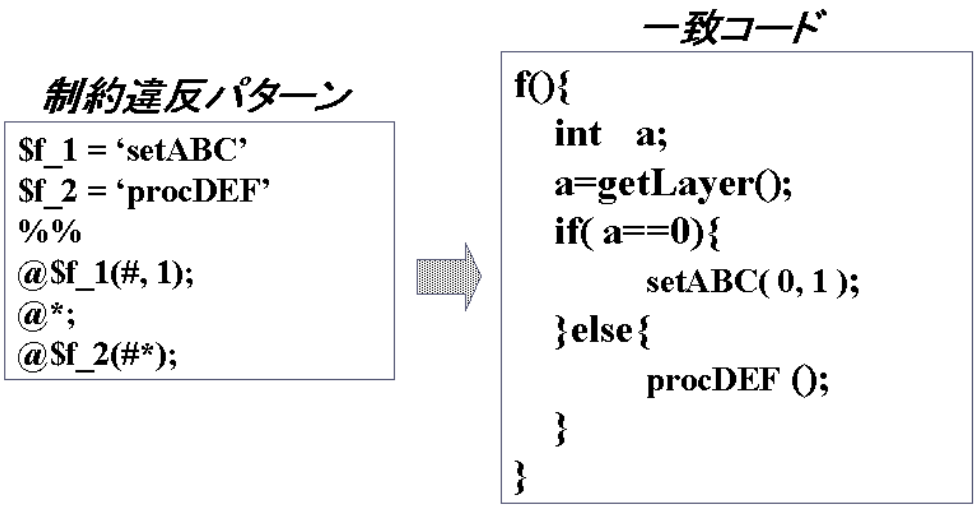


図 21　検出されないことが望ましい一致コード例

```
<stmt_list> ::= <stmt1>;[<stmt>;]*<stmt2>; |
                <stmt1>; [<stmt>; ]*<if_stmt2> |
                <if_stmt1>[<stmt>;]*<stmt2>; |
                <if_stmt1>[<stmt>;]*<if_stmt2> ......
<stmt1>     ::= <call><funcname>(<param>, 1 );
<stmt2>     ::= <call><funcname>([<param>]*);
<if_stmt1>  ::= if<relexpr>{<stmt1>} [else{[<stmt>]*}] |
                if<relexpr>{<if_stmt1>} [else{[<stmt>]*}] |
                if<relexpr>[{[<stmt>]*}]else{<stmt1>} |
                if<relexpr>[{[<stmt>]*}]else{<if_stmt1>}.....
<if_stmt2>  ::= if<relexpr>{<stmt2>} [else{[<stmt>]*}] |
                if<relexpr>{<if_stmt2>} [else{[<stmt>]*}] |
                if<relexpr>[{[<stmt>]*}]else{<stmt2>} |
                if<relexpr>[{[<stmt>]*}]else{<if_stmt2>}.....
....(以下略).....
```

図 22　図 21 のパターンの BNF 記述例

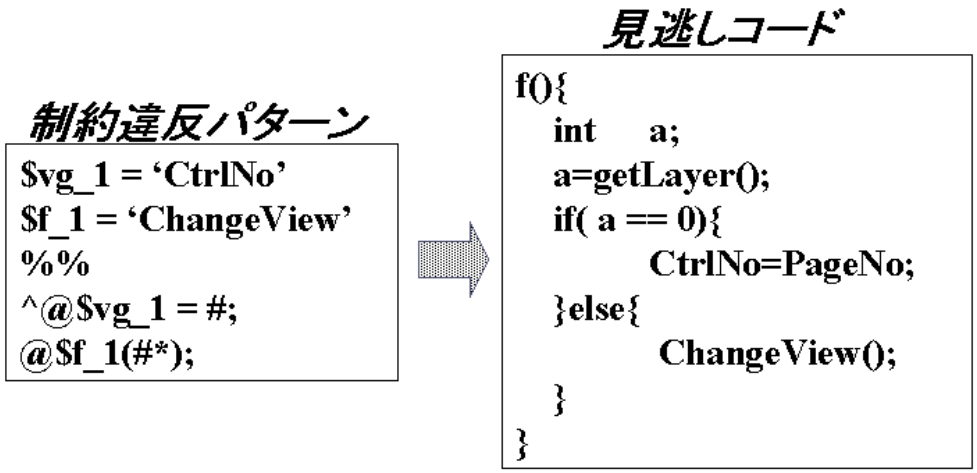


図 23 検出できない制約違反コード例

```
<stmt_list> ::=   <NOT_stmt1>;[<stmt>;]*<stmt2>; |
                  <NOT_stmt1>; [<stmt>; ]*<if_stmt2> |
                  <if_stmt1>[<stmt>;]*<stmt2>; |
                  <if_stmt1>[<stmt>;]*<if_stmt2> |
                  if<relexpr>{<stmt1>} [if else{[<stmt>]*}] else{< stmt2>} |
                  if<relexpr>{<stmt1>}if else{< stmt2>} [else{[<stmt>]*}] |
                  if<relexpr> [{[<stmt>]*}] if else{<stmt1>} else{<stmt2>}.....
<NOT_stmt1>     ::= <NOT><stmt1>;
<stmt1>     ::= <assign>;
<stmt2>     ::= <call> <funcname>([<param>]*);
<if_stmt1>  ::= if<relexpr>{<NOT_stmt1>} [if else{<NOT_stmt1>}]
                          [else{<NOT_stmt1>}]
<if_stmt2>  ::= if<relexpr>{<stmt2>} [else{[<stmt>]*}] |
                  if<relexpr>{<if_stmt2>} [else{[<stmt>]*}] |
                  if<relexpr>[{[<stmt>]*}]else{<stmt2>} |
                  if<relexpr>[{[<stmt>]*}]else{<if_stmt2>}.....
....(以下略).....
```

図 24 図 23 のパターンの BNF 記述例

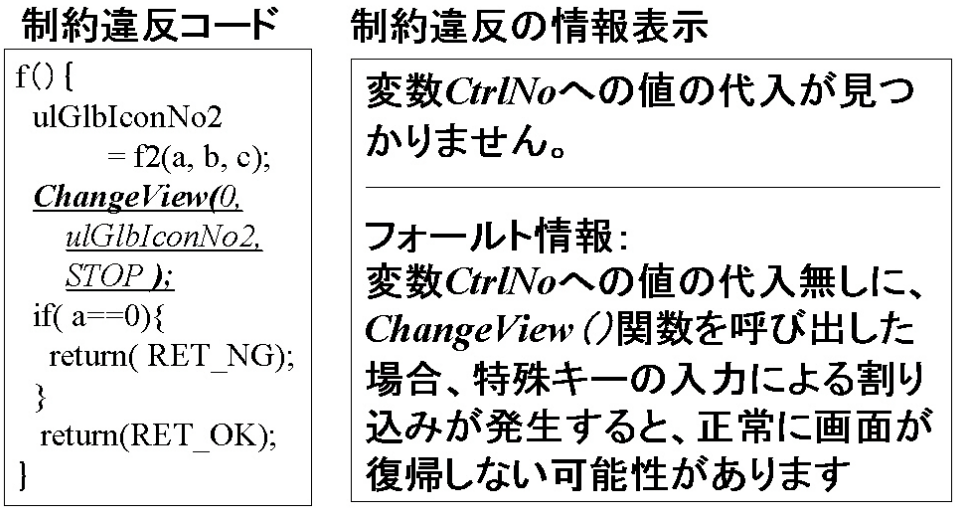


図 25 パターン照合結果の提示例

を BNF で記述する例の一部を図 22 に示す．この他に，switch-case 文や goto 文による分離パターンなどを考慮すると，記述量が膨大になると共に，記述に誤りが混入する可能性も高くなる．

- 「命令が存在しない」というパターンの記述は，現状の文法記述言語では対応していない．「指定の命令以外の命令」という記述方法を追加することは可能であるが，図 21 とは逆に図 23 のようなケースを検出しなければいけないため，パターンの記述量は図 24 のように，さらに膨大になると考えられる．

## 4.5 マッチング結果の提示と適用

マッチングの結果，制約違反パターンと一致した制約違反コードを，保守作業者がチェックするために必要な情報と共に提示する．図 25 に図 9 例 1 の制約違反コードの提示例を示す．提示すべき情報としては，過去に報告されている故障の症状，故障発生時のプログラムの状態，操作のタイミング，フォールトの修正方法，修正後の回帰テストの方法などが挙げられる．これらの情報を参照することで，コードレビューあるいは実機でのテストにおいて，検出されたコードがフォールトが含まれる（故障を発生させる）か否かが確認できる．また，実際にフォールトが発見された場合，修正方法やテスト内容を参照することで，対応を効率的に行うことが可能になる．

## 4.6 提案システムの使用方法

提案する制約違反検出システムの使用方法としては，以下の２つのパターンが考えられる．これらは，また，3.3 節の現状の２つの制約違反の検出方法に対応する．

1. コードレビュー時：

   開発者・保守作業者がコーディングを行ったときに，変更コードとその関連コードを対象にして，制約違反検出システム上に蓄積されているすべての制約違反パターンを使って，フォールト検索を行う．これによって，次工程に進む前に既存の暗黙的コード制約に違反するフォールトを発見・修正できる．

2. 暗黙的コード制約発見時：

   新たな暗黙的コード制約が発見されたとき，プログラム全体を対象に新しい制約違反パターンの検索を行う．これによって過去に知らないうちに混入していた潜在フォールトを発見できる．

# 5. ケーススタディ

## 5.1 目的

このケーススタディでは，あるレガシーソフトウェアを題材として，提案方法の有効性を２つの観点から評価する．

- 提案方法の有用性

  制約違反コードを検出することが，実際に保守作業に対して有益であるかを評価する．3.3節で述べたとおり，あるレガシーソフトウェアにおいて，暗黙的コード制約違反によって発生する故障は 32.7%に上るが，それらのフォールトが他の方法 (テストなど) により容易に発見できるのであれば，本方法が有効であるとは必ずしも言えない．また，検出されるコードはフォールトの可能性があるコードであり，実際にそれがフォールトであるかどうかは，レビュアによるコードチェックが必要である．検出されるコード中に故障の原因となるフォールトがほとんど存在しない場合，本方法はコードレビューのコストを増大させるだけとなる．そのため，以下の点を調査する．

  – システムによって検出されたコードのうち，フォールトの割合

  – システムによって検出されたコードのうち，テストや運用工程で故障発生が報告されていない潜在的なフォールトの割合

- 提案方法の性能

  提案するパターンマッチング技術が，パターンに一致するコード部分の検出に期待する性能を発揮するかを評価する．提案するパターン記述言語で制約違反パターンを記述できなければ，検出はできない．そのため，パターン化率は重要である．また，実際に存在するフォールトが確実に検出されることを確認する必要がある．そのため，以下の点を計測する．

  – すべての制約違反パターンのうち，拡張前のパターン記述言語で記述できたパターンの割合

  – すべての制約違反パターンのうち，拡張したパターン記述言語で記述できたパターンの割合

  – すべての検出されるべき既知のフォールトのうち，検出されたコード部分に含まれるフォールトの割合

## 5.2 評価事例ソフトウェア

今回研究の評価に用いたソフトウェアは，組込み系のハードウェア制御とユーザインターフェースを含むレガシーソフトウェアの一サブシステムである．記述言語は C で，最終リリース版でのファイル数は C・H ファイルを合わせて 621 個，サイズは約 447,000 行である．このソフトウェアの開発は，1991 年に開始され，現在も保守・運用されているが，このケーススタディで使用するシステムは，その一バージョンで，1997 年 4 月に開発が開始され，1999 年 5 月に実質的な保守作業は完了している．この開発とは，実際には別システムのソフトウェアを再利用し，ハードウェア変更や仕様変更に対して改造することであり，保守工程とは出荷後の保守ではなく，改造が一段落して製品メーカ側で品質保証テスト（システムテスト）開始後の変更・追加作業のことである．

このソフトウェアを研究事例として用いるにあたり，以下の資料を参照した．

- 不具合連絡書（1997 年 12 月〜1999 年 3 月）（紙媒体）
- 不具合報告書（1997 年 12 月〜1999 年 3 月）（紙媒体）
- テスト仕様書（1997 年 12 月〜1999 年 3 月）（紙媒体）
- ファイル更新通知（1997 年 12 月〜1999 年 3 月）（紙媒体）
- 最終リリースソースコード（1999 年 5 月）（電子媒体）
- マイナーリリースバックアップコード（1997 年 9 月〜1999 年 5 月）（電子媒体）

不具合連絡書，不具合報告書，テスト仕様書，ファイル更新通知については，その内容例を付録 A に示す．各ソースコードファイルには，修正理由（不具合番号）と修正年月日が記載され，ファイル更新通知に記述されているリビジョン番号は，ソースコード中の変更箇所（追加，修正，削除）部分に付加される．それによって，各ドキュメントの内容がソースコード上で確認できるような構造になっている．

## 5.3 実験方法

4.6 節で述べたとおり，フォールト検出システムの利用パターンは 2 通りある．このケーススタディでは，２番目の利用パターン，すなわち新しく暗黙的コード制約が発見された場合に，プログラムコード全体を対象に新しい制約違反パターンの検出を行う実験のみを行った．

今回の実験の対象は，図 26 の保守開始時点（B）でのプログラムコード全体を対象に，保守期間（B〜C）に報告されたフォールト報告データから抽出された暗黙的コード制約を使用して

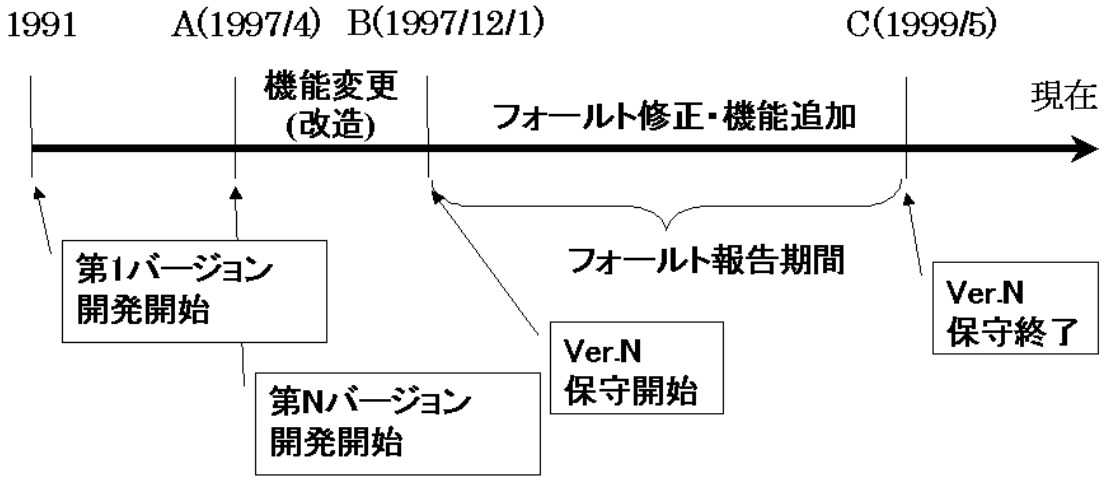


図 26 事例研究ソフトウェアの開発・保守期間

フォールト検出を行う．保守期間（B〜C）に報告されるフォールト報告データの中には，開発当初〜保守終了時点（C）の間に開発・変更されたコードによって発生した暗黙的コード制約が含まれるが，今回用いるのは開発当初〜保守開始時点（B）に発生した制約のみである．

本来，各暗黙的コード制約が発見された時点でのソースコードに対して，検出を行うべきだが，各時点での完全なプログラムコードが提供されていないため，このような方法を採用した．

## 5.4 違反検出実験用プロトタイプシステム

パターンマッチングを行うために，以下のプロトタイププログラムを作成した．

[開発・実行環境]

- CPU：Alpha21164A(300MHz)
- 主記憶装置：128MB
- 二次記憶装置：4.3GB
- OS：Digial Unix, IRIX
- 言語：C

[開発プログラム]

表 2 検出結果分類

| | | |
|---|---|---|
| A | 検出総事例 | 検出システムから出力された総事例 |
| B | フォールト発見事例 | フォールトが含まれていた事例 |
| C | フォールト報告事例 | 検出されたコードが原因である故障が発生・報告されている事例 |
| A-B | 誤検出事例 | フォールトが含まれていなかった事例 |

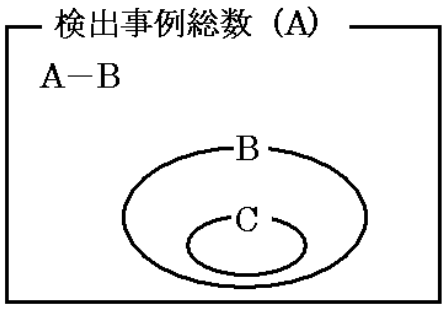


図 27 検出事例分類関係

- パターン解析・パターンマッチングプログラム:約 6800 行（一部，字句・構文解析コードは，オープンソースから流用）
- 対象コード作成ツール:610 行（注１）
- 複数照合連続実行用シェルスクリプト:20 行（注２）

（注１）提供されたプログラムコードは，保守終了時のソースコード以外は，リリース単位ごとにディレクトリ分けし，変更ファイルのみ保存したもの．そのため，特定の日時（この場合は 1997 年 12 月 1 日）の完全なソースファイル群を取り出すために，作成したツール．

（注２）パターン解析・パターンマッチングプログラムは，入力されたパターンファイルに対してマッチングを行う I/F で作成したが，複数の入力パターンファイルを連続処理するために，マッチングプログラムを連続処理するシェルスクリプトを作成．

## 5.5 評価結果

検出されたコードは，表 2 のとおりに分類する．各分類の包含関係を図 27 に示す.「事例数」というのは，実際に制約違反検出システムから出力された制約違反パターンに一致するコードの数である．

実際に検出実験に利用した制約違反パターン（付録 B 参照）に対するパターンごとの検出結果を表 3 に示す．制約番号は，暗黙的コード制約を取り出した順で，抜けている番号は実際の検出実験には使用できなかった制約である（16，34 が欠番であるため，実際の制約数は 45 個である）．

表 3: 検出結果

| 制約番号 | 検出事例数 A | フォールト発見事例数 B | フォールト報告事例数 C | 誤検出事例数 A-B | フォールト発見事例率 B/A | フォールト報告率 C/B |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 100.0% | 0.0% |
| 2 | 8 | 3 | 1 | 5 | 37.5% | 33.3% |
| 3 | 1 | 1 | 1 | 0 | 100.0% | 100.0% |
| 4 | 15 | 10 | 1 | 5 | 66.7% | 10.0% |
| 6 | 14 | 4 | 2 | 10 | 28.6% | 50.0% |
| 7 | 9 | 5 | 5 | 4 | 55.6% | 100.0% |
| 8 | 51 | 3 | 1 | 48 | 5.9% | 33.3% |
| 9 | 123 | 13 | 6 | 110 | 10.6% | 46.2% |
| 10 | 24 | 8 | 1 | 16 | 33.3% | 12.5% |
| 11 | 3 | 0 | 0 | 3 | 0.0% | 0.0% |
| 12 | 98 | 6 | 3 | 92 | 6.1% | 50.0% |
| 13 | 23 | 4 | 1 | 19 | 17.4% | 25.0% |
| 15 | 15 | 4 | 1 | 11 | 26.7% | 25.0% |
| 20 | 31 | 2 | 2 | 29 | 6.5% | 100.0% |
| 22 | 20 | 7 | 1 | 13 | 35.0% | 14.3% |
| 23 | 60 | 2 | 2 | 58 | 3.3% | 100.0% |
| 24 | 30 | 1 | 0 | 29 | 3.3% | 0.0% |
| 25 | 2 | 2 | 0 | 0 | 100.0% | 0.0% |
| 30 | 32 | 1 | 1 | 31 | 3.1% | 100.0% |
| 31 | 17 | 16 | 3 | 1 | 94.1% | 18.8% |
| 33 | 7 | 3 | 1 | 4 | 42.9% | 33.3% |
| 37 | 63 | 29 | 1 | 34 | 46.0% | 3.4% |
| 38 | 52 | 12 | 1 | 40 | 23.1% | 8.3% |

表 3: 検出結果 (前頁からの続き)

| 制約番号 | 検出事例数 A | フォールト発見事例数 B | フォールト報告事例数 C | 誤検出事例数 A-B | フォールト発見事例率 B/A | フォールト報告率 C/B |
|---|---|---|---|---|---|---|
| 39 | 10 | 4 | 1 | 6 | 40.0% | 25.0% |
| 42 | 3 | 0 | 0 | 3 | 0.0% | 0.0% |
| 43 | 6 | 2 | 0 | 4 | 33.3% | 0.0% |
| 44 | 13 | 1 | 0 | 12 | 7.7% | 0.0% |
| 45 | 14 | 5 | 2 | 9 | 35.7% | 40.0% |
| 46 | 5 | 1 | 1 | 4 | 20.0% | 100.0% |
| 47 | 22 | 2 | 2 | 20 | 9.1% | 100.0% |
| 平均 | 25.7 | 5.1 | 1.4 | 20.7 | 33.0% | 37.6% |
| 計 | 772 | 152 | 41 | 620 | 19.7% | 27.0% |

評価結果の詳細を表 4 に示す.

表 4 中の「故障数」「制約数」は，フォールト報告データから手作業で抽出し計測した．保守開始時点で存在する暗黙的コード制約は 39 個あったが，拡張前のパターン記述言語で記述できる制約は 17 個で，拡張後に記述できたのは 30 個だった．この 30 個の制約違反パターンを用いて，パターンマッチングを行い，フォールト報告データから把握している故障 38 個のうち 33 個の故障原因コードを，実際にシステムによって検出された一致コード中に確認できた.

出力された 772 事例に対して手作業でフォールトの有無をチェックし，故障が実際に発生すると考えられる 152 事例を確認できた（ただし，実機での確認は行っていない)．さらにその事例に起因する故障の発生報告がされていないもの（潜在的なフォールト）を抽出したところ，111 事例が確認できた.

表 4 の D，F から，9 個の制約違反パターンをパターン記述言語で記述できなかったことがわかる．この中には,「関数 A への変更を関数 B にも反映させなくてはいけない」という制約が 5 個,検出数が多くチェックが不可能なため制約違反パターンとして実用的ではないと判断されたものが 4 個あった．前者については，変更前と変更後のコードの差分を取得し，さらに差分のコードをパターン記述するなどの方法で検出しなければならない．このケースでは，記述言語の拡張とともにマッチング方法を単純な構文木によるマッチングから大幅に拡張する必要がある.

表 4 の G，H からは，制約を抽出する元になった 5 個の故障の原因となるコードが検出できな

表 4 評価結果

| | 項目 | 件数 | 割合 | 備考 |
|---|---|---|---|---|
| A | 全故障数 | 165 | | 原因が報告された故障 |
| B | A のうち暗黙的コード制約違反による故障数 | 54 | 32.7% | ＝ 54 ÷ 165 |
| C | B から抽出される制約数 | 45 | | ９件は，共通の制約違反が原因による不具合 |
| D | 保守開始時点での制約数 | 39 | | 保守開始時点＝ 1997 年 12 月 1 日 |
| E | D のうちで拡張前のパターン記述言語で記述できる制約数 | 17 | 43.6% | ＝ 17 ÷ 39 |
| F | D のうちで拡張したパターン記述言語で記述できる制約数 | 30 | 76.9% | ＝ 30 ÷ 39 |
| G | F に違反することで発生した故障数 | 38 | | テストや運用工程で発生し，発生報告が行われた故障 |
| H | G のうち制約違反検出システムによって検出できた故障数 | 33 | 86.8% | ＝ 33 ÷ 38 |
| I | 制約違反検出システムによる検出事例総数 | 772 | | F の全パターンによる |
| J | I のうち実際にフォールトが含まれていた事例数 | 152 | 19.7% | ＝ 152 ÷ 772 |
| K | J のうち潜在的なフォールト事例数 | 111 | 73.0% | ＝ 111 ÷ 152 |

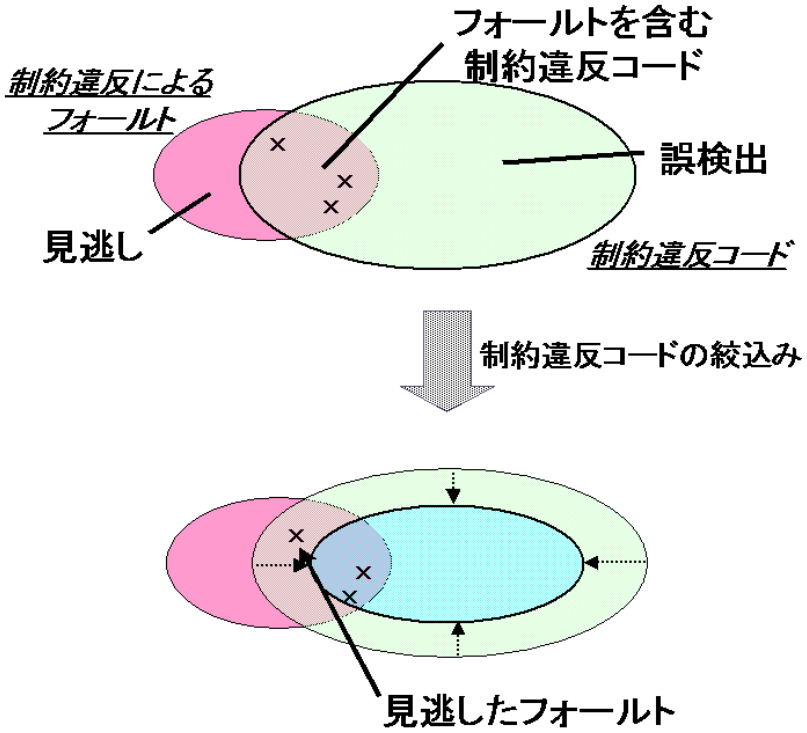


図 28 誤検出と見逃しの関連図

かったことが確認できる．その内訳は，メモリ不足による検出処理の中断が発生したものが 1 個，検出される制約違反コード数が多すぎることから絞り込むための条件を付加することによって検出できなかったものが 4 個である．

メモリ不足による検出処理の中断は，呼び出し関数先まで検索範囲を拡大した結果である．呼び出し関数の検索は，深さを 3 まで，かつ再帰呼び出しや頻繁に使用される汎用関数を考慮して同じ関数は 2 回検索を行わない，という制限を設けた．しかし，検索を開始する関数のサイズ (ステップ数) が非常に大きかったり，呼び出す関数が多い，あるいは呼び出された関数のサイズが大きいなどの理由で，検索処理が膨大なメモリを必要とするケースが存在した．他にも，複数の連鎖したパターンの検索時にもメモリ不足が発生したが，これは，パターンを分割して検索するなどの対応で検出を実施し，有効な結果を取得できた．この問題に関しては，制約違反検出システムにおける検出方法の改良など実装上の対応で，検出が可能になると思われる．

検出数を絞り込むための条件による検出漏れは，誤検出（フォールトが含まれない制約違反コード）と見逃し (制約違反コードではないフォールト）間に存在するトレードオフにより，現状では

***制約違反パターン***

```
$v_1 = '.CDData¥[[0-z]*¥].String'
$f_1 = 'strncpy'
$f_2 = 'strncat'
$f_3 = 'memcpy'
%%
@[$f_1 | $f_2 | $f_3]( $v, $v_1, # );
%
```

***見逃し事例***

```
struct infdata stINFData;
struct shmdat *pstShmDat;

void func_a(){
   char    buf[100];

   pstShmDat = malloc(sizeof(struct shmdat));
   strcpy ( pstShmDat->String,  stINFData.CDData[0].String);
   strncpy(buf,  pstShmDat->String, 99 );
}
```

図 29 見逃しフォールト例

対応が困難である．検出事例数が膨大になると，手作業でチェックを実施するのは困難であるが，図 28 のとおり，検出数を絞り込もうとすると，見逃しが発生する可能性がある．誤検出，見逃し共に極力少なくするように制約違反パターンを作成することが理想であるが，パターンによってはそれは困難である．例えば，図 29 のようなケースがある．このケースでは，グローバル変数 stINFData.CDData[0].String に格納されている文字列情報を単純にバイト数でコピー，カットしてはいけない，という暗黙的コード制約によるパターンである（詳細は，付録 B のパターン 1 参照）．この制約違反パターンでは，焦点となる文字列情報 stINFData.CDData[0].String が直接バイト数指定のコピーやカットを行う関数 strncpy などに引数として渡される，もしくは関数引数経由で渡されるパターンは検出できるが，図 29 の見逃し例のようにデータそのものが別領域にコピーされた上で参照される，もしくは，別のポインタ変数を経由して参照されるなどのケースを検出することはできない．それらをすべて検出するためには関数$f_1，$f_2，$f_3 の呼び出し箇所すべてを検出し，そこに与えられる引数の内容が stINFData.CDData[0].String と同じになることがないかを確認する必要があるが，検出数が膨大になるため，最も典型的な例を制約違反パターンとした．その結果，見逃しが発生した．

今回のケーススタディでは，いくつかのパターンにおいて誤検出を減らすための調整を行っている（付録 B のパターン 1，10，22）．これらのパターンについては，暗黙的コード制約抽出元のフォールト以外の多数のフォールトが検出できたことから，実用上は制約違反パターンとして有用であると考えられる．

## 5.6 考察

[提案方法の有用性]

表 4 の J から，検出された事例の 19.7%が実際にフォールトであることが確認できた．さらに，表 4 の K から，故障の発生などで発見されるものが，そのうちの 27.0%のみであることがわかった．それ以外の 73.0%のフォールトは，従来発見するのに人手によるコードレビューのような作業が必要だったが，提案方法により自動検出できることになると，人的・時間的なコストの低減につながると言える．また，この値から，暗黙的コード制約違反によって発生する故障は，氷山の一角であり，潜在しているフォールトが提案方法によって容易に発見できると考えられる．

これらの結果から，暗黙的コード制約に違反するコードの検出は，実際に故障を発生させるフォールトを検出できるばかりでなく，潜在しているフォールトを検出することが可能で，ソフトウェアの信頼性の向上に役立つと考えられる. さらに，潜在しているフォールトが実際に運用工程などで故障を発生させる前に発見・対処が可能になることによって，保守コストの削減が期待できる．

ケーススタディでは，実際には故障の原因とはならない誤検出コードは，検出されたコードの80.3%に上る．これらのコードを分析した結果，誤検出を低減するためには次の 2 つの対処方法が考えられる．

- 実際に実行される可能性が無いためフォールトにはなりえないコード（デッドコード）を，あらかじめマッチング対象のプログラムコードから除去する．
- switch-case 文や if-else 文によって分離された排他的な処理を「同時に実行されない処理」とみなしてマッチングを行う（図 21 参照）．

[提案方法の性能]

表 4 の E と F から，制約違反パターンを形式的に記述するために選択したパターン記述言語とその拡張が適切であったと確認できた．拡張は部分的であり，パターンマッチングのアーキテクチャに大きな変更を加える必要がなくこれだけのパターン化率の改善が見られたことは，元の記述言語の拡張性の高さを示しているとともに，制約違反パターンの特徴に対して適切な拡張が行われたと考えられる．

表 4 の H から，既知のフォールトで制約違反検出システムによって検出されることが期待されたほとんどのフォールトが，実際に検出できたことを確認できた．

これらの結果から，提案方法による制約違反コードの検出は，実際のフォールトの検出に対して有用であり，妥当な性能を持つことが確認できた．

# 6. 関連研究

## 6.1 フォールト検出支援

レガシーソフトウェアの保守工程において，膨大なプログラムコードからコードレビューでフォールトを見つけることはコストがかかるうえ，人手によるレビューでは確実に検出できる保証はない．さらに，すべてのフォールトを検出する完璧なテストを行うことは不可能である [38]．本節では，自動的にプログラムコードからフォールトを検出する，あるいは，フォールトの存在するモジュールを予測することでテストやレビューを効率的に行うための研究について述べる．

### 6.1.1 パターンマッチング

パターンマッチングの手法を使って具体的にフォールトとなるコードの検出を支援する研究には，以下のようなものがある．

小田，掛下の C プログラムの落とし穴検出ツール Fall-in C[39] では，コンパイラや lint などでは検出できなかったり，適切なコメントができなかったりする字句・構文上の問題を検出し，警告を行う．正規表現を用いた字句解析レベルでの検出と，構文解析によって得られる構文解析木を用いたパターンマッチングの手法を使用して，高速で正確に，C プログラムのコーディング上の危険箇所を指摘することを可能にした．

関本らのプラン認識を用いた方法 [44][45] では，誰が見ても理解しやすいプログラムを書くためのコーディング規約をプログラミングスタイルと呼び，これに違反するコードを検出することを目的とする．パターンマッチングの手法としては，文献 [41] で紹介された SCRUPLE システムを応用するが，パターン記述をパターン部と制約部に分けることで，検索をシンプルにしている．

プログラムの細粒度リポジトリに基づく CASE ツール・プラットフォーム Sapid（Sophisticated APIs for CASE tool Development）[12][1] を用いて，倉内らはプログラミング言語の構文規則に基づいた問い合わせを可能にした [24]．同じく Sapid をベースとして，河合らは，ソースコードから多用される関数に関して規範パターンを取り出し，それに一致しないパターンを検出することを提案している [20]．例えば，fopen 関数と fclose 関数は同一モジュールから呼び出されいるパターンが多いという「規範パターン」をプログラムコードを解析することによって抽出し，その規範に違反するコードをフォールトである可能性が高いとして指摘する．これによって，パターンを探したりパターン記述をしたりという作業を自動化することができる．

海尻 [19] は，プログラムレポジトリに基づくデータベース検索では検索命令への変換が困難であり，一方プログラムパターンを記述言語を用いて記述し検索する方法は，文法的に表現できるパターンが限定されるなど拡張性や分散パターンの認識に難があることから，関数単位でレポジ

トリを作成し関数本体に対しては解析木を用いた検索方法を提案している．

Engler らは，Meta-level Comlilation(MC) と呼ばれる拡張されたコンパイラをを用いて，ドメイン固有のフォールト検出を可能にする研究を行っている [8][14]．これは，いくつかのコードパターンのテンプレートに従ってモデルを記述し，これに違反するコードを MC を使って検出することができる．コンパイラをベースにしているためシステム構築の労力も最小限で，テストやコードレビューで発見が困難なフォールトを多数検出した．さらに，そこから偽のフォールトコードを除外したり，フォールトである可能性が高い順にランキングをしたりすることで，フォールト検出をより短時間に簡易に行うための改良も行っている [14]．また，この手法を応用し，検出されたモデル違反コードがフォールトであるかどうかを動的に確認する機能を追加する研究も行われている [26]．

これらは，いずれもプログラムコードの静的解析を基本にパターンマッチングの手法で規約違反や文法上の不適切な記述を検出する方法を採用しており，コーディング工程でのフォールト検出が可能である．しかし，汎用的なコードパターン検索をするのには記述言語の記述が困難であったり，記述言語が容易なものはパターンに制約を設けたりすることで，検索処理を容易にしている．本論文で抽出したような制約違反パターンの検索には，パターン記述の柔軟性とパターンの拡張性が要求されるが，これらを充足するパターン検索手法としては，文献 [41] の方法が最適であると考えられる．

### 6.1.2 不具合傾向モジュールの予測

モジュール単位のコード劣化を判定し，重点的なテストやレビューを必要とするモジュールや再設計するモジュールの選択に役立てる手法がある．MaCabe は複雑度を静的に計測し、プログラム構造の劣化を定量的に明らかにした [33]．Eick 等は，コード劣化をモジュール間の結合度やモジュールの凝集度によって計測し，結合度が高いモジュールや凝集度が低いモジュールは変更時に多くのモジュールを変更する必要があると言っている [7]．過去に頻繁に変更されたモジュールほどより多くのフォールトを含んでいることから，コード劣化はフォールト発生を予測するための重要な指標になると考えられる．Andrews 等は，フォールトの修正履歴に基づいてこれらの指標を計測し,「不具合を含み易いコンポーネント」と共に「不具合を含み易いコンポーネント関係」を検出する手法を研究している [3]．具体的には，各フォールトに対する修正対象のコンポーネントを分析し，コンポーネント間の結合度や各コンポーネントの凝集度を計測し，コード劣化を判定する．その情報を，重点的なテストやコードレビューに利用することによって，フォールトの発見を支援する．

一方で，ソースコードやフォールト報告データ，修正履歴から定量的に計測できるデータ（コー

ド行数，関数呼び出し数，ループ数，修正回数など）と実際に発生した不具合をモジュール単位で分析し，計測データと不具合の有無を関連付けることで，モジュール単位での不具合有無の予測を行う研究がある．不具合を含み易いモジュールと不具合を含みにくいモジュールに分類できると，不具合を含み易いモジュールを重点的にチェックしたりテストしたりして，実際のフォールトの発見を支援することができる．また，再設計を行うモジュールの選択に使用することも可能である．Khoshgoftaar 等は，新規・修正コード数，インストール回数，設計者が行った更新数などが予測に有効な因子になると言っている [21]．Graves 等は，修正履歴を用い，各モジュールにおける変更の回数と各変更が行われた時期を考慮した重み付けで予測精度が向上したと発表した [13]．

これらの方法では，モジュールやコンポーネント単位で不具合有無の予測しかできないため，実際のフォールトの発見やフォールト修正の手順としては，従来どおりのテストやコードレビューを必要とする．具体的なフォールトコードを指摘する情報や故障現象の予測ができないと，実際のフォールト検出には，コストのかかる作業を伴う．

### 6.1.3 フォールトの動的検出

フォールトの動的検出を行う方法としては，アサーションや Purify といったツールを使う方法がある．アサーションはプログラムコード中に assert() といった関数を埋め込み，故障発生時にエラーメッセージを表示する．関数の引数や変数の参照時にその内容に関する条件を指定しておくと，その条件外のケースで実行されると診断情報（式やファイル名，行番号）を表示する．これによって，フォールトの発生箇所の特定が容易にできる．Rosenblum は，ソースコード中にコメント形式のフォールト条件を記述することで，自動的にアサーションによるチェック機能を持った実行プログラムを作成する方法を提案している [42]．Purify はメモリ操作上のフォールトを動的に発見するための商用ツールである．メモリ操作命令に自動的に検査コードを追加することによって，未確保の領域が操作された場合に警告を行う [49]．

これらは，プログラムが実行可能状態であることを前提とする．従って，本手法のようなコードレビューレベルでのソースコードのチェックには利用できないことがある．また，故障が発生するような動作が実行されないとフォールトを検出できないため，故障状態の発生条件を満たす操作頻度が低い場合などは，検出が困難である．むしろ，故障発生時のデバッグ・原因解析などの支援に有用であると考えられる．

### 6.1.4 動的モデル抽出と違反検出

特定のシステムからモデルとなるパターンを自動的に抽出し，そのパターンに違反するケースを検出する研究が行われている [9][15]．Ernst らによる Daikon は，特定のシステムを様々なケースで実行して動的に注目するメモリ値（グローバル変数等）の値をトレースし，正常ケースでの取りうる値の範囲を「Invariants」として抽出する [9]．この Invariants を逸脱する状態が「プログラムが正しいふるまいをしていない状態」であると捉え，Invariants をモデルとして，これに違反するケースをシステム動作中に動的に検出することを目的としている．Hangal らによる DIDUCE は，この考え方を応用し，Invariant の検出と同時にその違反ケースのチェックを行い，Invariant の範囲を拡張することで，より正確なメモリ値の取りうる範囲を推測することができる，としている [15].

この手法では，モデルの手作業での抽出や違反検出を行うための特殊な作業をせずに，対象システム特有のフォールトを検出することが可能であると共に，故障の原因を具体的に指摘することが可能である．ただし，Invariants を検出するためのテストケースが適当であるか，もしくは，検出された Invariants が適当であるかを判断する明確な指針が与えられていないため，検出された Invariants に対する逸脱ケースが 100%フォールトである保証はない．また，異常ケースを発生させるケースを実行させることがタイミングや環境によって困難である場合，フォールトを検出できなかったり，もしくは，検出できてもシステムテスト工程やリリース後になってしまう可能性がある．

Musuvathi らによる CMC(C Model Checker) は，システムを実行することによって得られるメモリ値をトレースすることによって，抽象的なモデルを作成する必要なくモデルチェックを行うツールである [37]．ただし，CMC では，ドメイン特有のフォールトはユーザが記述する assertion によって検出され，その他には一般的な固定されたモデル違反を検出するツールであり，モデルチェックの範囲が限られている．また，上記と同様に，膨大なテストケースを実行する必要がある．

## 6.2 レビューの支援

設計やコードのレビューは，次工程での問題発生を防ぐために有効であり，コストの増大を防ぎ，品質を向上させる．しかし，人手で行うことが主流であり，その効率や信頼性を向上させる為に，レビュー作業を支援する方法が研究されている．

### 6.2.1 チェックリスト

設計書やソースコードに対するチェック項目をリスト化し，各工程で開発者・レビュア・保守担当者がチェックする手法である．

設計・コーディング（言語に依存する）に関する一般的な注意事項のリストに関しては，古くから研究が行われ，一般に普及しているリストが存在する [17][18][38]．また，これらのチェックリストを分類し，検証を行うコードに必要なもののみを精選することも提案されている [27]．

これらのチェックリストでは，システムに依存した問題はチェックできない．この点に関して，過去に発生したフォールトをチェックリストに追加し，そのリストを使ったレビュー方法が提案されている [36]．この方法では，システムに依存する問題がチェックでき，その頻度や発見難度・修正難度にしたがってリストを精選することで，チェック効率が良くなる．しかし，チェックを人手に頼るため，チェックのコストがかかったりチェック漏れが発生する可能性がある．また，リストから外れてしまったチェック項目によるフォールトは発見できない．

### 6.2.2 レビューレポートによる知識管理モデル

レビュー情報を利用して開発に関わる重要な知識を共有し，開発を支援するシステムモデルの提案が行われている [23]．その中で，レビュー情報は以下の点で有用であると述べている．

- 問題解決 (決定) のプロセスを含む
- 成功・拒否の両方の情報を持つ
- 開発者の実際の用語で記述される
- 材料と参照された情報源をレビューする関係を与える
- ソフトウェア開発の全フェーズをカバーする

この論文では，レビューを行う議論環境を Web 上で提供し，議論の内容をデータベース化することによって，議論に加わっていない開発者が必要な知識をデータベースから得られるシステムを構築した．情報を求める人は，キーワードを入力して必要な情報を検索する．実際に運用し，評価した結果，設計用語による検索が多いことから，設計工程におけるレビュー情報共有の有効性を示している．

レビュー情報の有用性という意味では，コードレビューに関しても上記の項目はあてはまる．しかし，キーワード入力による情報検索の場合，コードレビューに関するキーワードの選択は難しく，キーワード入力の手間に対する取得情報量が少ないことが予想される．

### 6.2.3 事例ベース推論

事例ベース推論とは,「知識ベース支援哲学を，過去の経験を用いた人間の推論のシミュレーションと組み合わせた技術，すなわち，頭の中で過去に起きた同じような状況を検索し，それらの状況から得られた経験を再利用すること」である．この技術を用いて，不具合報告データを新規開発の設計時に利用する手法が提案されている [16]．これは，類似したシステムの開発を繰り返し請け負う会社単位での導入に有効であると考えられている．あるプロジェクトで発生した問題やその解決方法を蓄積し，その検索システムを構築することで，新たなプロジェクトや新規の開発にもその情報を適用することができる．検索には，キーワード入力を行い，複数のキーワードとデータベース中の問題記述文の類似度を計算し，類似度の高い順にその結果を提示する．しかし，問題情報として保存する内容や入力するキーワードによって類似度が変わってくるため，キーワードや問題報告データの入力時に適切な選択が必要である．そのため，確実に関連する情報のみを抽出するには，すべての利用者に熟練が求められる．

この手法を下流工程に適用することは可能である．例えば，修正コードに関するキーワードを入力し，過去のフォールト情報との類似度を計算し，類似する情報を提示するといったシステムが考えられる．しかし，適切なキーワードの入力が必要であり，利用者に熟練が求められることは同じである．

### 6.2.4 レビュープロセス支援ツール

Macdonald らによって開発された ASSIST は，レビュープロセスを自動化することによってレビュー作業の効率化を目指している [27]．このシステムは，コードや設計をレビューするための準備・実施・フィードバックまでの作業を効率的に行うための様々な機能を提供すると共に，IPDL（Inspection Process Definition Language）によってプロセスをカスタマイズできるようになっている．具体的には，レビューメンバー・レビューコード・結果 (defect) の管理，レビュープロセスのスケジューリング，個人やグループによるレビュー実施用ブラウザなどの機能を提供する．

この研究によって，レビューに伴うコストの低減が期待される．また，レビュー結果の有効な利用方法も考えられている．

## 6.3 コード理解の支援

レガシーソフトウェアにおいて，フォールトの検出を困難にする一因として，プログラムコードの膨大なサイズと複雑化が挙げられる．このようなコードは理解することが困難なためである．コードの理解性が向上することは，フォールトの発見を容易にすると共に，フォールトの混入を

防ぐことにも繋がる．そのため，レガシーソフトウェアにおいて，プログラムコードの理解支援は重要なテーマである．

### 6.3.1 リファクタリング

リファクタリングとは,「外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させること」[11] である．具体的には，プログラムコードの中にいくつかの問題のパターンを見つけ，それを模範的な形式に変換することを言う．これを自動化する研究も行われている．この手法では，問題のコードを抽出するばかりでなく，将来的に修正しやすい模範コードを提示することで，一貫性を保って改善することができる．従って，この手法を使うことで，１つのプログラムを長期にわたって保守していくことが容易になると考えられている．

暗黙的コード制約は，本質的にリファクタリングの対象となるような問題構造に起因するため，この手法の適用は問題の根本解決として推奨される．しかし，保守工程のすでに複雑化したプログラムに関しては，修正するプログラムの理解に時間的人的コストがかかり，変更による信頼性低下の危険もあるため，容易には適用できないという問題がある．

### 6.3.2 SPIE

コードの読解を直接的に支援するという視点から，大橋等はソースプログラム理解支援ツール SPIE を開発した [40]．この研究では，Sapid[12][1] を用いて DB 化されたソースコードをハイパーテキストで表示し，関数仕様書や関数・変数の出現個所を簡単に参照することができるようにしている．またレビュー結果をソースコードに対応させて保存することで，過去のレビュー内容を行単位で参照することができる．紙媒体でのコードレビューでは，必要な情報を探すことだけで大きなコストがかかるという問題に着目している．

これは，大規模レガシーソフトウェアにおいて複雑化したコードの理解や検索に能力を発揮すると考えられ，特に設計書や仕様書がない状態での保守作業に有効と思われる．

### 6.3.3 コーディング規約の活用

コーディング規約は一般的に広く普及しており，各分野毎や対象システム毎に必要な規約を規定している．コーディング規約は可読性（理解性）や移植性の向上を目的とするが，Fang は，コーディング規約の適用による品質の向上を目指している [10]．JAVA における一般的なコーディング規約 (Java Code Conventions) に対し，独自にいくつかの厳格な規約を追加することによって，

エラー傾向の要因の縮小，開発者の教育期間の縮小，コードの理解性の向上等の改善が見られたと報告している．

# 7. おわりに

## 7.1 まとめ

本論文では，レガシーソフトウェアの保守におけるプログラム変更時の信頼性を向上させる方法として，複雑化したプログラムコード中に存在する暗黙的コード制約をフォールトの原因の１つとして捉え，制約違反コードをコードパターンとして形式化し，プログラムコードから一致するコードを検出する方法を提案した．暗黙的コード制約に違反することで混入するフォールトは，これまで開発・保守技術者の記憶や経験に頼って検出するか，システムテストや運用中に偶然に発見されることが多かった．そのため，保守作業者の入れ替わり等で制約を知らない作業者が制約に違反するコーディングをすることで，同じ制約違反によるフォールトが繰り返し混入される可能性がある．本論文での提案方法を用いることで，制約を知らない保守作業者も制約違反のコードをコーディング工程で検出できることになり，潜在的なフォールトの混入を防止することができる．

提案方法では，まず，システムに潜在している制約をフォールト報告データから抽出し，フォールトの原因となる制約違反のコードパターンを形式的な言語で記述する方法を提案した．形式的記述言語は文献 [41] を拡張して用いる．次に，形式的に記述された制約違反パターンを用いて，プログラムコードから一般的な方法では検出できなかった潜在するフォールトを検出する方法を提案した．静的プログラムコード解析による属性付き構文木と制約違反パターンから作成した Code Pattern Automata を用いて，制約違反パターンに一致するプログラムコードを検出する．検出されたプログラムコードの検証や修正のために，制約違反パターンを作成するもとになったフォールト報告データを用いることにより，より低コストで確実な対応が可能であると考えられる．

提案方法を用いてあるレガシーソフトウェアからのフォールト検出実験を行った結果，保守工程で報告された故障の 32.7%は暗黙的コード制約に違反したために混入したものであり，試作したマッチングシステムによる実験結果，772 個所の検出コード部分中に 111 件の未報告フォールトが検出できた．また，抽出された暗黙的コード制約の 76.9%は提案したパターン記述言語で記述可能で，更に，報告されたフォールトのうちパターン記述できたものの 86.8%が保守開始時のソースコード中から検出できた．これによって，提案方法を導入することで，保守作業により変更されたコードの信頼性の向上と保守コストの低減が期待できるとともに，提案方法で用いたパターン記述言語やパターンマッチング方法は，フォールトの検出に有効であると言える．

従来研究では，レガシーソフトウェアが次第に複雑化するという問題を，Code Decay などの現象面から主に分析されてきた [7][35]．しかし，モジュール間の結合度の増大やモジュール凝集度の低下といった現象が確認できたとしても，その結果を保守現場にフィードバックし，実際の

保守に役立てることは容易ではない．一方，本論文では，保守作業者の立場から問題を分析し，直接的に解決する手段を提供している．すなわち，ソフトウェアの劣化＝暗黙的コード制約の発生，と捉え，制約違反コードを検出することの意義やその効果について整理した点に特長がある．また，制約違反コードを検出する具体的な方法を提案し，ケーススタディを通してその有用性と性能を評価した．

ただし，提案方法では，プログラムコード上の静的なパターンのみが制約違反パターンとして記述可能であるため，故障の発生条件となるような実行時のプログラムの状態は，静的なパターンに置き換えて記述するか，パターンから取り除く必要がある．そのため，実際に故障を発生させるフォールトではないコードが多く検出された．また，パターンによっては，検出された事例数中の実際のフォールトの確率が非常に低いものがあった．逆に検出する事例を特定するために効率的なパターンを作成することで，制約違反パターンの基となった既知のフォールトを検出できなかったケースもある．

## 7.2 今後の課題

まず，制約違反パターンの形式的記述言語の改善と，それに伴うマッチング方法の拡張が必要と考えている．今回，約 23.1％の制約違反パターンが提案する記述言語で記述できなかったが，これらのパターンにはフォールト検出に有効なものがあるので，検出に利用できるような記述言語の改善が必要と考えている．

また，今回は，保守開始時点でのプログラム全体に対して照合作業を行い，フォールトを検出する評価実験を行ったが，これを保守工程における変更ソースコードに対しても適用し，評価することが必要である．保守作業中にプログラムの変更コードのみを対象にして，潜在フォールトを検出することは，検出にかかる時間や検出される事例を絞り込むことができ，より効率的なフォールトのチェックが可能になる．さらに，テスト工程に入る前にフォールトを検出することで，修正方法を変更する等の対応が早期に可能になり，保守コストの低減を図ることができる．そのために，変更コードによって混入する制約違反コードのみを検出するパターンマッチング方法の工夫が必要である．

最後に，提案方法の有効な活用のためには，暗黙的コード制約を早期に認識し，制約違反コードの検出をプログラム変更のプロセスの一環として確実に行うことが必要である．本論文では，暗黙的コード制約を取り出すためにフォールト報告データを用いたが，実際には開発・保守作業中に担当者が制約の発生を認識している場合もあり，また，コードレビューやコード変更時に作成されるドキュメントから抽出することも可能である．また，複雑度の計測や修正履歴などを使った制約抽出方法の工夫も，必要になると考えている．暗黙的コード制約を早期に発見し形式化す

ることによって，より早期により多くのフォールトを検出することが可能になると考えられる．

# 謝辞

本研究を進めるに当たり，研究テーマの決定前から全研究過程において，貴重なご助言と共に忍耐強いご指導を賜りました，奈良先端科学技術大学院大学情報科学研究科　松本　健一教授に，心から深く感謝申し上げます.

本研究を進めるに当たり，貴重で開明的なご助言を頂きました，奈良先端科学技術大学院大学情報科学研究科　小山　正樹教授に，心から深く感謝申し上げます.

本研究を進めるに当たり，丁寧なご指導と貴重なご助言を頂きました，奈良先端科学技術大学院大学情報科学研究科　関　浩之教授に，心から深く感謝申し上げます.

本研究を進めるに当たり，常に貴重なご助言とご指導を頂きました，奈良先端科学技術大学院大学情報科学研究科　飯田　元助教授に，心から深く感謝申し上げます.

本研究を進めるに当たり，貴重なご助言とご指導を頂きました，大阪大学大学院基礎工学研究科　井上　克郎教授に，心から深く感謝申し上げます.

本研究を進めるに当たり，研究テーマの決定前から全研究過程において，忍耐強く細部にわたる熱心なご指導と貴重なご助言を頂きました，奈良先端科学技術大学院大学情報科学研究科　門田　暁人助手に，心から深く感謝申し上げます.

本研究を進めるに当たり，研究テーマの決定前から研究方法などに関して，熱心に丁寧なご指導を頂きました，現　広島市立大学　島　和之助教授に，心から深く感謝申し上げます.

本研究を進めるに当たり，研究の整理や論文作成などに関して，熱心に丁寧なご指導を頂きました，奈良先端科学技術大学院大学情報科学研究科　中村　匡秀助手に，心から深く感謝申し上げます.

本研究を進めるに当たり，研究テーマの決定や発表準備などにおいてご助言やご協力を頂きました，奈良先端科学技術大学院大学情報科学研究科　ソフトウェア工学講座の皆様に，心から深く感謝申し上げます.

本研究を進めるに当たり，快くデータをご提供いただき，様々な面でご助言やご協力を頂きました皆様に，心から深く感謝申し上げます.

# 参考文献


[1] *Sapid Home Page*. http://www.sapid.org/.

[2] *IEEE Standard Glossary of Software Engineering Terminology ANSI/IEEE Std 729*. The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1983.

[3] A. A. Andrews, M. C. Ohlsson, and C. Wohlin. Deriving fault architectures from defect history. *Journal of Software Maintenance: Research and Practice*, Vol. 12, No. 5, pp. 287–304, 2000.

[4] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, Vol. 12, No. 1, pp. 19–23, 1995.

[5] K. Bennett. Software maintenance: A tutorial. In *Software Engineering*, pp. 289–303. IEEE Computer Society Press, 1997.

[6] G. Canfora and A. Cimitile. Software maintenance. In *Software Engineering and Knowledge Engineering, Vol.1 Fundamentales*, pp. 91–120. World Scientific Pub. Co., 2001.

[7] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. on Software Engineering*, Vol. 27, No. 1, pp. 1–12, 2001.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, Vol. 27, No. 2, pp. 99–123, 2001.

[10] X. Fang. Using a coding standard to improve program quality. In *Proc. APAQS2001 2nd Asia-Pacific Conference on Quality Software*, pp. 73–78, Hong Kong, 2001.

[11] M. Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley, 1999.

[12] 福安直樹, 山本晋一郎, 阿草清滋. 細粒度レポジトリに基づいた case ツール・プラットフォーム sapid. 情報処理学会論文誌, Vol. 39, No. 6, pp. 1990–1998, 1998.

[13] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering*, Vol. 26, No. 7, pp. 653–661, 2000.

[14] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pp. 69–82, 2002.

[15] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. the International Conference on Software Engineering*, May 2002.

[16] B. U. Hanque, R. A. Belecheanu, R. J. Barson, and K. S. Pawar. Towards the application of case based reasoning to decision-making in concurrent product development (concurrent engineering). *Knowledge-Based Systems*, Vol. 13, pp. 101–112, 2000.

[17] C. P. Hollocker. *Software reviews and audit handbook*. John Wiley & Sons, 1990.

[18] W. S. Humphrey. *A discipline for software engineering*. Addison-Wesley, 1995.

[19] 海尻賢二. プログラムパターンの認識及びリバースエンジニアリングツール. 信学技報, SS2000-20, pp. 17–24, 2000.

[20] 河合茂樹, 山本晋一郎, 阿草清滋. 既存プログラムからの規範パターン獲得とそれに基づくコーディングチェッカ. 日本ソフトウェア科学会 FOSE'97, pp. 99–106, 1997.

[21] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictors of software quality. *Int'l J. of Software Engineering and Knowledge Engineering*, Vol. 9, No. 5, pp. 547–563, 1999.

[22] 小山明. エンベッデドシステム開発の課題. ソフトウェアシンポジウム 2001, pp. 101–102, May 2001.

[23] Y. Kudo, C. Hirai, Y. Furuhata, T. Watanabe, and O. Ohno. A proposal of a review-report-oriented knowledge-management model. In *Proc. 2WCSQ 2nd World Congress for Software Quality*, 2000.

[24] 倉内伸和, 山本晋一郎, 阿草清滋. ソースプログラムに対する構文規則に基づいた高度な問合せシステムに関する研究. 信学技報, SS97-53, pp. 25–32, 1998.

[25] M. M. Lehman and L. Belady. *Program evolution: Processes of software change*. Academic Press, 1985.

[26] D. J. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proc. 28th Annual International Symposium on Computer Architecture*, pp. 192–203, 2001.

[27] F. Macdonald and J. Miller. A comparison of tool-based and paper-based software inspection. *Empirical Software Engineering*, Vol. 3, No. 3, 1998.

[28] T. Matsumura, A. Monden, and K. Matsumoto. The detection of faulty code violating implicit coding rules. In *Proc. 2002 International Symposium on Empirical Software Engineering (ISESE 2002)*, pp. 173–182, October 2002.

[29] T. Matsumura, A. Monden, and K. Matsumoto. A method for detecting faulty code violating implicit coding rules. In *Proc. 5th International Workshop on Principles of Software Evolution (IWPSE2002)*, pp. 15–21, May 2002.

[30] 松村知子, 門田暁人, 松本健一. バグ報告データを用いたプログラムコード検証方法の提案. 信学技法, SS2001-34, Vol. 101, No. 628, pp. 1–8, 2002.

[31] 松村知子, 門田暁人, 松本健一. 潜在コーディング規則に基づくバグ検出方法の提案. ソフトウェアシンポジウム 2002, pp. 105–114, July 2002.

[32] 松村知子, 門田暁人, 松本健一. 潜在コーディング規則違反を原因とするフォールトの検出支援方法の提案. 情報処理学会論文誌, Vol. 44, No. 4, pp. 1070–1082, April 2003.

[33] T. J. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, Vol. 2, No. 4, pp. 308–320, 1976.

[34] A. Monden, S. Sato, and K. Matsumoto. Capturing industrial experiences of software maintenance using product metrics. In *Proc. 5th World Multi-Conference on Systemics, Cybernetics and Informatics*, Vol. 2, pp. 394–399, 2001.

[35] A. Monden, S. Sato, K. Matsumoto, and K. Inoue. Modeling and analysis of software aging process. *F. Bomarius and M. Oivo (Eds), Lecture Notes in Computer Science*, Vol. 1840, pp. 140–153, 2000.

[36] 毛利幸雄, 菊野亨, 鳥居宏次. レビューで用いるチェックリストの作成法の提案. 第 11 回ソフトウェア信頼性シンポジウム論文集, pp. 7–11, 1990.

[37] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. In *Proc. the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[38] G. J. Myers. *The art of software testing.* John Wiley, 1979.

[39] 小田まり子, 掛下哲郎. パターンマッチングに基づいた c プログラムの落とし穴検出方法. 情報処理学会論文誌, Vol. 35, No. 11, pp. 2427–2436, 1994.

[40] 大橋洋貴, 山本晋一郎, 阿草清滋. ハイパーテキストに基づいたソースプログラム・レビュー支援ツール. 信学技報, SS98-28, pp. 15–22, 1998.

[41] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. on Software Engineering*, Vol. 20, No. 6, pp. 463–475, 1994.

[42] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Software Engineering*, Vol. 21, No. 1, pp. 19–31, January 1995.

[43] N. F. Schneidewind and C. Ebert. Preserve or redesign legacy systems? *IEEE Software*, Vol. 15, No. 4, pp. 14–17, 1998.

[44] 関本理佳, 海尻賢二. プラン認識を利用したプログラミングスタイルの診断. 信学技報, Vol. 97, No. 175, pp. 9–16, 1997.

[45] R. Sekimoto and K. Kaijiri. A diagnosis system of programming styles using program petterns. *IEICE Trans. on Information and Systems*, Vol. 83, No. 4, pp. 722–728, 2000.

[46] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Engineering*, Vol. 10, No. 5, pp. 595–609, 1984.

[47] 高田広章. 組込みシステム開発の現状と展望. 情報処理学会論文誌, Vol. 42, No. 4, pp. 930–938, April 2001.

[48] 高田広章. テストから見る組込みシステムの特徴. ソフトウェアシンポジウム 2002, p. 101, July 2002.

[49] 脇田健. バッファあふれ攻撃とその防御. コンピュータソフトウェア, Vol. 19, No. 1, pp. 49–63, 2002.

# 付録

# A. フォールト報告書例

## A.1 不具合連絡書

| 不具合連絡書 | | | |
|---|---|---|---|
| 不具合 No. | *B-1000* | 報告日時 | *1999 年 2 月 9 日* |
| 不具合内容 | *画面の表示状態を A の状態にし，機能 B を実行.*<br>*機能 B の実行完了メッセージ表示，消去直後にキー C を押下し，直後に操作 D で画面をスクロール.*<br>*本来表示状態 A でスクロールするべきだが，画面半分ずれた画面がスクロールする* | | |
| 復帰条件 | *「リターン」キーで復帰.（画面表示状態 A になる.）* | | |
| 発生 ROM バージョン | | *Ver.3.72, Ver.3.86* | |
| 優先度 | | *１０* | |
| 発生頻度 | | *100%* | |
| 詳細内容 | 操作 | *画面状態を A にした後，キー B を押下. 機能 B 実行完了メッセジ表示・消去直後にキー C 押下，直後に操作 D* | |
| | 現象 | *画面が半分ずれた位置のままスクロール* | |
| | 修正方針 | *画面が正しい位置に表示された状態でスクロールするように修正* | |
| 希望納期 | *1999 年 3 月 5 日* | | |
| 備考 | | | |

## A.2 フォールト修正報告書

<table>
<tr><td colspan="5">不具合報告書</td></tr>
<tr><td>不具合 No.</td><td colspan="2">B-1000（不具合連絡書の No.に対応）</td><td>報告日時</td><td>1999 年 2 月 13 日</td></tr>
<tr><td>不具合内容</td><td colspan="4">画面の表示状態を A の状態にし，機能 B を実行.<br>機能 B の実行完了メッセージ表示，消去直後にキーC を押下し，<br>直後に操作 D で画面をスクロール.<br>本来表示状態 A でスクロールするべきだが，画面半分ずれた画面がスクロールする</td></tr>
<tr><td>不具合原因</td><td colspan="4">画面を正しい位置に表示するためのフラグ x が<br>メッセージ表示・消去直後は，正しい値に設定されていない.<br>メッセージ消去後に動作するべき処理で正しく再設定されるが，<br>この間にキーC を押下されると，その処理が行われず，<br>スクロールをする際にこの値を参照して半分ずれた位置でスクロールする.</td></tr>
<tr><td>不具合混入時期・分類</td><td colspan="4">ファイル f の修正リビジョン番号$20 で行った修正による.<br>不具合 B-200 の対応による 2 次不具合</td></tr>
<tr><td colspan="5">修正計画書</td></tr>
<tr><td>DIR1/file1.c<br><br>DIR2/file2.c</td><td colspan="4">1）フラグ x の設定条件の変更フラグ x を正しい値に設定するように，機能 B 実行前の設定条件を変更する<br><br>2）フラグ x を削除し，それに代わって同じ情報を持っている変数 y を参照するように変更する.</td></tr>
<tr><td rowspan="3">予想工数</td><td>解析</td><td>12h</td><td colspan="2"></td></tr>
<tr><td>修正</td><td>1h</td><td colspan="2"></td></tr>
<tr><td>テスト</td><td>3h</td><td colspan="2"></td></tr>
</table>

## A.3 テスト仕様書

| テスト仕様書 | | | |
|---|---|---|---|
| 不具合 No. | *B-1000(不具合連絡書の No.に対応)* | 報告日時 | *1999 年 2 月 24 日* |
| 対応内容 | *画面の表示状態を A の状態にし，機能 B を実行.*<br>*機能 B の実行完了メッセージ表示，消去直後にキー C を押下し，*<br>*直後に操作 D で画面をスクロール.*<br>*本来表示状態 A でスクロールするべきだが，画面半分ずれた画面がスクロールする* | | |
| テスト概要 | *＊機能 B 実行後のキー C 押下，操作 D で正しい位置の画面がスクロールすることを確認する.*<br>*＊その他，画面が半分ずれた表示状態からキー C 押下，操作 D で正しい位置の画面がスクロールすることを確認する.*<br>*＊通常の状態でキー C 押下，操作 D で正しい位置の画面がスクロールすることを確認する.*<br>*＊コードレビューで抜けがないか確認* | | |

| テスト項目 | テスト詳細 | テスト実施日 | 結果 |
|---|---|---|---|
| *画面状態 A* | *機能 B 実行後のキー C 押下，操作 D（操作・正しい動作）* | | |
| | *画面が正しい位置の表示される* | *2 月 24 日* | *O.K.* |
| | *画面中央の正しい位置にカーソルが表示される* | *2 月 24 日* | *O.K.* |
| | *スクロールが画面中央位置に対して正しく動作する* | *2 月 24 日* | *O.K.* |
| *画面状態 a* | *同上* | *2 月 24 日* | *O.K.* |
| *画面状態 A* | *キー C 押下* | | |
| | *画面中央の正しい位置にカーソルが表示される* | *2 月 24 日* | *O.K.* |
| | *スクロールが画面中央位置に対して正しく動作する* | *2 月 24 日* | *O.K.* |
| *画面状態 a* | *同上* | *2 月 24 日* | *O.K.* |
| *あらゆる状態の画面での操作 D* | *同上* | *2 月 24 日* | *O.K.* |
| 備考 | | | |

## A.4 ファイル更新通知

| ファイル更新通知 | | | | |
|---|---|---|---|---|
| 管理 No. | *B-1000* | 更新年月日 | *1999 年 3 月 5 日* | |
| 更新先 Path | *dir1/home1/user1* | | | |
| 更新ファイル | *BBBB/ap_base.c　　BBBB/ap_scrl.c* | | | |
| 更新詳細 | | | | |
| ファイル名 | リビジョン番号 | 分類（※） | 削除行数 | 追加行数 |
| *BBBB/ap_base.c* | *$96* | *1* | *1* | *4* |
| *BBBB/ap_scrl.c* | *$34* | *1* | *1 1* | *1* |
| 変更理由 | *（不具合番号）B-1000* | | | |
| 不具合内容 | *画面の表示状態を A の状態にし，機能 B を実行.*<br>*機能 B の実行完了メッセージ表示，消去直後にキーC を押下し，直後に操作 D で画面をスクロール.*<br>*本来表示状態 A でスクロールするべきだが，画面半分ずれた画面がスクロールする* | | | |
| 変更内容 | | | | |
| ファイル名 | リビジョン番号 | 具体的な変更内容 | | |
| *BBBB/ap_base.c* | *$96* | *フラグ x の設定条件を変更* | | |
| *BBBB/ap_scrl.c* | *$34* | *同様に画面状態を保持するフラグ y の設定箇所を整理*<br>*不必要に変更している箇所を削除* | | |
| 設計書 No. | *B-1000* | 報告書 No. | *B-1000* | |
| 方針書 No. | *B-1000* | テスト仕様書 No. | *B-1000* | |
| 実績工数 | 解析 | *12h* | | |
| | 修正 | *1h* | | |
| | テスト | *1.5h* | | |
| | コードレビュー | *1h* | | |
| （※）分類 | １．C ソース<br>２．C インクルード<br>３．アセンブラソース | ４．アセンブラインクルード<br>５．その他 | | |

# B. 制約違反パターンの例

以下は，実際にケーススタディで作成された制約違反パターンである.（識別子名などは変更している．また，暗黙的コード制約・フォールト例の記述は，一部省略したり，簡易化したりしている.）

## B.1 パターン１

- 暗黙的コード制約
  構造体に含まれる文字列情報は，２バイトコードと１バイトコードが混在するため，無条件にバイト数で編集してはいけない．バイト数でカットすると，２バイトコードの１バイト目でカットされてゴミが表示される可能性がある.
- フォールト例
  ある画面領域にこの情報を表示する際，１行１４半角文字分の領域で折り返しをするため，strncpy で１４バイト分で分割した結果，１行目の最後と２行目の先頭にゴミが表示された.
- 制約違反パターン

```
$v_1 = '.CDData\[[0-z]*\].String'   *
$f_1 = 'strncpy'
$f_2 = 'strncat'
$f_3 = 'memcpy'
%%
@[$f_1 | $f_2 | $f_3]( $v, $v~_1, # );
%
```

- パターンの補足説明
  $v~_1 は，その変数が別の関数の引数を経由してきた場合を含むことを示す.
  例）

```
void func_x()
{
     func_a(DATA.CDData[0].String);
}

void func_a(
char *str)
{
     strncpy( buf, str, 10 );
}
```

## B.2 パターン２

- 暗黙的コード制約
  １つのイベントテーブル上で処理が行われる画面で，あるイベント$m_1 に対して処理を行う機能を持ち，かつ，別のイベントに対する処理を行う関数内で特定の関数$f_1～$f_6 を呼び出す場合に，$f_1～$f_6 処理直後にイベント$m_1 が発生すると，$f_1～$f_6 による処理が中断され，異常が発生する可能性があるため，問題がないことを確認しなければならない.

- フォールト例

  あるイベントに対する処理は，画面上の表示変更後，$f_1～$f_6 によって一定時間後に処理を継続するが，２つ目のイベント$m_1 が発生したため，１つ目のイベントの継続処理が中断され，２つ目のイベントによる表示処理が１つ目のイベントに対する表示処理をクリアする処理がなかったため，画面上にゴミが残ってしまった．

- 制約違反パターン

```
$f_1 = 'GotoA'
$f_2 = 'GotoB '
$f_3 = 'GotoC '
$f_4 = 'GotoD'
$f_5 = 'GotoE'
$f_6 = 'GotoAll'
$m_1 = 'OPERATE_A'*
$t_1 = 'EventTbl'
$m_2 = 'EVENTPROC'
$m_3 = 'SUBEVENT'
%%
struct $t_1 $v_1[] = {
*#,
$m_2( $m_1, #, # ),
*#,
};
%
struct $t_1 $v[] = {
*#,
$m_3( #, $v_1 ),
*#,
$m_2( #, $f_7, # ),
*#,
};
%
$f_7()
{
*@;
@[$f_1 | $f_2 | $f_3 | $f_4 | $f_5 | $f_6]($*v);
}
%
```

- パターンの補足説明

  $t_1 は本システムの基本的な画面制御イベントテーブルの構造体．再帰構造になっている．

## B.3 パターン３

- 暗黙的コード制約

  関数$f_2 は$f_1 で設定した内容を打ち消す処理を含んでいるため，$f_1 の設定が必要な場合は，この順序で関数を呼び出してはいけない．

- フォールト例

  $f_2 は従来$f_1 の設定を打ち消す必要があるケースでのみ使用されており，そのため，$f_1 の機能追加をした際に，ここに追加した．たまたま，特殊なケースで$f_2 が必要で$f_1 の設定も必要な処理があり，そこで呼出し順序を間違えたため，$f_1 の機能が動作しなくなった．

- 制約違反パターン

```
$f_1 = 'SetKeyX_On'
$f_2 = 'ScreenInit'*
%%
@$f_1($*v);
*@;
@$f_2;
%
```

- パターンの補足説明

  なし.

## B.4 パターン4

- 暗黙的コード制約

  1つのイベントテーブル上で処理が行われる画面で，あるイベント$m_2 に対して$f_2 による処理を行う機能を持ち，かつ，別のイベントに対する処理を行う関数内で第1引数が 3 である関数$f_1 を呼び出すと，$m_2 による処理をキャンセルしてしまうため，問題がないか確認しなければならない.

- フォールト例

  1つ目のイベントで画面操作機能をロックし，一定時間後にロックを解除するが，その間に別イベントでロック解除用のタイマーをキャンセルしてしまったため，画面操作がロックされたままになる.

- 制約違反パターン

```
$f_1 = 'func_XYZ'
$f_2 = 'func_OPQ'
$m_2 = 'OPERATE_X'*
$m_3 = 'EVENTPROC'
$m_4 = 'SUBEVENT'
$t_1 = 'EventTbl'
%%
struct $t_1 $v_1[] = {
*#,
        $m_3( $m_2, $~f_2, # ),
*#,
};
%
struct $t_1 $v[] = {
        *#,
        $m_4( #, $v_1 ),
        *#,
        $m_3( #, $f~_3, # ),
        *#,
};
%
$f_3()
{
        *@;
        @$f_1( 3, #, # );
        *@;
}
%
```

- パターンの補足説明
  $~f_2 は，この処理が実際のテーブルの記述される関数から呼び出される関数であることを許可し，$f~_3 は$f_3 に記述された処理が$f_3 から呼び出されるどの関数に存在してもよいことを示す．

## B.5 パターン６

- 暗黙的コード制約
  関数$f_1 はある情報の取得要求を発行する処理を行うが，その際，あるグローバル変数の情報を参照している．この変数に必要な情報が正しく設定されていないと情報の取得に失敗するため，この関数$f_1 を呼出すときは，このグローバル変数の値が関数$f_1 の処理が必要とする情報をに正しく設定されていることを確認しなければならない．(このケースでは，元々問題のグローバル変数は，関数$f_1 が必要とする情報の保持を保証されているものではないので，関数$f_1 の仕様に問題がある．)
- フォールト例
  ある画面で画面に表示する情報を取得するために関数$f_1 を呼び出したが，エラーが発生し，その後画面がフリーズする．
- 制約違反パターン

```
$f_1 = 'func_defgh'
%%
$f_1(*#);
%
```

- パターンの補足説明
  なし．

## B.6 パターン７

- 暗黙的コード制約
  ある機能を実現する関数$f_1 は，その機能の完了時の処理をコールバック関数処理$f_4 によって行うが，そこで画面表示を行う$f_2，$f_3 を呼び出す場合，その時点でその表示が可能な画面でない場合があるため，$f_2，$f_3 の呼び出しに条件を付加しなければいけない．
- フォールト例
  ある画面表示中に，画面をスクロールさせ，直後にあるキーを２回押下．すると，表示されている画面上に別の画面用のボタンが表示される．押下しても反応しない．
- 制約違反パターン

```
$f_1 = 'func_rstuv'   *
$f_2 = 'item_[a-z]*'
$f_3 = 'func_draw'
%%
@$f_1(#, #, #, $f_4 );
%
$f_4()
{
*@;
@[ $~f_2 | $~f_3 ](#);
*@;
}
%
```

- パターンの補足説明
  $~f_2，$~f_3 は，$f_4 から直接呼び出されるものだけでなく，$f_4 以下で処理されるすべての関数内から検索される．@[$~f_2 | $~f_3] は，$~f_2 もしくは$~f_3 を検索する．

## B.7 パターン８

- 暗黙的コード制約
  ある画面が終了する時に何らかの設定を行うグローバル変数には，その画面が起動する時に必ず初期化しなければならないものが多いため，初期化されていないものに関しては問題がないか確認しなければならない．
- フォールト例
  画面 A 表示中に，キー押下で画面 B に遷移し，そこからタイムアウトで元の画面 A に戻った時，表示されるべき情報が表示されない．画面 A 表示中に行った操作によって，画面 B から戻ってきた時の初期処理が途中で中断され，本来行われるべき変数の初期化が行われていない．
- 制約違反パターン

```
$m_1 = 'SCREEN'*
%%
$m_1(#, #, #, $f_2 );
%
$f_2()
{
*@;
@$vg = #;
*@;
}
%
```

- パターンの補足説明
  $m_1 は，画面の起動・終了処理関数を規定するマクロ定義．$f_2 は画面終了処理を示す．

## B.8 パターン９

- 暗黙的コード制約
  ある特殊なキー押下で割り込み画面遷移をする場合，他の一般的な画面に遷移をする場合と元の画面に戻る時のルートが異なるため，遷移前の処理が 2 つのケースを考慮しているか確認しなければならない．特に，グローバル変数への設定は，元の画面を表示する際に影響するものが多いので，注意する必要がある．
- フォールト例
  ある画面 A 表示中に特殊なキー B を押下し，タイムアウトで画面 A に戻る．画面 A が真っ黒な表示になり，別の特殊なキー C を連続押下すると，システムがリセット．特殊なキー押下時に，カウントアップしてはいけないグローバル変数をアップしたため，画面 A がその変数を参照して画面を表示しようとして，異常が発生する．
- 制約違反パターン

```
$f_1 = 'GotoA'*
$f_2 = 'GotoB '*
$f_3 = 'GotoC '*
$f_4 = 'GotoD'*
$f_5 = 'GotoE'*
$f_6 = 'GotoAll'*
%%
@$vg = #;
*@;
@[$f_1 | $f_2 | $f_3 | $f_4 | $f_5 | $f_6](*#);
%
```

- パターンの補足説明
  $f_1〜$f_6 は画面遷移をする関数．

## B.9 パターン１０

- 暗黙的コード制約

  ある機能 A を持つ画面群（$m_1 に対するイベント処理を行う画面群）で，関数$f_2，$f_3 を呼び出す場合，機能 A に関わる処理中でないことを条件にしなければならない．

- フォールト例

  システム起動時に画面が完全に表示完了する前に，操作が可能になり，ある操作をすると，画面が表示途中のままになってしまう．

- 制約違反パターン

```
$m_1 = 'OPERATE_X'
$m_2 = 'EVENTPROC'
$t_1 = 'EventTbl'
$f_1 = 'func_X_proc'*
$f_2 = 'item_set'
$f_3 = 'func_X_item'
%%
struct $t_1 $v[] = {
*#,
$m_2( $m_1, $f_1, # ),
*#,
$m_2( #, $f~_4, # ),
*#,
};
%
$f_4()
{
*@;
@[ $f_2 | $f_3 ](*#);
*@;
}
%
```

- パターンの補足説明

  $f~_4 は，$f_4 に記述された処理が$f_4 から呼び出されるどの関数に存在してもよいことを示す．

## B.10 パターン１１

- 暗黙的コード制約

  画面上のボタンを押下状態で表示する関数$f_1，$f_2 を処理直後に$f_3〜$f_7 を呼び出すと，画面遷移がすぐに行われ，ボタンの表示状態が視覚的に確認できないため，タイマー付きの画面遷移関数を使用しなければならない．

- フォールト例

  画面 A を表示中にボタン B を押下すると，直ぐに画面が切り替わって，ボタンの押下状態が確認できない．

- 制約違反パターン

```
$f_1 = 'item_on'
$f_2 = 'item_mode'
$f_3 = 'ChangeA'*
$f_4 = 'ChangeB'*
$f_5 = 'ChangeC'*
$f_6 = 'ChangeD'*
$f_7 = 'ChangeE'*
%%
@[$~f_1 | $~f_2 ]($*v);
*@;
@[$f_3 | $f_4 | $f_5 | $f_6 | $f_7 ]($*v);
%
```

- パターンの補足説明

  $~f_1，$~f_2 は，この処理を呼び出す関数が，$f_3〜$f_7 を呼び出す関数と同じ関数中もしくは，そこから呼び出される関数にまで拡張して検索されることを示す.

## B.11 パターン１２

- 暗黙的コード制約

  変数$v_1，$v_2 への設定は，規定された関数経由で行うことが原則で，通常その関数以外で設定してはいけない.

- フォールト例

  ある状態で画面が切り替わる時，切り替わった画面 A 上で表示されるべきアイテムが完全に表示されず，前の画面のボタンなどが表示されたままになる．切り替わった画面 A からさらに自動的に画面 B が起動した時に，画面 B の起動処理で$v_1，$v_2 の値が変更されたため，画面 B の処理が正常のルートを通らなくなり，画面の切り替えが中断してしまう.

- 制約違反パターン

```
$v_1 = 'Data.Mode'*
$v_2 = 'Data2.Mode'*
%%
@[$v_1 | $v_2] = #;
%
```

- パターンの補足説明

  なし.

## B.12 パターン１３

- 暗黙的コード制約

  $v_1 は本来の情報を正確に表していないため，ある特定の条件下以外で参照してはいけない.（後日，再設計により解消）

- フォールト例

  ある画面 A で，キー B 操作による処理直後に表示が完全に切り替わる前に別の操作を行うと，画面 A が完全な状態にならない．$v_1 は画面 A の表示状態を表すが，キー B 操作直後完全に画面が切り替わる前は初期化されてしまい，正しい状態を保持していないため，その途中でこの変数を参照して画面を表示すると，画面が正しい状態にならない.

- 制約違反パターン

```
$v_1 = 'Screen_flg'*
%%
@<$v_1>;
%
```

- パターンの補足説明
  なし.

## B.13 パターン１５

- 暗黙的コード制約
  イベントテーブル上でイベント$m_1 に対して呼び出される関数$f_1 に対して，引数$f_2 がある場合，それが頻繁に処理されることがあるため，問題ないことを確認しなければならない.
- フォールト例
  特定の処理を持つ画面で，ある情報を受信するとイベント$m_1 が頻繁に発生し，それに対する処理を行うが，このとき$f_2 に画面をクリアして表示する処理があったため，画面が頻繁にちらつく．$f_2 内での画面をクリアする処理は，特定の条件付で行わなければならない.
- 制約違反パターン

```
$m_1 = 'OPERATE_Y'
$m_2 = 'EVENTPROC'
$t_1 = 'EventTbl'
$f_1 = 'func_Y_proc'*
%%
struct $t_1 $v[] = {
        *#,
        $m_2( $m_1, $f_1, $f_2 ),
        *#,
};
%
```

- パターンの補足説明
  このパターンでは，$f_2 の関数名を検索する.

## B.14 パターン２０

- 暗黙的コード制約
  あるイベント$m_2 に対して呼び出される関数では，return(0) をすると，それから以降のテーブル上の処理が行われないため，特殊なケースを除いて$m_2 に対して呼び出される関数内に return(0) を用いてはならない.
- フォールト例
  ある画面上でキー B 操作直後にキー C で別の画面に遷移．本来タイムアウトするべきだが，タイムアウトせず，画面がフリーズする．画面遷移のために 2 つのイベントを処理するが，その一方のみ$f_1 のなかで return(0) によって正常に処理を行うテーブル上の処理まで到達せず，遷移が中途半端であるため，正常なタイムアウト処理ができなくなった.

- 制約違反パターン

```
$m_1 = 'OPERATE_R'*
$m_2 = 'EVENTPROC'
$t_1 = 'EventTbl'
%%
struct $t_1 $v[] = {
        *#,
        $m_2( $m_1, $f_1, # ),
        *#,
};
%
$f_1()
{
*@;
        return(0);
*@;
}
%
```

- パターンの補足説明

  なし.

## B.15 パターン２２

- 暗黙的コード制約

  １つのイベントテーブル上であるイベント$m_1 に対して処理を行う場合，その処理中に$f_1 による処理が行われると画面表示に異常が発生するため，イベント$m_1 処理中に$f_1 が処理されないように制御を行わなければならない.

- フォールト例

  システム起動直後，あるキー操作による処理を行うと，その処理中の画面上に不要なボタンが表示され，さらに別の操作で画面上にゴミが残ってしまう.

- 制約違反パターン

```
$f_1 = 'item_start'
$m_1 = 'OPERATE_K'*
$t_1 = 'EventTbl'
$m_2 = 'EVENTPROC'
$m_3 = 'SUBEVENT'
%% 第１サブパターン
struct $t_1 $v_1[] = {
*#,
        $m_2( $m_1, #, # ),
*#,
};
% 第２サブパターン
struct $t_1 $v[] = {
        *#,
        $m_3( #, $v_1 ),
        *#,
        $m_2( #, $f~_3, # ),
        *#,
};
% 第３サブパターン
```

```
$f_3()
{
        @$f_1( *# );
}
%
```

- パターンの補足説明
  $f~_3 は第３サブパターンに記述された処理が$f_3 から呼び出されるどの関数に存在してもよいことを示す.

## B.16 パターン２３

- 暗黙的コード制約
  変数$v_1，$v_2 への設定は，ある処理中に行われると，その処理後に参照されて，別の意味に解釈されてしまうため，$v_1，$v_2 への設定を行うのは，問題の処理中にでないという条件下で行わなければならない.
- フォールト例
  あるキー操作を 1 回目行い，一定時間後に再度実行すると，次からその処理がブロックされて，完全に機能しなくなる.
- 制約違反パターン

```
$v_1 = 'Data.No'*
$v_2 = 'Data2.No'*
%%
@[$v_1 | $v_2]  = #;
%
```

- パターンの補足説明
  [$v_1 | $v_2] は，この変数のいずれかを示す．どちらの変数も，同じ制約を持っている.

## B.17 パターン２４

- 暗黙的コード制約
  $f_1 もしくは，第３引数に 37 を持つ$f_2 は動作中のある情報を取得するが，機能追加により，ある状態では従来考えられていた情報を取得できないケースが発生したため，この関数呼出し時には，必要な情報を取得できる状態であることを確認しなければならない.
- フォールト例
  ある画面状態で，デバイス A を抜き挿しすると，通常初期画面状態で起動するべきだが，デバイス A 抜出時の特殊な画面が初期画面上に表示されてしまう.
- 制約違反パターン

```
$f_1 = 'func_get_no'*
%%
@$f_1();
%
------------------------------
$f_2 = 'get_no'*
%%
@$f_2( #, #, 37, #);
%
```

- パターンの補足説明
  この問題は，２つの関数に共通する問題なので，２つの独立したパターンによって，検出される．

## B.18 パターン２５

- 暗黙的コード制約
  $f_1 によって得られる文字列情報は，0 バイトの場合（データ無し）の場合が存在するため，この文字列サイズを参照する箇所が，0 バイトであるケースを想定した処理になっていなければならない．
- フォールト例
  $f_1 で得られた情報を画面上する際に，他の情報を組み合わせて表示するが，その表示サイズを計算する際に$f_1 で得られた情報が 0 の場合，不定なバッファの内容を表示領域にコピーしている．実際に故障として報告はされていないが，別の不具合対応のため，この部分は 0 バイト対応を含めてリメークされている．
- 制約違反パターン

```
$f_1 = 'func_get_string'
$f_2 = '*strlen'
%%
@$f_1(#, $v_1);
*@;
@$f_2($v_1);
%
```

- パターンの補足説明
  $v_1 は取得する情報の格納用のバッファとなる．

## B.19 パターン３０

- 暗黙的コード制約
  $f_1 は，ある画面状態を終了する関数で，その中でである情報を取得してグローバル変数に設定しているが，この情報をグローバル変数に設定してはいけない状態があるため，これを呼び出す時にはその状態にないことを確認しなければならない．
- フォールト例
  ある画面上で操作中に特殊なキーで割り込みを行い，割り込み処理終了後に元の画面に戻ると，画面の状態が完全に元の状態に戻らない．
- 制約違反パターン

```
$f_1 = 'ScreenInit'*
%%
@$f_1( *# );
%
```

- パターンの補足説明
  なし．

## B.20 パターン３１

- 暗黙的コード制約

  $f_1 は引数として$m_1，$m_2 を使用してはならない．本来，$m_1，$m_2 に類似した別のマクロ定義を渡さなければいけないが，ある時点で$m_1，$m_2 を渡す特殊なケースが混入し，$f_1 は引数として$m_1，$m_2 が使用できるような誤認識が発生し，間違った使用が行われる可能性がある．

- フォールト例

  ある情報画面でキー A,B を同時に押下．キー B に対応する画面が表示されるが，タイムアウトで本来戻るべき画面に戻らない上，タイムアウト時間経過後は画面操作が効かなくなる．

- 制約違反パターン

```
$f_1 = 'event_cancel'*
$m_1 = 'OPERATEG_[0-9A-Z]*'
$m_2 = 'OPERATE _[0-9A-Z]*'
%%
@$f_1( [ $m_1 | $m_2 ]);
%
```

- パターンの補足説明

  [ $m_1 | $m_2 ] は$f_1 は引数として$m_1，$m_2 のいずれかのケースを示す．

## B.21 パターン３３

- 暗黙的コード制約

  関数$f_1 はその内部で，ある変数を参照してイベントの処理を禁止する処理を行っているため，これを使用するイベントテーブルにおいて，その変数が正常に制御されていることを確認する必要がある．

- フォールト例

  ある画面表示中にキー操作で別の画面に遷移しようとしたときに，割り込みのイベントが発生して，別画面に遷移．そこで，あるキー操作が全く効かなくなる．

- 制約違反パターン

```
$f_1 = 'func_stop'*
$t_1 = 'EventTbl'
$m_2 = 'EVENTPROC'
%%
struct $t_1 $v[] = {
*#,
        $m_2( #, $f_1, # ),
*#,
};
%
```

- パターンの補足説明

  なし．

## B.22 パターン３７

- 暗黙的コード制約

  $f_1 は，$f_2 で設定される位置・フォーマット情報に従ってあるアイテムを画面上に表示するので，$f_1 呼び出し前には，必ず一度は$f_2 が呼び出されていなければならない.（ただし，システム起動後一度でも$f_2 が呼出されていればよいので，このパターンの故障発生率は高くないが，このシステムの別の改造版では，これらの関数を必ず対で呼出すように再設計が行われている.）

- フォールト例

  システム起動直後にキー A →キー B →キー C を連続して押下すると，キー B に応じた画面上で画面が固まり，操作が効かなくなる.

- 制約違反パターン

```
$f_1 = 'disp_string'*
$f_2 = 'set_disp'
%%
^@$f_2( *# );
*@;
@$f_1( *# );
%
```

- パターンの補足説明

  なし.

## B.23 パターン３８

- 暗黙的コード制約

  このシステムでは，符号なし（unsigned）変数の先頭は “qp_” で始まるというコーディング規約があるため，ループ条件にこれらの変数からの引き算した値がある場合，ループ回数が異常に大きな値にならないことを確認しなければならない.（qp_a < b の場合の (qp_a - b) > 0 のような条件文）

- フォールト例

  ある画面表示で必要な情報を取得し，編集する作業中に，システムが無限ループでストップ．情報が 0 バイトであることを考慮しないコードになっていたため，ある値を引き算した結果のループ回数が，異常に大きい値になり（0-5 など），ループが終了しなくなったため.

- 制約違反パターン

```
$v_1 = 'qp_[a-zA-Z0-9_]*'
$f_1 = 'for'*
$f_2 = 'while'*
%%
@[ $f_1 | $f_2 ]( *# $v_1 - *# )
%
```

- パターンの補足説明

  for，while の条件文中からは，単純に変数名とマイナス “-” 記号を検索する．従って，$v_1 がループ条件になっているかは，保証されない．#には，“;” 等の記号を含む.

## B.24 パターン３９

- 暗黙的コード制約

  $m_2 のイベントに対して，関数$f_2 を呼び出す処理があるイベントテーブル上で，第３引数が$f_3 である関数$f_1 を呼び出す処理がある場合，関数$f_2 処理中に$f_1 が呼び出されると，本来行われる$f_3 の実行が行われなくなる可能性があるため，問題ないか確認しなければならない.

- フォールト例

  ある画面でキーによる操作を実行中に，状態が変化して画面を切り替える処理が自動的に発生するが，実際に画面が切り替わらない.

- 制約違反パターン

```
$f_1 = 'func_set'
$f_3 = 'func_request'
$m_2 = 'OPERATE_Q'*
$f_2 = 'func_Q_proc'
$m_3 = 'EVENTPROC'
$m_4 = 'SUBEVENT'
$t_1 = 'EventTbl'
%%
struct $t_1 $v_1[] = {
*#,
        $m_3( $m_2, $~f_2, # ),
*#,
};
%
struct $t_1 $v[] = {
        *#,
        $m_4( #, $v_1 ),
        *#,
        $m_3( #, $f~_4, # ),
        *#,
};
%
$f_4()
{
*@;
@$f_1( #, #, $f_3 );
*@;
}
%
```

- パターンの補足説明

  $~f_2 は，この処理が実際のテーブルの記述される関数から呼び出される関数であることを許可し，$f~_4 は$f_4 に記述された処理が$f_4 から呼び出されるどの関数に存在してもよいことを示す.

## B.25 パターン４２

- 暗黙的コード制約

  $f_2 は指定するバイト数分のみ文字列情報のコード変換を行うが，２バイト・１バイトコードが混在している場合があるので，正しい文字の切れ目になるバイト数を引数として指定しなければならない．

- フォールト例

  ある画面上に表示する文字列は，20 バイトを上限とするが，変換するバイト数を 20 で指定したため，その時変換した情報の最後に 1 バイトのゴミコードが表示される場合がある．

- 制約違反パターン

```
$f_1 = 'func_data_chg_to_info'*
%%
@$f_1( *# );
%
```

- パターンの補足説明

  なし．

## B.26 パターン４３

- 暗黙的コード制約

  $v_1 はある情報の表示・非表示状態を表すが，これを表示状態と関係なく設定すると，その後の表示状態の制御がおかしくなるため，設定箇所，特に初期化箇所について，問題ないことを確認しなければならない．

- フォールト例

  ある画面で画面上に表示されるマークが，ある操作によって画面が移動する時は，消去されなければならないが，表示されたままとなる．

- 制約違反パターン

```
$v_1 = 'pq_draw'*
%%
@<$v_1>;
%
```

- パターンの補足説明

  <$v_1>は，$v_1 がプログラムコード上で使用されるすべてのケースを示す．参照，設定，宣言に依存しない．

## B.27 パターン４４

- 暗黙的コード制約

  $v_1，$v_2 は，画面を描画する時にある情報を指定するために使用するが，この変数を設定して，画面描画要求を発行しない場合は，画面状態と変数値が不一致のまま放置されてしまい，その後の処理で異常が発生する可能性があるため，問題ないことを確認しなければならない．

- フォールト例

  ある画面表示中にキー A 操作で別画面に遷移．そこでキー B 操作で画面状態を変更後，前の画面に戻り，再度キー B 操作．画面状態が変化しないはずなのに，操作処理完了後，自動的に画面が変わってしまう．

- 制約違反パターン

```
$v_1 = 'Data.No'*
$v_2 = 'Data.Len'*
$f_1 = 'func_request'
%%
@[$v_1 | $v_2] = #;
*@;
^@<$f_1>;
%
```

- パターンの補足説明

  <$f_1>は通常の関数呼び出しと違い，関数への引数として使われ，呼び出されるケースもあるため，単純に$f_1 と記述しない．

## B.28 パターン４５

- 暗黙的コード制約

  $f_1 は画面の状態を示すある情報を取得する関数であるが，$f_1 呼出し時にその画面が正常に表示されていないと，不定な値を取得してしまうため，画面が正常に表示されている状態下でのみ呼び出されることを確認しなければならない．

- フォールト例

  ある画面表示中にキー A 操作．処理中にキー B 操作で別の画面に遷移し，タイムアウトで元の画面に戻り，画面表示が完了する前に再度キー B 操作→タイムアウト．この時，元の画面表示が，正常に行われない．

- 制約違反パターン

```
$f_1 = 'abc_get_from_opf'*
%%
@$f_1(*#);
%
```

- パターンの補足説明

  なし．

## B.29 パターン４６

- 暗黙的コード制約

  ある変数$v_1 は，このシステムを開発するために再利用した元のシステムで使用されていた変数で，このシステムにおいては正確な情報の保持を保証されていないため，使用してはいけない．

- フォールト例

  ある画面 A 表示中にキー操作で画面 B に遷移し，さらに別のキーで画面 C に遷移．その後タイムアウトで画面 B にもどるが，この時画面 B は真っ黒になる．

- 制約違反パターン

```
$v_1 = 'hi_pos_t'*
%%
@<$v_1>;
%
```

- パターンの補足説明

  <$v_1>は，$v_1 がプログラムコード上で使用されるすべてのケースを示す．参照，設定，宣言に依存しない．

## B.30 パターン４７

- 暗黙的コード制約

  ある変数$v_1 は，本来画面の状態を表す変数であるが，ある処理を行うと，この値が必ず本来の状態ではない状態に設定されてしまうため，この変数の参照・設定箇所は基本的に変更・追加してはいけない．どうしても変更の必要があるときは，従来の設定・参照箇所すべてについて，問題がないことを確認しなければならない．

- フォールト例

  ある画面 A 表示中にキー操作で画面 B に遷移し，さらに別のキーで画面 C に遷移．その後タイムアウトで画面 B にもどるが，この時画面 B は真っ黒になる．

- 制約違反パターン

```
$v_1 = 'gh_pre_t'*
%%
@<$v_1>;
%
```

- パターンの補足説明

  <$v_1>は，$v_1 がプログラムコード上で使用されるすべてのケースを示す．参照，設定，宣言に依存しない．