JAIST Repository
https://dspace.jaist.ac.jp/

Title	ソースコード理解支援機能を持つ開発環境の研究
Author(s)	新倉，諭
Citation	
Issue Date	2008-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/4321
Rights	
Description	Supervisor: 鈴木正人，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修　士　論　文

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
情報科学研究科情報システム学専攻

新倉 諭

2008 年 3 月

修 士 論 文

ソースコード理解支援機能を持つ開発環境の研究

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

410203 新倉 諭

提出年月: 2008 年 2 月

Copyright ⓒ 2008 by Satoshi Niikura

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、ソースコードに対してその理解支援が求められている。ここで理解支援とは、ソフトウェアの修正や変更等の要求に対して対象となる部分を抽出して提示する機能と定義する。理解支援に関する既存研究はいくつか存在するが、これらの多くは情報量の制御が行われていない、あるいは構造や意味を考慮した絞り込みを行うことができないなどの問題を抱えている。

本研究ではC言語を対象にし、多様な要求に対して必要な情報のみを開発者に提供する理解支援機能をもつツールを開発する。情報の抽出は抽象構文木を対象に定義した細粒度のフィルタによって行う。このフィルタを組み合わせることで、必要とする情報の抽出を可能としている。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

第1章　はじめに **1**
1.1　研究の背景 1
1.2　研究の目的 1
1.3　論文の構成 1

第2章　理解支援と関連研究 **3**
2.1　理解支援の必要性 3
2.2　既存ツール 4
2.2.1　GNU GLOBAL 4
2.2.2　C Cross Referencing and Documenting tool 6
2.2.3　Semantic Grep 7
2.3　理解支援ツールへの要求 8

第3章　研究のアプローチ **9**
3.1　ツールに必要な情報 9

第4章　ツールの設計 **11**
4.1　理解支援ツールの構成 11
4.2　フィルタによる構造の抽出 12
4.3　フィルタの合成 12

第5章　フィルタとパラメータ **14**
5.1　制御構造の実行ブロックを抽出するフィルタ 14
5.2　ブロックを定義部と実行部に分離するフィルタ 21
5.3　変数の出現範囲を抽出するフィルタ 22
5.4　代入文の要素に依存する式を追跡して抽出するフィルタ 25
5.5　注目部の近傍を抽出するフィルタ 26
5.6　大域変数の定義部と実行部を抽出するフィルタ 29
5.7　変数および型の定義部を抽出するフィルタ 29

第6章　実装と評価 **31**
6.1　実装 31

6.2 ツールによる実験 33
6.3 評価 37

第 7 章 終わりに 38
7.1 まとめ 38
7.2 今後の課題 38

図 目 次

2.1 チーム内に生じる知識の差 3
2.2 GNU GLOBAL による実行例 5
2.3 Cxref の実行例 6
2.4 Sgrep の実行例 7
2.5 本研究の目指すツール 8

3.1 単一のファイルの AST 9
3.2 ソースコード全体の AST 10

4.1 理解支援ツールの構成 11
4.2 単一のフィルタと OR によるフィルタの合成の結果 13

5.1 for 文の AST の構造 16
5.2 if 文の AST の構造 17
5.3 else if 節の AST の構造 17
5.4 while 文の AST の構造 18
5.5 do 文の AST の構造 18
5.6 switch 文の AST の構造 19
5.7 depth=1, des=FOR のパラメータによる抽出 20
5.8 pos>0, depth=2, des=FOR のパラメータによる抽出 20
5.9 pos=-1, depth=2, des=FOR のパラメータによる抽出 21
5.10 ブロックを定義部と実行部に分離するフィルタによる各抽出範囲 23
5.11 変数の出現範囲を抽出するフィルタによる抽出範囲 24
5.12 パラメータに渡された範囲に変数定義がない場合の抽出する AST ノード . . 25
5.13 表 5.4 の AST と、指定した代入文を追跡する様子 27
5.14 図 5.13 で代入文を構成する変数を追跡するための変数の意味木 27

6.1 ツールの実行画面 32
6.2 実験の流れ 33
6.3 ツールの抽出結果 1 34
6.4 ツールの抽出結果 2 34
6.5 ツールの抽出結果 3 34

6.6 ツールの抽出結果 4 34
6.7 ツールの抽出結果 5 35
6.8 ツールの抽出結果 6 35
6.9 ツールの抽出結果 7 36

1 ユースケース 42
2 フィルタ選択のアクティビティ図 43
3 フィルタ合成のアクティビティ図 44
4 ツール全体のクラス図 45
5 フィルタ生成のクラス図 46

表 目 次

5.1 制御構造の実行ブロックを抽出するフィルタのデータ定義 15
5.2 Part 型のデータ定義 22
5.3 同じ名前で全く別のローカル変数 24
5.4 hoge.c 28

6.1 フィルタ (6.2) に対応する差分 36
6.2 フィルタ (6.3) に対応する差分 36

第1章　はじめに

1.1　研究の背景

現在、ソフトウェア開発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を修正することで機能の追加が行われている。これはソースコードを再利用することで、ソフトウェアの開発を低コストで行えるためである。だがこの再利用によってソースコードの量は肥大化し、また構造も複雑化されているという問題が発生している。

しかし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あげられる。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のかは、ソースコードを読まなければわからない。しかし膨大なソースコードを全て読むことは困難であるため、ソースコードを理解する能力が求められてきている。そこで、そのような支援を行うための環境が必要とされてきている。

1.2　研究の目的

背景で述べたように、ソースコードの理解支援が求められてきている。ここで理解支援とは、ソフトウェアの修正や変更等の要求に対して対象となる部分を抽出して提示する機能と定義する。

本研究の目的は、開発者が既存の大規模なソースコードを効率的に理解できるようにするツールを開発することである。また理解支援の行える開発環境を構築することで、他人のソースコードに対する知識を引き継ぐ狙いがある。これによって開発の保守のコストの低減が期待できる。

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。既存ツールの問題点と、その解決策を提案する。

- 第 3 章では、本研究で理解支援ツールを提供することを述べる。さらにソースコードを解析した抽象構文木と、それをツールに取り入れることについて述べる。
- 第 4 章では、実際に作成した理解支援ツールの設計について述べる。
- 第 5 章では、ツールの中心である情報抽出のアルゴリズムについて詳しく述べる。
- 第 6 章では、実装したツールを用いて実際に実験を行い、その結果と考察について述べる。
- 第 7 章では、本研究のまとめを述べている。

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。

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。そしてそれらはソースコードを最初から書くことは少なく、多くの場合はすでにあるソフトウェアのソースコードを再利用して行われている。

ソフトウェアのソースコードは再利用されるため、開発チームはソフトウェアを作って開発終了になることはない。その後開発チームは、自分たちのソフトウェアに機能拡張やバグフィックスを行って運営していくことになる。そしてその過程でチームに新人を迎え入れることもある。

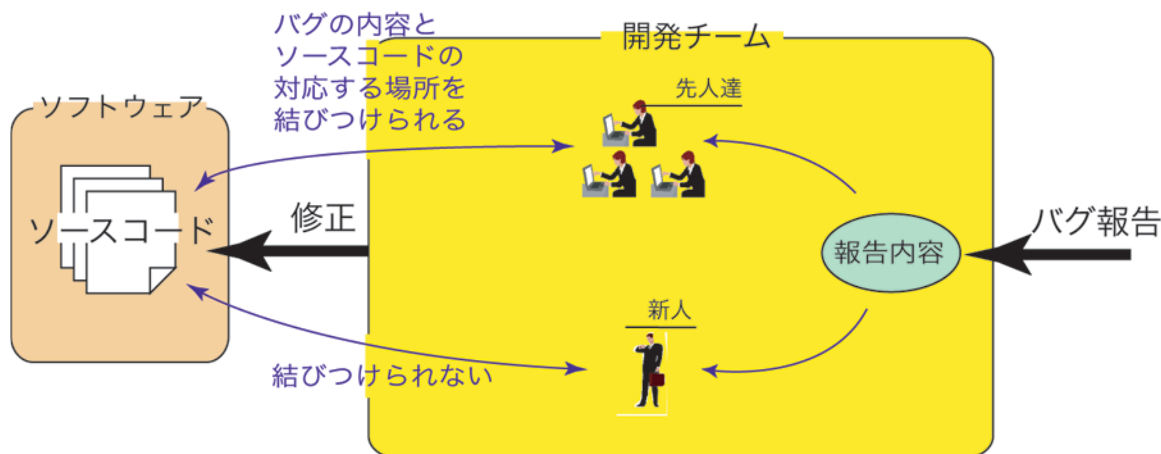


図 2.1: チーム内に生じる知識の差

そこで発生する問題として、次のような例を考える。

ユーザからはバグ報告や機能拡張の要求があり、それが開発チームへと届く。開発チームにはじめからいた人は、そのソースコードの構造を自分で設計しているため、修正箇所を経験的に予想できる。しかし新人にはそのような経緯がないため、最悪の場合、修正箇

所を特定するためにソースコードを全て読まなくてはならなくなる。これらの経緯から、この先人と新人の開発効率の差は明らかである。この流れを図 2.1 に示す。

よって、このような場合でも新人がスムーズに開発に加わることができるような理解支援を行うことが必要となってくる。

2.2 既存ツール

現在、理解支援として利用することのできる開発支援ツールがいくつかある。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可能なことより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。このツールのよいところは、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より見つけ、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こととなる。

図 2.2 は、あるオープンソース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であるが、その情報量を制御することはできない。またツール利用者がそのソースコードを初めて読むのであれば、ソースコードの注目点を決定する方法がないという問題が残る。

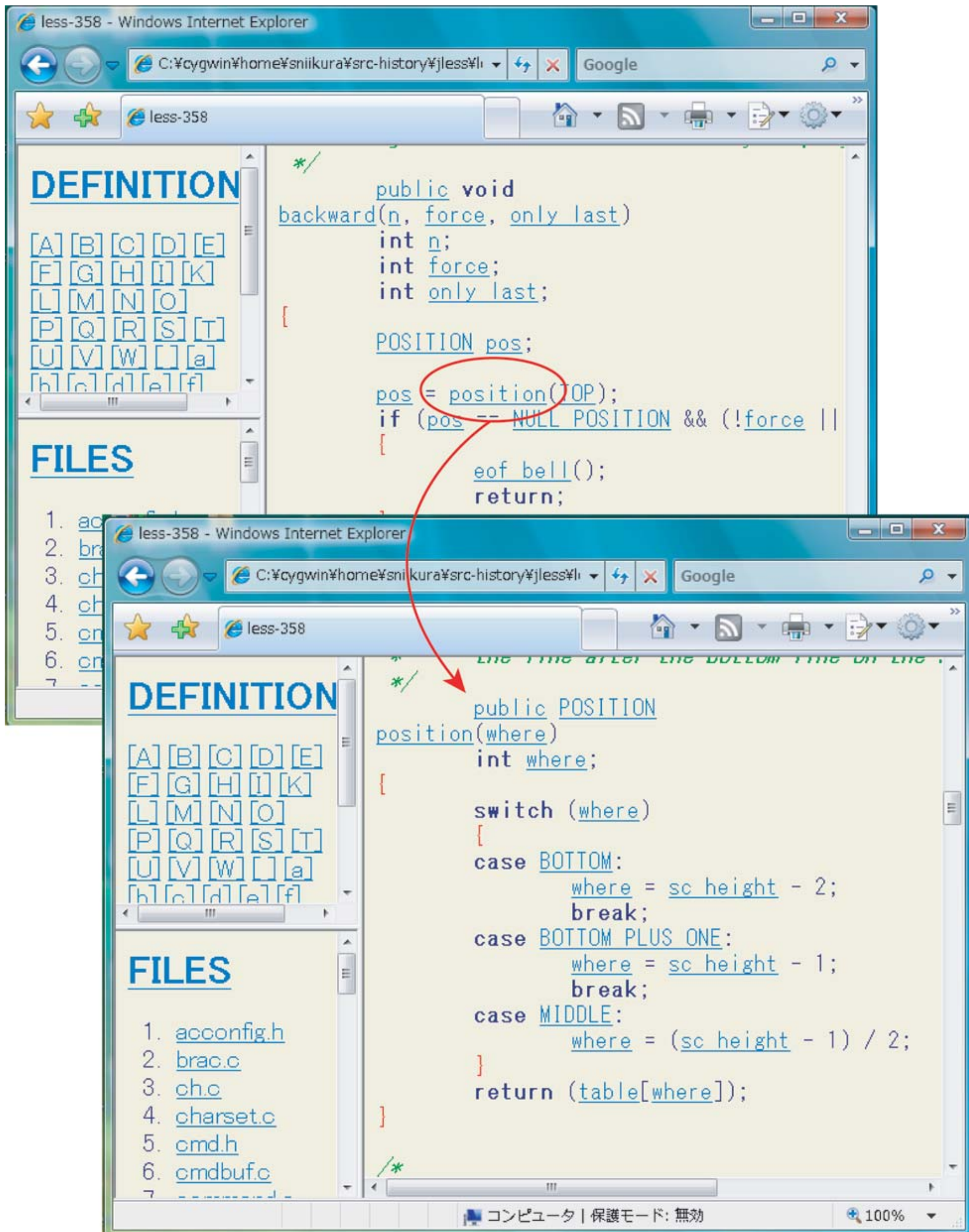


図 2.2: GNU GLOBAL による実行例

2.2.2 C Cross Referencing and Documenting tool

C Cross Referencing and Documenting tool[3](以下 Cxref) は C のソースコードを解析して、クロスリファレンス表を出力するツールである。全てのシンボルの定義場所および参照されている場所を行番号で表示することができる。ここでシンボルが定義されている箇所の場合は、アスタリスク (*) が付けられる。図 2.3 に、Cxref の実行例を示す。

```
[sniikura@lss2-is26] 65 % cat -n a.c
     1  void fx();
     2
     3  int main()
     4  {
     5          int i;
     6
     7          for(i=0; i<10; i++){
     8                  fx(i);
     9          }
    10          return 0;
    11  }
[sniikura@lss2-is26] 66 % cat -n b.c
     1  void fx(int a)
     2  {
     3          printf("%d\n", a^2);
     4  }
[sniikura@lss2-is26] 67 % cxref -c [ab].c
a.c:
b.c:
NAME            FILE          FUNCTION        LINE
__func__        b.c           fx                 2*
__func__        a.c           main               4*
a               b.c           fx                 1*    3
fx              a.c           ---                1-    8
                b.c           ---                2*
i               a.c           main               5*    7=    7     8     9=
main            a.c           ---                4*
printf          b.c           ---                3-    3
[sniikura@lss2-is26] 68 %
```

図 2.3: Cxref の実行例

しかし Cxref は、関数の定義参照関係およびその場所については明確に解析できるが、その内容まではわからない。変数や関数がどのような役割を持っているかは、そのソースコードに触れたことのある人にしかわからないため、関数名だけではこの問題は解決できないと言える。

2.2.3 Semantic Grep

Semantic Grep[4](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することが可能である。構造化されたテキストの例としては、SGML, HTML, Cソースコード, TeXやメールファイルがあげられる。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。

図2.4はソースコードfib.cに対してSgrepを適用した結果である。ここで入力されたコマンドは、
”コメントやプリプロセッサディレクティブに入っていないif”から”)”までを抽出
を意味する問い合わせである。そのため、/∗と∗/で囲まれた`if(x<0)`は抽出されず、また`else if`節の`else`句は出力には入らない。

```
[sniikura@lss2-is26] 81 % cat -n fib.c
     1  int fib(x)
     2  int x;
     3  {
     4  /*
     5          if(x<0){
     6                  printf("Error\n");
     7                  return -1;
     8          }
     9  */
    10          if(x<0){
    11                  return fib(x) * (x%2 ? 1 : -1);
    12          }
    13          if(x==0){
    14                  return 0;
    15          }else if(x==1){
    16                  return 1;
    17          }else{
    18                  return fib(x-1)+fib(x-2);
    19          }
    20  }
[sniikura@lss2-is26] 82 % sgrep '"if" not in ("/*" quote "*/" or ("\n#" .. "\n")) .. ("(" .. ")")' fib.c
if(x<0)if(x==0)if(x==1)
[sniikura@lss2-is26] 83 % █
```

図 2.4: Sgrepの実行例

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困難である。よって入力をより直観的な表現にする必要がある。

2.3 理解支援ツールへの要求

2.2 節で示したように、これらのツールは理解支援ツールとして使用するには不十分である。その理由として、

- 提示される情報が多すぎる
- 構造や意味に基づく絞り込みができない

などが挙げられる。

しかし、2.1 節で述べたようなケースは多く存在する。そのため理解支援ツールには、ソースコードの構造を知らない人でも目的の機能に関係のある場所を抽出できるような機能が必要となる。そこで本研究では、図 2.5 のような機能をもつツールの実装を目指すことにした。

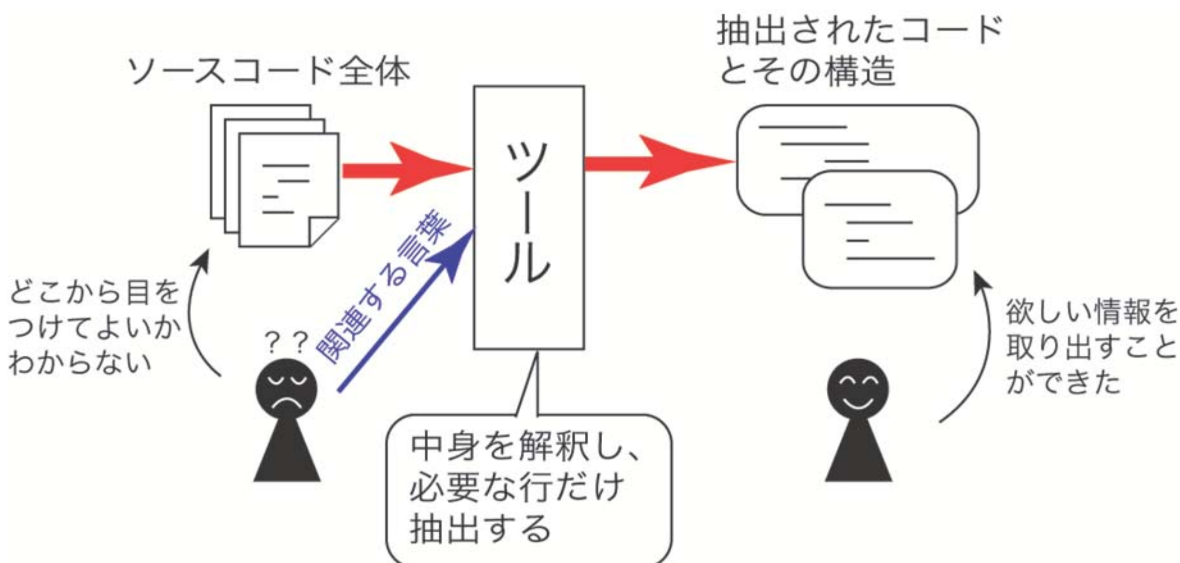


図 2.5: 本研究の目指すツール

第3章　研究のアプローチ

本章では、2.3 節で挙げた要求をもとに、本研究で作成する理解支援ツールの設計方針について述べる

3.1　ツールに必要な情報

図 2.5 の機能をもつツールを開発するために、本研究では永井 [1] の提案をもとにフィルタを利用したツールを作成することにした。ここでフィルタとは、ソースコードから構造化された情報を含む部分ソースコードを抜き出すための仕組みである。フィルタを用いることで、ソースコードから必要な情報を抜き出すことができるためである。

ソースコードは文字列の集合であり、そのままでは構造を抽出する上で扱いにくい。よってソースコードの構造を解析したデータが必要となる。そこで本研究では、ソースコードを構文解析した結果として得られる抽象構文木 (以下 AST [1]) を用いることにした。これは AST がソースコードの情報ごとにノードを作成して木構造をしているため、効率的に情報を抽出できるためである。単一のファイルから作成される AST は図 3.1 のような形をしている。

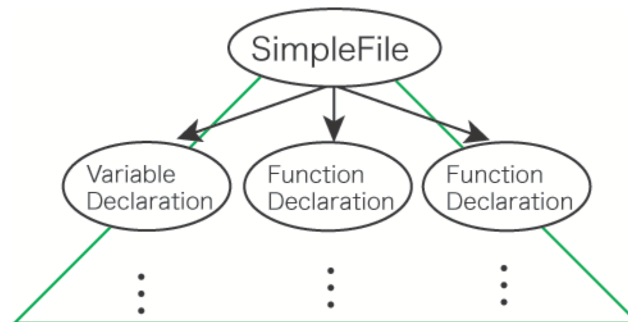


図 3.1: 単一のファイルの AST

またソースコードはたいていの場合、複数のファイルからできている。そこで各ファイルの AST の根を子ノードにもっている AST ノードを用意して、ソースコード全体で大きな 1 つの AST として扱うことにした。図 3.2 にこの AST を示す。ここで file1, file2, file3

[1]Abstract Syntax Tree

は、それぞれが図 3.1 のような AST の根であり、これらにはファイル固有の情報が与えられている。

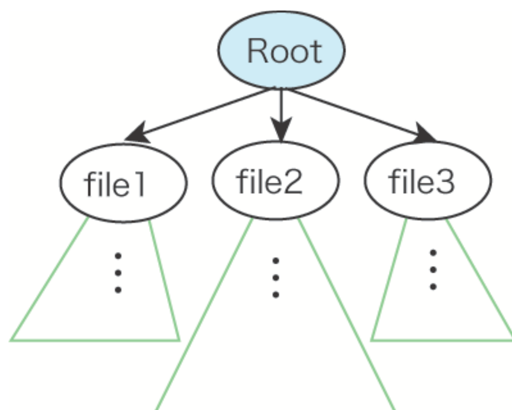


図 3.2: ソースコード全体の AST

第4章　ツールの設計

本章では、本研究で作成した理解支援ツールの設計についてまとめる。

4.1　理解支援ツールの構成

本研究で作成したツールの解析対象はC言語である。このC言語で書かれたソースコードに対して構文解析を行い、ASTを得る。ただし、本研究では解析器の開発は行っていない。これは解析器の開発がツールにとって必須ではあるものの、その作成に大きな労力がかかること、優良な解析ツールが存在すること、理解支援に着目した場合に解析器の開発が研究の本質でないことから、外部ツールの利用を考えたためである。

そしてソースコードに対して必要な情報を抽出できるよう、フィルタを定義した。実際にはASTのあるノードに注目し、フィルタの種類とパラメータを決定することで、目的とする役割を持つノードを抽出できるようになっている。またこれらのフィルタを合成することで、新たなフィルタを作成できるしくみになっている。このフィルタの選択と合成を繰り返し、必要な情報が得られるまで繰り返す。このようにして得たノードを、対応するソースコードに変換し出力することで、構造を持った部分ソースコードを出力できるように設計した。

この一連の流れの外観を、図??に示す。

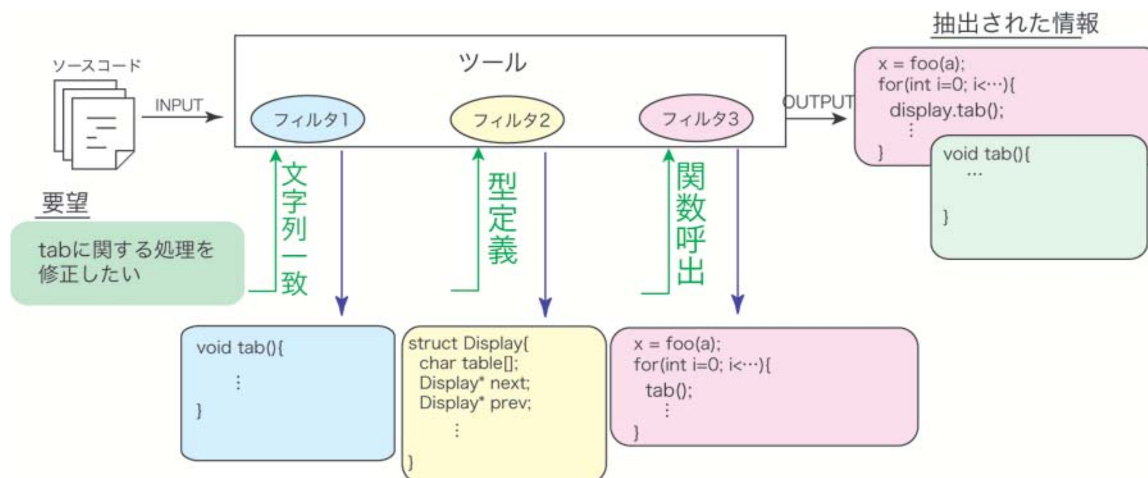


図 4.1: 理解支援ツールの構成

4.2 フィルタによる構造の抽出

4.1 節で述べたとおり、このツールは AST に対するいくつかのフィルタにパラメータを与えることで、必要な情報を含む構造を抽出する仕組みになっている。本研究で定義・作成したフィルタは以下の通りである。

1. 制御構造の実行ブロックを抽出するフィルタ
2. ブロックを定義部と実行部に分離するフィルタ
3. 変数の出現範囲を抽出するフィルタ
4. 代入文の要素に依存する式を追跡して抽出するフィルタ
5. 注目部の近傍を抽出するフィルタ
6. 大域変数の定義部と実行部を抽出するフィルタ
7. 変数および型の定義部を抽出するフィルタ

例えば、1. 制御構造の実行ブロックを抽出するフィルタは、以下のように形式的に定義できる。

$$\mathcal{F}_{Ctl}(\mathrm{AST\ ast,\ BlockSet\ bs,\ int\ pos,\ int\ depth,\ Descripter\ des})$$

ここで式に与えている値 ast, bs, pos, depth, des を、このフィルタのパラメータと呼ぶ。これらのフィルタの詳しい説明は 5 章で述べることにする。

これら各フィルタに対してフィルタ固有のパラメータを与えることで、情報の抽出を行えるようにした。これは基本的なフィルタのみを作成して、その挙動をパラメータによって制御することで、多くの要求に対応できるようにするためである。

4.3 フィルタの合成

フィルタの合成とは、これら個々のフィルタの出力に対して AND や OR といった合成を行うことと定義する。図 4.2 に単純なフィルタと合成したフィルタの選択の様子を示す。また個々のフィルタあるいは合成したフィルタの出力を、別のフィルタの入力として使用できるようにして、絞り込み等の抽出が行えるようにようにした。ただし、個々のフィルタの入出力のフォーマットは統一することができるが、フィルタ内部の抽出アルゴリズムは独立しているため、同じフィルタ、同じパラメータであってもフィルタを選択する順番によって異なった結果が得られる。よって単純な絞り込みではない。

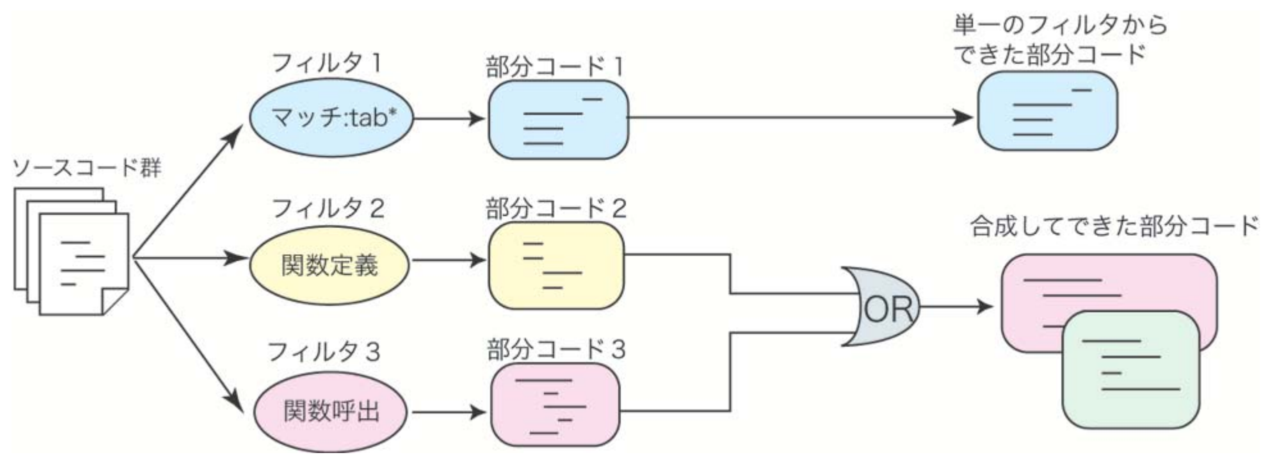


図 4.2: 単一のフィルタと OR によるフィルタの合成の結果

第5章　フィルタとパラメータ

4 章で述べたように、この理解支援ツールはフィルタによる情報の抽出によって作られている。本章では、4.2 節で紹介した各フィルタの定義と、そのフィルタによる抽出の具体的なアルゴリズムについて述べることとする。

5.1　制御構造の実行ブロックを抽出するフィルタ

C 言語のソースコードの実行単位は Statement である。そして 0 個以上の Statement 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。C 言語において、制御文による制御対象は 1 つの Statement 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内容を抽出することが目的である。

本研究では、このフィルタに対して以下の式を定義した。

$$\mathcal{F}_{Ctl}(\text{AST ast},\ \text{BlockSet bs},\ \text{int pos},\ \text{int depth},\ \text{Descripter des}) \tag{5.1}$$

このパラメータはそれぞれ、次のような意味を持っている。

AST ast ソースコード全体

BlockSet bs フィルタの適用範囲

int pos 注目している行の番号

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

SourceCode, BlockSet, Descripter のデータ構造の定義を、擬似コードで表 5.1 に示す。なおここで定義したデータ型の Block とはソースコード上の物理的な行の範囲であり、ソースコードの制御ブロックとは独立したデータである。

ソースコード全体は常にフィルタのパラメータとして用いる値であり、この値は不変である。

フィルタの適用範囲は、出力に不要な部分をマスクするために存在するパラメータである。単一のフィルタを利用する場合、大抵はソースコード全体、あるいはファイル全体か

表 5.1: 制御構造の実行ブロックを抽出するフィルタのデータ定義

```
struct AST{
    AST NODE;
    AST *parent;
    AST **child;
    int childNumber;
}

struct BlockSet{
    Block *block;
}

struct Block{
    int fileId;
    int beginLine;
    int endLine;
}

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

関数全体であるべきであり、このパラメータはあまり意味を持たない。しかし別のフィルタの出力を利用した絞り込みをするためには必須といえるパラメータである。

ソースコードを読むときに、大抵はある1つのStatementに注目することとなる。そこでそのStatementのある行番号をパラメータとしてフィルタに与えることにした。そのStatement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Statementが複数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するべき時と、内側のループに注目するべき時と、それぞれの要求があり得る。よってこれをパラメータとして与えることにした。

C言語における制御構造は**for, if, while, do, switch**の5種類である(ただし#ifdefなどのPreprocessorは考えないものとする)。これらは1つ以上の制御対象ブロックを持つこと以外、ほぼ独立した作りになっている。よってどの制御構造に着目するかをパラメータとして与えることにした。このフィルタによる抽出アルゴリズムについて、以下に示す。

1. for文

for文をASTにしたときの一般形は、図5.1のようになる。

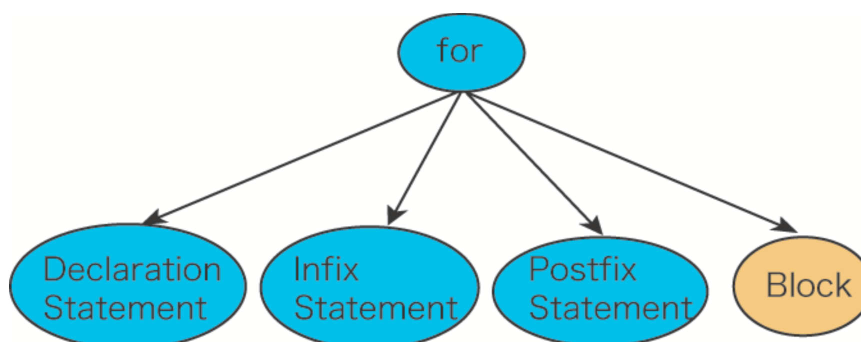


図5.1: for文のASTの構造

ここで各ノードは、DeclarationStatementが初期化式、InfixExpressionが反復条件、PostfixExpressionがインクリメント、Blockが反復対象となる実行ブロックを示している。ただし、DeclarationStatementとInfixExpression、およびPostfixExpressionは省略可能であり、そのとき対応するノードが存在しない構造になる。しかし実行ブロックは{ }を用いない単一のStatementとなることもあるが、必ず唯一つだけ存在する(セミコロン(;)のみのStatementでもEmptyStatementというノードが生成される)。

よってforを示すノードの子ノードのうち、一番最後のノードを抽出することで、実行ブロックを抽出できる。これはASTが構文解析の結果であり、ノードの出現する順番に意味を持つためである。

2. if 文

if 文を AST にしたときの一般形は、図 5.2 のようになる。

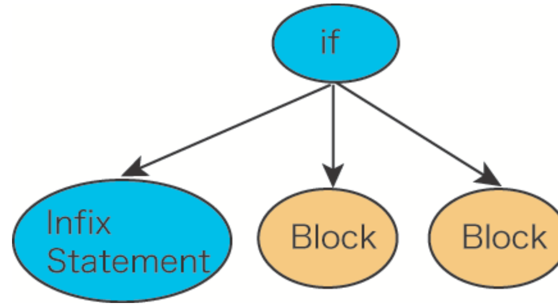


図 5.2: if 文の AST の構造

ここで InfixExpression はブロックの実行条件、真ん中の Block は条件式が true のときに実行されるブロック、最後の Block は条件式が false のときに実行されるブロックを示している。for 文とは異なり、InfixStatement のノードが省略されることはない。しかし 2 つの Block ノードのうち最後のノードは、ソースコードに else 節が存在する場合に限って存在する。真ん中の Block ノードは必ず存在するので、if ノードの子ノードの数が 2 個か 3 個かで決定することができ、2 番目と 3 番目が抽出対象となる。

三項演算子は Statement として処理されるとして考え、抽出対象としない。また else if 節に関しては、図 5.3 のような AST となり、同様に処理することができる。ただしこのとき探索深さの問題が発生する。探索深さについては後述する。

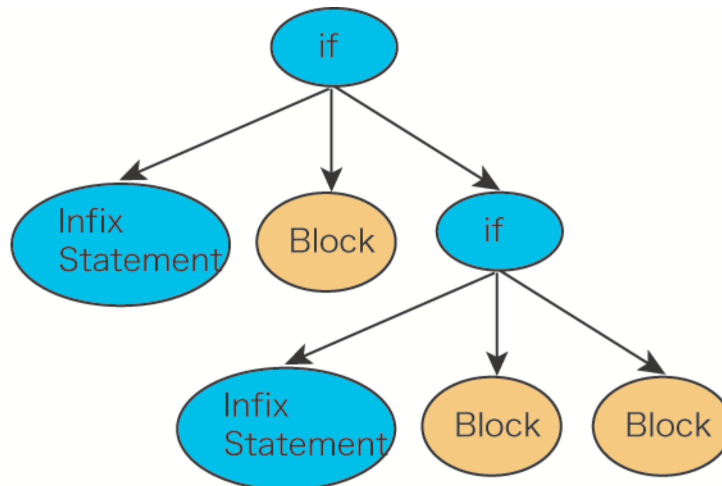


図 5.3: else if 節の AST の構造

3. while 文

while 文を AST にしたときの一般形は、図 5.4 のようになる。

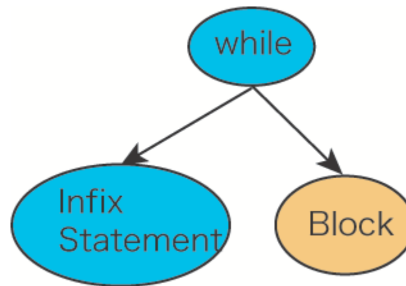


図 5.4: while 文の AST の構造

ここで InfixStatement は、実行ブロックの反復条件を示している。抽出対象は Block であり、これら 2 つのノードは必ず唯一つずつ存在する。よって 2 番目のノードを選択することで、実行ブロックを抽出できる。

4. do 文

do 文を AST にしたときの一般形は、図 5.5 のようになる。

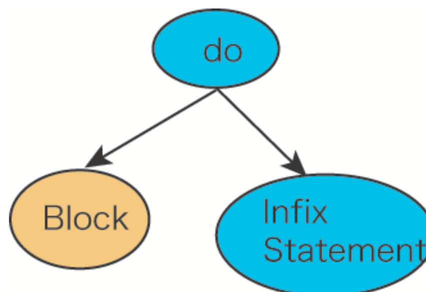


図 5.5: do 文の AST の構造

do 文は通常、while 節と併用して利用される。C 言語では while 節が省略不可能であり、この InfixStatement のノードは必ず存在する。よってこの AST も固定であり、実行ブロックの抽出は最初のノードということになる。

本研究で作成したフィルタでは、do 文と while 文は別の制御文としてとらえて作成している。C 言語だけならば while 制御構造とまとめることはできるが、他言語への拡張性を考えて分けることとした。

5. switch 文

switch 文を AST にしたときの一般形は、図 5.6 のようになる。

ここで SimpleName ノードは switch 文によって評価される値である。このノードと Block のノードは両方とも省略は不可能であり、構造は固定である。

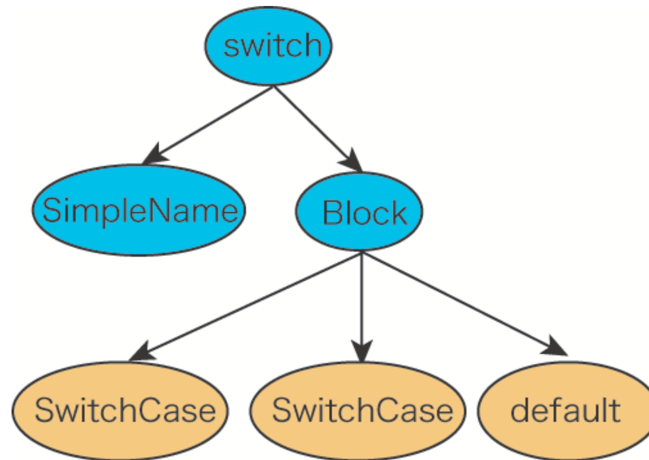


図 5.6: switch 文の AST の構造

しかしながらこの構造に限って、Block ノードが抽出対象にはならない。SwitchCase の開始行は case ノードであるが、終了行は BreakStatement, ReturnStatement, GotoStatement のいずれかが出現するまでであり、Block の範囲が重なりうるためである。

よって case 数 (+1) 個の範囲について、ノードの集合で抽出することとした。

ところで、これら制御ブロックは混在することが多い。単純な for のみの 2 重ループ等なら上記の入れ子の定義で十分であるが、for 文の中に if 文が入ることは典型的な例である。図 5.7 にその AST の例を示す。

本ツールに実装したフィルタで、図の通り if 文の中の Statement に注目して for 文の実行ブロックを抽出するためには、

depth=1, des=FOR

をパラメータとして与えることとする。すなわち for 文の入れ子を数えるときには、if 文の制御構造を無視するように設計した。

ただし純粋な入れ子の数を調べたい場合も存在するため、パラメータに **WHOLE** というパラメータを渡せるようにした。このパラメータを与えた場合、あらゆる制御構造の実行ブロックに加えて、単純な { } で囲まれたブロックも抽出対象となる。

ところで、1 つの Statement に注目することを前提として上記のフィルタの設計を行ったが、ブロック全体に注目して構造を抽出したいケースも存在する。この場合、葉ノードでなく root に近い Block ノードに注目して、その子ノードに対する探索となる。そのような抽出を行うためには、さらにパラメータを増やす必要があるが、このとき注目行をパラメータに与える必要がなくなる。そこでパラメータに **pos=-1** を与えることで、入力された範囲に対する探索を行うようにした。**pos>0** と **pos=-1** の比較について、図 5.8 と図 5.9 に示す。

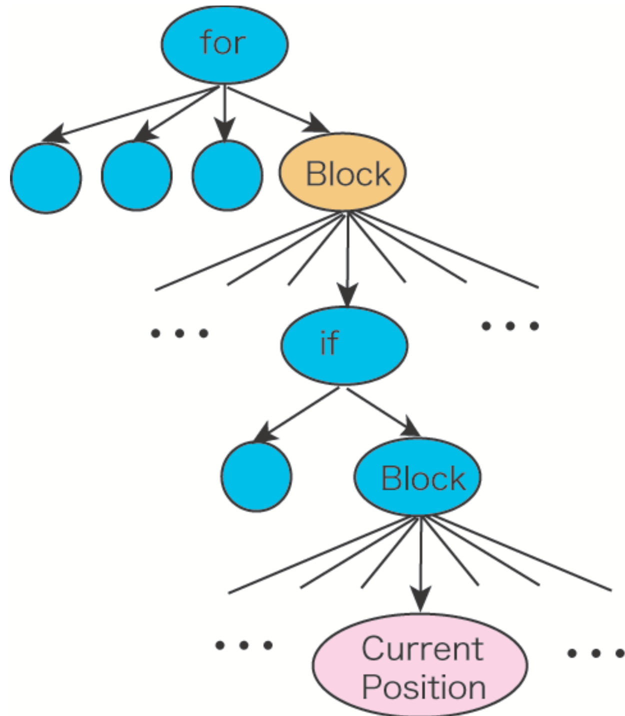


図 5.7: depth=1, des=FOR のパラメータによる抽出

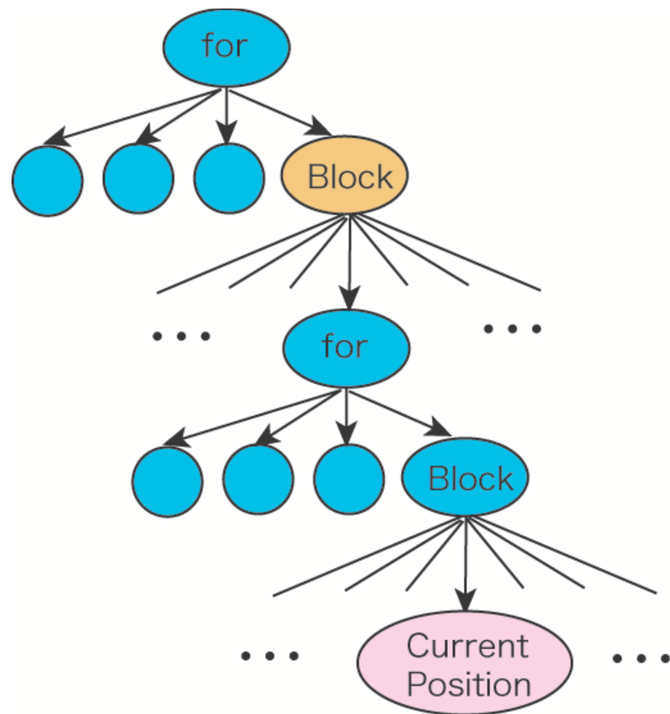


図 5.8: pos>0, depth=2, des=FOR のパラメータによる抽出

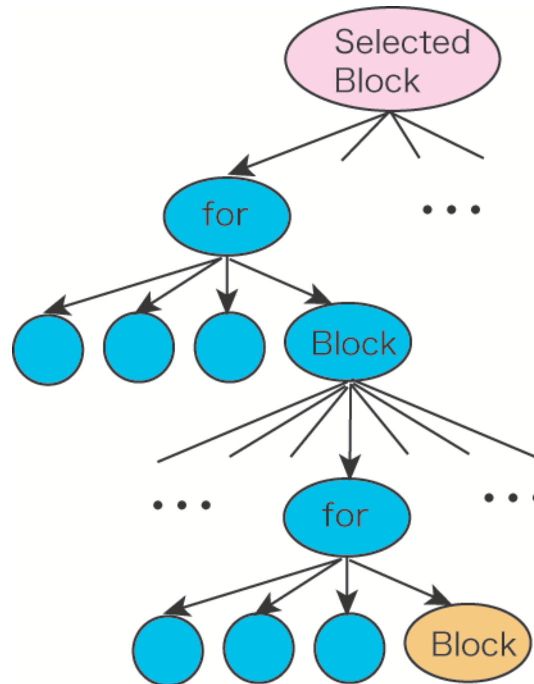


図 5.9: pos=-1, depth=2, des=FOR のパラメータによる抽出

この機能の実装により、ある関数の中で”for 文の中にある if 文のみに注目して実行ブロックを抽出する”といった要求に対して、

1: $\mathcal{F}_{Ctl}(S,\ B_f,\ -1,\ 1,\ FOR) = \mathcal{BS}_1;$
2: $\mathcal{F}_{Ctl}(S,\ \mathcal{BS}_1,\ -1,\ 1,\ IF) = \mathcal{BS}_2;$

のように、フィルタを組み合わせることで柔軟に対応できる (ただし S はソースコード全体、B_f は該当関数の開始行から終了行の範囲とする。)。

5.2 ブロックを定義部と実行部に分離するフィルタ

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り、ブロックの中で閉じている (Preprocessor を除く)。またブロック中の各 Statement は、変数定義のための DeclarationStatement か、代入や関数呼び出しなどを行う ExpressionStatement に大きく分けられる。そこで定義部と実行部を分離して、その一方を抽出するフィルタを設計した。これによってブロック変数の特定が行えることを目的としている。

本研究では、このフィルタに対して以下の式を定義した。

$$\mathcal{F}_{Sep}(\text{AST ast},\ \text{BlockSet bs},\ \text{int pos},\ \text{int depth},\ \text{Part part}) \tag{5.2}$$

このパラメータはそれぞれ、次のような意味を持っている。

AST ast ソースコード全体

BlockSet bs フィルタの適用範囲

int pos 注目している行の番号

int depth ブロックの入れ子の深さ

Part part 抽出対象: 定義部または実行部

ここで、AST と BlockSet のデータ構造は、$\mathcal{F}_{Ctl}$ と同様である。Part のデータ構造の定義を、擬似コードで表 5.2 に示す。

表 5.2: Part 型のデータ定義

```
enum Part{
  DECL, EXEC;
}
```

このフィルタは制御文の実行ブロックを抽出するフィルタとは異なり、全てのブロックに対して depth のカウントを行う。これは特定の制御ブロックに注目したい場合に、制御文の実行ブロックを抽出するフィルタと組み合わせることで可能であるからである。

さらにこのフィルタでは、あるブロックに着目した場合に、そのサブブロックを 1 つの実行部として扱う。これもサブブロックについて調べたい場合に、depth によって調節可能なためである。

このフィルタによる AST ノード分離の様子を、図 5.10 に示す。

5.3 変数の出現範囲を抽出するフィルタ

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で計算が閉じている。よって、その変数が定義された行から最後に呼ばれる行までに注目するだけで、その変数にかかる計算は全て抜き出されると考えられる。

本研究では、このフィルタに対して以下の式を定義した。

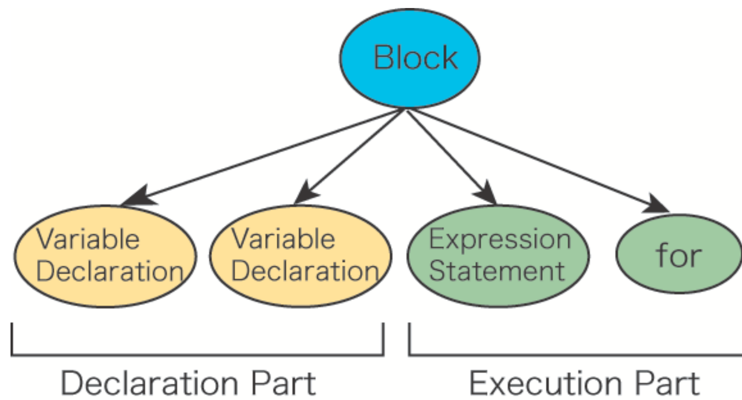


図 5.10: ブロックを定義部と実行部に分離するフィルタによる各抽出範囲

$$\mathcal{F}_{Range}(\text{AST ast}, \text{ BlockSet bs}, \text{ int pos}, \text{ String var}) \tag{5.3}$$

このパラメータはそれぞれ、次のような意味を持っている。

AST ast ソースコード全体

BlockSet bs フィルタの適用範囲

int pos 注目している行の番号

String var 特定の変数名

ここで、AST と BlockSet のデータ構造は、$\mathcal{F}_{Ctl}$ と同様である。

このフィルタによる抽出範囲を、図 5.11 に示す。

ただし上記 2 種類のフィルタと異なり、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。

表 5.3 に例を挙げる。このコードは 5 行目にローカル変数 x を定義している。しかし 8 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 9 行目に注目した場合、呼び出されている変数 x は、8 行目で定義された変数である。しかし 11 行目に注目した場合、呼び出されている変数 x は、5 行目で定義された変数であり、8 行目の変数 x は全く関与しないことがわかる。

このように変数名をパラメータで与えても、行に注目しない限り注目する変数を特定できない。

ここでパラメータに渡された範囲 (BlockSet) 内に変数の定義が含まれていない場合を考える。このとき、このフィルタが抽出する範囲はパラメータに渡された範囲の中で一番最初に出てきた変数の呼出を開始行とする。しかしながら、該当する変数の定義がどのブ

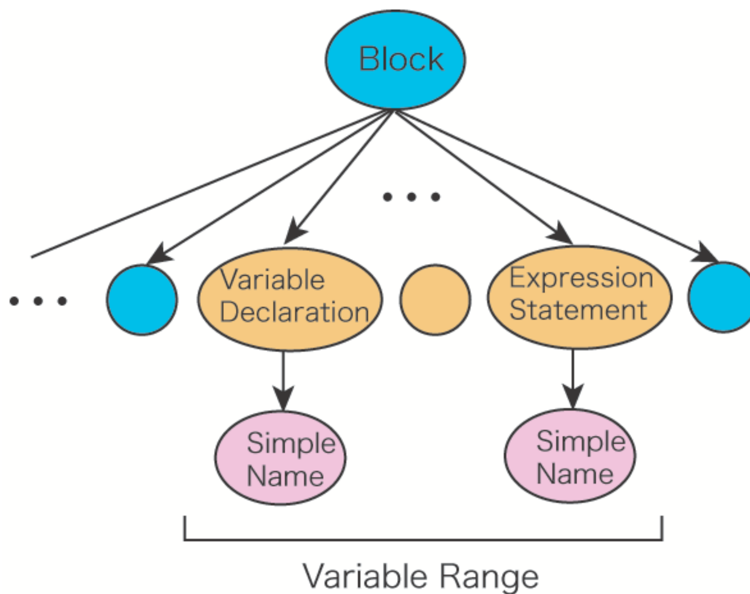


図 5.11: 変数の出現範囲を抽出するフィルタによる抽出範囲

表 5.3: 同じ名前で全く別のローカル変数

```
01  #include <stdio.h>
02
03  int main(){
04     int i;
05     int x=3;
06     printf("x1=%d\n", x);
07     for(i=0; i<3; i++){
08        int x=50;
09        printf("x2(%d)=%d\n", i, ++x);
10     }
11     printf("x3=%d\n", x);
12     return 0;
13  }
```

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

ロックにあるかの特定は、パラメータの範囲を超えても必ず探索する。これは上記の理由から、変数の有効範囲を確認する必要があるためである。この場合の抽出の様子を図 5.12 に示す。

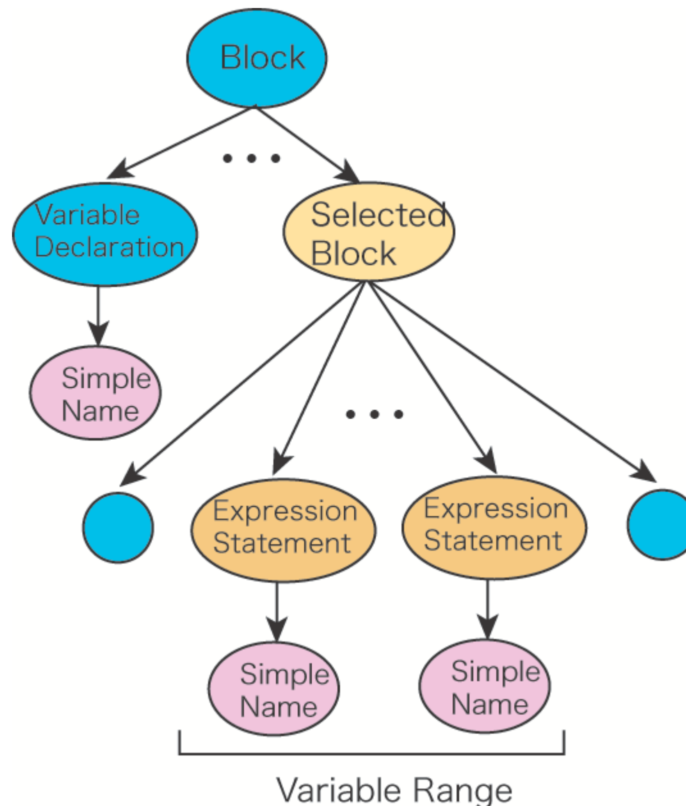


図 5.12: パラメータに渡された範囲に変数定義がない場合の抽出する AST ノード

またパラメータに与える変数名を大域変数にした場合も、図 5.12 の例が適用される。このときフィルタのパラメータとして入力する範囲は、最大で、注目している行のある関数全体とする。大域変数は複数の関数に対して影響を及ぼすが、関数間の物理行は一般的にその変数と何ら関係がないからである。

5.4 代入文の要素に依存する式を追跡して抽出するフィルタ

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 Statement 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)。そこまでであれば変数の出現範囲を抽出するフィルタで足りることであるが、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、上記フィルタでは要求に応えることができない場合が発生しうる。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。
本研究では、このフィルタに対して以下の式を定義した。

$$\mathcal{F}_{Trace}(\mathrm{AST\ ast,\ BlockSet\ bs,\ int\ pos,\ int\ depth,\ Booleanargs}) \tag{5.4}$$

このパラメータはそれぞれ、次のような意味を持っている。

AST ast ソースコード全体

BlockSet bs フィルタの適用範囲

int pos 注目している行の番号

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

ここで、AST と BlockSet のデータ構造は、$\mathcal{F}_{Ctl}$ と同様である。
このフィルタの特徴は、ある代入文に注目したときに、その変数を root とする意味木を内部で作成するところである。例として、表 5.4 のソースコードの AST である図 5.13 に対して作成される意味木を図 5.14 に示す。ただし、注目している行は 125 行目とする。
この代入式は、変数 a と変数 b の積を変数 x に代入している。すなわち x は a と b に依存しているといえて、この a と b の値を決定している式を抽出することで、x を知ることができる。それは a や b への直前の代入文か定義文である。結果として a については 111 行目、b については 121 行目が抽出される。ここでパラメータに depth=1 を与えていた場合、このフィルタは 111, 121, 125 行目を Statement List のような形で出力する。
もし depth=2 であれば、さらに a と b について意味木を作成する。ここで a は自分自身を引数として関数 bar() に与えている。そしてこのフィルタのパラメータが args=true であった場合のみ、この bar() への引数 a を追跡する。この変数 a は自信を参照しているため、意味木は図 5.14 のような形になる。b については i によって決定されているため、120 行目が depth=1 の出力に追加される。depth=3 の場合も、i について同様に処理する。
もし a のように自身あるいは探索済みのノードへの循環参照が発生した場合、そこで探索をを終了する。

5.5 注目部の近傍を抽出するフィルタ

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ことがしばしばある。これは処理の内容やデータ構造とは依存性の低い操作であるが、人間にとって非常に直観的な操作である。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。
本研究では、このフィルタに対して以下の式を定義した。

図 5.13: 表 5.4 の AST と、指定した代入文を追跡する様子

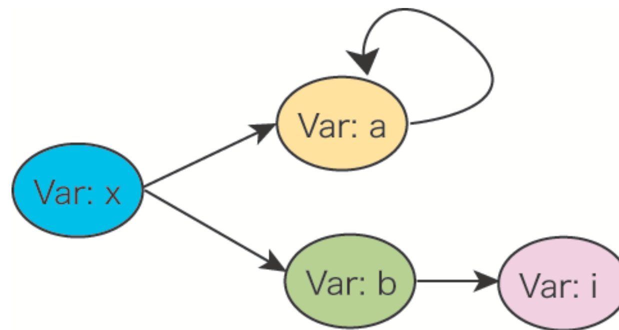


図 5.14: 図 5.13 で代入文を構成する変数を追跡するための変数の意味木

表 5.4: hoge.c

```
 :      :
100  int foo(int x){
101    int a = 0;
 :        :
110    while(x--){
111      a = bar(a);
 :          :
120      for(int i=0; i<N; i++){
121        b += i;
 :            :
125        x = a * b;
 :            :
130      }
 :          :
```

$$\mathcal{F}_{Neiborhood}(\text{AST ast},\ \text{BlockSet bs},\ \text{int prev},\ \text{int follow}) \tag{5.5}$$

このパラメータはそれぞれ、次のような意味を持っている。

AST ast ソースコード全体

BlockSet bs フィルタの適用範囲

int prev 前方物理行数

int follow 後方物理行数

ここで、AST と BlockSet のデータ構造は、$\mathcal{F}_{Ctl}$ と同様である。

なおこのフィルタのパラメータには注目行を与えていない。抽出される範囲は、bs の範囲の前後で与えることができるためである。このようにすることで、他のフィルタの出力を別のフィルタの入力にするときに、入力の幅を広げることも可能となる。

実際には AST の木構造は前後の行というデータを持っていない。しかし抽出対象は取り込んだソースコードの物理行数で決定できるため、このフィルタは AST の依存なしに抽出を行っている。

5.6 大域変数の定義部と実行部を抽出するフィルタ

これまでの5種類のフィルタは、注目するブロックとノードを決定して、探索を行うことで作成してきた。そのため変数の扱いはすべてローカル変数のみであり、大域変数については基本的に触れていなかった。しかしソースコードに大域変数が利用される場合も多く存在する。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に影響する場合、ソースコードを読もうとする人間にとって非常に労力を伴う。よって、大域変数がどこで定義されているか、そしてどこで利用されているかを抽出することで、この労力を省くことができると考えられる。

本研究では、このフィルタに対して以下の式を定義した。

$$\mathcal{F}_{Global}(\text{AST ast, String var}) \tag{5.6}$$

このパラメータはそれぞれ、次のような意味を持っている。

AST ast ソースコード全体

String var 大域変数名

ここでASTのデータ構造は、$\mathcal{F}_{Ctl}$と同様である。

このフィルタを作るために、ツールの前処理を行うようにした。すなわちソースコードを取り込んだときに、すべての大域変数定義, 関数定義, 型定義を行うノードを割り出し、いつでも参照できるようにした。関数の外に対する抽出は、その影響範囲が非常に大きくなるため、パフォーマンスを考慮する必要があるからである。

このようにして得た前処理を下に、パラメータで得た変数名に一致する大域変数定義を調査した。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)、このフィルタは空集合を返す。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Statement Listのような形で抽出している。

5.7 変数および型の定義部を抽出するフィルタ

ソースコードを読むとき、ある変数の型が気になることが往々にしてある。あるいはPrimitiveでない型で定義されている定義文をみたときに、その型の正体が気になることがある。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、困難な場合も多い。そこでこれらを簡単に調べることが、このフィルタの目的である。

本研究では、このフィルタに対して以下の式を定義した。

$$\mathcal{F}_{Decl}(\text{AST ast, int file, int pos}) \tag{5.7}$$

このパラメータはそれぞれ、次のような意味を持っている。

AST ast ソースコード全体

int file ファイル ID

int pos 注目している行

ここで、SourceCode のデータ構造は、$\mathcal{F}_{Ctl}$ と同様である。ファイル ID とは、ツール内部でファイルの特定に使用している値である。

このフィルタの抽出範囲も、場合によっては関数の外に及ぶため、影響範囲が全てのファイルとなる。よって前準備で作成した型定義のリストをもとに抽出を行う。ここには struct, union, typedef が含まれている。

フィルタの出力は型定義を行っている範囲と、変数定義を行っている Statement となる。

第6章　実装と評価

本研究では4章および5章をもとに理解支援ツールを作成した。そしてそのツールを用いて、実際のソフトウェアのソースコードで実験を行った。本章では、その結果について述べる。

6.1　実装

このツールを用いてユーザが行うことは、以下の通りである。

1. フィルタの選択
2. フィルタの合成

ここでフィルタの選択が選ばれた場合、以下の操作が発生する。

1. ユーザは使用するフィルタの種類とパラメータを決定する。
2. システムはUIからデータを受け取り、フィルタを生成する。
3. システムはフィルタにパラメータを与え、パラメータを検査する。
4. システムはフィルタを適用する。
5. システムは結果をUIに表示する。

このとき、これらのフィルタを選択して得られた中間結果を履歴として保存する。これは以下に続く合成に利用される。

フィルタの合成が選択された場合、以下の操作が発生する。

1. ユーザは合成の種類(AND/OR)と、合成するフィルタの結果のインデクスを選択する。
2. システムはUIからデータを受け取り、インデクスに対応するフィルタ履歴の出力を取得する。
3. システムは結果を合成する。
4. システムは結果をUIに表示する。

なお付録にツールのユースケース図、アクティビティ図、クラス図を添付する。

このようにして本ツールをJava言語を用いてEclipse SWTアプリケーション[1]として作成を行った。

図6.1に、作成したツールの実行画面を示す。

[1]Standard Widget Toolkit

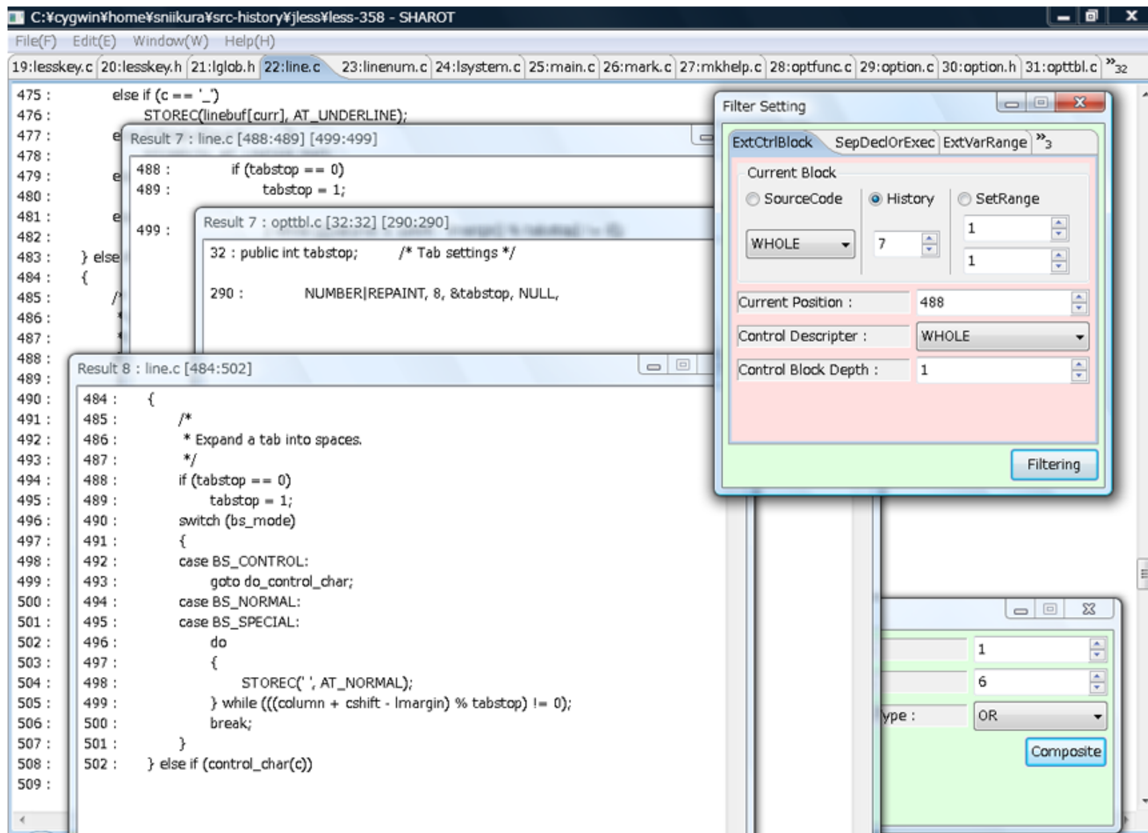


図 6.1: ツールの実行画面

6.2　ツールによる実験

本研究で作成したツールを利用して抽出されたデータの有用性を確認するために、オープンソースのソフトウェアを用いて実験を行った。ソフトウェアの 2 つのバージョンを利用することで、そのバージョン間のリリースノートを古いバージョンでの要望内容としてとらえ、新しいバージョンとの差分を模範解答としてとらえることができる。それに相当するデータがツールから抽出できれば、このツールによってユーザの必要とする情報が抽出できたと言える。この条件を満たすためにオープンソースのソフトウェアを利用した。

この実験の流れを図 6.2 に示す。

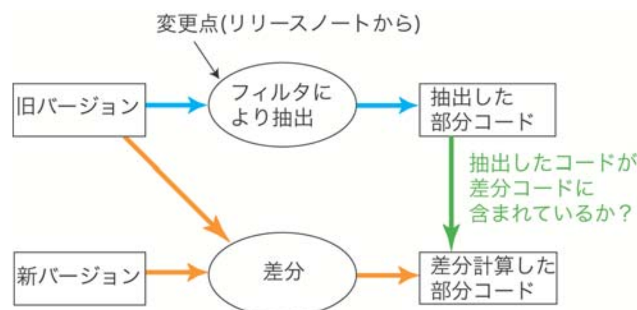


図 6.2: 実験の流れ

実際の実験には、jless[9] の Ver.358 と 378 を用いて行った。またここでは、
-x オプションが複数の可変幅のタブ · ストップに対応した
という機能変更に注目した。

1. ソースコード全体からの抽出

まず最初は、各ソースコードの役割がわからないので、ソースコード全体を対象とした抽出を行った。よって使用したフィルタは大域変数フィルタである。

$$\mathcal{F}_{Global}(\mathrm{S}, \text{"/tab/i"}) \\ = \langle\{\mathrm{line.c}, 488, 489\}, \{\mathrm{line.c}, 499, 499\}, \{\mathrm{opttbl.c}, 32, 32\}, \{\mathrm{opttbl.c}, 290, 290\}\rangle \quad (6.1)$$

ここで S はソースコード全体を意味している。また”/tab/i”であるが、名前検索には正規表現が使えるように設計した (ただしオプションは i のみ実装)。これによって、tab という文字列を含む変数の抽出を行った。

このフィルタによって抽出する様子を図 6.3 に示す。

フィルタ (6.1) の出力は、大域変数 tabstop の定義場所と参照場所のみ抽出されていた。そこで line.c の中で行われている操作に注目し、制御構造の実行ブロックの抽出フィルタを用いた。よって、フィルタの式は (6.2) のようになる。

$$\mathcal{F}_{Ctrl}(\mathrm{S}, \langle\{\mathrm{line.c}, 0, \mathrm{EOF}\}\rangle, 488, 1, \mathrm{WHOLE}) = \langle\{\mathrm{line.c}, 484, 502\}\rangle \quad (6.2)$$

このフィルタによって抽出する様子を図 6.4 に示す。ここで抽出された結果から、タブをスペースに変換する処理を行っていることが読み取れる。

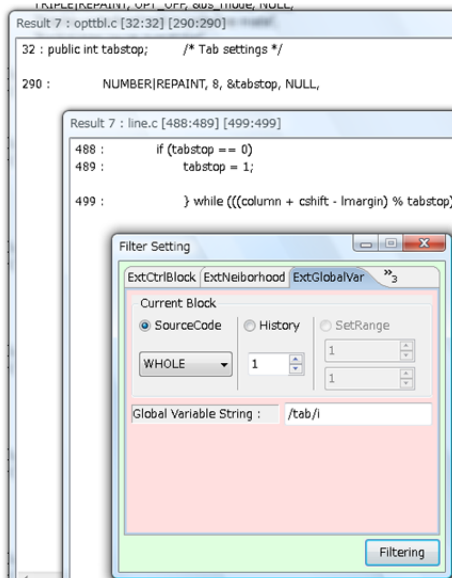


図 6.3: ツールの抽出結果 1

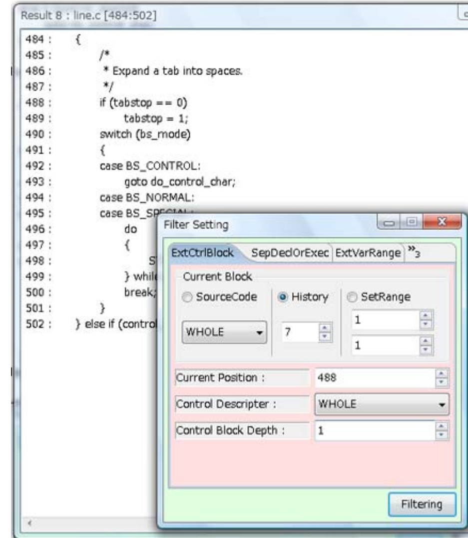


図 6.4: ツールの抽出結果 2

また opttbl.c についても同様に、注目している行のブロックの意味を知るために、フィルタ (6.3) を適用した。このフィルタによる抽出の様子を図 6.5 に示す。

$$\mathcal{F}_{Ctrl}(\mathrm{S}, \langle\{\mathrm{opttbl.c}, 0, \mathrm{EOF}\}\rangle, 290, 1, \mathrm{WHOLE}) = \langle\{\mathrm{opttbl.c}, 289, 294\}\rangle \tag{6.3}$$

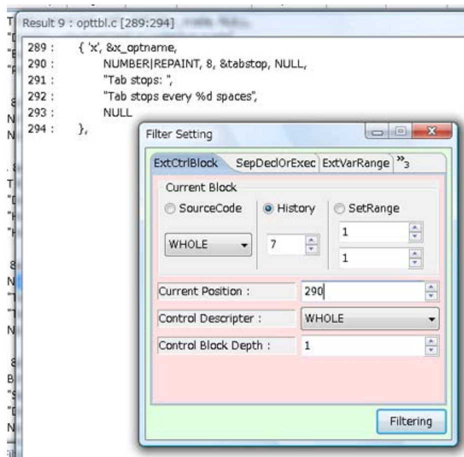


図 6.5: ツールの抽出結果 3

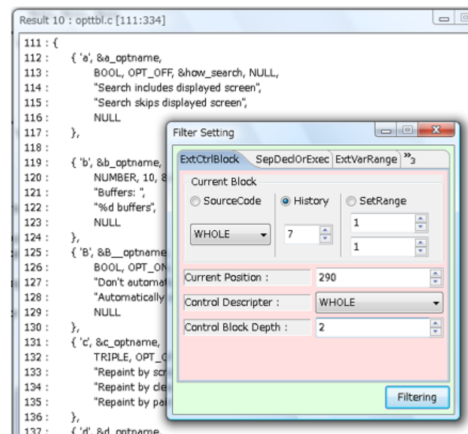


図 6.6: ツールの抽出結果 4

フィルタ (6.3) の結果は、値の列挙で作られたブロックである。これを含むステートメントの正体を知るために、フィルタ (6.3) のパラメータ depth を変化させて抽出を行った。このフィルタを (6.4) に示す。またその抽出結果を図 6.6 に示す。

$$\mathcal{F}_{Ctrl}(\mathrm{S}, \langle\{\mathrm{opttbl.c}, 0, \mathrm{EOF}\}\rangle, 290, 2, \mathrm{WHOLE}) = \langle\{\mathrm{opttbl.c}, 110, 334\}\rangle \tag{6.4}$$

ここで抽出された結果も、やはり値の列挙で作られたブロックであった。そこでさらにパラメータの depth 値を変化させて抽出を行ったところ、(6.5) のようにファイル全体が抽出されてしまった。これは (6.4) のブロックが、一番外のブロックであることを示している。この抽出結果を図 6.7 に示す。

$$\mathcal{F}_{Ctrl}(\mathrm{S}, \langle\{\mathrm{opttbl.c}, 0, \mathrm{EOF}\}\rangle, 290, 3, \mathrm{WHOLE}) = \langle\{\mathrm{opttbl.c}, 0, \mathrm{EOF}\}\rangle \tag{6.5}$$

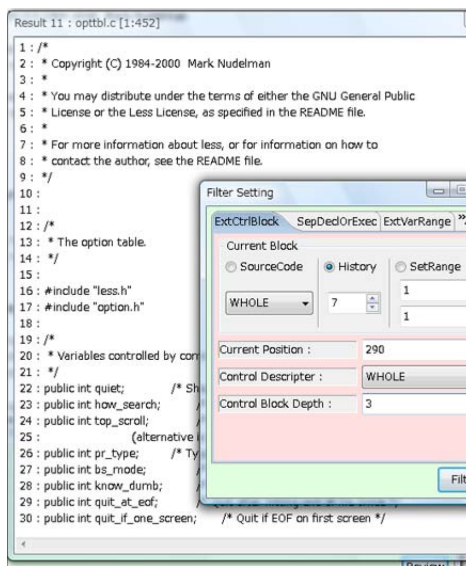


図 6.7: ツールの抽出結果 5

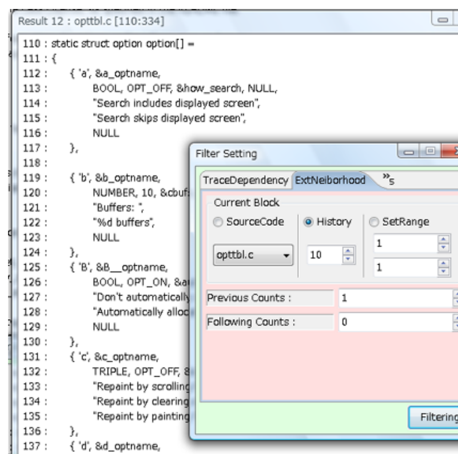


図 6.8: ツールの抽出結果 6

そこで、このブロックの近傍を抽出して、このステートメントの正体を知ることにした。フィルタ (6.6) の抽出の様子を図 6.8 に示す。

$$\mathcal{F}_{Neiborhood}(\mathrm{S}, \langle\{\mathrm{opttbl.c}, 111, 334\}\rangle, 1, 0) = \langle\{\mathrm{opttbl.c}, 110, 334\}\rangle \tag{6.6}$$

この結果から、この Statement が構造体 option 型の配列の初期化であることがわかった。そこでこの構造体の定義を調べるために、定義部抽出フィルタを使用した。フィルタの式は (6.7) となり、ツールでは図 6.9 のようになった。

$$\mathcal{F}_{Decl}(\mathrm{S}, \langle\mathrm{opttbl.c}\rangle, 110) = \langle\{\mathrm{option.h}, 52, 61\}\rangle \tag{6.7}$$

これら 3-7 の結果から、この Statement では各オプションの対応表を作成することが目的であると読み取れる。大域変数 tabstop は、そこで共有変数として使われている。

ところで、Ver.378 について diff を利用してその差分を求めた。フィルタ (6.2) の出力結果に対応する差分を表 6.1 に示す。ここで、タブからスペースに変換する処理の部分が変わり、専用のマクロを呼び出すように変更されていることが読み取れる。

さらにフィルタ (6.3) の結果に対応する差分をとった。その差分を表??に示す。なお Ver.378 では構造体 option 型が構造体 loption 型に変更されていたが、option.h に記述されているその型定義は全く変わっていなかった。

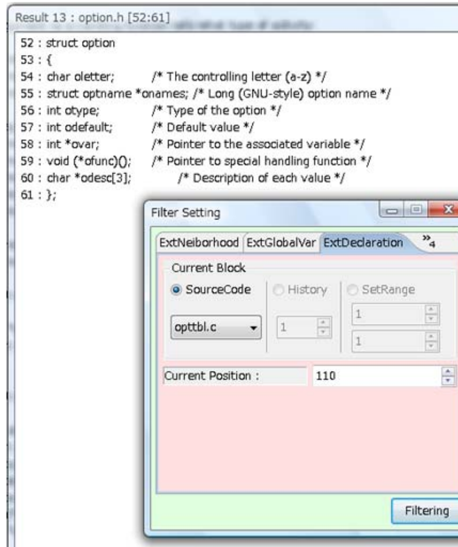


図 6.9: ツールの抽出結果 7

表 6.1: フィルタ (6.2) に対応する差分

```
* * *485, 502 * * * *
          /*
           * Expand a tab into spaces.
           */
!         if (tabstop == 0)
!             tabstop = 1;
          switch (bs_mode)
          {
          case BS_CONTROL:
              goto do_control_char;
          case BS_NORMAL:
          case BS_SPECIAL:
!             do
!             {
!                 STOREC(' ', AT_NORMAL);
!             } while (((column + cshift - lmargin) % tabstop) != 0);
              break;
          }
      } else if (control_char(c))
```

→

```
- - - 646,658 - - - -
          /*
           * Expand a tab into spaces.
           */
          switch (bs_mode)
          {
          case BS_CONTROL:
              goto do_control_char;
          case BS_NORMAL:
          case BS_SPECIAL:
!             STORE_TAB(AT_NORMAL, pos);
              break;
          }
      } else if (control_char(c))
```

表 6.2: フィルタ (6.3) に対応する差分

```
* * *289, 294 * * * *
      { 'x', &x_optname,
!     NUMBER|REPAINT, 8, &tabstop, NULL,
!     "Tab stops: ",
!     "Tab stops every %d spaces",
!     NULL
      },
```

→

```
- - - 346,353 - - - -
      { 'x', &x_optname,
!     STRING|REPAINT, 0, NULL, opt_x,
!     {
!         "Tab stops: ",
!         "0123456789,",
!         NULL
      }
      }
```

構造体定義の変化を表 6.2 に示す。

ここで option の表に現れた大域変数 tabstop に注目する。Ver.378 では tabstop は使用されておらず、対応する値が NULL となっている。そして opt_x という名前の新しい関数が呼ばれていることがわかる。

この結果より、ツールによって抽出されていた情報は、Ver.378 で確かに注目されて変更のあった場所であることがわかる。ここから、本ツールによって抽出された情報が機能の変更に有用な情報であるといえる。

6.3 評価

6.2 節で行った実験結果について、Cxref や Sgrep などでは抽出できないデータを抽出できた。GNU GLOBAL の場合は抽出不可能ではないが、本ツールにとって単純な操作であったことに対し、GNU GLOBAL は発見に多くのステップ数がかかる (リンクをたどるうちに中心としていた点がわからなくなる迷子問題も内包している)。

これよりユーザは必要な情報を短時間で得ることが可能となったことがわかった。

第7章　終わりに

7.1　まとめ

本研究ではソースコードの理解支援として、実際にツールの開発を行った。永井の提案した 5 種類のフィルタの概念を元にして、2 種類のフィルタを新たに定義し、これら 7 種類のフィルタについて厳密な定義を行った。実装では、フィルタの動作をパラメータにより細かく制御し、その結果を別のフィルタの入力とする、あるいは AND や OR によるフィルタの合成を行うことでユーザの多様な要求に応えることを可能にした。

7.2　今後の課題

現在は必要なフィルタの選択や合成を行うために、ユーザは多くの試行錯誤を必要とする。このような入力はユーザにとって大きな負担となるため、簡単に利用できる仕組みを設けたい。特定の状況で使用されるフィルタとパラメータの組を事前に提供することで、利便性を向上させることが今後の方針として挙げられる。

謝辞

本研究を行うにあたり、終始御指導して頂きました鈴木正人准教授に深く感謝申し上げます。自身の体調を顧みず指導してくださった姿は、指導者の鑑として強く感銘を受けました。

また研究の節目ごとに的確な指摘をしてくださった落水浩一郎教授に深く感謝申し上げます。本研究を客観的な立場から評価して頂いたおかげで、常に研究の立ち位置を意識しながら進めることができました。

そして他研究室でありながら公私にわたり御世話をしてくださった小谷正行氏に深く感謝申し上げます。研究面で、あるいは生活面で、先輩として、指導者として、友人として、様々な角度から支えてくださったおかげで、本研究を実らせることができました。

その他、研究室での生活を支えてくださった鈴木研究室、落水研究室の皆様、さらに本学で再出発の機会を与えてくださった島津明研究科長に、ここで感謝のことばとさせて頂きます。

最後に、ここまで勉学を支援して頂いた家族に、心より感謝申し上げます。

参考文献

[1] 永井路人, ソースコード理解支援のための表示自由度の高い視覚化ツールの研究, 北陸先端科学技術大学院大学 情報科学研究科 修士論文, 2006.03.

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

[3] Andrew M. Bishop, C Cross Referencing and Documenting tool, http://www.gedanken.demon.co.uk/cxref/

[4] 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)

[5] 吉田敦, 山本晋一郎, 阿草清滋, 意味を考慮した差分抽出ツール, 情報処理学会論文誌 Vol.38 No.6 P.1163-1171, 1997.06.

[6] 荻原剛志, 會沢実, 鳥居宏次, 構文木の相互比較による複数バージョン比較分析方法の提案, ソフトウェア工学の基礎 II P.21-30, 日本ソフトウェア科学会 FOSE'95

[7] 五月女健治, JavaCC コンパイラ・コンパイラ for Java, テクノプレス, 2003.

[8] Diomidis Spinellis 著, (株) トップスタジオ 訳, 鵜飼文敏/平林俊一/まつもとゆきひろ 監訳, Code Reading オープンソースから学ぶソフトウェア開発技法

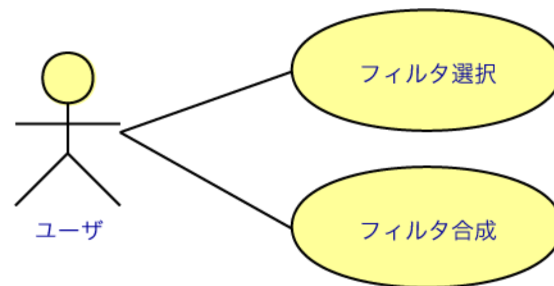
[9] Kazushi (Jam) Marukawa, Jam less (jless), http://www25.big.jp/ jam/

[10] Bjarne Stroustrup 著, 長尾高弘 訳, プログラミング言語 C++ 第3版, Addison-Weslry Publishers Japan Ltd., 1998.11.

本研究に関する発表論文

[1] 新倉諭, 鈴木正人, ソースコード理解支援機能を持つ開発環境, 情報処理学会, 第159回ソフトウェア工学研究発表会, 2008.03.

付録



フィルタ選択
目的: ソースコードから必要な情報を抽出する。
起動アクタ: ユーザ
主シーケンス:
1. ユーザはフィルタ名とパラメータを与える。
2. システムはフィルタを生成する。
3. システムはパラメータを検査する。
4. システムはソースコードに選択されたフィルタを適用する。
5. システムは適用結果をユーザに返す。
代替シーケンス:
A1. パラメータがフィルタの仕様と一致しない場合
システムはパラメータ不一致エラーを返す。
A2. パラメータの値の一部がNULLの場合
システムはフィルタの前回使用された値を
不足しているパラメータの値として、4に戻る。
事後条件: フィルタは有効な結果を返している。

フィルタ合成
目的: フィルタ選択によって得られた結果を拡大または縮小する。
事前条件: 少なくとも1つのフィルタ選択が実行され、結果が得られている。
主シーケンス:
1. ユーザはフィルタ選択結果のインデクス列と合成の種類をシステムに与える。
2. システムはインデクスを用いて履歴を取得する。
3. システムは結果を合成する。
4. システムは合成結果をユーザに返す。
代替シーケンス:
B1. 履歴の取得に失敗した場合
システムは履歴取得エラーを返す。
事後条件: フィルタは有効な結果を返している。

図 1: ユースケース

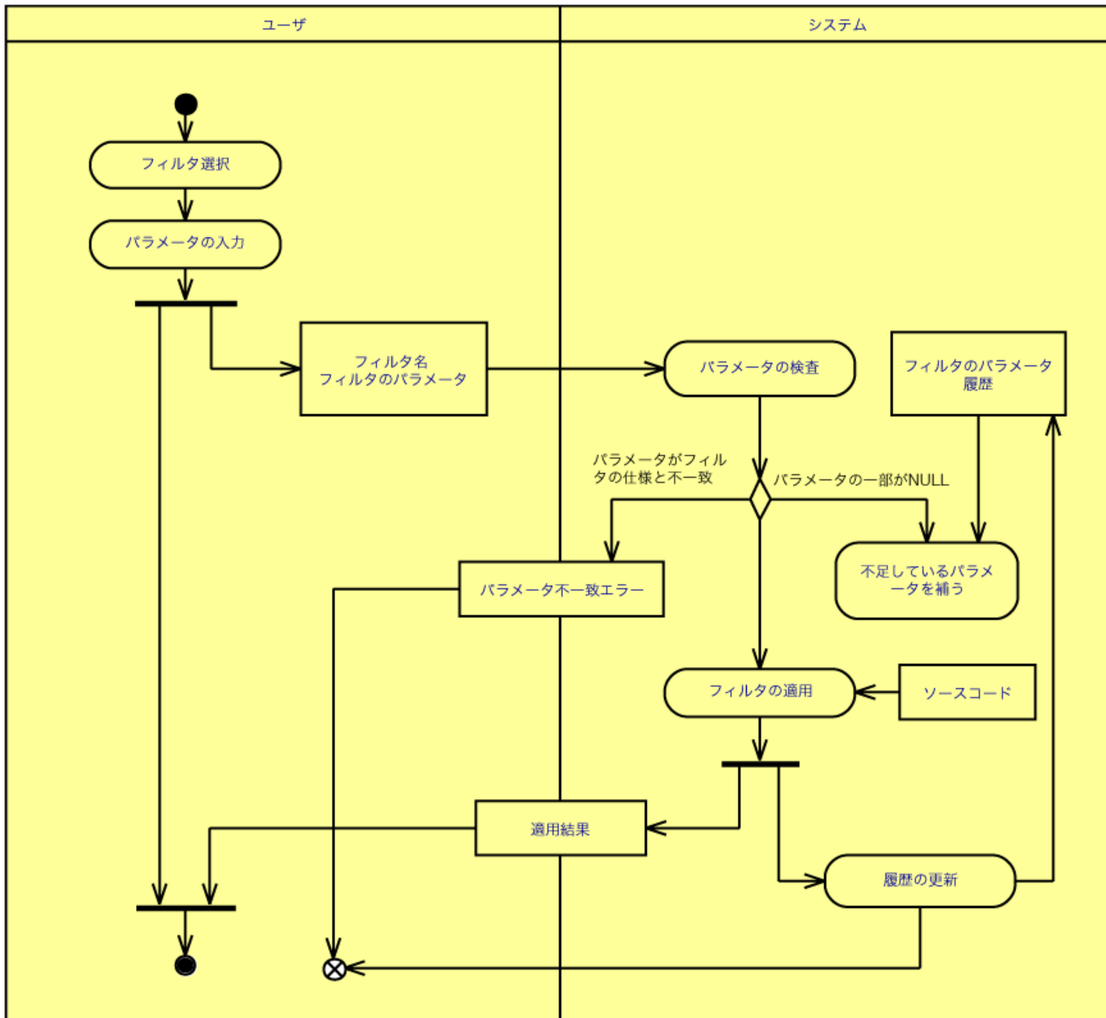


図 2: フィルタ選択のアクティビティ図

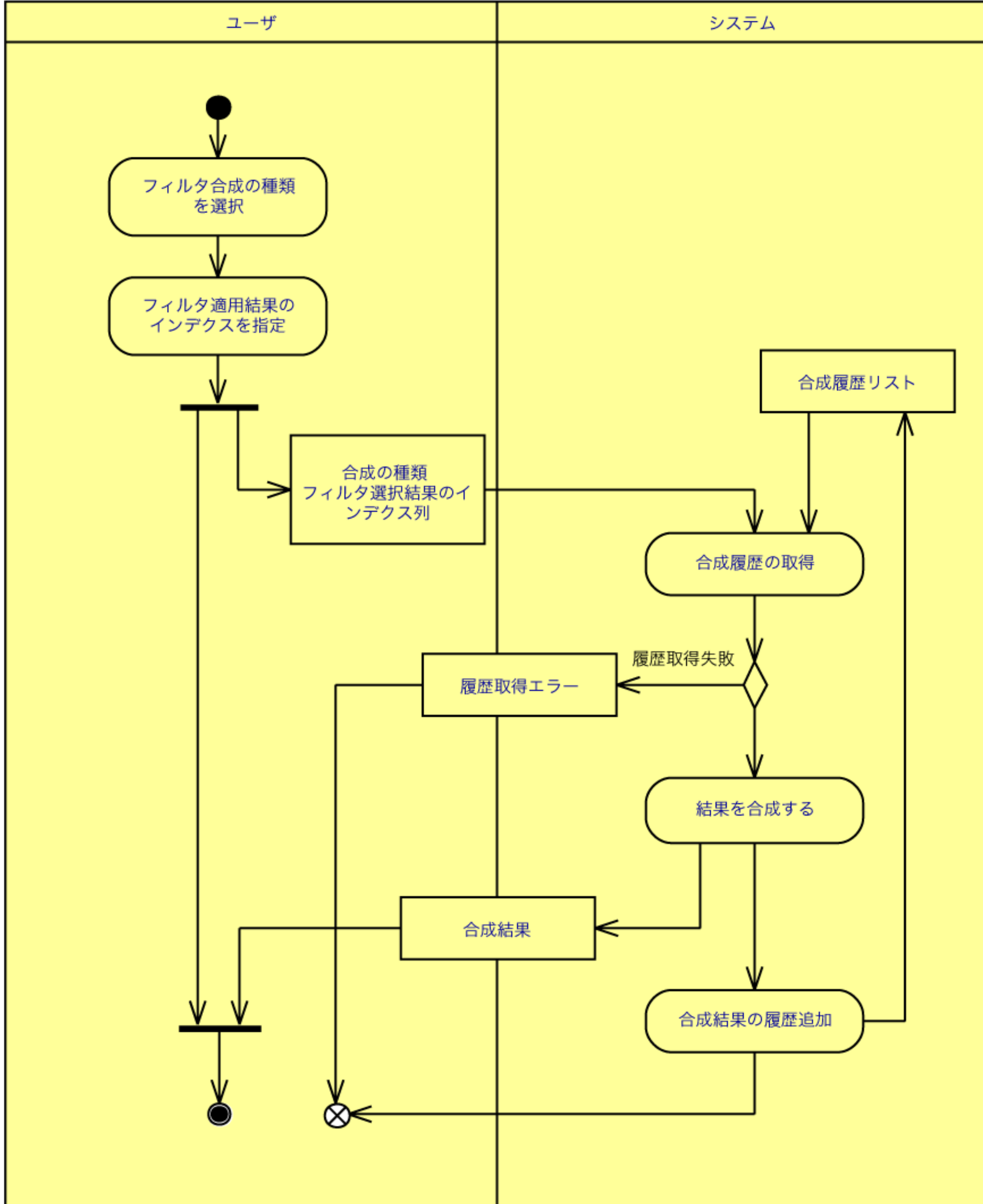


図 3: フィルタ合成のアクティビティ図

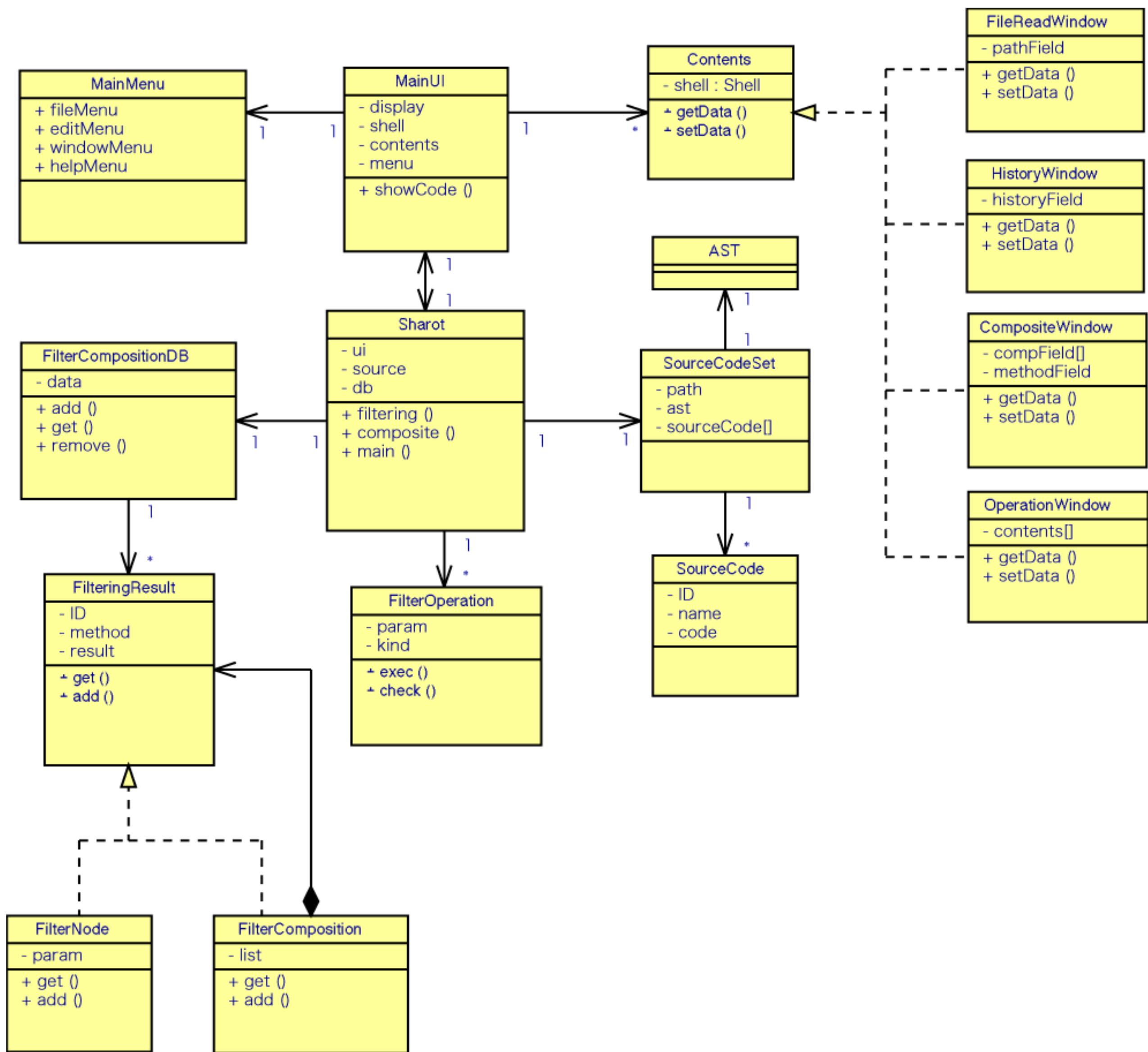


図 4: ツール全体のクラス図

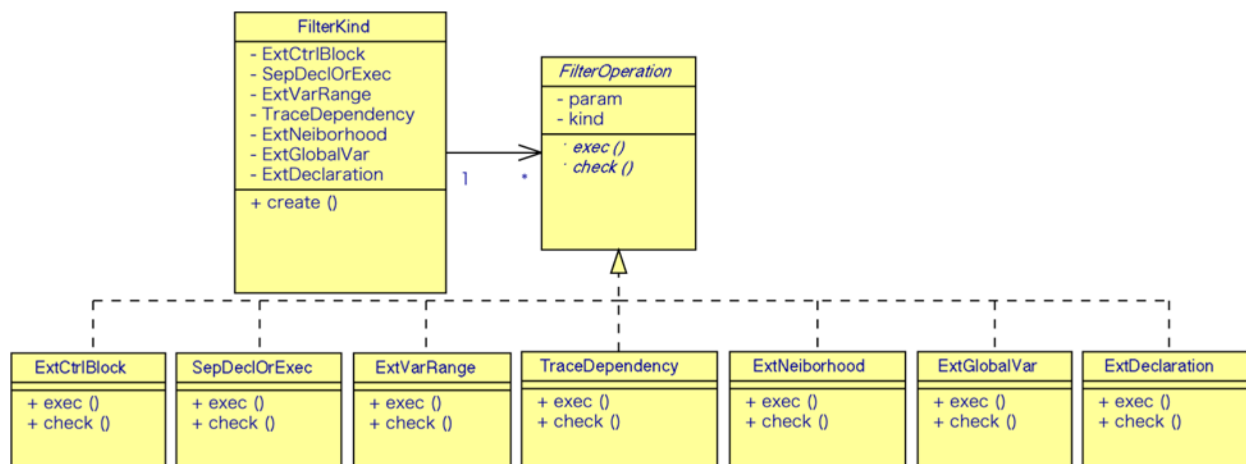


図 5: フィルタ生成のクラス図