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

松村知子[†] 門田暁人[†] 松本健一[†]

長年修正や機能追加が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）には，仕様書や設計書に記載されない潜在的なコーディング規則（潜在コーディング規則）が多数存在し，規則に反したコーディングを行うことはフォールト混入の原因となる．保守の過程で保守作業者が入れ替わると，規則の存在に気づいていない作業者が各規則にたびたび違反し，同種のフォールトが繰り返し混入するという問題が起こる．本論文では，レガシーソフトウェアを対象として，潜在コーディング規則に違反するプログラムコードを検出する方法を提案する．提案方法では，熟練した保守作業者が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潜在コーディング規則を列挙する．次に，各規則に違反する典型的なプログラムパターン（規則違反パターン）を形式言語により記述する．そして，対象とするソフトウェアと規則違反パターンとのマッチングを行うことで，規則に違反する箇所を検出する．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%は潜在コーディング規則に違反したために混入したものであり，保守開始時に 39 個の規則が存在したことが分かった．さらに，試作したマッチングシステムによる実験の結果，規則違反パターンに違反する箇所が 772 個検出され，そのうちの 152 カ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 カ所が未報告のフォールトであり，提案方法がフォールトの検出に有効であることを確認した．

A Method for Detecting Fault Caused by the Violation of Implicit Coding Rules

Tomoko Matsumura,[†] Akito Monden[†] and Ken-ichi Matsumoto[†]

In the field of legacy software maintenance, there unexpectedly arise a large number of implicit coding rules, which are usually undocumented. Violating such rules causes injection of a new fault. As maintainers move between companies and job assignments in the long maintenance process, some maintainers who are not aware of a rule often violate the rule, and, the same kind of faults is repeatedly introduced. This paper proposes a method for detecting code fragments that violate implicit coding rules. 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 coding rules that lie behind the faults. Then, code patterns violating the rules (which we call violation patterns) are formally described in a pattern description language. Finally, potential faulty code fragments are automatically detected by a pattern matching technique. 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 coding rules, and there were 39 rules at the start point of the maintenance process. Moreover, 152 faults were existed in 772 code fragments detected by the prototype matching system, and 111 of faults were not reported in the fault reports. It shows that the method we proposed is effective in detecting faults.

1. はじめに

今日，大規模レガシーソフトウェアの信頼性の向上と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[15),22)]．レガシーソフトウェアとは，長期にわたって開発・保守が繰り返されている遺産的なソフトウェアのことで，30 年以上保守されているものも存在する．長期にわたる保守の過程では，たび重なる機能変更や追加，修正のために Code Decay[2)]（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）が起こり，ソフトウェアに変更を加えることが次

† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology

第に困難となる．Eick らは Code Decay の症状としてモジュール間の結合度の増大，およびモジュールの凝集度の低下が起こることを示した[2]．

レガシーソフトウェアの保守作業の現場では，このような複雑化したプログラムコードに対する知識・経験への依存が大きい．たとえば，モジュール間の依存関係は，一見して認識できないものも多く，見落とされがちである．その結果，フォールトが混入することもしばしば発生する．また，作業中に依存関係が発見されたとしても，作業担当者の記憶にとどめられるのみで，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，経験的に得られた知識は流出してしまうことが多い[14]．特に複数の会社やグループが並行して保守作業を行うプロジェクトでは，知識の共有を確実に行うことは困難である．そのため，プログラムコード間の依存関係の見落としによって生じるフォールトは繰り返し混入し，潜在化したり，故障を発生させたりする．

本論文では，このようにフォールトの原因となりやすく，かつ保守対象のレガシーソフトウェアに熟知した者だけが知っているプログラムコード間の潜在的な依存関係を，保守作業者が守るべき規則（以降では，「潜在コーディング規則」と呼ぶ）としてとらえる[10)~13)]．機能拡張・変更が繰り返されてきたレガシーソフトウェアには，「大域変数 A に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 の呼び出し直後に関数 Y を呼び出すと異常が発生する」といったフォールトに結び付く多くの規則が存在する．保守作業者は，これらすべての規則に違反していないかどうかを確認しながらコーディングを行う必要がある．しかし，これらの規則は開発・保守過程で予期せずに発生し，明文化されていないため，規則を知らない保守作業者が規則に違反するコードを作成してしまう危険が大きく，慎重なコードレビューやテストが要求される．さらに，レビュアがこれらの規則違反を見つけるには，規則に関する知識とともに手作業によるコードチェックが必要で，非常にコストがかかり，かつ不確実である．2 章で述べるように，我々の調査したレガシーソフトウェアでは，保守工程で発見された故障の約 32.7%は潜在コーディング規則の違反により混入したものであった．

このような問題背景から，本論文では，潜在コーディング規則に違反していると思われる部分のソースコードを自動的に検出・警告する方法を提案し，ある大規模レガシーソフトウェアのサブシステムに適用したケーススタディについて報告する．保守作業者がソフトウェアの機能拡張や変更を行った直後に，変更されたソースコードから潜在コーディング規則に違反している箇所を自動的に検出・警告できれば，潜在コーディング規則の違反によるフォールトを次工程に進む前に発見し除去できると期待される．また，コードレビューなどの手作業によって潜在コーディング規則違反コードを見つける必要がなくなり，保守コストの削減とチェック漏れによるフォールトの混入を防ぐことが可能になる．

提案方法では，「潜在コーディング規則に違反する典型的なプログラムパターン」を形式言語によりあらかじめ記述しておく（以降，これを「規則違反パターン」と呼ぶ）．そして，対象となるレガシーソフトウェアと規則違反パターンのマッチングを行い，マッチした部分のソースコードを保守作業者へ提示する．規則違反パターンは，当該ソフトウェアの保守工程において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できる．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，および，フォールトの除去方法などについての情報である．抽出した規則違反パターンの形式的な記述には，文献 19) 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．提案方法では，プログラムコードの静的な解析から得られるパターンのみが規則違反パターンとして記述可能である．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（変数の値や関数の実行順序など）は，実行時にその状態が成立しうるプログラムコード上の静的なパターンとして（可能な範囲で）記述することとなる．

この提案方法の有効性の評価のため，パターンマッチングのプロトタイププログラムを作成し，実際に運用されているレガシーソフトウェアの一サブシステムのフォールト報告データやソースコードを用いて，適用実験を行った．

以降，2 章では，潜在コーディング規則と規則違反パターンについて述べる．3 章では提案方法について述べ，4 章ではそのケーススタディを行う．5 章では関連研究について述べ，6 章ではまとめと今後の課題を述べる．

2. 潜在コーディング規則と規則違反パターン

2.1 潜在コーディング規則の特徴と問題

潜在コーディング規則は，「遵守すべきプログラムコード間の依存関係」のうち，仕様書や設計書に明記

表 1 潜在コーディング規則と規則違反パターンの例
Table 1 Examples of implicit coding rules for violation pattern.

潜在コーディング規則の例	規則違反パターンの例
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．
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，関数 Y と関数 X を連続して呼び出すと，ある機能が正常に動作しない．	関数 Y の呼び出し後に関数 X が呼び出される．

されておらず，将来にわたって繰り返し違反が行われると予想されるものを指す．保守作業者が機能変更，機能追加，修正などを行う際に，規則に違反したコーディングを行うと，フォールト混入の原因となる．潜在コーディング規則の特徴を以下に記す．

- 保守作業者の頭の中に暗黙的に記憶されていることが多く，保守作業者の退職時や入れ替わり時に規則に関する知識が失われやすい．
- いわゆる「コーディング規約」のようにソフトウェア開発・保守組織においてあらかじめ定められたものではなく，長期にわたる保守の過程で予期せず発生するものである．
- 保守対象のソフトウェアに特化して発生する規則であり，一般的なプログラミング方法を定めた規則（文法やライブラリ仕様など）とは異なる．そのため，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潜在コーディング規則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 [20),24)] 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規則違反の抽出には利用できない．

潜在コーディング規則の存在は，信頼性の低下，および，保守コストの増大の大きな要因となる．具体的には，以下の点である．

- 潜在コーディング規則に違反することはフォールト混入の原因となるため，慎重なレビューや膨大な再テストが必要である．
- 潜在コーディング規則は，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，規則を知らない保守作業者によって，規則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．
- 潜在コーディング規則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，必ずしも容易ではない．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り，時には新しい規則を混入してしまうことがある．再設計手法として Refactoring [3)] 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．

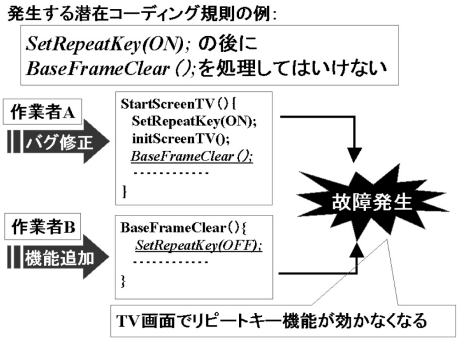

図 1 規則違反パターン発生状況の例
Fig. 1 Example of the condition in which a violation pattern was generated.

2.2 規則違反パターンとフォールト検出

潜在コーディング規則に対して，規則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「規則違反パターン」と呼ぶ．実際にケーススタディで抽出された潜在コーディング規則の例に対する規則違反パターンを**表 1** に例示する．

このような規則違反パターンが発生する状況の一例を**図 1** に示す．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 1 のような状況で，コードレビューや単体テストで発見できないフォールトが混入することがしばしば発生する．実際にこのケースでは，作業者 A が担当するコードにおいて，規則違反を修正した．一方，作業者 B の担当コードを変更して，規則が発生しないようにすることも考えられるが，このケースでは，他の多くのモジュールに影響を及ぼすために規則の解消は困難であった．

このような規則違反パターンに起因するフォールト（故障を発生させるコード）の検出方法としては，現状では主に 2 通りの方法がある．

(1) 故障が発生・報告されたときに，その調査から

規則違反パターンが見つかることがある．見つかった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パターンの検索を手作業でやることは大きな負担となる．(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規則違反パターンを把握していることが必要である．2.1 節で述べたとおり，現状では保守作業者の記憶に頼って行われている．

2.3 潜在コーディング規則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，フォールトと潜在コーディング規則の件数を調査した（このソフトウェアの詳細については，4.2 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 件が潜在コーディング規則の違反によるものであることが分かった．実際に存在する潜在コーディング規則は 45 個で，9 件の故障は原因となる規則が共通のフォールトであった．このうち，2 つの潜在コーディング規則は，後日規則を解消するように再設計され修正されたが，残りの 43 個の潜在コーディング規則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規則違反によるフォールトが存在する．ある規則に関しては，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，潜在コーディング規則に基づくフォールトは，人手による検出は必ずしも容易ではなく，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．

3. フォールト検出のための提案方法

3.1 提案するフォールト検出方法の概要

提案方法は，規則違反パターンに一致する部分のソースコードを検出・警告する．以降，提案方法をシステム化したものを「規則違反検出システム」と呼ぶ．提案方法による規則違反コードの検出手順は，以下のとおりである（**図 2** 参照）．

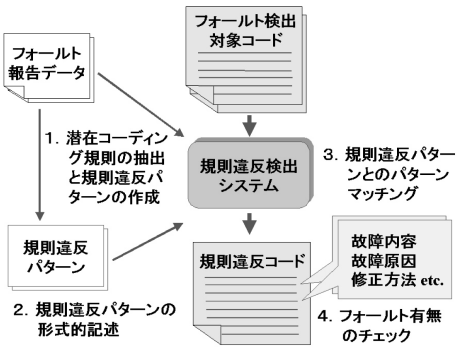

図 2 規則違反検出システムの手順
Fig. 2 Overview of fault detecting system.

(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潜在的コーディング規則を抽出する．次に，抽出された潜在コーディング規則に違反する規則違反パターンを作成する．

(2) 規則違反パターン形式的に記述し，規則違反検出システムに保存する．

(3) 規則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規則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，規則違反検出システムから出力される規則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．

3.2 潜在コーディング規則の抽出と規則違反パターンの作成

3.2.1 フォールト報告データ

潜在コーディング規則は，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 1 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ともなって，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，結果，リリース情報などの一連の情報が記載されている．本方法でこのフォールト報告

故障内容:ある画面の表示中に特殊なキー入力による割り込み処理が発生すると、元の画面に復帰するときに正しい画面が表示されない。

```
F(){
  int c;
  c=0;
  for( ; ; c++){
    if( F2(c)== 0 )
      break;
  }
  ChangeView( );
  return(RET_OK);
}
```

故障原因(フォールト):復帰画面表示のために必要なページ番号が、変数*CtrlNo*へ保存されずに、*ChangeView*()関数を呼び出したため。

修正方法:*ChangeView*()関数を呼び出す前に、ページ番号を変数*CtrlNo*に保存する。

図 3 フォールト報告データの例
Fig. 3 Example of fault report.

パターン表現記号			
宣言	$d	$*d	$d_{name}
型	$t		$t_{name}
変数	$v	$*v	$v_{name}
関数	$f		$f_{name}
式	#	#*	#_{name}
命令	@	@*	@_{name}

$v	不特定の変数
$*v	不特定の変数集合
$v_{name}	特定の変数
@[stmt1| stmt2]	いくつかの命令のどれか
@<id_1>	特定の識別子を参照
%%	キーワード宣言とパターン記述の分離記号

図 4 パターン表現記号一覧(文献 19) から)
Fig. 4 Pattern description symbols (from Ref. 19)).

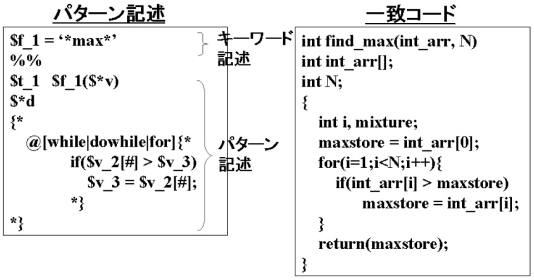

図 5 "配列から最大値を求めるアルゴリズム" のパターンの記述例(文献 19) から)
Fig. 5 Pattern for finding the maximum in an array of integers (from Ref. 19)).

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,修正方法などがあげられる(**図 3** 参照).潜在コーディング規則は主に故障の原因から抽出される.しかし,その他の情報も規則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をチェックしたり,故障の発生を確認したり,フォールトを修正したりする際に重要な情報となる.

3.2.2 規則の抽出とパターン作成

潜在コーディング規則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).

潜在コーディング規則からの規則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.規則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.また,マッチング元になる規則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,開発プロセス中に規則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規則違反パターンの有無を確認したり,抽出された規則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 3)から抽出される潜在コーディング規則は「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,規則違反パターンとしては「CtrlNo への代入なしに ChangeView() 関数を呼び出す」というように作成する.

3.3 規則違反パターンの形式的記述

規則違反パターンの形式的記述には,文献 19) 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.実際に問合せを行う場合に使用する構文要素の主なものは**図 4** にあげるもので,「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 19) には述べられている.さらに,もし他の構文要素を追加が必要な場合,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 5** 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規則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,規則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

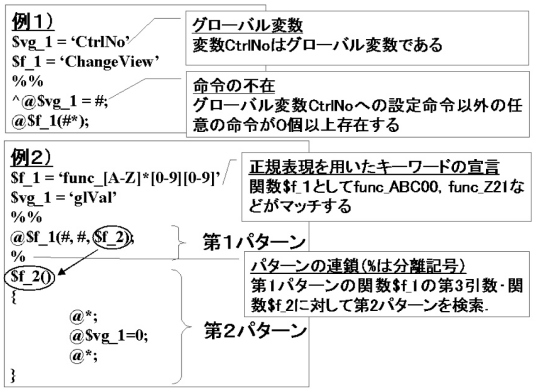

図 6 パターン記述の拡張
Fig. 6 Extension of pattern description symbols.

行う（**図 6** 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規則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 6 例 1 の “^@” が命令の不在を表現している．

（2） $vg 記号の追加：ローカル変数に対して大域変数が持つ特徴に起因したフォールトがある．たとえば，1 つの大域変数が多数のモジュールで設定・参照されている場合，ある変更が他のモジュールに影響し，フォールトの原因になる．このように，ローカル変数と大域変数の意味の違いは，フォールトに関しては大きいため，特に大域変数を示す記号が必要となる．基本的な使い方は$v と同じ．図 6 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ことが多く，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．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 6 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 6 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．

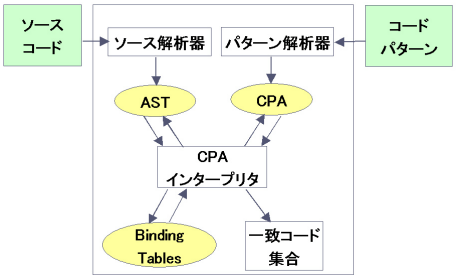

図 7 SCRUPLE システムのアーキテクチャ（文献 19) から）
Fig. 7 The architecture of the SCRUPLE system (from Ref. 19)).

上記の規則違反パターンを記述言語で記述した実例のいくつかを付録に示す.

3.4 コード パターンのマッチング

文献 19) 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 7** のとおりである．ソースコードはソース解析器によって AST（属性構文木）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 8** に示す．また，図 6 の例 1 に示す規則違反パターンを CPA に変換したものを**図 9** に示す（図中の記号の詳細は，文献 19) を参照されたい）．CPA インタプリタは AST 上で CPA をシミュレートし，マッチしたものを保存していく．Binding Tables はパターン中の特定要素（名前つき要素：図 6 の$f_2 など）の情報を保持するために利用される．

本システムでは，C 言語をターゲットとしたマッチングシステムを開発した．プログラム中の各関数に対して AST を作成し，マッチングを行う．3.3 節のパターン記述言語の拡張や規則違反パターンの特徴にともない，以下の処理を AST 解析器に追加する．

（1） 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．

（2） 通常のソース解析器では，コードを 1 ステップずつ解析し入力するが，規則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．したがって，我々は CPA インタプリタへの入力となる AST を作成する際，呼び出される関数内のコードまで深さ優先

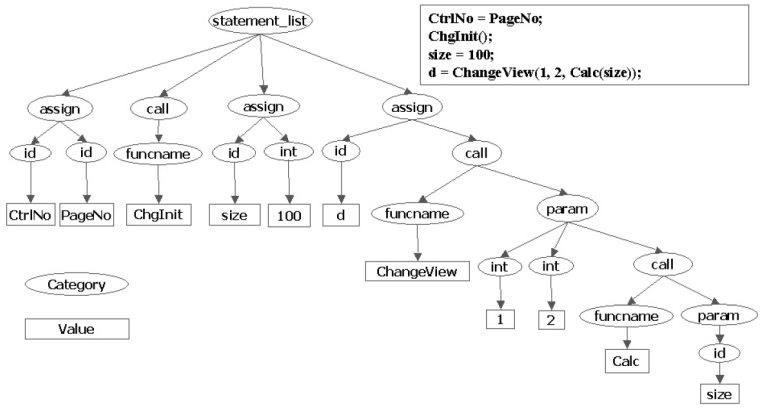

図 8 AST（属性構文木）の例
Fig. 8 Example of AST (Attributed Syntax Tree).

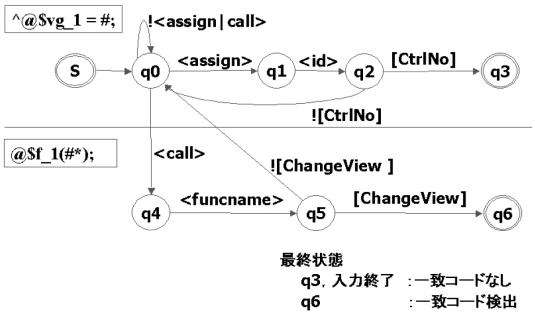

図 9 CPA の例（図 6 例 1 による）
Fig. 9 Example of CPA from Fig. 6.

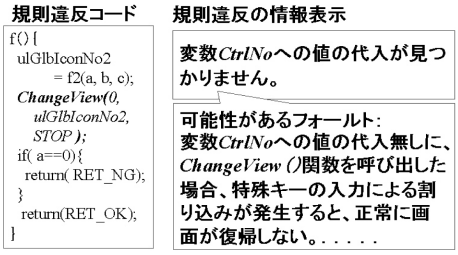

図 10 パターン照合結果の提示例
Fig. 10 Example of presentation of result from the fault detection system.

で解析するようにする．ただし，この場合一度検索した関数は再検索しないといった制御が必要である．

3.5 マッチング結果の提示と適用

マッチングの結果，規則違反パターンと一致したコードを，保守作業者がチェックするために必要な情報とともに提示する．**図 10** に図 6 例 1 の規則違反パターンに一致した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あげられる．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である（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．

3.6 提案システムの使用方法

提案する規則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，2.2 節の 2 つの規則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，規則違反検出システム上に蓄積されているすべての規則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潜在コーディング規則に違反するフォールトを発見・修正できる．

(2) 潜在コーディング規則発見時：新たな潜在コーディング規則が発見されたとき，プログラム全体を対象に新しい規則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．

4. ケーススタディ

4.1 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：規則違反パターンに一致するコード部分を検出することが，実際に保守作業に

対して有益であるかを評価する．2.3 節で述べたとおり，あるレガシーソフトウェアにおいて，潜在コーディング規則違反によって発生する故障は 32.7%にのぼるが，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いえない．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規則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．
  - すべての規則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
  - すべての規則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

4.2 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 月に開発開始され，1999 年 5 月に実質的な保守作業は完了している．

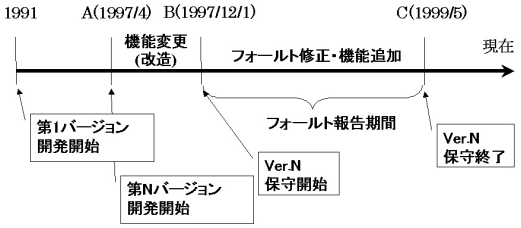

図 11 事例研究ソフトウェアの開発・保守期間
Fig. 11 Schedule of the software used by case study.

4.3 実験方法

3.6 節で述べたとおり，バグ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，すなわち新しく潜在コーディング規則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規則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 11** 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潜在コーディング規則を使用して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潜在コーディング規則が含まれるが，今回用いるのは開発当初〜保守開始時点（B）に発生した規則のみである．

本来，各潜在コーディング規則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．

4.4 評価結果

評価結果の詳細を**表 2** に示す．また，実際に検出実験に利用した規則違反コードパターンのいくつかを付録に示す．

表 2 中の「故障数」,「規則数」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潜在コーディング規則は 39 個あったが，拡張前のパターン記述言語で記述できる規則は 17 個で，拡張後に記述できたのは 30 個だった．この 30 個の規則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．

「事例数」というのは，実際に規則違反検出システムから出力された規則違反パターンに一致するコードの数である．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

表 2 評価結果
Table 2 The results of the evaluation.

	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 のうち潜在コーディング規則違反による故障数	54	32.7%	= 54 ÷ 165
C	B から抽出される規則数	45		9 件は，共通の規則違反が原因による不具合
D	保守開始時点での規則数	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規則数	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規則数	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 のうち規則違反検出システムのよって検出できた故障数	33	86.8%	= 33 ÷ 38
I	規則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発見された事例数	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

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 2 の D，F から，9 個の規則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分かる．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」という規則が 5 個，検出数が多くチェックが不可能なため規則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 2 の G，H からは，規則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，実行時のプログラムの状態（故障の発生条件）をプログラムコード上の静的なパターンに置き換えて規則違反パターンを作成したが，作成元になったフォールトが規則違反パターンにマッチしなかったものが 4 個である．前者に関しては，マッチングアルゴリズムの改良など実装上の対応で，検出が可能と思われる．後者に関しては，作成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，実用上は規則違反パターンとして有用であると考えられる．

4.5 考　　察

［提案方法の有用性］

表 2 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 2 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%のみであることが分かった．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いえる．また，この値から，潜在コーディング規則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，潜在コーディング規則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フォールトは，検出されたコードの 80.3%にのぼる．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な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」と見なしてマッチングを行う．

［提案方法の性能］

表 2 の E と F から，規則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，規則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2のHから，既知のフォールトで規則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，提案方法による規則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．

5. 関連研究

5.1 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コード検証を支援する研究には，以下のようなものがある．

小田らのCプログラムの落とし穴検出ツールFall-in C[18]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．

Sekimotoらのプラン認識を用いた方法[23]では，だれ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．

河合らのSapid[21]を用いた方法[7]では，あるまとまった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（たとえば，fopen–fclose）．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．

これらは，いずれも一般的な問題の検出を対象としており，本研究で対象とするようなシステム依存のフォールトの検出への適用は考えられていない．

5.2 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[5),6),17)]．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[9)]．

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[16)]．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・修正難度に従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したりする可能性がある．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．

5.3 不具合傾向モジュールの予測

ソースコードやフォールト報告データ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，ループ数など）を予測因子として，モジュール単位で不具合傾向予測を行う．不具合を含む傾向にあるモジュールとないモジュールに分類できると，不具合傾向にある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．

たとえば，Khoshgoftaarらは，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[8)]．Gravesらは，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[4)]．これに対して，Andrewsらは，フォールト履歴からコンポーネントやそれを含むモジュール・サブシステムの不具合傾向を判定する研究を行っている[1)]．

これらの方法では，モジュール単位での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抽出やフォールトへの対応は従来どおりのテストやデバッグを必要とする．

6. まとめと今後の課題

本論文では，フォールトの検出を支援する方法として，フォールト報告データを用いたパターンマッチングによる方法を提案し，それ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．その結果，保守工程で報告された故障の32.7%は潜在コーディング規則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カ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，抽出された潜在コーディング規則の76.9%は提案したパターン記述言語で記述可能で，さらに，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いえる．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規則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．

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[2),15)]．しかし，モジュール間の結合度の増大や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，ソフトウェアの複雑化＝潜在コーディング規則の発生，ととらえ，規則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，規則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．

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．

今回は潜在コーディング規則を取り出すためにフォールト報告データを用いたが，潜在コーディング規則を開発過程や保守作業過程で抽出することも可能である．開発プロセスの一環として，コードレビューなどで潜在コーディング規則を抽出することによって，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．

この提案方法を保守工程におけるフォールト検出方法として実用的なシステムにするためには，以下のような点が今後の課題になると考えられる．

- 規則違反パターンをすべて記述できるようなパターン記述方法の改善（4.4 節で述べたとおり，記述言語の拡張とともにマッチング方法の拡張も必要）．
- 高速で有効なパターンマッチングシステムの開発．
- パターン記述化の支援．

謝辞 本研究の一部は，新エネルギー・産業技術総合開発機構産業技術研究助成事業の援助によるものである．

参 考 文 献

1) Andrews, A.A., Ohlsson, M.C. and Wohlin, C.: Deriving fault architectures from defect history, *J. of Software Maintenance: Research and Practice*, Vol.12, No.5, pp.287–304 (2000).
2) Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S. and Mockus, A.: Does code decay? Assessing the evidence from change management data, *IEEE Trans. Softw. Eng.*, Vol.27, No.1, pp.1–12 (2001).
3) Fowler, M.: *Refactoring: Improving the design of existing code*, Addison-Wesley (1999).
4) Graves, T.L., Karr, A.F., Marron, J.S. and Siy, H.: Predicting fault incidence using software change history, *IEEE Trans. Softw. Eng.*, Vol.26, No.7, pp.653–661 (2000).
5) Hollocker, C.P.: *Software reviews and audit handbook*, John Wiley & Sons (1990).
6) Humphrey, W.S.: *A discipline for software engineering*, Addison-Wesley (1995).
7) 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).
8) Khoshgoftaar, T.M., Allen, E.B., Jones, W.D. and Hudepohl, J.P.: Data mining for predictors of software quality, *Int'l J. of Software Engineering and Knowledge Engineering*, Vol.9, No.5, pp.547–563 (1999).
9) Macdonald, F. and Miller, J.: A comparison of tool-based and paper-based software inspection, *Empirical Software Engineering*, Vol.3, No.3 (1998).
10) Matsumura, T., Monden, A. and Matsumoto, K.: The Detection of Faulty Code Violating Implicit Coding Rules, *Proc. 2002 International Symposium on Empirical Software Engineering* (*ISESE 2002*), pp.173–182 (2002).
11) Matsumura, T., Monden, A. and Matsumoto, K.: A Method for Detecting Faulty Code Violating Implicit Coding Rules, *Proc. 5th International Workshop on Principles of Software Evolution* (*IWPSE2002*), pp.15–21 (2002).
12) 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).
13) 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).
14) Monden, A., Sato, S. and Matsumoto, K.: Capturing industrial experiences of software maintenance using product metrics, *Proc. 5th World Multi-Conference on Systemics, Cybernetics and Informatics*, Vol.2, pp.394–399 (2001).
15) Monden, A., Sato, S., Matsumoto, K. and Inoue, K.: Modeling and analysis of software aging process, *Lecture Notes in Computer*

Science, Bomarius, F. and Oivo, M. (Eds), Vol.1840, pp.140–153 (2000).
16) 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).
17) Myers, G.J.: *The art of software testing*, John Wiley (1979).
18) 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).
19) Paul, S. and Prakash, A.: A Framework for Source Code Search Using Program Patterns, *IEEE Trans. Softw. Eng.*, Vol.20, No.6, pp.463–475 (1994).
20) Rational Purify. http://www.rational.co.jp/products/purify_nt/
21) Sapid Home Page. http://www.sapid.org/
22) Schneidewind, N.F. and Ebert, C.: Preserve or redesign legacy systems?, *IEEE Software*, Vol.15, No.4, pp.14–17 (1998).
23) Sekimoto, R. and Kaijiri, K.: A Diagnosis System of Programming Styles Using Program Petterns, *IEICE Trans. Information and Systems*, Vol.83, No.4, pp.722–728 (2000).
24) 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 規則違反パターンの例

以下は，実際にケーススタディで作成された規則違反パターンから取り出された典型的な例である（一部，名称などは変更している）．

A.1 パターン A

```
$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);
}
%
```

A.2 パターン B

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

A.3 パターン C

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

A.4 パターン D

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

A.5 パターン E

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

(平成 14 年 7 月 1 日受付)
(平成 15 年 2 月 4 日採録)

松村　知子

平成 14 年奈良先端科学技術大学院大学博士前期課程修了．現在，同大学院博士後期課程に在学中．ソフトウェア品質保証，ソフトウェア開発プロセスに興味を持つ．電子情報通信学会，IEEE 各会員．

門田　暁人（正会員）

平成 6 年名古屋大学工学部電気学科卒業．平成 10 年奈良先端科学技術大学院大学博士後期課程修了．同年同大学情報科学研究科助手．博士（工学）．ソフトウェアの知的財産権の保護，ソフトウェアメトリクス，ヒューマンインタフェース等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．

松本　健一（正会員）

昭和 60 年大阪大学基礎工学部情報工学科卒業．平成元年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 5 年奈良先端科学技術大学院大学助教授．平成 13 年同大学教授．工学博士．ソフトウェア品質保証，ユーザインタフェース，ソフトウェアプロセス等の研究に従事．電子情報通信学会，IEEE，ACM 各会員．