修 士 論 文

多言語に対応した解析・理解支援のフレームワークに関する研究

北陸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

伊藤 徹弥

2009 年 3 月

修 士 論 文

多言語に対応した解析・理解支援のフレームワークに関する研究

指導教官　鈴木正人 准教授

審査委員主査　鈴木正人 准教授
審査委員　落水浩一郎 教授
審査委員　青木利晃 准教授

北陸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

0710008 伊藤 徹弥

提出年月: 2009 年 2 月

概 要

ソフトウェアの高機能化によって，ソースコードの量は肥大化し，またその構造も複雑化している．そのため，ソースコードを理解するという作業が非常に困難となっている．しかしながら，現状はソースコードを理解するという作業はソースコードを読む人間の経験による部分が非常に大きい．

本研究では多言語に対応した解析・理解支援のフレームワークを作成するために，C 言語と Java を対象にし，それぞれの言語固有の特性について検討した．さらにソースコードを読む際のナビゲーションを提案した．ナビゲーションの実現の手段としてソースコードを理解する際の関心を基にしたナビゲーション木を作成した．このナビゲーション木を利用することでソースコードを理解する際，ユーザが理解に必要な情報を正確に抽出する支援を行うことが可能となった．これによって，開発保守のコストの抑制が期待できる．

目 次

図 目 次

表 目 次

ソースコード目次

第1章　はじめに

本章では，ソフトウェア開発の現状について述べる．また，本研究の目的，本論文の構成について述べる．

1.1　研究背景

現在，ソフトウェア開発の現状はソースコードをはじめから作ることが少なく，多くの場合は既存のソースコードに対してコードを追加，修正を行うことで機能の追加が行われている．既に運用されているソースコードの信頼性を再確認する必要性がなくなり，ソフトウェアの開発を低コストで抑えている．ただしこの再利用が原因でソースコードの量は肥大化し，また構造も複雑化しているという問題が発生している．

一方，他人の書いたソースコードに触れる機会も増えてきている．オープンソースのソフトウェアなどが，手軽に入手できる他人の書いたソースコードの例として挙げられる．ソースコード自体は SourceForge[1] において様々なソースコードを入手することができる．他にもソフトウェア開発者が自身のホームページ上等でソースコードを公開している例もある．しかしながら，入手は可能であるが，目的の機能がソースコードのどこに実装されているかは，ソースコードを詳細にわたって解読，理解しなければならない．しかし膨大なソースコードを全て理解することは困難であるため，支援を行うための環境が必要とされる．

1.2　研究の目的

ここで理解支援とは，ソフトウェアの修正や機能追加等の変更要求を実現するために開発者が機能を理解し，変更に必要な箇所を抽出して提示する機能と定義する．

本研究の目的は，開発者が大規模なソースコードを効率的に理解できるように支援するツールの開発と，理解する際のナビゲーションを提示することである．また，そのツールを多言語に対応させることである．ただし，C や Java のソースコードが混在し，一つのソフトウェアとなっているようなものは対象としていない．一つのソフトウェアは単一の言語で構成されていることが前提である．

1.3 論文の構成

本論文の構成について簡単に説明したものを以下に示す.

- 第 2 章では，ソースコード理解支援が必要とされている現状について指摘する．さらに，既存ツールの問題点と，先行研究の紹介とその問題点と解決策について述べる.
- 第 3 章では，先行研究で行われたフィルタについて述べる．さらに本研究で提案する新たなフィルタについて述べる.
- 第 4 章では，本研究における理解支援のアプローチ.
- 第 5 章では，提案した手法の有用性実験を行った.
- 第 6 章では，まとめと今後の課題について述べる.

第2章　理解支援と関連研究

本章では，ソフトウェア開発における理解支援の必要性について述べる．また今日，ソフトウェア開発を支援するために，様々な研究が行われている．これらと理解支援との関係についても述べることとする．さらに，関連研究について簡単に述べる．

2.1　理解支援の必要性

現在のソフトウェア開発はチームで行われることが多い．そして大規模なプロジェクトでは数万から数百万行のソースコードを全て書くことは現実的でないため，通常既存のソースコードを再利用して行われている．

作成されたソフトウェアを運用する時には様々な機能の修正 (バグフィックス) や追加が行われる．その際，変更要求を実現するには，その機能を実現している箇所を特定し適切な修正や追加を行う必要がある．しかしながら，一つのソフトウェアのソースコードは膨大なため，変更すべき箇所を短時間で効率よく発見するのは困難であり，計算機による支援が求められている．

2.2　既存ツール

現在，理解支援に利用できる開発支援ツールがいくつか存在している．ここにその例を挙げる．

2.2.1　GNU GLOBAL

GNU GLOBAL[2] はソースコードに索引付けを行うことで，変数や関数の宣言部や実行部が簡単に発見することができる理解支援に利用できるツールである．このツールは，以下の特徴を持っている．

- ソースコードからハイパーテキストを生成する．
- 様々なツールから呼び出して使用することが可能である．

現在も開発が行われており，関数や変数をリンク付けして交互に参照することを可能にする．対応環境としてはシェルのコマンドラインや bash, vi,Web ブラウザなど様々である．このツールの利点は，関数名や変数名に索引付けを行い，その結果から HTML を生成することが可能であることである．このことから，ユーザは関数名などを索引付けされたページから見つけ，ハイパーリンクによって対応するソースコードに移動することが容易にできるようになる．図 2.1 はあるオープンソースソフトウェアのソースコードに対してこのツールを適用して，その出力を Web ブラウザで表示させた結果である．今，
pos = position(TOP);
という行に注目していたとして，この関数名 position のリンクをクリックするだけでその宣言部へ移動することができる．

しかしながら，このツールは索引付けの結果を利用して関連する変数を参照できるようにしただけなので，ソースコードの構造に関する解析を行っていない．よって，目的の関数や変数の宣言部を表示することは可能だが，ソースコードから抽出された情報は膨大で，その情報量を制御することはできない．

2.2.2 Semantic Grep

Semantic Grep[3](以下 Sgrep) は，問い合わせ言語 (SQL) によりテキストファイルを検索したり，テキストストリームをフィルタリングするためのツールである．Sgrep は，正規表現に基づく SQL を実装しているため，grep と同様にあらゆる種類のテキストファイルから文字列を抽出することが可能なほか，構造化されたテキストを含むテキストファイルに対して構造を指定した文字列の抽出を行うことが可能である．構造化されたテキストの例としては，SGML，HTML，C 言語のソースコード， TEX やメールファイルが挙げられる．構造化されたテキストを含むファイルは，HTML や C などそれぞれの文法に従うファイルとして定義され，その文法がパーザに定義されている．この文法定義によって構造の抽出を可能とするツールである．Sgrep では，元のソースコードの構造をユーザが指定できることが前提で抽出を行っている．しかし実際には，抽出するためのコードに対応した問い合わせ言語による入力を作成することは Sgrep の機能に対して熟練を要する．よって入力をより直感的な表現にする必要がある．

また，GNU GLOBAL と Sgrep 双方に残る問題として，ソースコードを理解するために必要な解析手順 (ガイドライン) が存在しないことである．例えば，ある関数に注目していて，その関数を参照している箇所を抽出するとき，その関数を参照している箇所を抽出した結果と，抽出結果を含んでいる関数全体の情報を組み合わせる必要がある．これは経験者でないと分からない．

2.3 先行研究

本研究の先行研究として，新倉 [4] が行った研究について述べる．

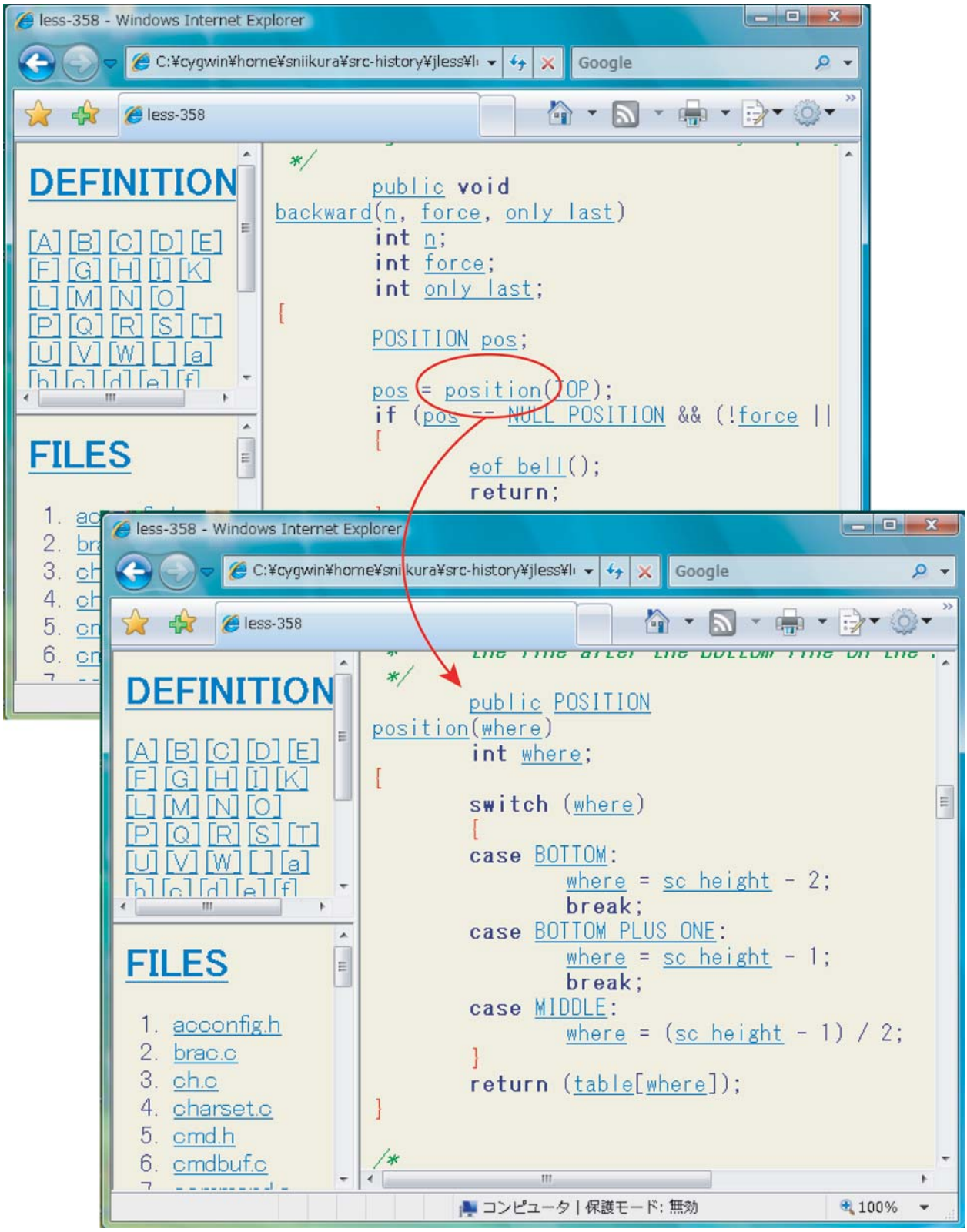

図 2.1: GNU GLOBAL による実行例

新倉はソースコードの理解に伴う情報量を抑制するために細粒度フィルタとその合成による情報抽出と抑制の仕組みを確立した．このプリミティブなフィルタを用いて情報を抽出し，抽出した情報を合成することによって変更すべき箇所を抽出することが可能となった．

与えられた変更要求に対し使用するフィルタの種類，パラメタや適用の順序などの手順を含んだ情報 (ナビゲーション) に関しては不十分な点，と C 言語のみを対象としているため他の言語には適用できないといった問題がある．そこで本研究においてソースコードを理解する際に関数の呼び出しフローに注目し，関数の役割を理解をするために必要な情報を抽出する手順をフィルタの組み合わせにより提案する．手戻りを直感的に表現するためナビゲーション木を導入する．また，C 以外の言語に適用するために，制御フローや定義，実行部の抽出方法等を拡張する．さらに

第3章　情報量の制御とナビゲーション

本章では先行研究において実装されたフィルタについて述べる．さらに，今回新しく提案するナビゲーション木という概念について述べ，ナビゲーションの必要性について述べる．

3.1　情報量の制御

本研究における情報量の制御には，先行研究で提案された，細粒度フィルタによる情報抽出を用いて行う．細粒度フィルタとは単純な機能を持ったフィルタのことで，細粒度フィルタから抽出された情報を組み合わせ，変更に必要な箇所を抽出する．

3.2　関心と関心の拡大

関心とはソースコードから変更に必要な情報を抽出する際，その時点でユーザが注目している箇所 (範囲) のことである．

関心の拡大とは，ユーザが関心を元に情報を抽出し，注目している範囲を変更することである．関心の拡大によってユーザがソースコードから変更に必要な情報を抽出することができる．関心の拡大の例を以下に示す．

まず，図 3.1 の上部のように 3 つのソースコードから成るソフトウェアがあるとする。その中でユーザが関心を持っている範囲を黄色で示す。ソースコード 1 の関心がユーザにとって情報量が多すぎるという場合は情報量を絞込む。これを関心の縮小と呼ぶ。そしてソースコード 2 の関心がユーザにとって情報量が少なすぎるという場合は情報量を増大させる。これを関心の拡大と呼ぶ。さらに、ソースコード 2 の関心からソースコード 3 の新たな関心を発見することも関心の拡大と呼ぶ。

また、ソースコード 2 から関心を拡大させ、その結果からさらに関心を拡大させ新たな関心を発見することも関心の拡大と呼ぶ。

このように関心を状況に応じて拡大・縮小することを関心の拡大と呼ぶ．

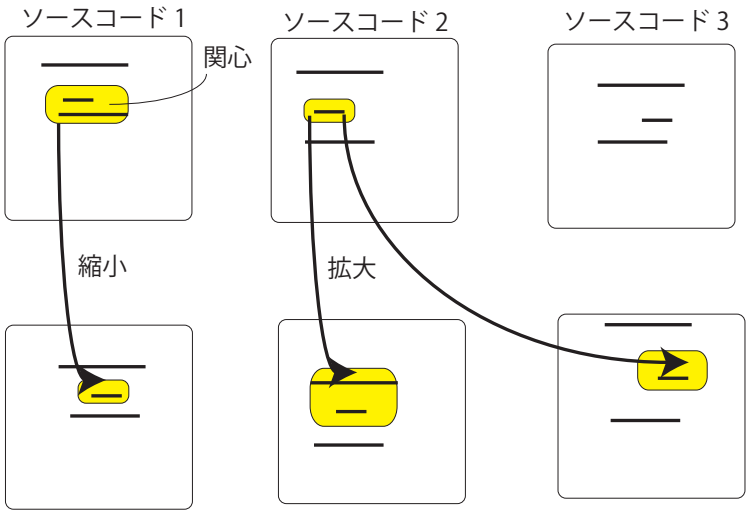

図 3.1: 関心の拡大 1

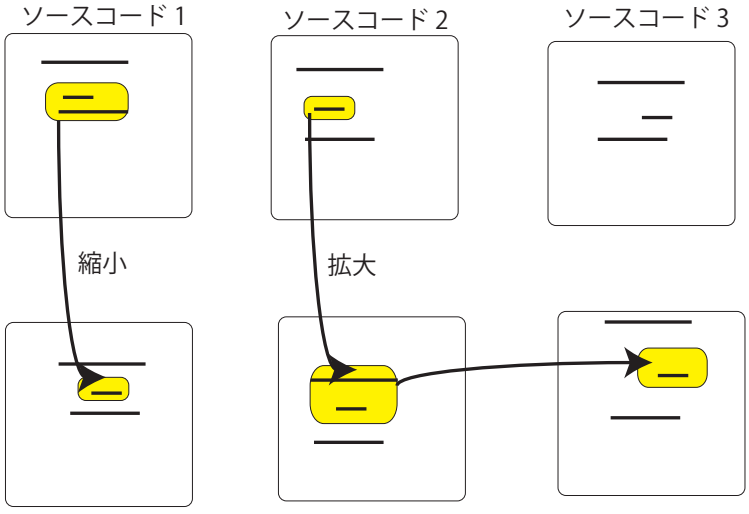

図 3.2: 関心の拡大 2

3.2.1 抽出結果の合成

先行研究で定義された7つの細粒度フィルタから抽出された情報を組み合わせることで変更に必要な箇所を抽出することは上で述べた．ここではそれをどのようにして組み合わせるかについて述べる．

細粒度フィルタにはいくつかの種類が存在しているが，どの細粒度フィルタから抽出された情報も同じデータ構造を持つように設計した．抽出されたデータから，ユーザが注目したい箇所を選択し，別のフィルタの入力として与えてやることによってand合成やor合成を実現する．情報量が多い時はand合成によって情報量を制限し，情報量が少ない時はor合成によって情報量を増やすことができる．

3.3 フィルタの機能

新倉が提案したフィルタには大まかに以下の3種類に分類され，計7つのフィルタが存在する．

- 関心を決定するフィルタ
  - 大域変数の宣言部と参照部を抽出 (GLOBAL)
  - 変数と型の宣言部を抽出 (DECL)
- 関心を拡大するフィルタ
  - 制御構造の実行ブロックを抽出 (CTRL)
  - 注目行の近傍を抽出 (NEIGHBOR)
  - 代入文の依存を追跡して抽出 (TRACE)
- 関心を絞るフィルタ
  - 範囲を宣言部と実行部に分離 (SEP)
  - 変数の出現範囲を抽出 (RANGE)

以下でフィルタの詳細について述べる．

3.4 フィルタの詳細

フィルタの入力に必要なパラメタのうち，複数のフィルタで共通に用いられているパラメタについて述べた後，個々のフィルタの詳細について述べる．なお，新倉はフィルタの実装についてC言語を対象にして行ったため，他の言語では動作を変更する必要がある

ものも存在するため，それぞれのフィルタについて述べる際に各言語依存性についても述べる．

3.3 で示したフィルタのうち，関心を決定するフィルタである 2 種類のフィルタは抽出結果が注目している行を含む関数の外に及ぶため，あらかじめ全ての大域変数定義，関数定義，型定義に対して名前と型定義等の対応表を作成しておく必要がある．

3.4.1 フィルタの共通パラメタの定義

フィルタに与える共通のパラメタとして抽象構文木である AST，関心の集合である BlockSet がある．AST とは，抽象構文木 (Abstruct Syntax Tree) のことを指し，ソースコードの情報ごとにノードを作成し，それを木構造で表したものである．木構造で表現されているため，効率的に情報を抽出できるため，本研究で定義されているフィルタはこの AST から情報を抽出する．AST の例を図 3.3 に示す．

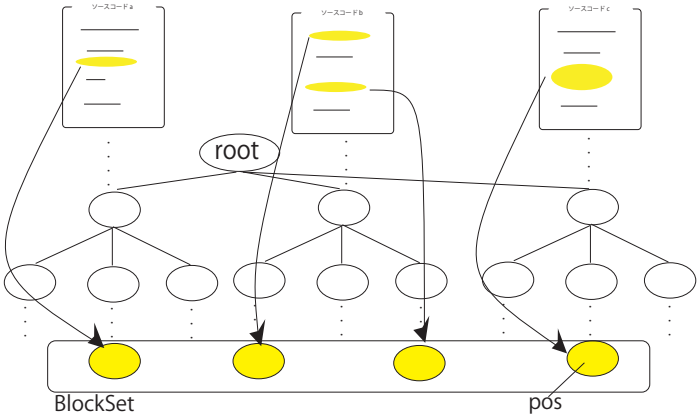

図 3.3: 単一のファイルの AST

AST は通常一つのファイルを元に木を形成している．しかし，それでは複数のソースファイルを扱うことができないため，本研究ではそれぞれのファイルの上にルート

BlockSet とは 0 個以上から成る Block の集合であり，関心の集合を表している．

Block とは注目しているファイルと注目範囲 (最初の行番号と最後の行番号) から成り，一つの関心を表している．初期状態は全ソースコードの全ての行が関心となる．

Pos は，現在の注目点を表している．BlockSet の中からユーザが注目したい行を選択してフィルタに入力する．

この 3 つのパラメタのデータ構造の定義をソースコード 3.1，3.2，3.3 に示す．

ソースコード 3.1: AST のデータ定義

```
struct AST{
        enum node;
        AST *parent;
```

```
        AST **child;
}
```

ソースコード 3.2: BlockSet のデータ定義

```
struct BlockSet{
        Block *block;
}
```

ソースコード 3.3: Block のデータ定義

```
struck Block{
        String fileName;
        int beginLine;
        int endLine;
}
```

ソースコード 3.4: Pos のデータ定義

```
struct Pos{
        String fileName;
        int pos;
}
```

ast はフィルタのパラメタとして常に用いる値であり，この値は不変である．

bs は情報抽出プロセスを開始する前の状態の場合はあまり意味を成さない．しかし，関心の拡大を行う際には出力に不要な部分をマスクすることができるため，複数のフィルタを組み合わせて情報を抽出するためには必須といえるパラメタである．このために，フィルタの戻り値は BlockSet にしてある．

3.4.2 大域変数の宣言部と参照部を抽出 (GLOBAL)

ソースコードに大域変数が利用されている場合が多く存在する．そして特にファイルが複数に及び大域変数が複数のファイルから参照されているとき，開発者にとってその大域変数の構造を理解することは非常に労力を伴う．よって大域変数の定義箇所と利用箇所が抽出できればこの作業が簡単になる．このフィルタは以下の式で定義される．

$$\mathcal{F}_{Global}(\text{ast},\ \text{bs},\ \text{var}) \tag{3.1}$$

このパラメタはそれぞれ，次のような意味を持っている．

String var 対象とする大域変数名

このフィルタはパラメタで得た変数名に一致する変数を含んでいるBlockSetを返す．もしここで定義されていない大域変数名を与えていた場合 (ローカル変数等)，このフィルタは不一致を表す空集合を返す．var と一致する大域変数を発見した場合，その大域変数をextern で呼び出しているファイルを調べ，それらのファイルに含まれている各関数に対して探索を行って，大域変数を利用している行に対して文の集合を抽出し，最終的にBlockSet にまとめて戻り値を返している．

なお，bs はこのフィルタの入力パラメタとして定義しているが，現在の動作では全ソースが入力であるため，使用していない．しかしながら，今後使用する可能性があるためパラメタとして定義している．

3.4.3 変数と型の宣言部を抽出 (DECL)

ソースコードから情報を抽出するとき，ある変数の型に関する情報が必要なことがある.. 例えば，Primitive でない型で定義されている定義文をみたときに，その型がどのような要素で構成されているかという情報が必要なことがある．しかしこれらを調べようとしたとき，その定義がヘッダファイルなどの別ファイルに記述されていることもあり，さらにその中で使用されている変数の型もPrimitive でない型の場合だと，ソースコードから必要な情報を抽出することが困難な場合がある．そこでこれら，このフィルタの目的である．

このフィルタは以下の式で定義される．

$$\mathcal{F}_{Decl}(\mathrm{ast},\ \mathrm{bs},\ \mathrm{pos}) \tag{3.2}$$

このフィルタの抽出結果も，注目している行を含んでいる関数の外に及ぶためGLOBAL と同様に型名と定義部の対応表のリストを元に抽出を行う．C 言語を対象とした場合ここにはenum，struct，union，typedef が含まれている．フィルタの出力であるbs に含まれる行は，型定義を行っている行である．

bs はフィルタの入力パラメタとして定義しているが，現在は使用していない．しかし，GLOBAL と同じ理由で入力として与えている．

3.4.4 制御構造の実行ブロックを抽出 (CTRL)

ソースコードを読む際に一つの文に注目するということは述べた．そして0 個以上の文の集まりを明示的に {} で囲ったものがブロックと呼ばれている．制御文による制御対象は1 つの文あるいは1 つのブロックであるため，このフィルタの実装によってプログラム

の制御文と制御文によって実行される可能性のあるブロックを全て抽出することが目的である．

このフィルタは以下の式で定義される．

$$\mathcal{F}_{Ctl}(\text{ast},\ \text{bs},\ \text{pos},\ \text{depth},\ \text{des}) \tag{3.3}$$

このパラメタはそれぞれ，次のような意味を持っている．

int depth 抽出対象の制御ブロックの入れ子の深さ

Descripter des 制御文の記述子の種類

ソースコード 3.5: Descripter のデータ構造

```
enum Descripter{
        IF, WHILE, DO-WHILE, FOR, SWITCH, WHOLE;
}
```

Descripter のデータ構造の定義を，ソースコード 3.5 に示す．

このフィルタの動作は文に対応する AST ノードを特定することで，その親ノードをたどり，Block の抽出を可能とした．

ある注目する文を含む制御ブロックが他の制御ブロックの入れ子になっていることがしばしばある．for 文による 2 重ループなどがこれに相当する．このとき外側のループに注目したい時と，内側のループに注目したい時と，それぞれの要求があり得る．よってこれを選択するためのパラメタとして depth を与えることにした．

C 言語における制御構造は **if**, **while**, **do-while**, **for**, **switch** の 5 種類である．よってどの制御構造に着目するかを選択するためのパラメタとして des を与えることにした．この制御構造の種類は各言語によって異なるため，他の言語に実装する場合は des を変更する必要がある．

3.4.5 注目行の近傍を抽出するフィルタ (NEIGHBOR)

ソースコードを読むとき，特定の処理を行う前後にも注目する必要がある．近傍行はデータ依存性は低いが，ある特定の関数内で処理が似通っている場合がある．初期化操作等がこれにあたる．よって，ある注目部分に対してその前後を抽出するフィルタを作成した．

このフィルタは以下の式で定義される．

$$\mathcal{F}_{Neighbor}(\text{ast},\ \text{bs},\ \text{back\_line},\ \text{forward\_line}) \tag{3.4}$$

このパラメタはそれぞれ，次のような意味を持っている．

int back_line BS の何行前まで抽出するか

int forward_line BS の何行後まで抽出するか

このパラメタはフィルタを適用する時点で注目している行から back_line 行前の情報から forward_line 行後までの行を抽出する．このフィルタは指定した値 (back_line や forward_line) が制御ブロックの外にまたいでいる場合も抽出することが可能である．

3.4.6 代入文の依存を追跡して抽出 (TRACE)

変数の値の変化を読み取りたいことがそのとき，値を変更している箇所ソースコードの流れを読み取る上で，ある代入文の値を計算するための変数に注目することがしばしばある．そのとき，それらの変数の値を決定している箇所を探すことになるが，通常それは直前のその変数の代入文である (大域変数を除く)．その際，直前の代入文に注目したとき，さらにその値を計算するための変数に注目したいこともある．このとき注目するべき変数が変化していくため，その変数を追跡するためのフィルタが必要である．そこで，その代入文を追跡するためのフィルタを作成した．

このフィルタは以下の式で定義される．

$$\mathcal{F}_{Trace}(\text{ast},\ \text{bs},\ \text{pos},\ \text{depth},\ \text{args}) \tag{3.5}$$

このパラメタはそれぞれ，次のような意味を持っている．

int depth 追跡を行う深さ

Boolean args 代入文が関数呼び出しを含む場合に，その変数を追跡対象とするか否か

3.4.7 範囲を宣言部と実行部に分離 (SEP)

C 言語のソースコードでは，ブロックの中で定義された変数はそのブロックの中でのみ有効である．またブロック中の各文は，変数定義のための宣言文か，代入や関数呼び出しなどを行う実行文に大きく分けられる．そこでブロック内の文を宣言部と実行部に分離して，その一方を抽出するフィルタを設計した．同じスコープを持つ変数のリストを特定する．

このフィルタは以下の式で定義される．

$$\mathcal{F}_{Sep}(\text{ast},\ \text{bs},\ \text{pos},\ \text{depth}\,\text{decl\_exec}) \tag{3.6}$$

このパラメタはそれぞれ，次のような意味を持っている．

int depth 抽出対象の制御ブロックの入れ子の深さ

int exec_decl 抽出対象:宣言部または実行部

depth は制御ブロックをどこまで抽出するかを指定する．Part は変数の定義部を抽出するか，実行部を抽出するかを指定する．

ソースコード 3.6: SEP フィルタで抽出可能なものの例

```
1  int main(){
2          int i;
3          int x = 3;
4          printf("x1 = %d\n", x);
5
6          for(i = 0; i < 3; i++){
7                  int y= 50;
8                  printf("y = %d\n",y);
9                  }
10 }
```

SEP の動作例をソースコード 3.6 に示す．このソースコード上で 6 行目から 8 行目があるフィルタの出力として出てきたとする．この際に SEP フィルタで，part に EXEC，depth に 1 を選択し，適用すると，8 行目の printf("y = % d \ n",y); という変数定義が抽出される．また，main 関数全体に注目していた場合に，part に DECL，depth に 2 を選択肢，SEP フィルタを適用すると，2 行目の int i; と，3 行目の int x = 3; と，7 行目の int y = 50; が出力として出てくる．

3.4.8 変数の出現範囲を抽出 (RANGE)

ソースコードを理解する上で，ある特定の変数に注目することがしばしば発生する．ローカル変数に限定した場合，その変数は定義されたブロックの中のみに出現する．よって，その変数が初期化された行から最後に呼ばれる行までに注目するだけで，その変数に関わる計算は全て抜き出されることになる．

このフィルタは以下の式で定義される．

$$\mathcal{F}_{Range}(\mathrm{ast},\ \mathrm{bs},\ \mathrm{pos},\ \mathrm{var}) \tag{3.7}$$

このパラメタはそれぞれ，次のような意味を持っている．

String vars 対象の変数名

ただし 3.4.4 や 3.4.7 と異なり，探索は必ず葉ノードから行っている．これは C 言語におけるローカル変数が，その位置を指定しないと特定できないことに由来している．

ソースコード 3.7: 同じ名前で全く別のローカル変数

```
1  int main(){
2          int i;
3          int x = 3;
4          printf("x1 = %d\n", x);
5          for(i = 0; i < 3; i++){
6                  int x= 50;
7                  printf("x2(%d) = %d\n",i ,++x);
8                  }
9          printf("x3 = %d\n",x);
10         return 0;
11 }
```

ソースコード 3.8: ソースコード 3.7 の実行結果

```
x1 = 3
x2(0) = 51
x2(1) = 51
x2(2) = 51
x3 = 3
```

ソースコード 3.7 に例を挙げる．このコードは 3 行目にローカル変数 x を定義している．しかし，6 行目で for 文のブロック内のローカル変数 x を定義している．このとき 7 行目に注目した場合，変数 x とは 6 行目で定義された変数である．しかし 9 行目に注目した場合，変数 x とは，3 行目で定義された変数であり，6 行目の変数 x は独立していることが分かる．

このように変数名をパラメタで与えても，行も与えない限り注目する変数を特定できない．

3.5　ナビゲーションの必要性とナビゲーション木

2 章で先行研究や関連研究について述べたが，そこで述べたツールには理解のためのナビゲーションについては考えていない．そのため，どれだけ変更が必要な箇所を抽出できる機能が備わっていても，全体的な手順がなければ有効に機能しない．したがって，ソースコードを読むためのガイドラインを与える必要がある．それによってソースコードを理解する際のコストを減少させることができる．

3.5.1　ナビゲーション木

ナビゲーション木とは，ユーザがフィルタを適用履歴を木構造で表したものである．それぞれのノードはユーザが関心を持った範囲を表している．ノードが保持している情報は，BlockSet である．矢印はユーザが適用したフィルタを表している．矢印は適用したフィルタの種類とそのパラメタの情報を保持している．最初の関心を決定し，そこから関心を移動すると，新たなノードがリンクされる．(図 3.4) ナビゲーションが停止した際

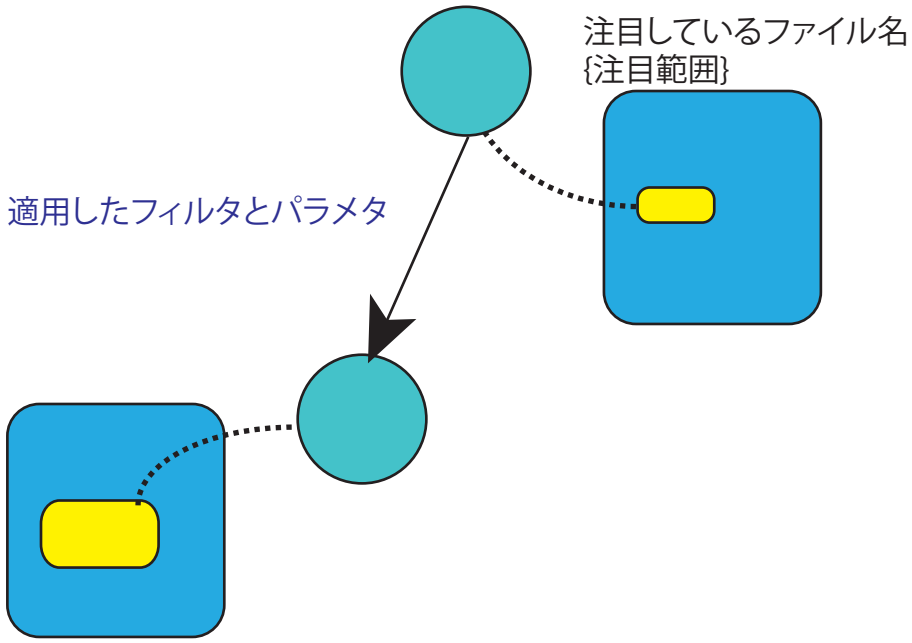

図 3.4: ナビゲーション木作成の過程 1

(図 3.5) は，ナビゲーション木を一つ遡り，そこから別の bs の中にあるを探し，そこからまたプロセスを再開すればよい．(図 3.6) それを繰り返し行い，ユーザが特定の役割を見つけたところでプロセスを停止し，図 3.7 が完成する．

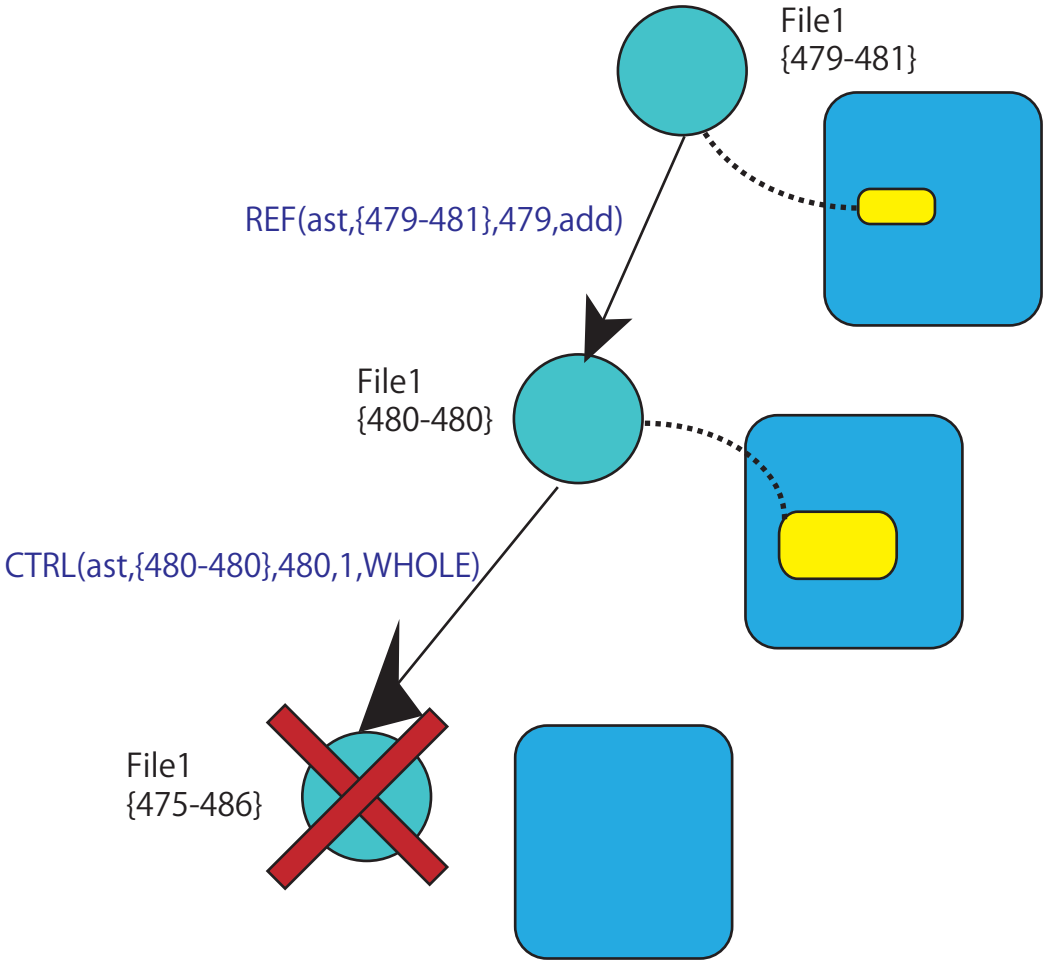

図 3.5: ナビゲーション木作成の過程 2

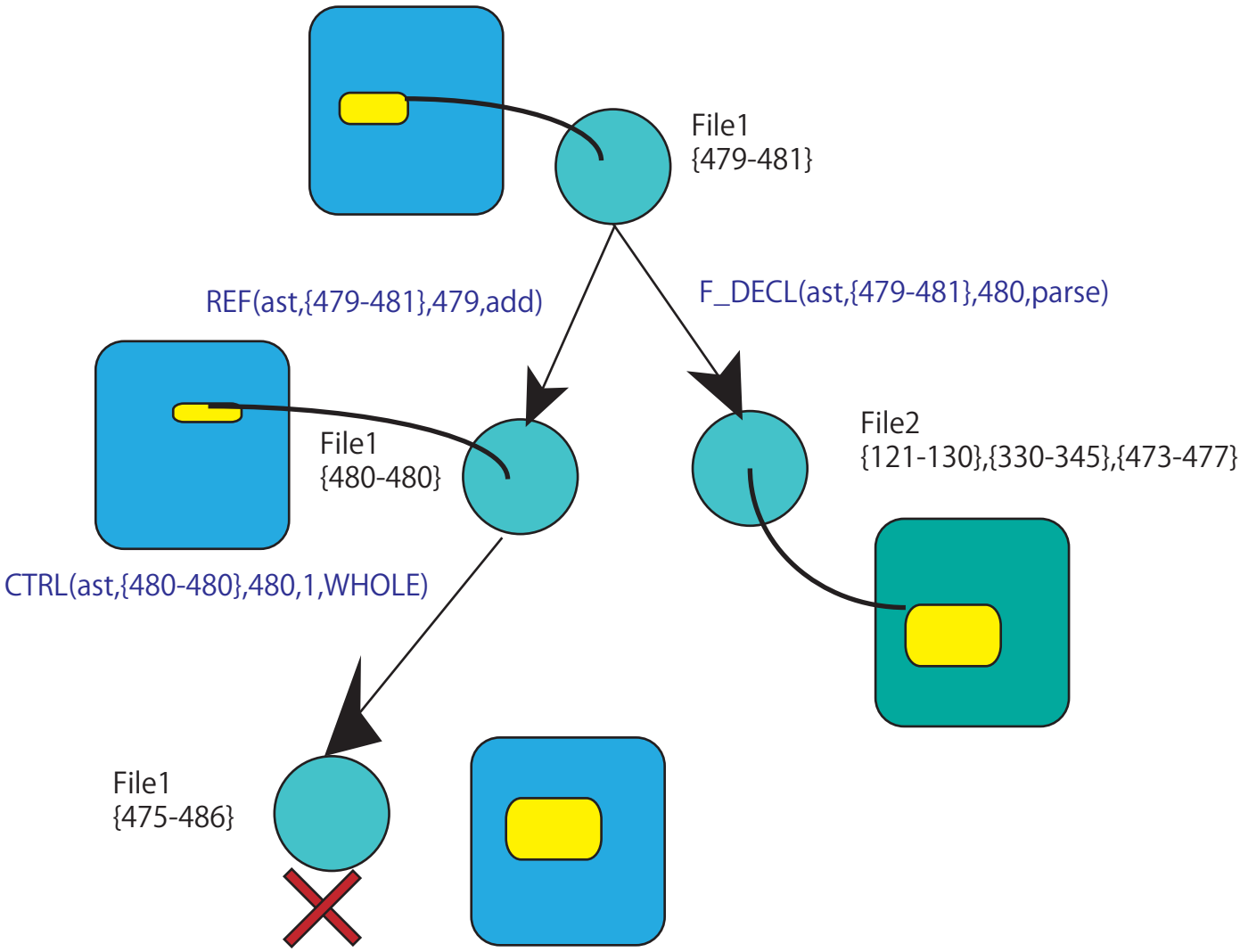

図 3.6: ナビゲーション木作成の過程3

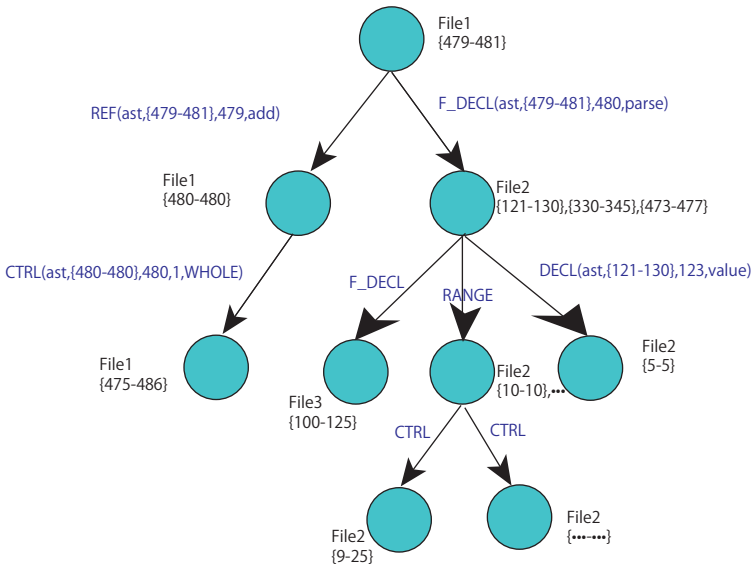

図 3.7: ナビゲーション木の例

3.5.2 ナビゲーションプロセス

プロセスの決定方法は対象言語やユーザの概念モデルによって異なる．本研究で扱った言語では一つの機能 (関数) にいくつかの要素 (引数) を代入してやり，その結果を得るといった処理の流れの存在する言語を対象としている．そのため，ソースコード上の関数の呼び出しフローを抽出することで理解に必要な多くの情報が得られる．

3.6 言語依存性

上で述べたフィルタを他の言語に用いる際に対応させる際に考慮しなければならないことににについて以下に示す．

- 大域変数と同等の機能を他の言語でどのように実現しているか
- 継承を実装している言語としていない言語
- クラス内クラスや関数内関数を定義している言語

また，他の言語に対応させる際，フィルタに代入するパラメタのうち言語依存なため，変更する必要があるのは des である．

第4章　フィルタを利用した理解支援手法

前章の 3.5 でプロセスの必要性について述べた．本章では変更に必要な箇所を抽出するために先行研究のフィルタに加え，新たなフィルタを提案し，定式化する．そしてフィルタを用いてどのような手順でソースコードから情報を抽出する手法について述べる．あるソフトウェアの開発者に対して変更要求が与えられた場合を想定し，その際の変更すべき箇所を抽出する手順について述べる．変更すべき箇所を変更箇所と呼ぶ.

4.1　提案手法の概要

変更要求が与えられたばかりの状態では開発者はどこを修正すればよいか分かっていない．ここでいう変更要求とは機能拡張やバグフィックスのことを言う．そして変更要求を分析し，その後ソースコードを理解する作業に入り，変更を変更すべき箇所を抽出するプロセスを以下に示す.

1. 機能に関連があると考えられる文字列を検索
2. 検索した結果に含まれる変数や関数呼び出しからソースの制御フローをたどる
3. 2 について以下の条件が成立するまで続ける

プロセスの停止条件

- コンストラクタの呼び出しに到達した場合
- 動作が既に分かっている関数 (ライブラリ等) に到達した場合
- 右辺に対してキャスト等の型変換操作をして型の依存関係がない場合

4.2　フィルタの改善

3.3 章で 7 つのフィルタについて述べたが，他の言語を扱うには現状では言語依存の点や不足している機能があるため不十分である．そのため，ここでは機能拡張と他の言語への対応のため，フィルタの改良と新たなフィルタを定義した.

4.2.1 フィルタの問題点と改良

3.3 で挙げたフィルタのうち，いくつかのフィルタには現状では不十分な点がある．まずはそれについて述べ，その解決策についても述べる．

大域変数の定義部と実行部を抽出するフィルタ (GLOBAL) の問題

C 言語には大域変数が存在する．しかし，大域変数という要素が存在しない言語も多く存在する．例えば Java では public 変数や static 変数は C の大域変数と同等の使われ方をすることがある．このように大域変数という要素が存在しない言語の場合は他の方法で実現されている．そのため，そのため，このフィルタを拡張する必要がある．

変数の出現範囲を抽出するフィルタ (RANGE)

関数の振る舞いを理解するためには特定の変数の値の変化に注目する必要があるのは述べた．しかしながらその際に注目する変数が代入文の左辺に出現するか右辺に出現するかで次に適用するフィルタが変わってくる．ここで，関数呼び出しの引数に注目する変数が用いられている場合も右辺に出現する場合と同じ扱いをしている．左辺の変数に注目する場合は，値が変わっているので，それがそのままその関数全体の動作に影響を及ぼす可能性が高い．右辺の変数に注目する場合は，そのため本研究では RANGE フィルタの機能を以下のように選択できるように改良した．

- 注目する変数が左辺に出現する行のみ抽出
- 注目する変数が右辺に出現する行のみ抽出
- 注目する変数が左辺，右辺に関わらず出現する行を抽出

また，改良した RANGE フィルタは，以下で定義される．

$$\mathcal{F}_{Range}(\mathrm{ast},\ \mathrm{bs},\ \mathrm{pos},\ \mathrm{var},\ \mathrm{left\_or\_right}) \tag{4.1}$$

このパラメタはそれぞれ次のような意味を持っている．

String var 対象となる変数名

int left_or_right 変数が代入文の左辺 (0)，右辺 (1)，両方 (2) のどの場合に出現する行を抽出するか

4.2.2 フィルタの新規定義

現状のフィルタのままでは機能的に不十分であることは既に述べた．そのため本研究では新たなフィルタを提案した．

特定の文字列を含むクラス，関数，変数の宣言部を抽出 (EXT_NAME)

変更要求が来た際，ソースコードの量が膨大だと対象とするソースコードに関する知識が十分でない場合は関心を決定する方法がない．そのため最初の関心を見つけるためにこのフィルタを定義した．

このフィルタ (EXT_NAME) は以下の式で定義される．

$$\mathcal{F}_{EXT\_NAME}(\mathrm{ast},\ \mathrm{var},\ \mathrm{obj},\ \mathrm{scope}) \tag{4.2}$$

このパラメタはそれぞれ，次のような意味を持っている．

String var 対象となる文字列

ObjType obj 探索対象

Scope scope 探索対象のスコープ

ここで，AST のデータ構造は 3.4.1 で示したものと同じである．ObjType と Scope のデータ構造の定義を，擬似コードで表 4.1 に示す．

ソースコード 4.1: ObjType と Scope のデータ定義

```
enum ObjType{
        CLASS, METHOD, VALUE, WHOLE;
}

enum Scope{
        PUBLIC, PACKAGE, PROTECT, PRIVATE, WHOLE;
}
```

このフィルタは全ソースコード中から入力した文字列を含むクラス，関数，変数の宣言部を抽出する．grep のような動作をするが，grep と違うのは宣言部のみを抽出するところである．ObjType とは検索対象をクラスのみ，関数のみ，変数のみ，全てを選択するために定義した．Scope とは検索対象とする obj のスコープである．Java 言語では変数，関数，クラスのそれぞれに private 等のスコープがついているためである．C 言語ではグローバルやローカルな変数があるため表に示した値とは違うデータが入る．スコープの概念がない言語に対してはこの値は使われない．

また，GLOBAL フィルタでは大域変数の定義部と参照部を抽出していたが，EXT_NAME では宣言部のみを抽出している．しかし参照部については宣言部を抽出した後にRANGE フィルタを用いることによって抽出することが可能である．よって GLOBAL フィルタで抽出可能な情報は改良後においても損なわれることはない．

注目した関数の宣言部を抽出 (F_DECL)

ソースコードを読むとき，ある変数の宣言部の情報が必要にとなることがあるということは述べた．先行研究では変数についてのみ抽出の対象しているが，ある関数 a に注目している時，そこで呼び出されている関数 a の役割を知るためには a の内部で呼び出されている関数 b の情報が必要になる場合がある．そして呼び出されている関数 b の内部を見ることで注目している関数 a の動作を理解することができることがある．

このフィルタ (F_DECL) は以下の式で定義される．

$$\mathcal{F}_{f\_decl}(\mathrm{ast},\ \mathrm{bs},\ \mathrm{pos},\ \mathrm{method}) \tag{4.3}$$

このパラメタはそれぞれ，次のような意味を持っている．

Method method 注目しているメソッドの情報

ここで，AST と BlockSet のデータ構造は 3.4.1 で示したものと同じである．Method のデータ構造の定義を，ソースコード 4.2 に示す．

ソースコード 4.2: Method と Type のデータ定義

```
struct Method{
        String methodName;
        Type returnValue;
        Type[] arg;
}
```

Method とは関数の情報を格納するデータで，関数の名前を格納する methodName と，戻り値の型を格納する returnValue，さらに引数の型を格納する arg で定義されている．引数の数は arg の配列の要素数で分かる．

Type は型情報を表している．そのため，戻り値の型や引数の型を表すために Type を定義した．このフィルタの動作は，注目している関数呼び出しから，その関数を含んでいるオブジェクトの定義部を抽出し，そこで型情報を得る．得た型情報から目的の関数を定義している箇所を抽出する．

注目した関数が呼び出されている箇所を抽出 (REF)

ソースコードの流れを読読み取る際，ある関数に注目している場合に，その関数がどこから呼び出されているかを知りたいことがしばしばある．そのため注目している関数がどこで利用されているかを抽出することで，この労力を軽減することができると考えられる．

このフィルタ (REF) は以下の式で定義される．

$$\mathcal{F}_{Ref}(\mathrm{ast},\ \mathrm{bs},\ \mathrm{pos},\ \mathrm{method}) \tag{4.4}$$

このパラメタはそれぞれ，次のような意味を持っている．

Method method 注目しているメソッドの情報

ここで，AST と BlockSet，Method のデータ構造は 3.4.1 ， 4.2.2 で示したものと同じである．

抽象クラスやインタフェースの実装部を抽出 (IMPLEMENTED)

ソースコードから必要な情報を抽出する過程で，抽出された箇所がインタフェースあるいはアブストラクトクラスのように実装が行われていない箇所が抽出されることがある．その際に実装が行われている箇所の情報が必要になる．そのため，インタフェースやアブストラクトクラスの実装を行っている箇所を抽出するフィルタを作成した．

このフィルタ (IMPLE)MENTED は以下の式で定義される．

$$\mathcal{F}_{Imple}\mathrm{ast},\ \mathrm{bs},\ \mathrm{pos}) \tag{4.5}$$

ここで，AST と BlockSet，Pos のデータ構造は 3.4.1 で示したものと同じである．

4.2.3 フィルタ改良の結果

フィルタを改良した結果，フィルタの一覧は以下のようになった．

- 関心を決定するフィルタ
  - 変数と型の宣言部を抽出 (DECL)
  - 入力した文字列を含むクラス，関数，変数の宣言部を抽出 (EXT_NAME)
- 関心を広げるフィルタ
  - 制御構造の実行ブロックを抽出 (CTRL)
  - 注目行の近傍を表示 (NEIGHBOUR)
  - 代入文の依存を抽出 (TRACE)
  - 関数の宣言部を抽出 (F_DECL)
  - 関数が呼び出されている箇所を抽出 (REF)
  - 抽象クラスやインタフェースの実装部を抽出 (IMPLEMENTED)
- 関心を絞るフィルタ
  - 範囲を宣言部と実行部に分離 (SEP)
  - 変数の出現範囲を抽出 (RANGE)

4.3 言語依存とフレームワーク

今回改良，新規定義したフィルタはある程度言語依存については考慮に入れてある．しかし，データ定義について一部依存しているため改良したフィルタを他の言語に用いる際に対応させる際に考慮しなければならない。その例を以下に示す。

- C++のテンプレート
- クラス内クラスや関数内関数
- 継承

またフレームワークを実装するに当たり，可変部を分離してやる必要がある．主な可変部を以下に示す．

- C++のテンプレート
- Java，C#のインタフェース

- C の大域変数
- Java のクラス内クラスや関数内関数
- C,C++のポインタ

さらに、同様の機能を実現してあっても言語によっては仕様が異なるケースがある。したがって仕様の違いを考慮した設計を行った。その例として、関数の宣言部を抽出する F_DECL が挙げられる。C 言語では同名の関数が存在することを許していないが、Java や C++ではシグネチャが異なれば同盟の関数が存在することを許している (オーバーロード)。

このように同じ機能でも言語によって仕様が異なる。しかしユーザ側にはそれを気にすることなく利用できるよう設計する必要がある (図 4.1)。

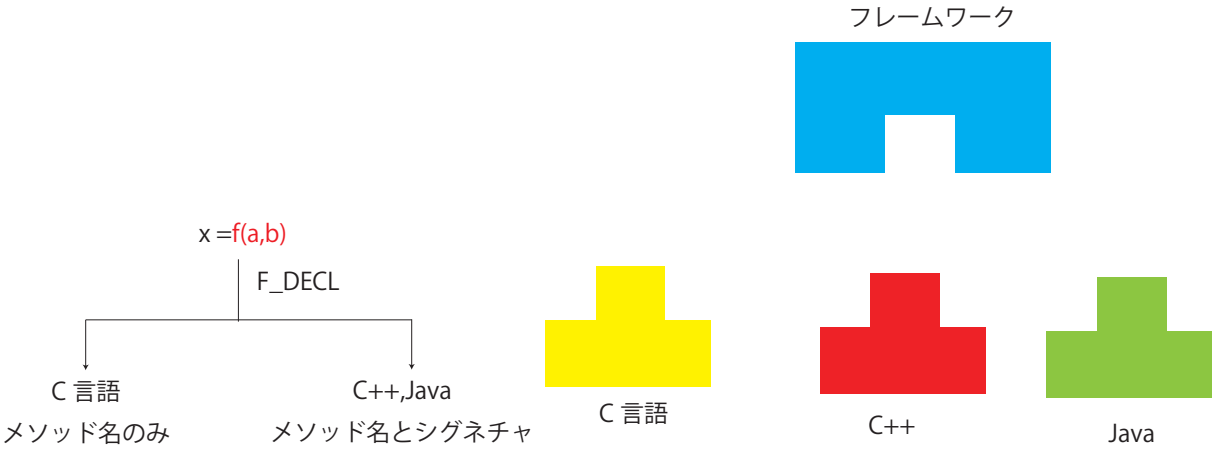

図 4.1: 言語依存性

第5章　評価

提案手法の有用性を確認するために実際のオープンソースソフトウェアを題材に評価実験を行った．本章では，その手順と結果について述べる．

5.1　実験の概要

本実験では機能変更を必要とする箇所を抽出する実験と，変更すべき箇所を全て抽出可能かどうかの実験の 2 つの実験を行った．対象としたソフトウェアを旧バージョンと新バージョンの 2 つのバージョンを用意し，新バージョンのリリースノートを参考に変更箇所を分析する．そして分析した変更箇所を旧バージョン時点の変更要求ととらえると，旧バージョンと新バージョンとの差分は変更箇所の解答としてとらえることができる．旧バージョンに対して提案手法を用いて情報を抽出し，抽出した情報が解答に相当するデータであれば，提案手法によって開発者が必要とする情報が抽出できたといえる．

なお，実験ではいくつかのフィルタを用いているが，その際のパラメタである bs,pos にはそれぞれファイル名の情報が含まれているが，抽出結果が単一クラス内のみの場合は注目しているファイル名についてはノード情報に記してあるため省略し，行数のみを示す．method に関しても，戻り値と引数の型についても同様に，メソッド名のみを示す．ナビゲーション木内の情報のフィルタのパラメタや注目点の位置についても，一部情報を省略している．ナビゲーション木のノードの色が変わっている箇所は注目しているソースファイルが異なっていることを表している．フィルタを適用した結果，関心がない場合，つまり null だった場合はノード内に null と表記している．フィルタを適用した結果が変更すべき箇所だとユーザが判断した場合はノードを表す円の中に印をつけている．

フィルタはまだ実装できていない．そのため grep 等の標準動作で実験した．

実験対象

本研究において用いた実験素材は Apach-log4j[5] である．Apach-log4j は Java を使ったシステムにログ機能を追加するためのクラスライブラリである．以下に示す．

- ソフトウェア名: Apach-log4j
- ソフトウェアのサイズ:8.6MB

- ソースファイル数:約 180 個
- ソフトウェアバージョン:後述

5.2 機能変更を必要とする箇所の抽出

ここでは，機能変更要求が与えられたとして，その機能を実現している箇所を変更箇所として抽出する実験を行った．まず変更要求の内容を述べ，その後実験の手順，実験の適用事例，最後に実験の結果得られたナビゲーション木を示す．この実験で用いた Apach-log4j のバージョンは旧バージョンとしたのは 1.2.14 で，新バージョンとしたのは 1.2.15 である．

5.2.1 変更要求

システムがメモリ不足を引き起こしている．stacktrace を見たら URL コネクションが閉じられていないというところまでは分かった．そのため一度開いた URL コネクションが開いたままで閉じられないのが原因でメモリ不足を起こすバグを修正する．

5.2.2 実験の手順

システムがリソース不足変更要求が URL コネクションが開いたままで閉じられないバグを修正すること．であるため，URL コネクションを開いているメソッドを抽出する．そして抽出した箇所が URL コネクションを開いたまま閉じていないのであればそこがバグの原因である可能性が高いといえる．以下のような手順で情報抽出操作を行った．

1. 全ソースコードから”URL”という文字列を含む名前 (メソッド，変数) を検索
2. 1 の結果全てに対して操作を行い，フィルタの
3. プロセスが停止した時点でそこが目的の箇所かどうかを調査
4. 目的の箇所ではなければ木を遡って別の関心箇所を定め，新しい木が生成されるまで続ける．

5.2.3 実験の詳細

まず，EXT_NAME 全ソースコードから”URL”という文字列を含むクラス，メソッド，変数を検索した．

$$\mathcal{F}_{EXT\_NAME}(\text{ast},\ \text{”URL”},\ \text{WHOLE},\ \text{WHOLE}) \tag{5.1}$$

その結果計 16 個のファイルから，大量に URL という文字列を含んでいるメソッド，変数を発見した．URL という文字列を含んでいるメソッド，変数を持っているファイルの一覧を表 5.1 に示す．

表 5.1: URL を含む文字列が出現しているファイルの一覧

LogManager.java
PropertyConfigurator.java
helpers/Loader.java
helpers/OptionConverter.java
helpers/SyslogWriter.java
jdbc/JDBCAppender.java
lf5/DefaultLF5Configurator.java
net/JMSAppender.java
spi/Configurator.java
varia/ReloadingPropertyConfigurator.java
xml/DOMConfigurator.java
lf5/util/Resource.java
lf5/util/ResourceUtils.java
lf5/viewer/LogBrokerMonitor.java
lf5/viewer/LogFactor5InputDialog.java
lf5/viewer/categoryexplorer/CategoryNodeRenderer.java
lf5/viewer/configure/MRUFileManager.java

まず検索結果の一番上に出てきた LogManager クラスに注目する．この LogManager クラスは，log4j の API 仕様書 [6] によると，Logger インスタンスを取得したり，LoggerRepository クラスの操作を行うとある．LogManager クラスの中で最初に URL という文字列を含んでいる 97 行目の

ソースコード 5.1: LogManager.java 中で変数 url を宣言している箇所

```
97 URL url = null;
```

という変数宣言に注目し，変数 url を関心の初期値とした．
そして変数 url の出現範囲を RANGE フィルタを用いて抽出した．

$$\mathcal{F}_{Range}(\text{ast},\ \{97-97\},\ 97,\ \text{"url"},2) \tag{5.2}$$

その結果ソースコード 5.2 のように 9 箇所で変数 url が使用されているのが分かった．

ソースコード 5.2: LogManager.java 中で変数 url を使っている箇所

```
97  URL url = null;

103 url = Loader.getResource(DEFAULT_XML_CONFIGURATION_FILE);
104 if(url == null) {
105 url = Loader.getResource(DEFAULT_CONFIGURATION_FILE);

109 url = new URL(configurationOptionStr);

113 url = Loader.getResource(configurationOptionStr);

120 if(url != null) {
121 LogLog.debug("Using URL ["+url+"] for automatic log4j configuration.");
122 OptionConverter.selectAndConfigure(url, configuratorClassName,
123         LogManager.getLoggerRepository());
```

この時点でのナビゲーション木を 5.1 に示す．この中から url が左辺に出現している箇所

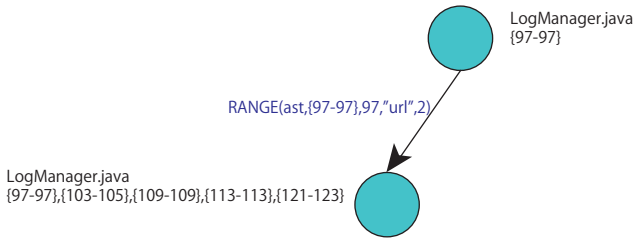

図 5.1: ナビゲーション木の生成過程 1

のみに注目すると，url の初期値を決定しているということが分かる．しかしこれは処理のフローを理解するのには重要ではないと考え，メソッド呼び出しの引数に変数 url を用いている，121 行目と 122-123 行目を注目する候補に絞った．

その中からまず，121 行目に注目し，LogLog.debug を宣言している箇所を F_DECL フィルタを用いて抽出した．

$$\mathcal{F}_{f\_decl}(\text{ast},\ \{121-123\},\ 121,\ \text{LogLog.debug}) \tag{5.3}$$

その結果ソースコード 5.3 のように LogLog クラスの 97 行目で宣言されていた．

ソースコード 5.3: LogLog クラスの debug メソッドの定義部

```
95  public
96  static
97  void debug(String msg) {
98    if(debugEnabled && !quietMode) {
99      System.out.println(PREFIX+msg);
100   }
101 }
```

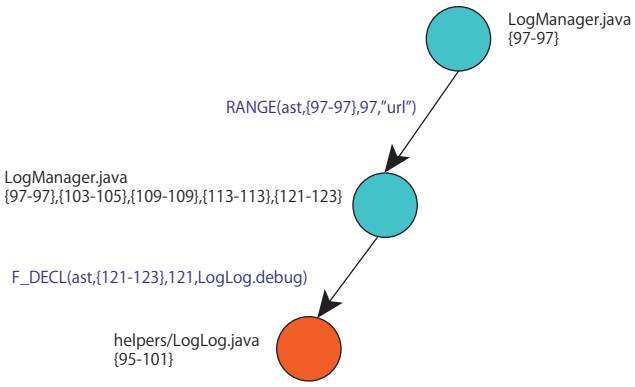

図 5.2: ナビゲーション木の生成過程 2

この時点でのナビゲーションツリーを図 5.2 に示す．ソースコード 5.3 から，この debug メソッドは入力された文字列を出力するだけで，今回の修正要求とは関係がないものであることが分かる．そのため今度は LogManager クラスの 122-123 行目 (5.2 に関心を移した．そして 122-123 行目のメソッド selectAndConfigure を宣言している箇所を F_DECL フィルタを用いて抽出した．

$$\mathcal{F}_{f\_decl}(\text{ast},\ \{121-123\},\ 122,\ \text{OptionConverter.selectAndConfigure}) \tag{5.4}$$

その結果ソースコード 5.4 のように OptionConverter クラスの 449 行目で宣言されていることが分かった．

ソースコード 5.4: OptionConverter クラスの selectAndConfigure メソッドの定義部

```
447 static
448 public
449 void selectAndConfigure(URL url, String clazz, LoggerRepository hierarchy)
      {
450   Configurator configurator = null;
451   String filename = url.getFile();
452
453   if(clazz == null && filename != null && filename.endsWith(".xml")) {
454     clazz = "org.apache.log4j.xml.DOMConfigurator";
455   }
456
457   if(clazz != null) {
458     LogLog.debug("Preferred configurator class: " + clazz);
459     configurator = (Configurator) instantiateByClassName(clazz,
460                                                 Configurator.class,
461                                                 null);
462     if(configurator == null) {
463         LogLog.error("Could not instantiate configurator ["+clazz+"].");
464         return;
```

```
465       }
466     } else {
467       configurator = new PropertyConfigurator();
468     }
469     configurator.doConfigure(url, hierarchy);
470   }
471 }
```

これまでの関心点はURL型の変数だったため，メソッドselectAndConfigureの引数である，URL型の変数urlに関心を移動し，変数urlに対してRANGEフィルタを用いて出現箇所を全て抽出した．

$$\mathcal{F}_{Range}(\text{ast},\ \{447-471\},\ 449,\ \text{"url"}, 2) \tag{5.5}$$

その結果ソースコード5.5のように451行目と470行目の2箇所で使用されていることが分かった．

ソースコード 5.5: selectAndConfigureメソッドの引数urlの出現箇所

```
447 static
448 public
449 void selectAndConfigure(URL url, String clazz, LoggerRepository hierarchy)
      {

451   String filename = url.getFile();

470   configurator.doConfigure(url, hierarchy);
```

抽出された2箇所のうち，451行目はString型の変数に変数url中の文字列を代入しているだけで，コネクションを開くといった動作は行っていない．そのため，今回の変更に関係はないと判断した．さらに，470行目はselectAndConfigureの引数であるurl，別のメソッド(Configuratorクラスのの doConfigure)の引数として用いている．そのため，今度はメソッドdoConfiguratorに関心を移動し，doConfigureを宣言している箇所をF_DECLフィルタを用いて抽出した．

$$\mathcal{F}_{f\_decl}(\text{ast},\ \{447-471\},\ 122,\ \text{OptionConverter.selectAndConfigure}) \tag{5.6}$$

その結果，メソッドdoConfigureの定義部を抽出したが，Configuratorクラスはインタフェースであるため中身は何もなかった．この時点でのナビゲーションツリーを5.3に示す．抽出した箇所がインタフェースやアブストラクトクラスであった場合はそれを実装している箇所を抽出したいが，その際の処理については検討していなかった．そのため，Configuratorクラスを実装しているクラスを探した．

その結果表5.2のように4つのクラスで実装されていることが分かった．

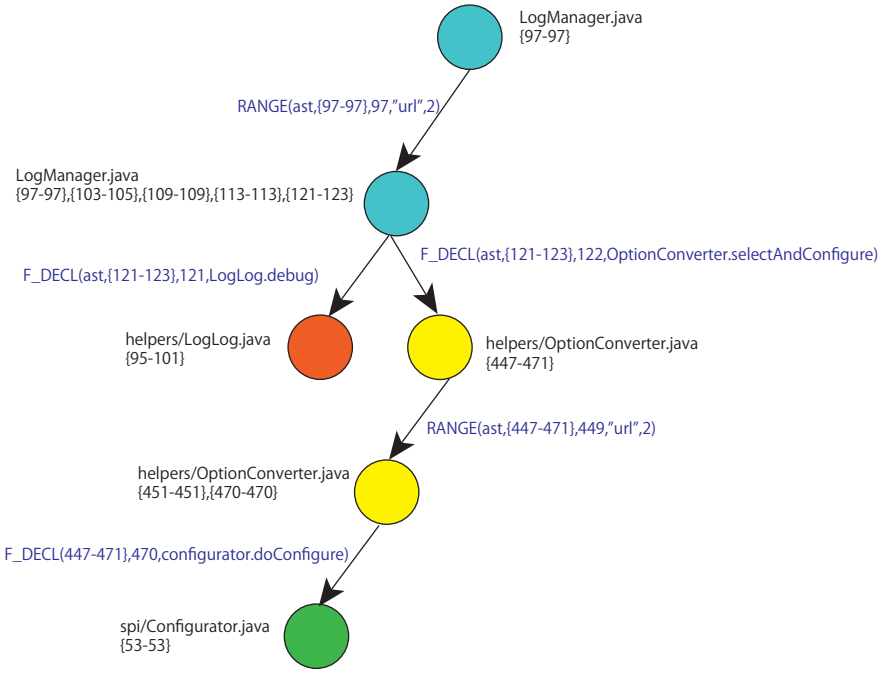

図 5.3: ナビゲーション木の生成過程 3

表 5.2: Configurator インタフェースを実装しているクラス一覧

PropertyConfigurator.java
lf5/DefaultLF5Configurator.java
varia/ReloadingPropertyConfigurator.java
xml/DOMConfigurator

この結果から，関心を PropertyConfigurator クラスのメソッド doConfigure に関心を移した．その中で，doConfigure の引数である configURL という変数に関心を移し，その出現箇所を RANGE フィルタを用いて抽出した．

$$\mathcal{F}_{Range}(\mathrm{ast},\ \{428-442\},\ 449,\ \text{"configURL"},\ 2) \tag{5.7}$$

その結果ソースコード 5.6 のように 431 行目，433 行目，436 行目，438 行目に出現していた．

ソースコード 5.6: PropertyConfigurator クラスの doConfigure メソッドの定義部

```
428 public
429 void doConfigure(java.net.URL configURL, LoggerRepository hierarchy) {

431   LogLog.debug("Reading configuration from URL " + configURL);

433     props.load(configURL.openStream());

436     LogLog.error("Could not read configuration file from URL [" + configURL
437    + "].", e);
438       LogLog.error("Ignoring configuration file [" + configURL +"].");
```

出現している箇所を順に見ていくと，431 行目で LogLog クラスの debug というメソッドに変数 configURL を引数として渡しているが，このメソッド debug は，既に見ていて，変更に関係がないことを確認しているため，関心を移す候補から除外した．そして 433 行目で configURL に対して URL 内のメソッド openStream というメソッドを呼び出している．呼び出されているメソッド名と言語の仕様書を見た結果，ここでコネクションを開いていることが分かった．

そのため，今度は関心をこのメソッド全体に移すために，CTRL フィルタを適用して，メソッド全体を抽出した．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{433-433\},\ 433,\ 2,\ \text{WHOLE}) \tag{5.8}$$

その結果ソースコード 5.7 で示す通り，このメソッド内でコネクションを閉じている様子がなかったため，ここがバグの原因と考えられる箇所である可能性が高いと考えられる．

ソースコード 5.7: PropertyConfigurator クラスの doConfigure メソッド全体

```
428   public
429   void doConfigure(java.net.URL configURL, LoggerRepository hierarchy) {
430     Properties props = new Properties();
431     LogLog.debug("Reading configuration from URL " + configURL);
432     try {
433       props.load(configURL.openStream());
434     }
435     catch (java.io.IOException e) {
436       LogLog.error("Could not read configuration file from URL [" +
              configURL
437                    + "].", e);
438       LogLog.error("Ignoring configuration file [" + configURL +"].");
439       return;
440     }
441     doConfigure(props, hierarchy);
442   }
```

バグの原因と考えられる箇所が抽出できたため，フィルタを適用したバージョンと次のバージョンの中で，PropertyConfigurator クラスの差分を取り，その違いを見てみると，新バージョンでは抽出されたメソッドが修正されていて，その変更内容も開いたコネクションを閉じるように修正されていた．

なお，PropertyConfigurator クラスの doConfigure メソッド内で，他にも変数 configURL を用いている箇所が抽出されていたが，それを見た結果，全て変更に影響を与えないものであった．

さらに，PropertyConfigurator クラス以外で Configurator インタフェースを実装している箇所があったが，そこを全て見た結果，これも全て URL コネクションを開く動作をしないものであった．

5.2.4 実験の結果

評価実験の結果をナビゲーション木に表したものを図 5.4 に示す．

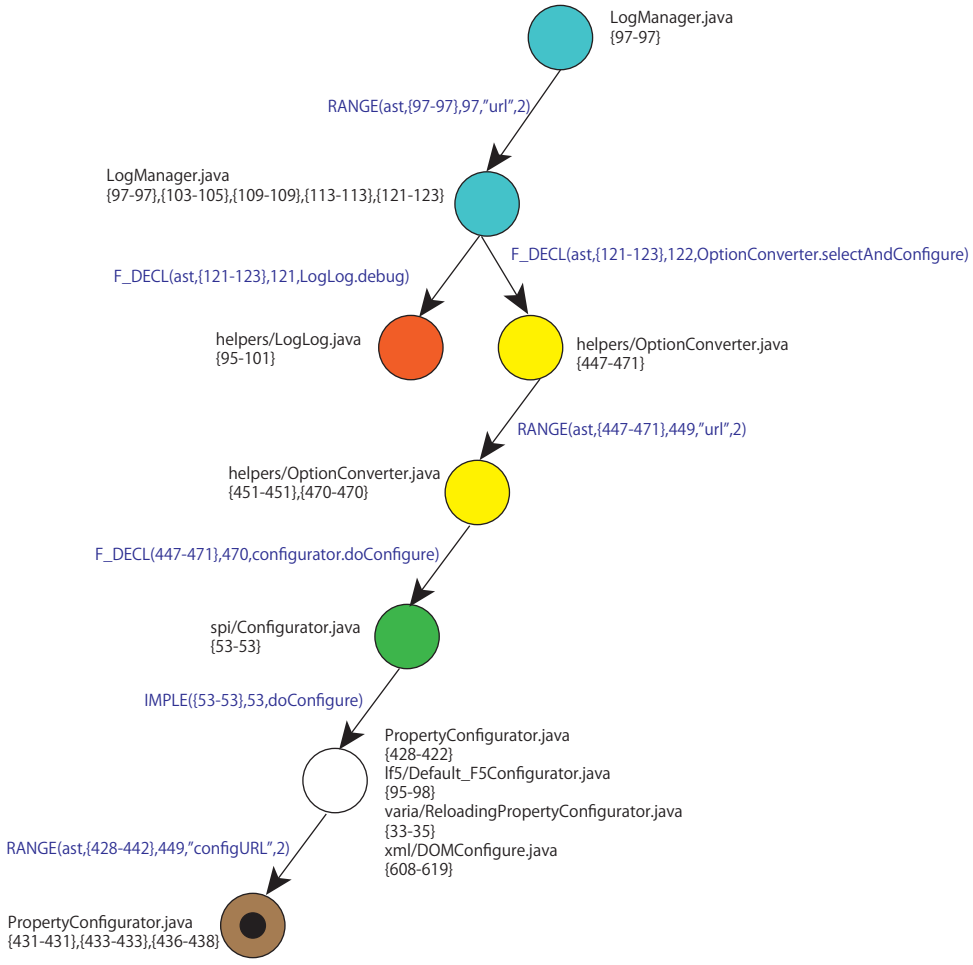

図 5.4: URL コネクションが閉じられないバグの変更箇所の抽出結果

5.3 変更すべき箇所を全て抽出可能かどうかの実験

ソフトウェアの構造を変更するソフトウェアの変更要求から，最初に変更する箇所を抽出し，変更したとしても，他の関数やクラスに影響を及ぼす可能性がある．そのためその変更に伴い，どこを変更すればよいかという情報を抽出する実験を行った．

この実験で用いた Apach-log4j のバージョンは 1.3.3 と 1.3.5 である．

その手順を以下に示す．

5.3.1 変更箇所

HashMap 限定だが他の Map でも適用できるようにするために抽象度を上げて Map に変更した．pattern/PatternParser.java 内の HashMap 型の変数の型を抽象度を上げて Map 型に変更した．なお，PatternParser クラスの概要は，PatternLayout クラスのほとんどのデータは PatternParser クラスに委譲され，このクラスで変換パターンを解析し，OptionConverter のチェインリストを作成する

5.3.2 操作の手順

まずはじめに最初に変更する箇所を見つける．そしてそのクラス内で，変更の影響を受け，戻り値の型や引数の数，型が変わる箇所を見つける．戻り値の引数の型が変わる箇所があれば

5.3.3 実験の詳細

型名 HashMap 型の変数を EXT_ まず，HashMap 型の変数を PatternParser クラス内で探索する．その結果ソースコード 5.8 で示すように 2 箇所で定義されていた．

ソースコード 5.8: pattern/PatternParser.java 中で HashMap 型の変数を定義している箇所

```
51  static HashMap globalRulesRegistry;

119 HashMap converterRegistry;
```

ソースコード 5.8 の中から，globalRulesRegistry を最初の関心に決め，その出現箇所を RANGE フィルタを用いて抽出した．

$$\mathcal{F}_{Range}(\text{ast},\ \{51-51\},\ 51,\ \text{"globalRulesRegistry"},2) \tag{5.9}$$

ソースコード 5.9 は一部を省略している．ソースコード 5.9 は注目点以外では 55 行目から 101 行目までの間の 30 箇所と 329 行目に 1 箇所出現していた．

ソースコード 5.9: pattern/PatternParser.java 中で globalRulesRegistry が出現している箇所

```
51  static HashMap globalRulesRegistry;

55    globalRulesRegistry = new HashMap(17);
56    globalRulesRegistry.put("c", LoggerPatternConverter.class.getName());
57    globalRulesRegistry.put("logger", LoggerPatternConverter.class.getName
          ());
58
59    globalRulesRegistry.put("C", ClassNamePatternConverter.class.getName());
60    globalRulesRegistry.put("class", ClassNamePatternConverter.class.getName
          ());
61
62    globalRulesRegistry.put("d", DatePatternConverter.class.getName());
63    globalRulesRegistry.put("date", DatePatternConverter.class.getName());

329 String r = (String) globalRulesRegistry.get(converterId);
```

55 行目から 101 行目までの間では変数の初期化を行っているため影響を与えない．もう一つは表に示したように 329 行目に代入文の右辺に出現していた．しかしながら，その行を見ると値を String にキャストして別の変数に代入している．そのためこの時点で変数の型の依存関係が存在しなくなるためこれ以上影響を与えない．

globalRulesRegistry では影響が確認できなかったため，次に converterRegistry に関心を定め，RANGE フィルタを用いて出現箇所を抽出した．

$$\mathcal{F}_{Range}(\text{ast},\ \{119-119\},\ 119,\ \text{"converterRegistry"},2) \tag{5.10}$$

ソースコード 5.10: pattern/PatternParser.java 中で converterRegistry が出現している箇所

```
321   if (converterRegistry != null) {
322     String r = (String) converterRegistry.get(converterId);

392 return converterRegistry;

399 this.converterRegistry = converterRegistry;
```

その結果はソースコード 5.10 に示したように，321 行目から 322 行目に 2 箇所，392 行目と 399 行目に 1 箇所ずつの計 4 箇所に出現していた．321 行目は if 文で converterRegistry が null でない場合という条件文であるため，この部分は影響を及ぼさないと判断し，無視した．次に 322 行目では globalRulesRegistry の 335 行目と同じく，値を持ってきて String 型にキャストして別の変数に代入しているため，影響を与えない．

次に 392 行目を見ると，converterRegistry を return している．今回の変更は converterRegistry の型を変更しているためこのメソッドを呼び出しているクラスには影響が出ると考えた．そのため，CTRL フィルタを用いて関心をこのメソッド全体に広げた．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{392-392\},\ 392,\ 1,\ \text{WHOLE}) \tag{5.11}$$

ソースコード 5.11: getConverterRegistry メソッド

```
391 public HashMap getConverterRegistry() {
392   return converterRegistry;
393 }
```

その結果このメソッドの戻り値はHashMap 型であったため，今回の変更には影響が出る．そのためこのメソッドを呼び出している箇所をREF フィルタを用いて抽出したが，どこからも呼び出されていなかった．

$$\mathcal{F}_{Ref}(\text{ast},\ \{391-393\},\ 391,\ \text{getConverterRegistry}) \tag{5.12}$$

この時点でのナビゲーション木を図 5.5 に示す．

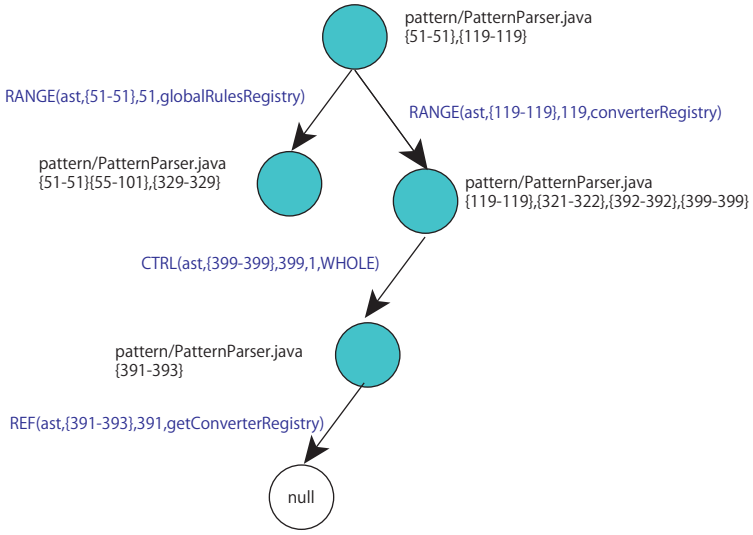

図 5.5: ナビゲーション木の生成過程 1

最後の出現箇所である，399 行目を見ると，converterRegistry に値を代入している．これも，このメソッドの引数の型が変更されることによって他のクラスにも影響を及ぼす可能性が高いため，CTRL フィルタを用いて関心をメソッド全体に広げた．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{399-399\},\ 399,\ 1,\ \text{WHOLE}) \tag{5.13}$$

ソースコード 5.12: setConverterRegistry メソッド

```
398 public void setConverterRegistry(HashMap converterRegistry) {
399   this.converterRegistry = converterRegistry;
400 }
```

その結果，ソースコード 5.12 で示すように引数として HashMap 型の変数を受け取っているため，このメソッドを呼び出しているクラスにも影響が出る．そのため，このメソッドを呼び出している箇所を REF フィルタを用いて抽出した．

$$\mathcal{F}_{Ref}(\text{ast},\ \{398-400\},\ 398,\ \text{setConverterRegistry}) \tag{5.14}$$

この時点でのナビゲーション木を図 5.6 に示す．その結果，PatternLayout.java の 505

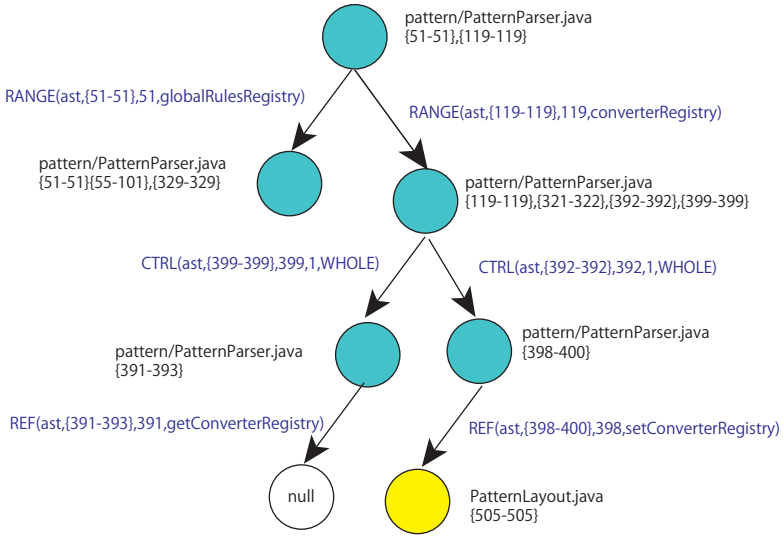

図 5.6: ナビゲーション木の生成過程 2

行目で呼び出されていることが分かった．この行のみではこのメソッドの動作が分からないのでソースコード 5.13 に示すように CTRL フィルタを用いて関心をこのメソッド全体に広げた．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{505-505\},\ 505,\ 1,\ \text{WHOLE}) \tag{5.15}$$

ソースコード 5.13: PatternLayout.java 内の activateOptions メソッド

```
503 public void activateOptions() {
504   PatternParser patternParser = new PatternParser(conversionPattern);
505   patternParser.setConverterRegistry(ruleRegistry);
506   head = patternParser.parse();
507   handlesExceptions = PatternConverter.chainHandlesThrowable(head);
508 }
```

ソースコード 5.13 に示したように，このメソッド自体は引数も戻り値もないため，このメソッドを呼び出すことによって直接他のクラスに影響を与えるということはない．しかし，505 行目で呼び出されている際に，ruleRegistry という値を引数として入れている．setConverterRegistry メソッドの引数は HashMap であるため，この ruleRegistry という変数も HashMap 型であるため ruleRegistry にも影響を与える．そのため今度は ruleRegistry という変数に関心を移動し，RANGE フィルタを用いて変数を追跡した．

$$\mathcal{F}_{Range}(\text{ast},\ \{503-508\},\ 505,\ \text{"ruleRegistry"}, 2) \tag{5.16}$$

ソースコード 5.14: 変数 ruleRegistry の出現箇所

```
439 private HashMap ruleRegistry = null;

469   if(ruleRegistry == null) {
470     ruleRegistry = new HashMap(5);
471
472   ruleRegistry.put(conversionWord, converterClass);

480 return ruleRegistry;

505 patternParser.setConverterRegistry(ruleRegistry);
```

その結果，ソースコード 5.14 に示したように，注目点以外では 439 行目と 469-472 行目と，480 行目で出現していることが分かった．そのうち，439 行目は変数宣言であった．今回の変更は変数の抽象度を上げているため，ここを HashMap から Map に変更する必要があると考えられる．

今度は ruleRegistry が出現している 469,470,472 行目に関心を移動した後，CTRL フィルタを用いてメソッド全体に関心を拡大した (ソースコード 5.15)．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{469-472\},\ 470,\ 2,\ \text{WHOLE}) \tag{5.17}$$

ソースコード 5.15: addConversionRule メソッド

```
468 public void addConversionRule(String conversionWord, String converterClass)
        {
469   if(ruleRegistry == null) {
470     ruleRegistry = new HashMap(5);
471   }
472   ruleRegistry.put(conversionWord, converterClass);
473 }
```

この時点でのナビゲーション木を図 5.7 に示す．469 行目は if 文で中身が null だった場合という条件文であったため無視した．470 行目では新たに HashMap を new しているが，ここを変更する必要はない．472 行目で，ruleRegistry に対して put をしている．しかし，

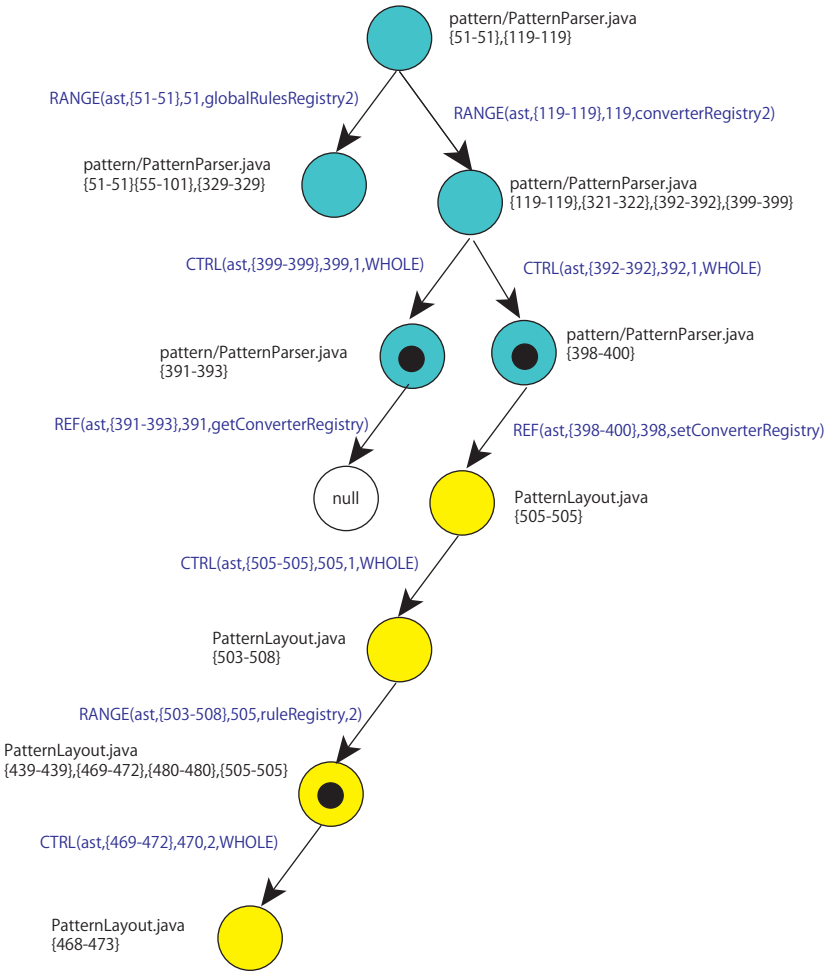

図 5.7: ナビゲーション木の生成過程 3

put する際の引数は String と String であり，なおかつこのメソッドには戻り値がないため，変更は不要であると考えられる．

次に，ソースコード 5.14 の 480 行目に注目し，CTRL フィルタを用いて関心をメソッド全体に広げた (ソースコード 5.16)．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{480-480\},\ 480,\ 1,\ \text{WHOLE}) \tag{5.18}$$

ソースコード 5.16: getRuleRegistry メソッド全体

```
479 public HashMap getRuleRegistry() {
480   return ruleRegistry;
481 }
```

ここでは HashMap 型の変数を戻り値としているため，このメソッドを読み出している箇所は変更が必要である．そのため，getRuleRegistry メソッドを呼び出している箇所を REF フィルタを用いて抽出したが，呼び出している箇所は存在しなかった．

$$\mathcal{F}_{Ref}(\text{ast},\ \{479-481\},\ 479,\ \text{getConverterRegistry}) \tag{5.19}$$

この時点でのナビゲーション木を図 5.8 に示す．これで ruleRegistry の変更に関しては全て抽出できたと考えられる．しかし，ソースコード 5.13 の 505 行目まで戻って見ると，

```
505 patternParser.setConverterRegistry(ruleRegistry);
506 head = patternParser.parse();
```

となっている．ここで patternParser に対する操作の戻り値を head に代入している．

今回の変更要求は PatternParser オブジェクト内の変数に対しての変更要求であるため，parse メソッドの動作にも影響がある可能性がある．

そのため，parse メソッドを定義している箇所を F_DECL フィルタを用いて抽出した．

ソースコード 5.17: PatternParser.java 内の parse メソッド

```
191 public PatternConverter parse() {
192   char c;
193   i = 0;
194
195   while (i < patternLength) {
196     c = pattern.charAt(i++);
197
198     switch (state) {
199     case LITERAL_STATE:
200
201       // In literal state, the last char is always a literal.
202       if (i == patternLength) {
203         currentLiteral.append(c);
```

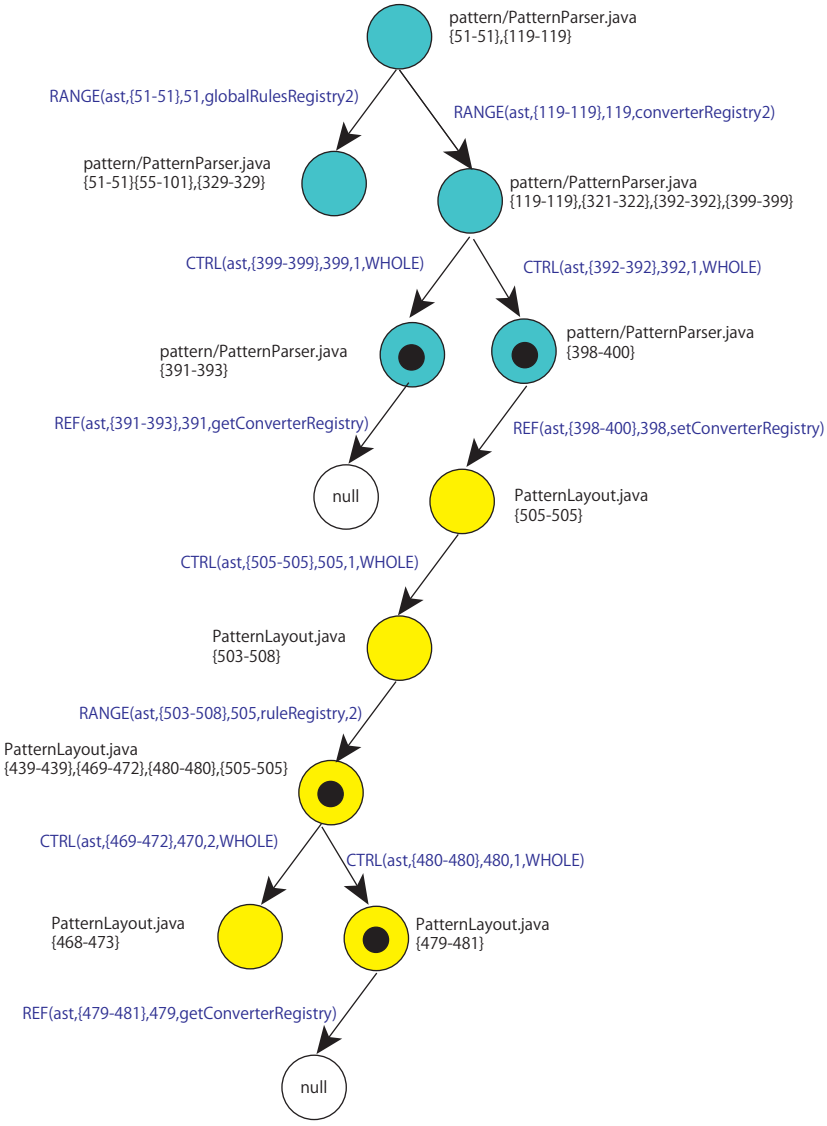

図 5.8: ナビゲーション木の生成過程 4

```
204
205         continue;
206       }
207
208       if (c == ESCAPE_CHAR) {
209         // peek at the next char.
210         switch (pattern.charAt(i)) {
211         case ESCAPE_CHAR:
212           currentLiteral.append(c);
213           i++; // move pointer
214
215           break;
216
217         default:
218
219           if (currentLiteral.length() != 0) {
220             addToList(
221               new LiteralPatternConverter(currentLiteral.toString()));
222
223             //LogLog.debug("Parsed LITERAL converter: \""
224             //           +currentLiteral+"\".");
225           }
226
227           currentLiteral.setLength(0);
228           currentLiteral.append(c); // append %
229           state = CONVERTER_STATE;
230           formattingInfo.reset();
231         }
232       } else {
233         currentLiteral.append(c);
234       }
235
236       break;
237
238     case CONVERTER_STATE:
239       currentLiteral.append(c);
240
241       switch (c) {
242       case '-':
243         formattingInfo.leftAlign = true;
244
245         break;
246
247       case '.':
248         state = DOT_STATE;
249
250         break;
251
252       default:
253
254         if ((c >= '0') && (c <= '9')) {
255           formattingInfo.min = c - '0';
256           state = MIN_STATE;
```

```
257         } else {
258           finalizeConverter(c);
259         }
260       } // switch
261
262       break;
263
264     case MIN_STATE:
265       currentLiteral.append(c);
266
267       if ((c >= '0') && (c <= '9')) {
268         formattingInfo.min = (formattingInfo.min * 10) + (c - '0');
269       } else if (c == '.') {
270         state = DOT_STATE;
271       } else {
272         finalizeConverter(c);
273       }
274
275       break;
276
277     case DOT_STATE:
278       currentLiteral.append(c);
279
280       if ((c >= '0') && (c <= '9')) {
281         formattingInfo.max = c - '0';
282         state = MAX_STATE;
283       } else {
284         logger.error(
285           "Error occured in position " + i
286           + ".\n Was expecting digit, instead got char \"" + c + "\".");
287         state = LITERAL_STATE;
288       }
289
290       break;
291
292     case MAX_STATE:
293       currentLiteral.append(c);
294
295       if ((c >= '0') && (c <= '9')) {
296         formattingInfo.max = (formattingInfo.max * 10) + (c - '0');
297       } else {
298         finalizeConverter(c);
299         state = LITERAL_STATE;
300       }
301
302       break;
303     } // switch
304   }
305
306   // while
307   if (currentLiteral.length() != 0) {
308     addToList(new LiteralPatternConverter(currentLiteral.toString()));
309
```

```
310         //LogLog.debug("Parsed LITERAL converter: \""+currentLiteral+"\".");
311     }
312
313     return head;
314 }
```

ソースコード 5.17 で示すメソッドを見て，return している箇所を探したところ，313 行目で head という変数を return していた．したがって，変数 head に対して代入が行われている箇所を抽出すれば今回の変更が parse メソッドの動作に影響を与えているかが判断できると考え，RANGE フィルタを用いて head が左辺に出現している箇所を抽出した．

$$\mathcal{F}_{Range}(\text{ast},\ 191-314,\ 313,\ \text{"head"},0) \tag{5.20}$$

その結果 131 行目のみが抽出された．これだけでは処理の内容が分からないため，CTRL フィルタを用いて 131 行目を含んでいるメソッドを抽出した．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{131-131\},\ 131,\ 2,\ \text{WHOLE}) \tag{5.21}$$

その結果 5.18 に示したように，メソッドの引数を head に代入していた．

ソースコード 5.18: PatternParser 内で変数 head が左辺に出現する箇所を含むメソッド全体

```
129 private void addToList(PatternConverter pc) {
130   if (head == null) {
131     head = tail = pc;
132   } else {
133     tail.next = pc;
134     tail = pc;
135   }
136 }
```

そのため，次はこのメソッドの引数がどの状況で決定するかを調べるため，REF フィルタを用いて addToList メソッドを参照している箇所を抽出した．

$$\mathcal{F}_{Ref}(\text{ast},\ \{129-136\},\ 129,\ \text{addToList}) \tag{5.22}$$

ソースコード 5.19: addToList を参照している箇所

```
220 addToList(
221                 new LiteralPatternConverter(currentLiteral.toString()));
```

```
308 addToList(new LiteralPatternConverter(currentLiteral.toString()));
```

```
379  addToList(pc);
```

その結果，ソースコード 5.19 に示すように 3 箇所から参照されていた．抽出結果のそれぞれの bs を見ると，220-221 と 308-308 は parse メソッド内の処理である．220-221 行目を見ると，String を引数として新しく LiteralPatternConverter オブジェクトを生成している．そのため，この行は今回の変更には何も影響を与えてないといえる．308 行目に関しても同様のことが言える．

次に 379 行目を見る．しかし，このままではどのような動作をするかが分からないため，CTRL フィルタを適用して関心をメソッド全体に広げた．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{379-379\},\ 379,\ 1,\ \text{WHOLE}) \tag{5.23}$$

ソースコード 5.20: addConverter メソッド全体

```
375  protected void addConverter(PatternConverter pc) {
376    currentLiteral.setLength(0);
377
378    // Add the pattern converter to the list.
379    addToList(pc);
380
381    // Next pattern is assumed to be a literal.
382    state = LITERAL_STATE;
383
384    // Reset formatting info
385    formattingInfo.reset();
386  }
```

ソースコード 5.20 を見ると，引数として渡されたものをそのまま代入している．そのため，この引数を追跡するために REF フィルタを用いて addConverter メソッドを参照している箇所を抽出した．

$$\mathcal{F}_{Ref}(\text{ast},\ \{375-386\},\ 375,\ \text{addConverter}) \tag{5.24}$$

その結果 372 行目からのみが抽出された．これだけでは処理の内容が分からないため，CTRL フィルタを用いて関心をメソッド全体に広げた．

$$\mathcal{F}_{Ctl}(\text{ast},\ \{372-372\},\ 372,\ 1,\ \text{WHOLE}) \tag{5.25}$$

ソースコード 5.21: finalizeConverter メソッド全体

```
341 protected void finalizeConverter(char c) {
342   PatternConverter pc = null;
343
344   String converterId = extractConverter(c);
345
346   //System.out.println("converter ID[" + converterId + "]");
347   //System.out.println("c is [" + c + "]");
348   String className = (String) findConverterClass(converterId);
349
350   //System.out.println("converter class [" + className + "]");
351
352   String option = extractOption();
353
354   //System.out.println("Option is [" + option + "]");
355   if (className != null) {
356     pc =
357       (PatternConverter) OptionConverter.instantiateByClassName(
358         className, PatternConverter.class, null);
359
360     // formattingInfo variable is an instance variable, occasionally reset
361     // and used over and over again
362     pc.setFormattingInfo(formattingInfo);
363     pc.setOption(option);
364     currentLiteral.setLength(0);
365   } else {
366     logger.error(
367         "Unexpected char [" + c + "] at position " + i
368       + " in conversion patterrn.");
369     pc = new LiteralPatternConverter(currentLiteral.toString());
370     currentLiteral.setLength(0);
371   }
372   addConverter(pc);
373 }
```

ソースコード 5.21 を見ると，342,356-358,369 行目で pc に値を代入している．そのうち 342 行目は右辺が null であるため今回の変更には関係がない．さらに，356-358 行目を見ると，className という String を元にして値を代入している．そのため今回の変更には関係がない．そして 369 行目については，ソースコード 5.19 の 220-221 行目と同じ処理をしているため変更には関係がない．ここで全てのプロセスが停止したため，変更の影響を受け，修正する必要のある箇所が特定できたと考えたため，抽出した結果を旧バージョンと新バージョンの差分と比較した．

その結果 PatternParser クラスに関しては変更した箇所と抽出した箇所が一致した．しかしながら，PatternLayout クラスに関しては，大幅な変更が加えられ，ソフトウェアの構造が変更されていた．PatternLayout クラスは旧バージョンではどのクラスとも継承関係はなかったが，新バージョンでは ComponentBase クラスを継承していた．そのため，ただ差分を比較しただけでは分からないため継承関係も含めて比較したところ，変更すべき箇所として抽出した箇所に関しては旧バージョンと新バージョンの PatternLayout クラ

スと ComponentBase クラスを比較してみると全て抽出できていた．
最後に完成したナビゲーション木を図 5.9 に示す．

5.4 実験のまとめ

バグ修正に対する変更箇所の抽出というケースに関しては提案手法は有効であるといえる．変更箇所のコアとなる部分を修正し，そこから波及する変更箇所の抽出というケースでは波及する変更すべき箇所は抽出できたが，ソフトウェアの構造の変更にまでは対応できていない．

実験を通して適用したフィルタを順に並べてみると，REF フィルタを用いた後は CTRL フィルタを常に用いていた．これは REF フィルタで抽出される範囲が 1 行だけであるが，それだけでは何をしているか理解できないためである．そのため，REF フィルタと CTRL フィルタを組み合わせて一つのフィルタのように扱うことが有効だといえる．

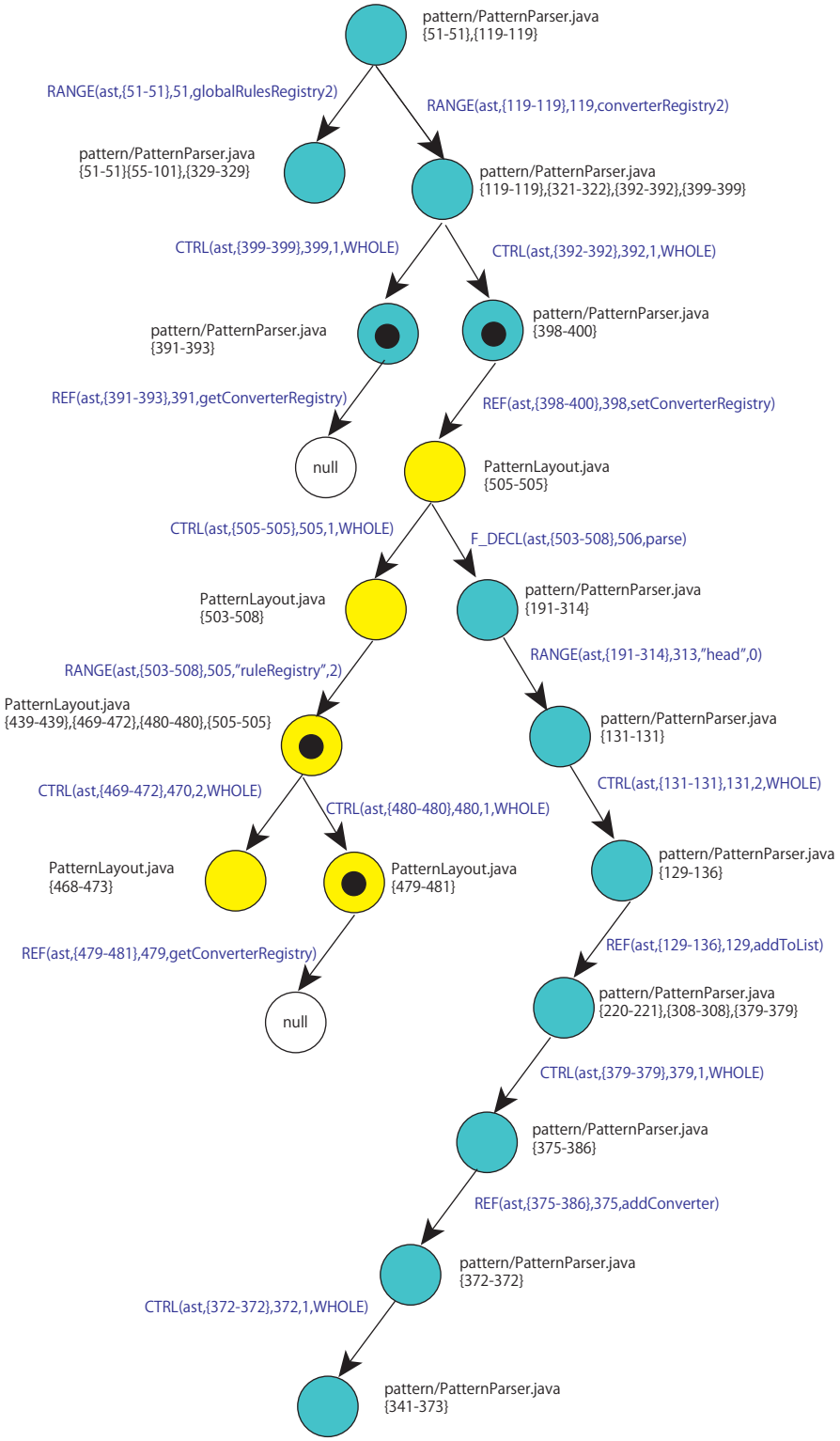

図 5.9: ナビゲーション木の生成過程

第6章　まとめ

6.1　まとめ

本研究ではソースコードの理解支援として、要求と関心とフィルタ適用の関係を示すためにナビゲーションを導入した。さらに、理解の際の補助としてナビゲーション木を導入した。また、7種類のフィルタを他の言語に対応させるために、1種類のフィルタの改良と新たなフィルタを4種類定義することによって先行研究において不十分であった点を補強した。そして多言語対応のために言語依存部を分離した。

6.2　今後の課題

本研究ではC，C++，Javaを対象にナビゲーションについて検討したが，インタフェース，テンプレートといった特定の言語固有の性質に十分に対応できていない．それに伴いフィルタを新たに追加することや，フィルタのパラメタを変更する必要があると考えられる．

そのため現状ではフレームワークの設計に不十分な点がある．また，フレームワークとして実装するためにどの言語に対しても共通に使用できる部分と言語毎の特性として独立している部分を明確に区別する必要がある．

さらに、recallやpresicionによって抽出結果の定量的な評価方法を確立する方法がある。

謝辞

本研究を行うにあたり，終始ご指導いただきました鈴木正人准教授に深く感謝申し上げます.

また本研究の審査員として大変有益な御助言をいただきました落水浩一郎教授，青木利晃准教授に心より感謝申し上げます.

また，研究生活全般にわたり，様々なサポートをしていただいた研究室の皆様に心より感謝申し上げます.

さらに，JAIST での生活において様々なサポートをしていただいた同期の友人たちに心より感謝申し上げます.

最後に，これまでの人生を支えていただき，さらに JAIST への進学を認め，様々なサポートをしていただいた両親や兄弟に心より感謝申し上げます.

参考文献

[1] SourceForge, SourceForge.net, http://sourceforge.net/

[2] Tama Communications Corporation, GNU GLOBAL source code tag system, http://www.gnu.org/software/global/

[3] R. Ian Bull, Andrew Trevors, Andrew J. Malton, and Michael W. Godfrey, Semantic Grep: Regular Expressions and Relational Abstraction, University of Wateloo Ontario, Canada, N2L 3G1, Proceedings of the Ninth Working Conference on Reverse Engineering(WCRE'02)

[4] 新倉諭, ソースコード理解支援機能を持つ開発環境の研究, 北陸先端科学技術大学院大学 情報科学研究科 修士論文, 2008.03.

[5] Apache Software Foundation, Apach-log4j, http://logging.apache.org/log4j/

[6] Apache Software Foundation, Apach-log4j API specification, http://www.jajakarta.org/log4j/jakarta-log4j-1.1.3/docs-ja/api/index.html

[7] Sun Microsystems, Java Platform Standard Edition 6 の API 仕様, http://java.sun.com/javase/ja/6/docs/ja/api/index.html