Diplomarbeit

Studiengang:	Informatik
Prüfer:	Prof. Erhard Plödereder
Betreuer:	Dr. Rainer Koschke
Beginn am:	08.07.2003
Beendet am:	08.01.2004
CR-Nummer:	D.2.2

Universität Stuttgart
Fakultät Informatik
Institut für Softwaretechnologie
Abteilung Programmiersprachen und Übersetzerbau

Diplomarbeit-Nr. 2128

Konzeption und Implementierung

eines Quellcode-Navigators

Leiqin Lu

Diploma Thesis

Conception and Implementation of a Source Code Navigator

Leiqin Lu

2003.07.08 – 2004.01.08

Supervisor: Dr. Rainer Koschke

Professor: Erhard Plödereder

University of Stuttgart

Institute of Software Technology

Department of Programming Languages and Compilers

Table of Contents

Table of Contents i

Table of Figures v

Abstract 1

1. Introduction 3

1.1 The Bauhaus Project 3
1.2 Levels of Program Understanding 5
1.3 A Source Code Navigator 7
1.4 Targets and Tasks 9
1.5 Developing Environment 10

2. Requirements Specification 11

2.1 Introduction 11
2.1.1 Purpose 11
2.1.2 Scope 11
2.1.3 Definitions, Acronyms and Abbreviations 12
2.1.4 References 12
2.1.5 Overview 12
2.2 Overall Description 13
2.2.1 Product Perspective 13
2.2.2 Product Functions 14
2.2.3 User Characteristics 15
2.2.4 Constraints 15
2.3 Specific Requirements 16
2.3.1 Integration with Gravis 16
2.3.2 Interaction with Users 16
2.3.3 Source-code-related Information 16
2.3.3.1 Reference Information 17
2.3.3.2 Class Information 20
2.3.3.3 Points-to and Side-effects Information 21

2.3.3.4 Program Slicing Information 27
2.3.3.5 Source Code Metrics 28
2.3.4 Navigation Semantics 28
2.3.5 Miscellaneous 29

3. Tools Evaluation 31

3.1 Source-Navigator 31
3.1.1 Introduction 31
3.1.2 Windows Layout 32
3.1.3 Symbol Browser Window 34
3.1.4 Source Viewer Window 34
3.2 CodeSurfer 40
3.2.1 Introduction 40
3.2.2 Windows Layout 40
3.2.3 Project Viewer 40
3.2.4 File Viewer 42
3.2.5 Call Graph Viewer 44
3.2.6 Property Sheets 45

4. Conception of SCN 49

4.1 Windows Layout 49
4.1.1 Browser Window 51
4.1.2 Viewer Window 53
4.2 User Actions 58
4.3 Source Code Construct Selection 60
4.4 Context Menu Construction 63
4.5 Source-code-related Information 64
4.5.1 Reference Information 64
4.5.2 Class Information 66
4.5.3 Calling Context 68
4.5.4 Points-to Information 69
4.5.5 Side-effects Information 70
4.5.6 Source Code Metrics 72

4.5.7 Miscellaneous 73
4.6 Navigation Semantics 74
4.6.1 Model of Navigation Process 74
4.6.2 Navigation Sequence and Navigation Map 76
5. Implementation of SCN 77
5.1 Software Architecture 77
5.2 Miscellaneous 78
Table of Definitions, Acronyms and Abbreviations 79
Bibliography 81

Table of Figures

Fig. 1-1 Bauhaus and Program Understanding 4

Fig. 1-2 Levels and GUI Tools of Program Understanding 5

Fig. 1-3 Source Code Navigator in Bauhaus (a) 7

Fig. 2-1 Source Code Navigator in Bauhaus (b) 13

Fig. 2-2 Inheritance Hierarchy of Classes 20

Fig. 2-3 Calling Contexts 23

Fig. 3-1 Windows Layout of Source-Navigator 33

Fig. 3-2 Symbol Browser of Source-Navigator 34

Fig. 3-3 Editor of Source-Navigator 35

Fig. 3-4 Hierarchy Browser of Source-Navigator 36

Fig. 3-5 Class Browser of Source-Navigator 37

Fig. 3-6 Cross-reference Browser of Source-Navigator 38

Fig. 3-7 Include Browser of Source-Navigator 39

Fig. 3-8 Project Viewer of CodeSurfer (a) 41

Fig. 3-9 Project Viewer of CodeSurfer (b) 42

Fig. 3-10 File Viewer of CodeSurfer 43

Fig. 3-11 Context Menu of CodeSurfer 44

Fig. 3-12 Call Graph Viewer of CodeSurfer 45

Fig. 3-13 Property Sheet of of CodeSurfer 47

Fig. 4-1 Windows Layout of SCN 51

Fig. 4-2 SCN Browser Window 53

Fig. 4-3 SCN Viewer Window 55

Fig. 4-4 User Actions 59

Fig. 4-5 Overlapping of Source Code Constructs 60

Fig. 4-6 Incremental and Interactive Selection of Source Code Constructs 61

Fig. 4-7 Reference Information in SCN (a) 64

Fig. 4-8 Reference Information in SCN (b) 65

Fig. 4-9 Class Information in SCN 67

Fig. 4-10 Points-to Information in SCN 69

Fig. 4-11 Side-effects Information in SCN 71

Fig. 4-12 Source Code Metrics in SCN 72

Fig. 4-13 Finite State Model of Navigation Process 74

Fig. 5-1 Software Architecture of SCN 77

Abstract

The Bauhaus project supports program understanding on both architectural level and source code level, which requires a graphical user interface tool for source code navigation. In this thesis a source code navigator is designed and implemented as part of the Bauhaus toolkit.

The source code navigator cooperates with other tools in the Bauhaus toolkit. In particular, it is fully integrated with Gravis, the tool for architectural view, to provide an integrated environment for program understanding from the general view on existing software to the particular view on source code in every detail.

Besides displaying source code, the source code navigator visualizes source-code-related information to pass users the power of Bauhaus. Source-code-related information includes reference information, points-to and side-effects information, class information, program slicing information and source code metrics information.

The source code navigator also makes an experimental study of navigation semantics. Navigation activities are modeled and navigation history is maintained to assist users in managing the progress of program understanding.

Some similar tools on the market are evaluated in this thesis, to give readers a picture of the state of the art in source code navigation.

Emphasis of this thesis is placed on conception of the source code navigator, while implementation is done in the associated program.

1. Introduction

This chapter introduces the background, motivation and basic requirements of a source code navigator. It starts with a brief description of Bauhaus project and program understanding, follows an introduction to the facility and difficulty of current tools for source code navigation in the Bauhaus project, which motivates the conception and implementation of a source code navigator. At last the basic requirements of the source code navigator are specified.

1.1 The Bauhaus Project

The Bauhaus project is devoted to software maintenance technologies, especially to the field of program understanding. In the following text, the Bauhaus project is referred to as Bauhaus. (See [Bauhaus])

Bauhaus develops software tools to extract, analyze, query and visualize information about existing software. The Bauhaus toolkit includes both back ends, which analyze programs and store the analysis result in some intermediate format (that is IML, the Intermediate Language), and front ends, which are graphical user interface tools for visualization and manipulation of program information on different levels. The basic work flow of program understanding in Bauhaus is illustrated in figure 1-1.

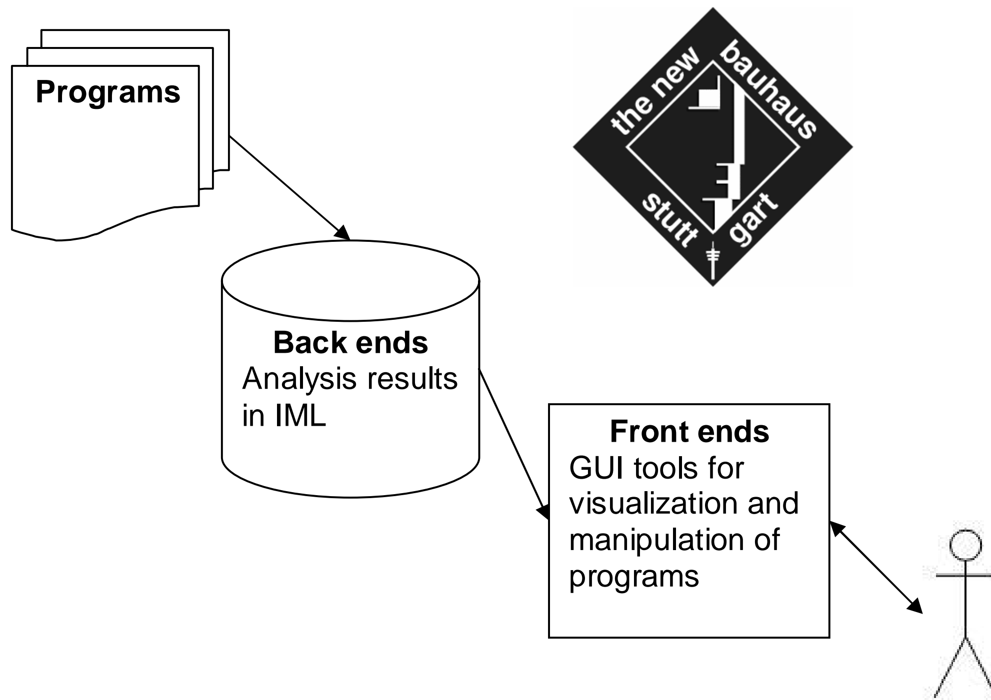

Fig. 1-1 Bauhaus and Program Understanding

The graphical user interfaces tools are only responsible for visualization and manipulation of program information. The program information is extracted and provided by corresponding back end tools. The graphical user interface tools retrieve program information by querying analysis results in the intermediate language IML, not by analyzing the programs directly. Thus the graphical user interface tools are almost independent with programming languages and other low level properties of the existing software being studied.

The user sign in the figure models the role of software maintainers or developers. Users explore existing software by using graphical user interface tools in the Bauhaus toolkit.

1.2 Levels of Program Understanding

Bauhaus supports program understanding on both architectural level and source code level. The architectural views model existing programs as graphs to provide users with overall information about the programs, such as the relationship between components, functions etc. The source code views give users textual information about the program in detail, that is, the source code itself and source-code-related information for specific source code constructs.

The graphical user interface tool for architectural information is Gravis, a generic graph editor. Source code views are currently supported by Emacs and HTML generators. The relationship between these tools and users is illustrated in figure 1-2.

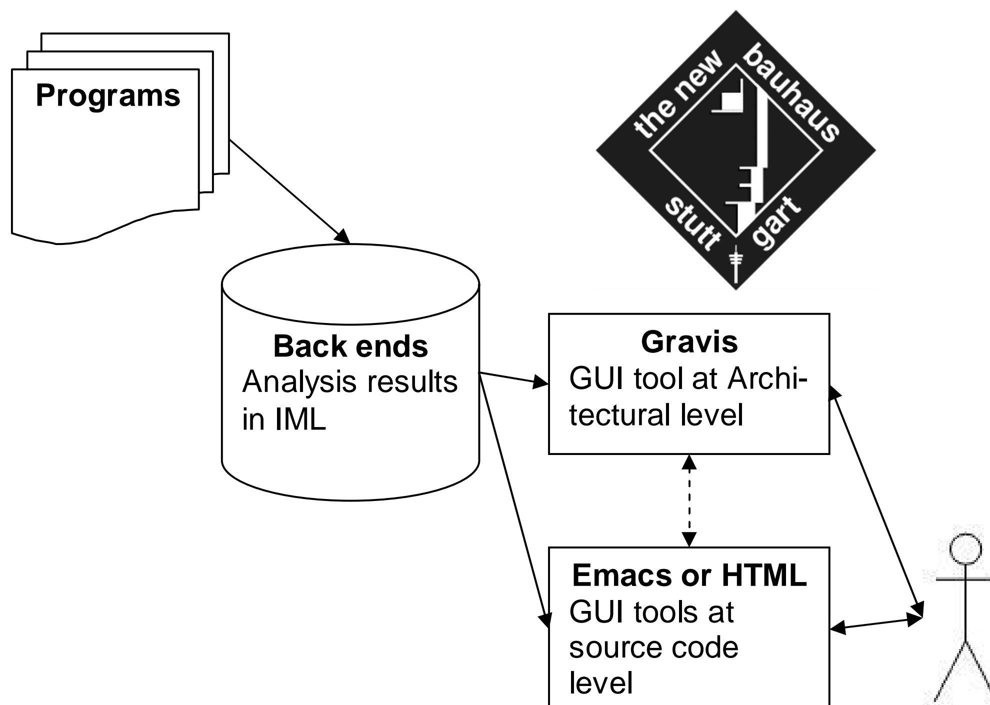

Fig. 1-2 Levels and GUI Tools of program understanding

In the figure 1-2, Gravis and Emacs/HTML are connected by a dotted line, indicating that they are loosely coupled. Gravis and Emacs/HTML are independent standalone tools.

Users interact with both levels of GUI tools. Switching between architectural and source code views is an explicit and separate step. For instance, to gain points-to information of a program, users must at first invoke a tool to generate HTML files from IML data, thereafter launch an HTML browser to view these HTML files. When viewing the points-to information in the HTML browser, there is no way to directly and naturally switch back to architectural views in Gravis.

On the other hand, attributes of Emacs and HTML gives a limitation on the facilities and feelings of source code navigation. Although Emacs is a very powerful tool for text processing, it's overpowered for source code navigation, and it is a foreign tool to the Bauhaus toolkit so that its integration with the toolkit is unnatural and brings unnecessary troubles.

HTML also has serious shortcomings for source code navigation. HTML is a markup language for static visual contents of pages, while source code navigation requires dynamic facilities by its nature. Except that, HTML is non-extensible. The main device in HTML for navigation is hyperlinks; there are no other sophisticated mechanisms, which are required by many different kinds of source-code-related information.

1.3 A Source Code Navigator

The shortcomings of Emacs/HTML for source code navigation motivate the conception and implementation of a source code navigator. The source code navigator is to substitute Emacs/HTML in the Bauhaus toolkit and integrate with Gravis fully. All tasks about navigation in source code level are to be taken by this single tool. See figure 1-3 for an overview of the source code navigator in Bauhaus. In the following text, the source code navigator is often abbreviated as SCN.

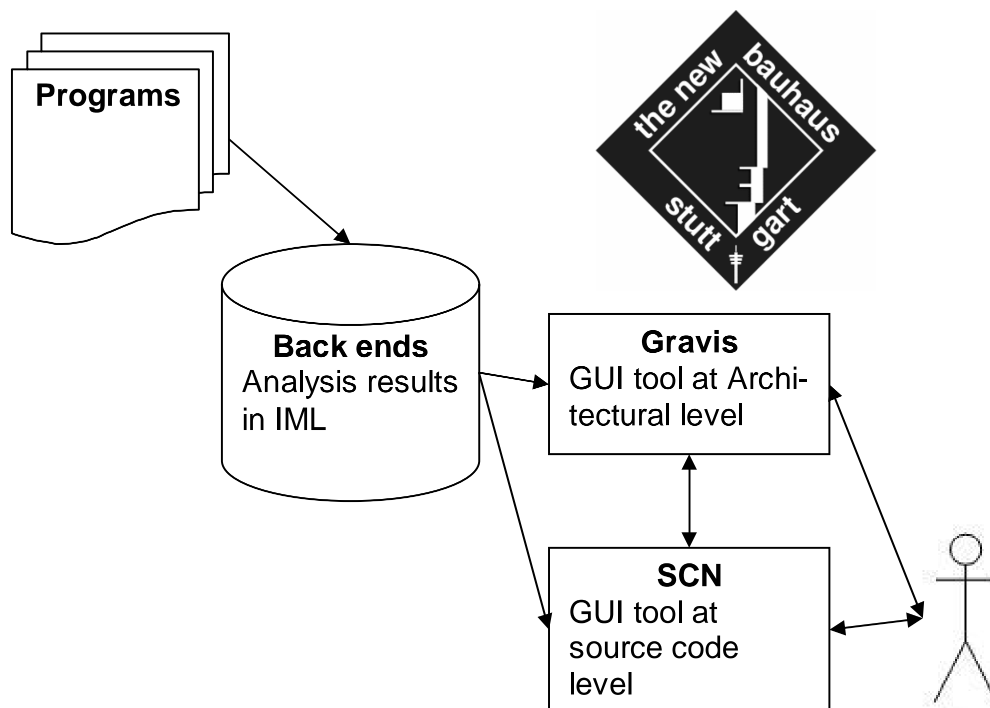

Fig. 1-3 Source Code Navigator in Bauhaus (a)

In comparison with the figure 1-2, the dotted line between Gravis and SCN is replaced by a normal line in the figure 1-3, indicating that Gravis and SCN are tightly coupled. They cooperate with each other and provide an integrated environment for program understanding on architectural and source code levels. Switching from Gravis to SCN or vice versa shall be intuitive and efficient.

Users are no longer required to invoke some tool explicitly to generate intermediate files or launch a viewer program.

1.4 Targets and Tasks

This section specifies targets and tasks of the SCN briefly, which are to be discussed in detail in next chapter.

The source code navigator shall implement these categories of functions:

- Integration with Gravis
- Interaction with users
- Visualization of source code and source-code-related information
- Management of navigation semantics

Emphasis is placed on source-code-related information, which includes:

- Reference information
- Points-to and side-effects information
- Program slicing information
- Class information
- Source code metrics information

Miscellaneous functionalities should be taken into consideration, for instance, syntax highlighting, macro representation, etc.

1.5 Developing Environment

The source code navigator is implemented in the Ada 95 programming language (see [Ada95]). The compilation system used is GCC 2.8.1 with GNAT 3.1.5 (See [Gnat]. The user interface is implemented with the Ada 95 graphical toolkit GtkAda 2.2.0 (see [GA-Web] and [GA2.2]), which is based on Gtk+. (See [Gtk+-Web])

The Bauhaus toolkit used is of version 4.4.3.

The native developing platform is Unix. A version of SCN on Windows systems are not considered currently, but should be easy to port.

The source code navigator is designed and implemented with an object-oriented approach, which is encouraged and eased by those object-oriented features of Ada 95 programming language and GtkAda 2.2.0 graphical toolkit.

Static structures, use cases and other perspectives of the source code navigator are modeled using UML, the unified modeling language. (See [UML-RJB] and [UML-OMG])

The extensible markup language, XML, is used to store data for navigation semantics. (See [XML-Web])

2. Requirements Specification

This chapter specifies the software requirements of the source code navigator. The structure and content of this document are adhering to the IEEE Standard 830-1998: IEEE Recommended Practice for Software Requirements Specifications (see [IEEE830-1998]).

In this specification, emphasis is placed on analysis of source-code-related information. A detailed conception of user interfaces is in chapter 4, and a basic description of software architecture is in chapter 5.

2.1 Introduction

2.1.1 Purpose

This chapter specifies the basic requirements of the source code navigator, from the view of users and other tools in Bauhaus. The conception and implementation of the source code navigator shall fulfill these requirements.

The intended audience for the software requirements specification is Bauhaus developers and users of the Bauhaus toolkit.

2.1.2 Scope

The software to be specified is "Source Code Navigator", often abbreviated as “SCN”. The source code navigator is a graphical user interface tool that supports users to navigate in source code of software products and provides users with source-code-related information to support program understanding on source code level.

The source code navigator is only responsible for visualization of source code and source-code-related information; it doesn't support editing or compiling of the source code. The source-code-related information is extracted by other tools in

the Bauhaus toolkit and stored in an intermediate format, so the source code navigator also needn't analyze source code by itself.

2.1.3 Definitions, Acronyms and Abbreviations

The definitions of all terms, acronyms, and abbreviations required to properly interpret this specification and the thesis is listed in the table of definitions, acronyms and abbreviations.

2.1.4 References

See the bibliography for references.

2.1.5 Overview

The specification is structured as follows. Section 1 introduces the software to be specified. The overall software requirements are described in section 2, to show readers perspectives and functions of the source code navigator. Finally, the third section contains specific requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and users to understand why the system is designed in this way. In particular, source-code-related information is learned and classified in detail in this section.

2.2 Overall Description

2.2.1 Product Perspective

The source code navigator is part of the Bauhaus toolkit. It cooperates with other Bauhaus tools to support program understanding. The architectural information of programs is visualized and manipulated by a graph editor, Gravis. The source-code-related information is organized and visualized by the source code navigator. See the following figure for an overview of the source code navigator in Bauhaus.

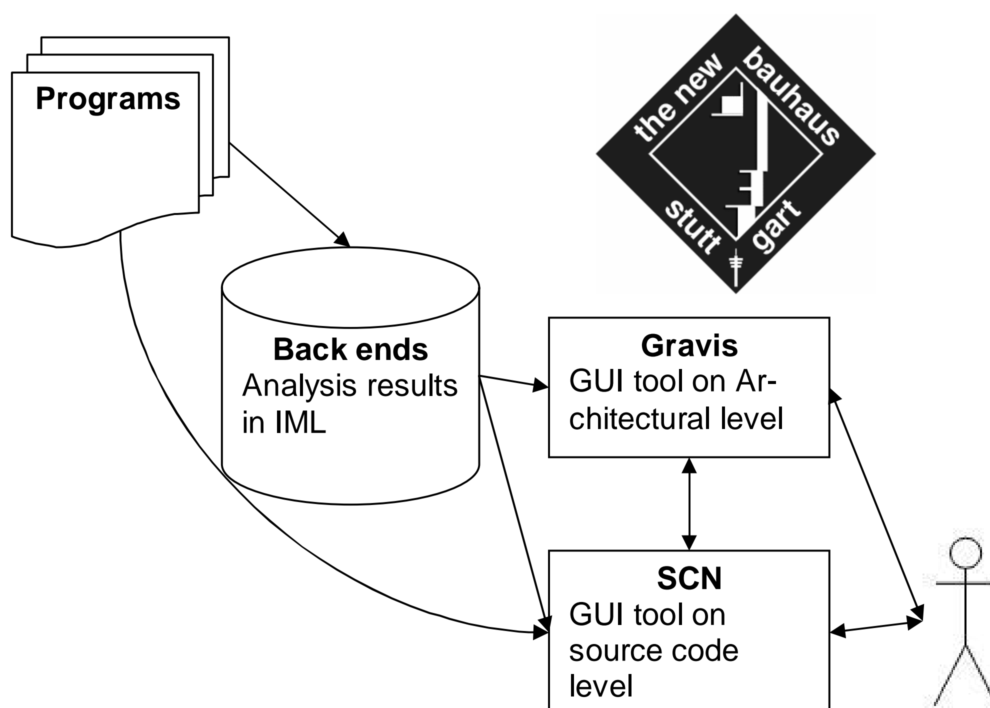

Fig. 2-1 Source Code Navigator in Bauhaus (b)

As illustrated in figure 2-1, Gravis and the source code navigator cooperate with each other to provide users a complete view on existing software from architectural views to source code views. That is, an integrated environment for program understanding, like those integrated environments for developing (IDE). Gravis

and SCN both retrieve program information from IML data, and visualize the information to users for navigation.

The source code navigator is fully integrated with Gravis. Users switch from architectural views to source code views and vice versa intuitively.

The source code navigator gets source-code-related information from IML data, not by analyzing source files by itself. It responds to user actions by querying IML data, organizing and visualizing the results to users.

To visualize source code and source-code-related information, the SCN must also access source code of the software product being studied. So a line in the above figure connects programs and the SCN. But note that the SCN only has read access to source code, and it is not allowed to make any change on the source code.

Benefited from the intermediate format of program information, the source code navigator is almost independent with the underlying programming languages of the programs in question, except for syntax highlighting.

2.2.2 Product Functions

The functions of SCN can be classified into following categories:

- Integration into the Bauhaus toolkit
- Interaction with users
- Source-code-related information
- Management of navigation semantics

The source code navigator is tightly integrated into the Bauhaus toolkit. It retrieves source-code-related information from IML data, which is generated by

back end tools; it cooperates with Gravis to switch between architectural views and source code views; it also accesses source files directly to get source code.

As a graphical user interface tool, the source code navigator responds to user actions reasonably.

Visualization of source-code-related information is central to the conception and implementation of the source code navigator. Such information passes the power of Bauhaus to users. The nature of source-code-related information is to be precisely specified in next section.

The source code navigator also tries to capture navigation semantics. The most basic form of navigation semantics is navigation history, which is similar to the browsing history of web surfing.

2.2.3 User Characteristics

The intended users of SCN are software developers or maintainers who want to understand programs to a level of source code.

2.2.4 Constraints

The functions of SCN are constrained by the facilities of back ends in the Bauhaus toolkit. Its current conception and implementation mainly concentrate on source-code-related information provided by current tools. In the future, if more kinds of source-code-related information are added, the source code navigator can also extend correspondingly to visualize them.

2.3 Specific Requirements

2.3.1 Integration with Gravis

The SCN is part of the Bauhaus toolkit, especially, it is to be fully integrated with Gravis, so to support program understanding on both architectural and source code levels in an integrated environment. Users shall be able to easily switch between architectural views and source code views of programs. The integration of SCN with Gravis is intuitive:

Nodes/edges in Gravis represent source code constructs of programs in question. By selecting or marking some node(s)/edge(s) in Gravis, the corresponding construct(s) is/are to be displayed in SCN.

On the other direction, by querying on some source code construct in SCN, Gravis is notified to display the corresponding node(s)/edge(s) to users.

2.3.2 Interaction with Users

SCN is a graphical user interface tool. The navigation in source code and source-code-related information is driven by user actions.

Use cases will be introduced in chapter 4.

2.3.3 Source-code-related Information

Except for source code itself, source-code-related information is also presented to users for navigation. There are several categories of source-code-related information to be visualized:

- Reference information (e.g. types, declarations, definitions)

- Class information (e.g. inheritance hierarchy, attributes, methods)

- Points-to and side-effects information
- Program slicing information
- Source code metrics

The source-code-related information is extracted and passed to the SCN by back end tools in the Bauhaus project. The SCN is just responsible for visualization of source-code-related information and navigation in source code.

Note that each category of source-code-related information is only meaningful for specific source code constructs. An obvious example is that class information is only applicable on object-oriented languages. This fact should be taken into consideration when designing the source code navigator.

To fully understand why the source code navigator is designed this way and how the source code navigator behaves under different circumstances, source-code-related information must be carefully classified and understood at first. Following subsections introduce each category of such information.

2.3.3.1 Reference Information

The most fundamental facility of source code navigation is to provide references of some source code constructs to some other constructs. Three kinds of references are considered in this thesis:

- Types
- Declarations
- Definitions

Examples for each kind of references are listed below.

A variable definition in strong-typing languages always refers to a type definition for the variable's type, as illustrated in the following example.

File "header.h":

```
......
typedef struct
{
      int x_coord;
      int y_coord;
} position;
......
```

File "main.c":

```
......
position start;
......
```

In the above example, the variable "*start*" in file "main.c" refers to the corresponding type definition of "*position*" in file "header.h".

An occurrence of a variable refers to the declaration of this variable. See the following example:

File main.c:

```
......
position start;
......
move(start);
```

……

In the example above, the variable "*start*" in the call to function "*move*" refers to the corresponding variable declaration.

A call to a procedure (function in procedural languages or method in object-oriented languages) creates a reference to the definition of this procedure:

File main.c:

```
……
int getNext()
{
    ……
}
......
int main()
{
    ……
    i = getNext();
    ……
}
```

In the example above, the function "*getNext*" is called inside "*main*", this call site refers to the function definition of "*getNext*" at some other position in source code.

Note that each kind of references may be, but not necessarily, crossing two different files. So it is common experience to spend a lot of time on looking up references of source code constructs in question for software developers or main-

tainers. By providing reference information, the most basic navigation requirement is satisfied.

2.3.3.2 Class Information

For object-oriented languages, class information is central to understand the systems being analyzed. Three kinds of class information are considered in this thesis:

- Inheritance
- Attributes
- Methods

An object-oriented system in reality normally contains hundreds or thousands of classes, which constitute a quite complex inheritance hierarchy. The source code navigator shall visualize the inheritance relationship to support users to navigate in the hierarchy easily. For instance, the following diagram shows a simple inheritance hierarchy.

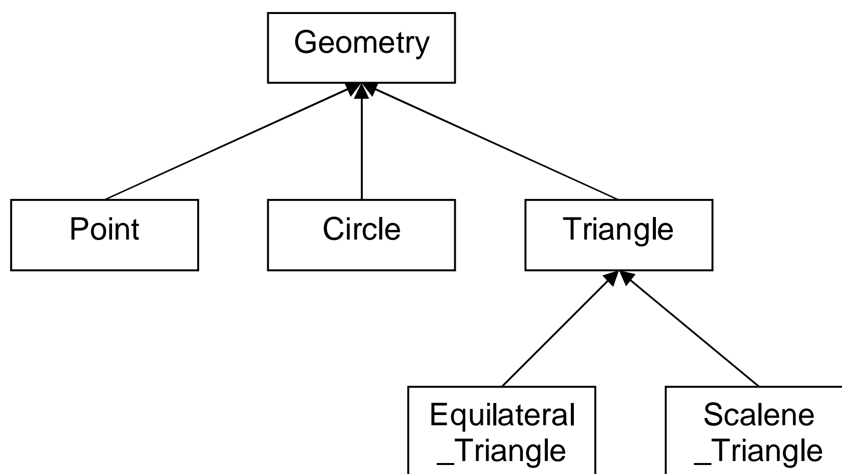

Fig. 2-2 Inheritance Hierarchy of Classes

The navigation in inheritance hierarchy follows either top-down direction or bottom-up direction. The top-down direction is from a super class to its subclasses. In the above example, when users are browsing the class `Geometry`, its children (the classes `Point`, `Circle` and `Triangle`) should be provided to users as requested. The bottom-up direction is from a subclass to its parent class or classes if multiple inheritance is allowed. In the above example, when users make inquiries about the class `Equilateral_Triangle`, its parent class `Triangle` should be provided to users.

The containment relation between classes and attributes/methods is to be visualized textually, too.

2.3.3.3 Points-to and Side-effects Information

Points-to and side-effects information is of great importance for program understanding. Without understanding of the nature of points-to and side-effects information, the source code navigator cannot be correctly designed and understood.

Both points-to and side-effects information depend on calling contexts heavily, so it is necessary to introduce calling contexts at first.

A calling context of a function describes a sequence of call sites that lead to this function. A same function can have multiple calling contexts. See the following example:

```
         ……
100      void g()
101      {
                 ……
110              f();
         ……
```

```
120  }
     ……
150  void f()
151  {
          ……
170  }
     ……
180  void main()
     {
          ……
190       if (…) g();
191       else f();
          ……
196       f();
          ……
200  }
```

In the example above, there are three calling contexts of function “*g*” from “*main*”, they are:

- *main* calls *g* at line 190; *g* calls *f* at line 110
- *main* calls *f* at line 191
- *main* calls *f* at line 196

A diagram to illustrate the calling contexts are shown in Fig. 2-3.

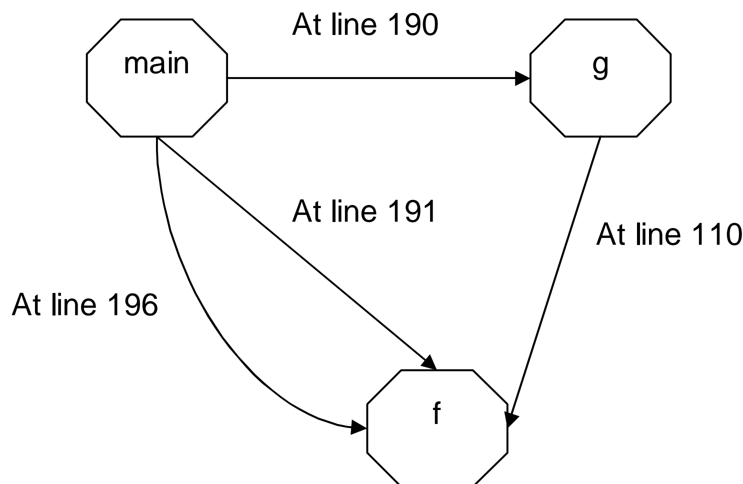

Fig. 2-3 Calling Contexts

The nodes in Fig. 2-3 represent functions themselves, while the edges represent function calls. The direction of edge arrows differentiates the caller and callee. For instance, the directed edge from "*main*" to "*g*" in the diagram indicates that "*g*" is called at line 190 in "*main*".

Note that different call sites of one function in some other function lead to different calling contexts, thus a calling context is identified by a sequence of 3-tuples constituted by the caller, the callee and the exact calling position.

Based on the introduction of calling contexts, some simple examples of points-to and side-effects information are given and explained below.

Programming languages like C supports pointer data types. A value of some pointer type contains an address to some data structure in memory. The program behavior often depends on the target data structure of pointers. The pointer targets may even be functions so that functions can be called through pointers.

The Bauhaus toolkit resolves pointers syntactically, so the source code navigator is able to provide targets of pointers to users. Points-to information is defined as all the pointer targets of some expression of pointer type.

Example of points-to information:

```
1       int a, b;
2       int *p;
3
4       void g()
5       {
6               int x;
7               a = 5;
8               b = 10;
9               *p = 0;
10              x = a+b;
11      }
12
13      void f1()
14      {
15              p = &a;
16      }
17
18      void f2()
19      {
20              p = &b;
```

```
21        }
22
23        void main()
24        {
25                f1();
26                g();
27                f2();
28                g();
29        }
```

In the example above, the expression at line 9, 15 and 20 are of a pointer type, so points-to information can be queried on these source code constructs.

Obviously, the pointer "*p*" at line 15 points to the variable "*a*"; the pointer "*p*" at line 20 points to the variable "*b*".

But the pointer "*p*" at line 9 can point to different targets, depending on calling contexts of the containing function "*g*". In a calling context with a calling site at line 26, the pointer "*p*" points to "*a*". In another calling context with a calling site at line 28, "*p*" points to "*b*".

Note that pointer targets may be memory locations allocated by some library routines. In this case, the targets themselves are identified by calling contexts to the memory allocation routines.

Functions and some other source code constructs often have side effects on object values. See the following example for a basic idea of side effects:

```
int a;
void f()
```

```
{
        a++;
}
void main()
{
        a = 10;
        f();
        println("a=%d", a);
}
```

In the example above, function "*f*" modifies the value of variable "*a*", thus the output generated by main is dependent not only on the assignment to the variable "*a*" in "`main`", but also on side effects on "*a*" by the function "*f*".

Assignment statements and some other source code constructs also have side effects on object values. The assignment "*a = 10*" in the example above also modifies the value of "*a*".

Bauhaus defines three categories of side effects. They are:

- Mayuse: an object value may be used by an expression
- Maydef: an object value is conditionally defined by an expression
- Mustdef: an object value is definitely defined by an expression

Note that the side effects may depend upon pointer values, if the object being used or defined are targets of pointers. The pointer values depend upon calling contexts in turn, as shown before in this subsection. Consequently, the side effects also depend upon calling contexts.

Also note that side effects can be passed along with calling paths. In the example above, the side effects of "`main`" must take into account the side effects of "`f`".

2.3.3.4 Program Slicing Information

Program slicing is basis technology for many other reengineering techniques. The source code navigator shall visualize program slicing to support users to navigate along by dependency graphs in source code.

There are two kinds of dependencies between expressions and predicates in programs:

- Data dependency: expression A depends on expression B, if A uses a value defined in B.
- Control dependency: expression A depends on predicate P, if P decides whether B is executed or not.

Example:

```
1     a = 1;
      ……
5     if (a>10)
6           i++;
      ……
10    b = a;
```

In this example, the value of "`b`" at line 10 depends on the assignment at line 1; the increment of variable "`i`" at line 6 depends upon the predicate at line 5.

All data and control dependencies inside a program constitute some dependency graphs. Program slicing is the process of traversing in the dependency graphs. A

program slice of some particular source code construct is defined as all the program points which are connected with this construct in dependency graphs.

According to the direction of dependencies, program slice of a particular source code construct is further divided into:

- Forward slice: follows forward all the data and control dependencies
- Backward slice: follows backward all the data and control dependencies

The source code navigator should support users to traverse along within dependency graphs and to visualize program slices as summary views.

2.3.3.5 Source Code Metrics

Source code metrics are overall measurements of certain system attributes relating to source code. A good example is the line number of each function definition. Other metrics supported by Bauhaus include the maximal syntactic nesting of statements in each function, and Cyclomatic McCabe Complexity.

As seen in the two examples mentioned above, source code metrics normally don't adhere to some particular source code constructs, but provide an overview on global system attributes. A mechanism is to be designed to visualize such source code metrics.

2.3.4 Navigation Semantics

Besides the static and syntactic source-code-related information, the source code navigator tries to provide a dynamic and semantic view of navigation activities: to capture the semantics of navigation.

Source code navigation implies two categories of functions: visualization of source code and source-code-related information, and navigation process itself

as mental activities. Valuable information is embedded in the navigation process that can support users to better control the program understanding progress.

The most basic form of navigation semantics is nowadays common in many tools. The best example is web surfing. Web browsers normally remember users' browsing history in a sequential manner. Users are able to go backward or forward in the sequential browsing history. Such form of navigation semantics is defined as navigation sequence in the thesis. There are two kinds of navigation history concerned in this thesis.

- Navigation sequence
- Navigation map

Navigation map is a sophisticated version of navigation history. It is non-linear and characterized by a directed graph whose nodes represent consulted source code constructs and whose edges represent trivial navigation sequence.

The navigation sequence and navigation map are modeled in detail in chapter four.

2.3.5 Miscellaneous

Miscellaneous functions and properties should be taken into consideration during conception and implementation of the source code navigator. Some important ones are listed below.

Syntax highlighting greatly improves user experience of source code navigation. Remember that SCN is independent with underlying programming languages of programs being inspected, but syntax highlighting must be language dependent. Caution should be taken when designing software architecture for SCN, so as to implement syntax highlighting without sticking SCN to particular programming languages.

Macro is a very useful feature in some programming languages like C, but it brings great difficulties regarding source code navigation. Source files can contain macros, but the IML data reflects programming text after macro expansion. A mechanism is to be designed to support macros in source code navigator.

Searching for a specific string pattern in source text is common requirement to text processing tools. The source code navigator should support basic searching functions.

3. Tools Evaluation

There are already several tools for source code navigation on the market. In this chapter, two representative tools among them are selected for evaluation, to gain an overview on the state of the art of source code navigation.

3.1 Source-Navigator

3.1.1 Introduction

Source-Navigator is a source code analysis and comprehension tool released under the GNU public license (See [SN-Web]). It was developed by *Red Hat, Inc.* originally and moved to *SourceForge* (see [SF-Web]) in Apr. 2002.

Source-Navigator is widely used on Linux platforms and the current version is 5.1.

Source-Navigator is an all-in-one tool for a list of tasks during software developing and maintaining life circle. It incorporates source code analysis, source code editing, architectural navigation and source code navigation into one a graphic framework for understanding and reengineering large or complex software products.

With Source-Navigator, users can

- Analyze dependencies between source modules
- Edit source code
- Invoke a compiler or a debugger to compile or debug the program
- Display relationships between classes and functions and members
- Display call trees

Currently, Source-Navigator supports C, C++, Java, Tcl, FORTRAN and COBOL.

For more information about Source-Navigator, see its web page at *http://sourcenav.sourceforge.net/*.

The navigation features of Source-Navigator are discussed in following sections. Some screenshots are taken from the online User's Guide of Source-Navigator (see [SN-Web]).

3.1.2 Windows Layout

The user interface of Source-Navigator consists of two top-level windows. They are:

- Symbol browser window
- Source viewer window

The following screenshot shows the basic windows layout.

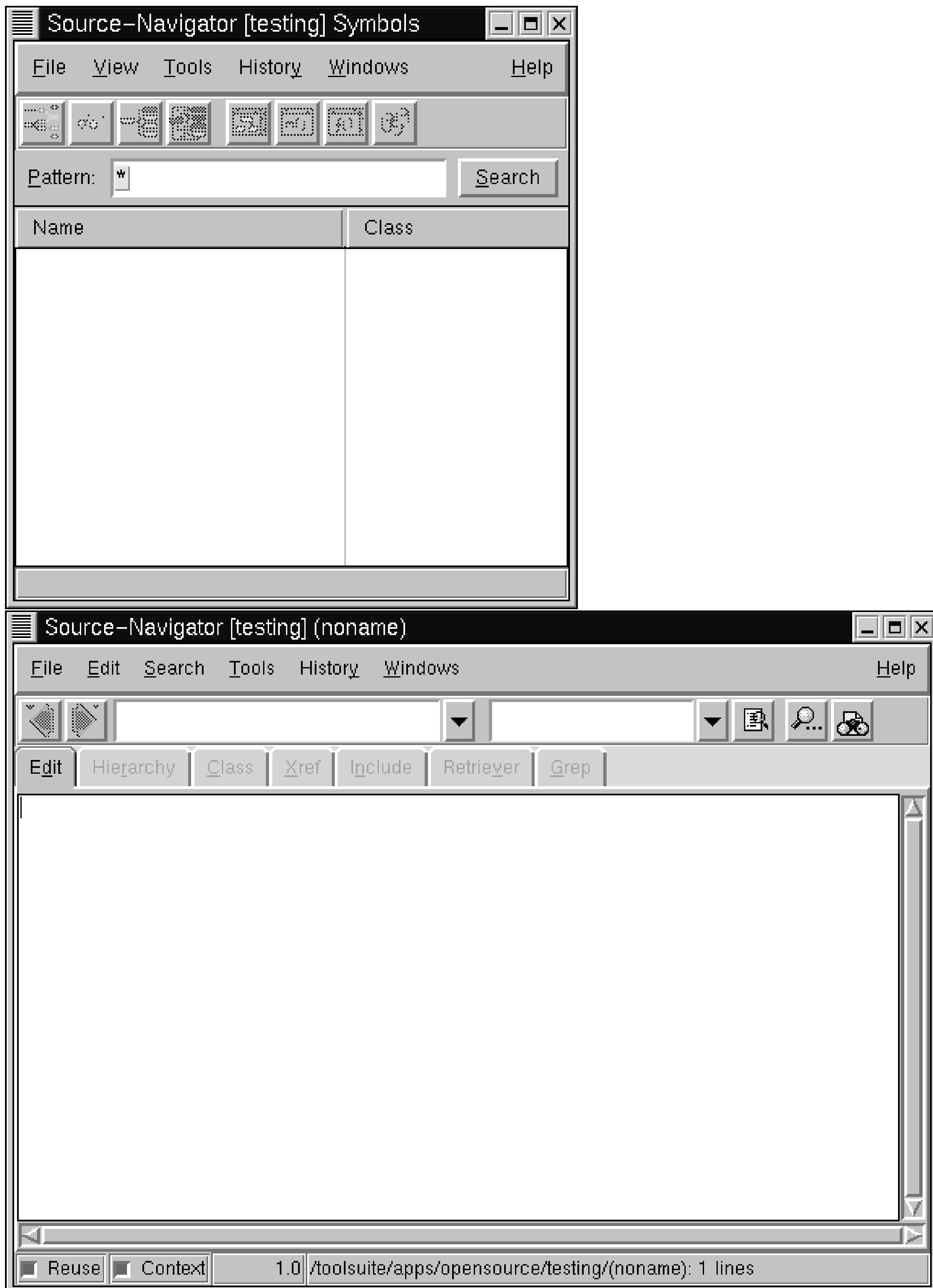

Fig. 3-1 Windows Layout of Source-Navigator

3.1.3 Symbol Browser Window

The symbol browser provides a tree view of the symbols within the program. Top level nodes of the tree view represent source files and the containing directories. Clicking on the icon to the left of the file or symbol name expands the node to a tree view of the file or symbol and its members like variables or functions, as illustrated below:

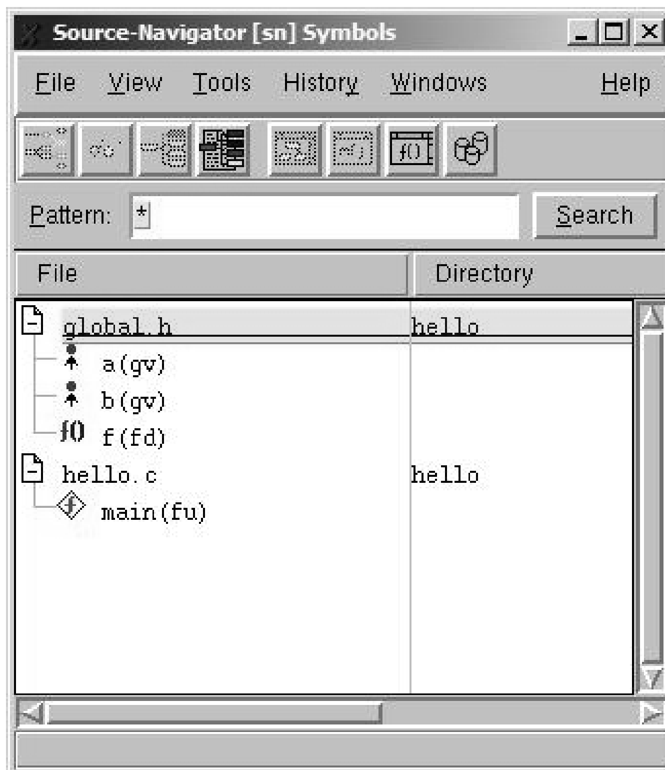

Fig. 3-2 Symbol Browser of Source-Navigator

3.1.4 Source Viewer Window

The source viewer window combines several tools in one window. The tools are:

- Editor
- Hierarchy browser

- Class browser
- Cross-reference browser
- Include browser
- Retriever
- Grep

Source code of programs is shown in the editor for viewing and editing, with supports for pattern searching and syntax highlighting, as illustrated below.

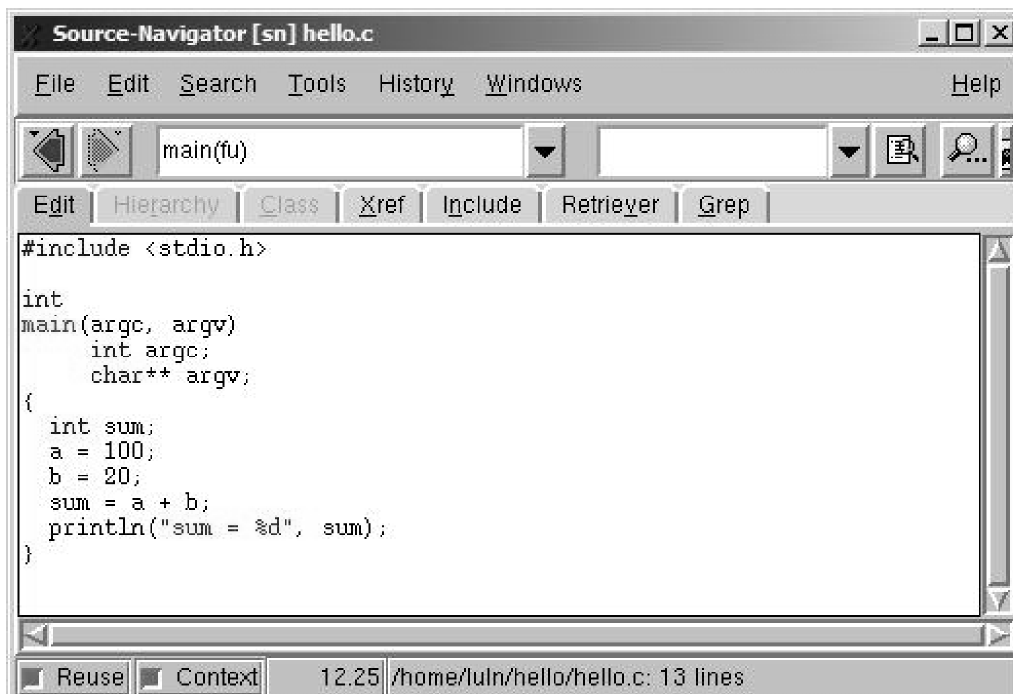

Fig. 3-3 Editor of Source-Navigator

The hierarchy browser visualizes inheritance relationships of classes. It can display either entire class hierarchy of the software project or base classes, super classes and subclasses of a selected class. The following figure shows an entire class hierarchy.

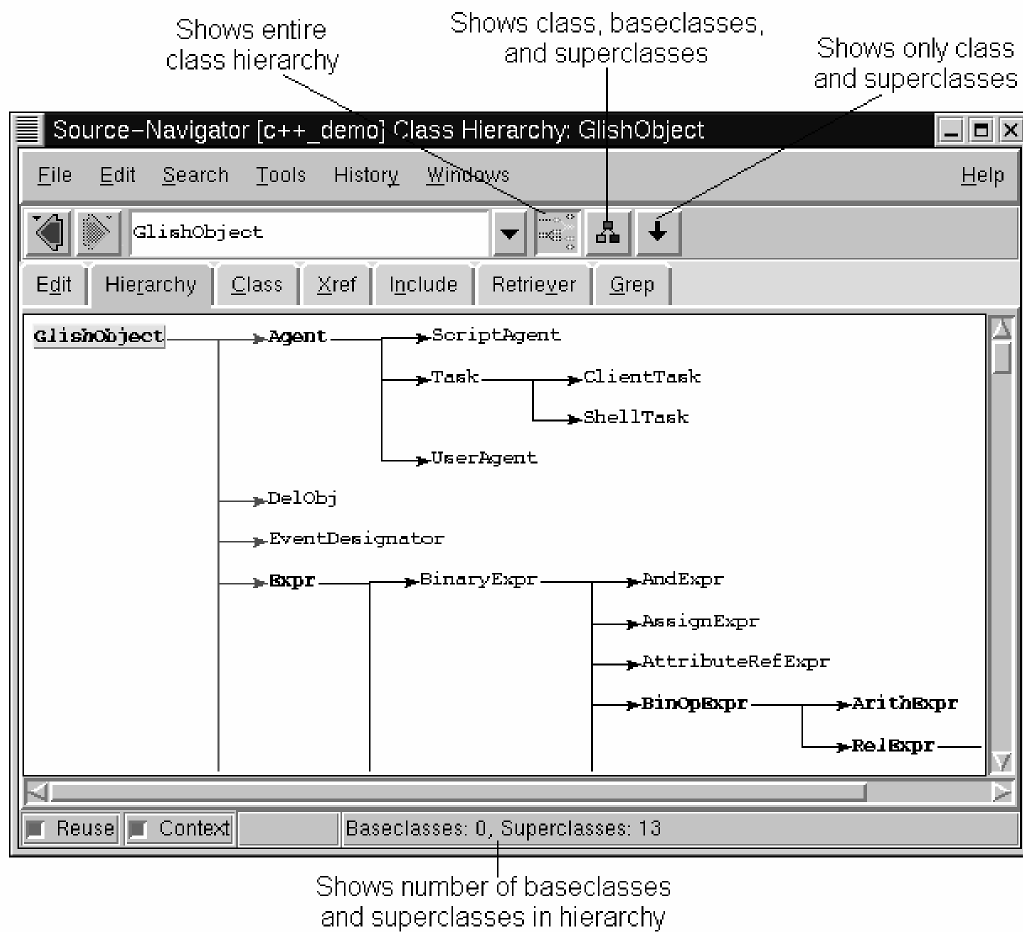

Fig. 3-4 Hierarchy Browser of Source-Navigator

For object-oriented programming languages, the class browser enables users to browse class hierarchies, access levels, and member types.

For traditional languages such as C, the class browser enables users to see the members of structures and common blocks.

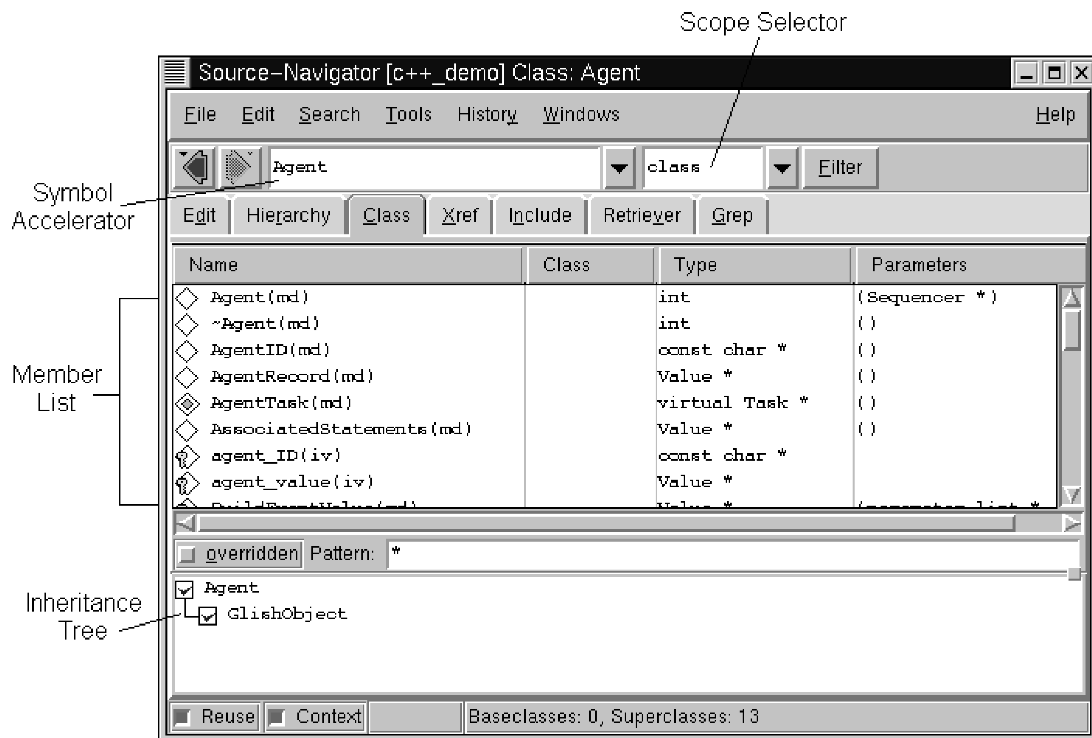

Fig. 3-5 Class Browser of Source-Navigator

The hierarchy browser and class browser thus provide a complete view of class information for object-oriented programs.

The cross-reference browser is responsible for reference information introduced in chapter 2. It shows, for a given symbol, all the other symbols it refers to, and all the symbols that refer to it. It creates tree diagrams that show refers-to and referred-by relationships, such as call trees of selected functions. The following figure shows a simple refers-to hierarchy.

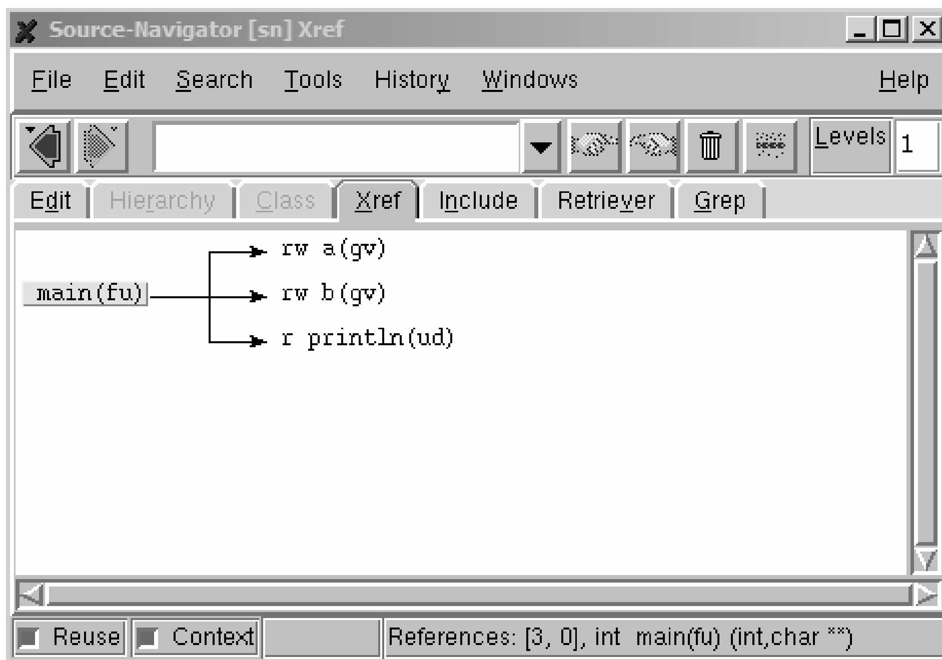

Fig. 3-6 Cross-reference Browser of Source-Navigator

The include browser is very helpful for programming languages which support inclusion of other source files. It displays the inclusion hierarchy in tree diagrams.

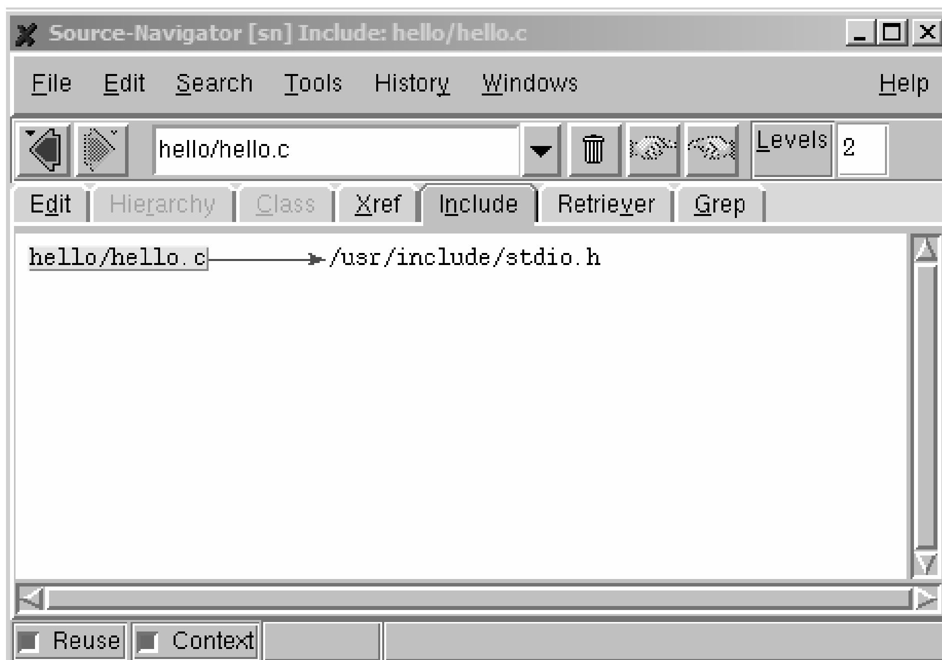

Fig. 3-7 Include Browser of Source-Navigator

Retriever and Grep support users to search for text pattern in symbol names and source files, respectively.

3.2 CodeSurfer

3.2.1 Introduction

CodeSurfer is a powerful source code analysis and navigation tool produced by *GrammaTech, Inc.* (See [GA-Web]) It shares many similarities with the Bauhaus toolkit. It understands pointers, side effects and complete call graphs even when indirect function calls through pointers are used.

CodeSurfer is a commercial product and its current version is 1.7. It runs on Unix, Linux or Windows platforms.

3.2.2 Windows layout

There are four types of main top-level windows:

- Project viewer
- File viewer
- Call graph viewer
- Property windows

Each type of top-level windows is introduced in following sections. Some screen-shots are taken from the CodeSurfer User Guide and Technical Reference (See [CS-Guide].

3.2.3 Project Viewer

The project viewer is like the symbol browser of Source-Navigator, it provides an overview on software being investigated.

Program elements – source files, object files, library files, functions, variables, functions calls, and many others - are organized in a tree in the project viewer. In the following screenshot, the node for source file “hello.c” is expanded to its functions, variables and even included files.

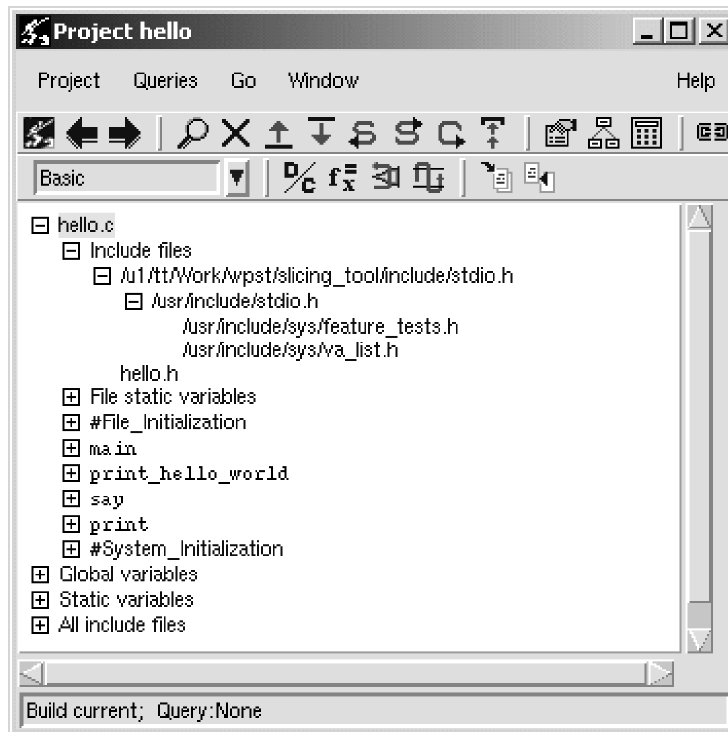

Fig. 3-8 Project Viewer of CodeSurfer (a)

The functions can be further expanded to variables and function calls that occur within it, executable code and dependence points.

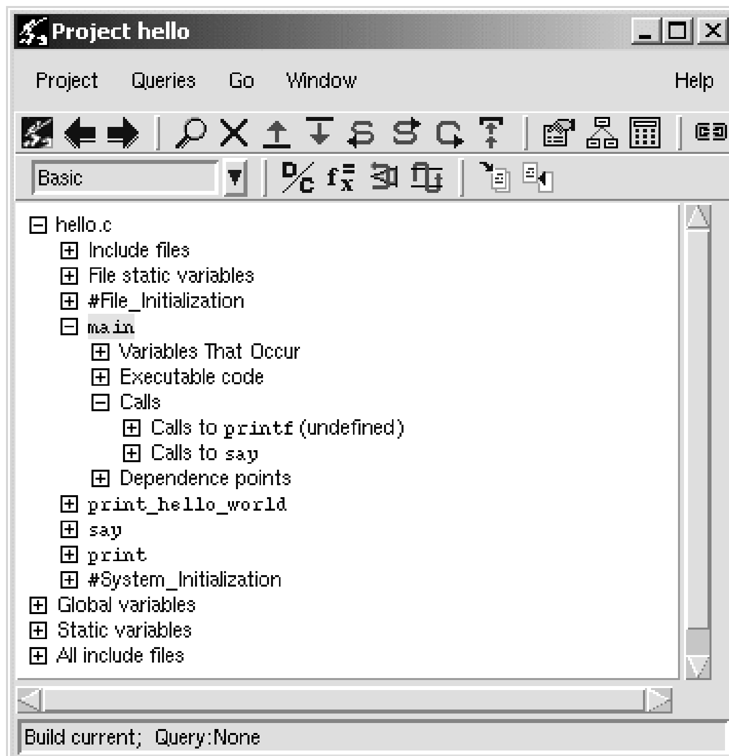

Fig. 3-9 Project Viewer of CodeSurfer (b)

The project viewer provides complete hierarchical views of the project at a higher-level of abstraction than the source code. Parts of its views are implemented by Gravis in the Bauhaus toolkit and other parts need to be implemented in SCN.

3.2.4 File Viewer

File viewer displays source files and support invocation of queries for source-code-related information. Text in file viewers is read-only.

hello.c

File Edit Functions Queries Go Window Help

```c
/* Demo CodeSurfer */
#include <stdio.h>
#include "hello.h"

#if DEBUG
# define DONE "Debugging Done."
#else
# define DONE "Done."
#endif /* DEBUG */

static int debug = 0; /* !=0 for debug output. */
int count = 0;        /* # of messages printed. */

int main(void)
{
  PARITY english = 0;    /* Parity of #msgs in English. */
  PARITY french = 0;     /* Parity of #msgs in French. */
  say("Hello World", &english);
  say("Bonjour", &french);
  printf("Total hellos=%d\n", count);
#ifdef DEBUG
  print_hello_world();
#endif
  count = 0;
  say("Goodbye", &english);
  say("Au Revoir", &french);
  printf("Total goodbyes=%d\n", count);
  printf( DONE );
}

/* Print "Hello World". */
void print_hello_world(void) { print("Hello World"); }

/* Print every other MSG in a given LANGUAGE. */
void say(char *msg, PARITY *language)
{
  if (debug) print("enter say");
  if (!*language) print(msg);
  *language = (PARITY)(!*language);
  if (debug) print("exit say");
}

/* Print MSG and increment COUNT. */
void print(char *msg) { printf("%s\n", msg); count++; }
```

Fig. 3-10 File Viewer of CodeSurfer

Invocation of queries results in a context menu which gives users further navigation possibilities based on the selected program points. See the following figure for an example of context menu.

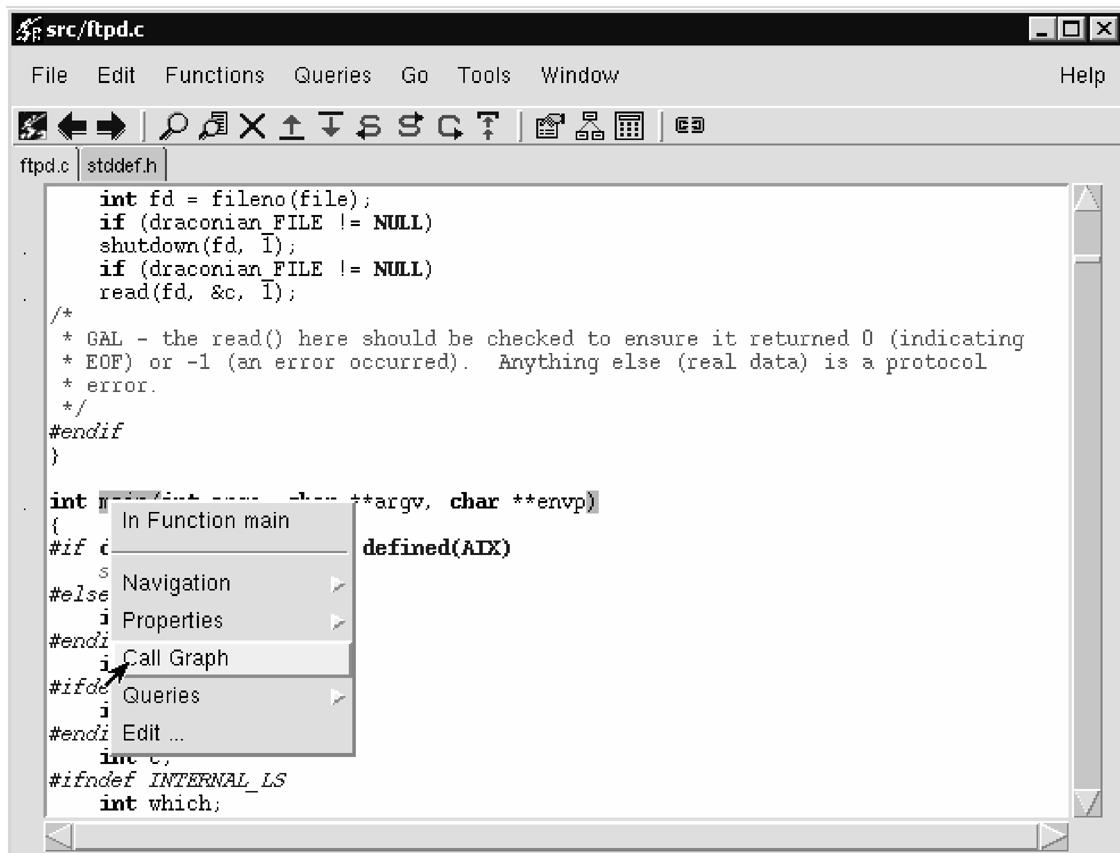

Fig. 3-11 Context Menu of CodeSurfer

3.2.5 Call Graph Viewer

Call graph viewers show calling relationships between functions as directed graphs. The call graphs model calling contexts discussed in chapter 2, but the visualization of pointer targets and side effects in CodeSurfer doesn't explicitly take calling context into account.

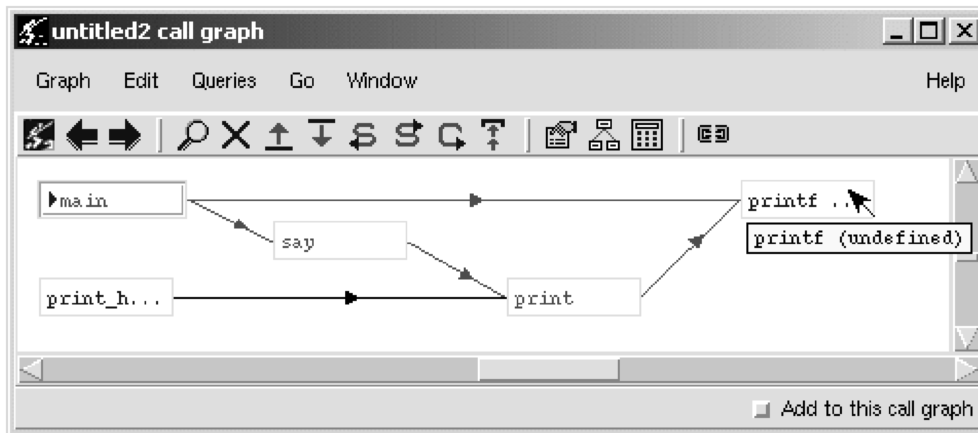

Fig. 3-12 Call Graph Viewer of CodeSurfer

3.2.6 Property Sheets

Each program element has an associated set of properties, which can be shown in property sheets. The property sheets are invoked on selected program points in project viewers or file viewers.

Different forms of property sheets are:

- Top-Level source file
- Include file
- Include file instance
- Compound object file
- Archive or library
- Variable
- Function definition
- Program point

- Program point set
- Call sites
- Calls
- Call
- Indirect call sites
- Indirect call

A property sheet for variables is illustrated below:

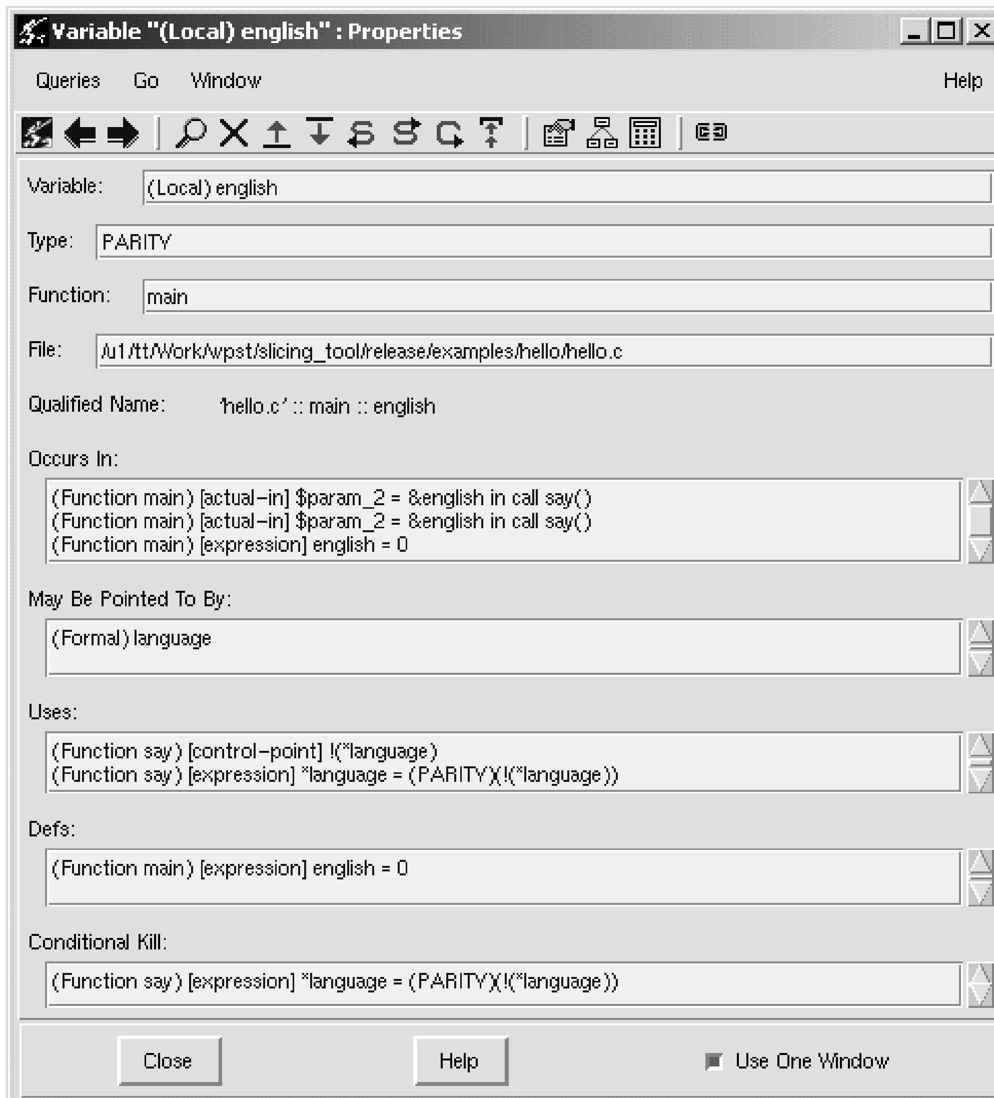

Fig. 3-13 Property Sheet of CodeSurfer

4. Conception of SCN

In this chapter a source code navigator is designed to satisfy the software requirements specified in chapter 2. To understand why it is designed this way and how it behaves in different scenarios, the software requirements specification must be understood at first. To compare the design with similar tools on the market, chapter 3 is recommended for reading.

In the conception of source code navigator, emphasis is placed on user interfaces for source-code-related information. An XML application is designed to capture navigation semantics.

4.1 Windows Layout

SCN is designed as a pure textual tool. All information is visualized in text, organized by basic graphical widgets such as tables, trees etc.

The basic windows layout heavily depends on the architecture of Bauhaus. Remember that source-code-related information is extracted by other tools in the Bauhaus toolkit and stored in an intermediate format, and that architectural views of software being inspected are visualized by Gravis, the source code navigator is thus only responsible for visualization of textual information, that is, source code and source-code-related information.

Benefited by this clean design of system architecture, the GUI of SCN can be designed in a simpler manner than the GUIs of Source-Navigator and Code-Surfer. In particular, less top-level windows are used in SCN.

There are only two kinds of top-level windows of SCN:

- Browser window
- Viewer window

The browser window is used to visualize overall structure and metrics of source code. Objects in the browser window are program elements of a higher-level of abstraction than source code. For instance, files, functions, methods etc. are to be presented here.

The viewer window is used to display source code and source-code-related information. Most navigation activities happen in this window.

At any time, a running instance of SCN contains only one browser window and one viewer window. This design makes the screen be clean and simplifies the management of windows. If multiple files are to be inspected simultaneously, a multiple document interface approach is used, see section 4.1.3.

The following figure is a screenshot of a running session of SCN.

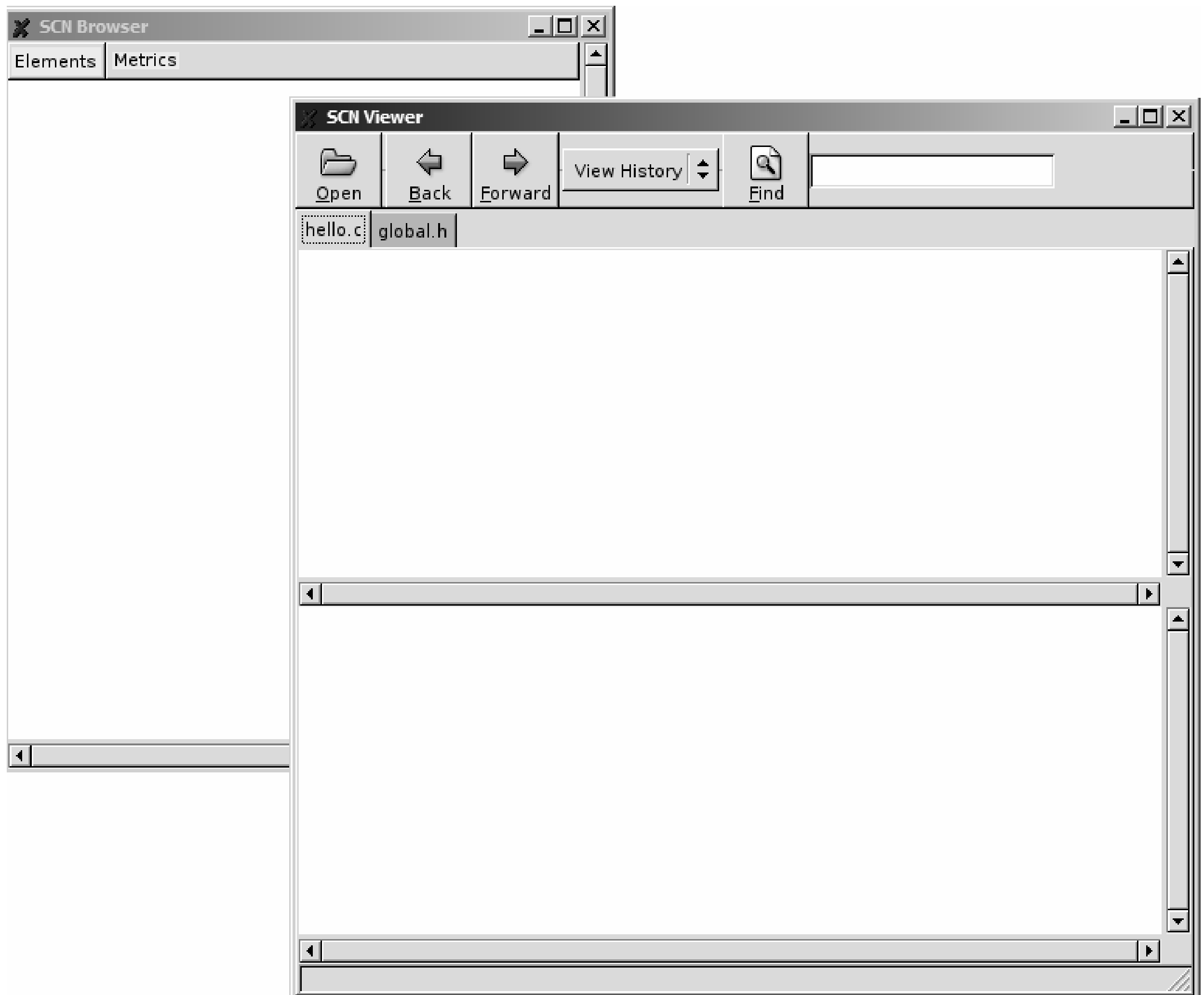

Fig. 4-1 Windows Layout of SCN

Besides the two top-level windows, context menus are heavily used. Context menus are popped up when users invoke a query on some program elements in either the browser window or the viewer window. Items of context menus are shortcuts to different functions applicable to the program element being queried, for instance, to invoke for specific source-code-related information. Context menus can be nested.

4.1.1 Browser Window

The browser window implements a generic tree view with an associated table.

The tree view on the left of the browser window presents program elements at a higher level of abstraction than source code and relationships between these elements. For instance, the tree view's top level nodes list directories, files, packages or classes of a software project. Child nodes can present subdirectories, functions, child packages, methods of its containing node.

The table on the right is entitled as "Metrics" because it visualizes source code metrics. Each row of the table corresponds to a node of the tree view on the left, and content of each cell shows the result of measurement for the program element represented by the corresponding node of the tree view.

When the tree view is folded or unfolded, the table also changes correspondingly to make the matching of tree nodes and table rows be consistent.

A screenshot of a simple tree view and its associated metrics table is illustrated below. Note that only the metric of line number is displayed. It's only for illustration and doesn't mean that the table cannot display more than one column at the same time. More columns can be easily added as requested.

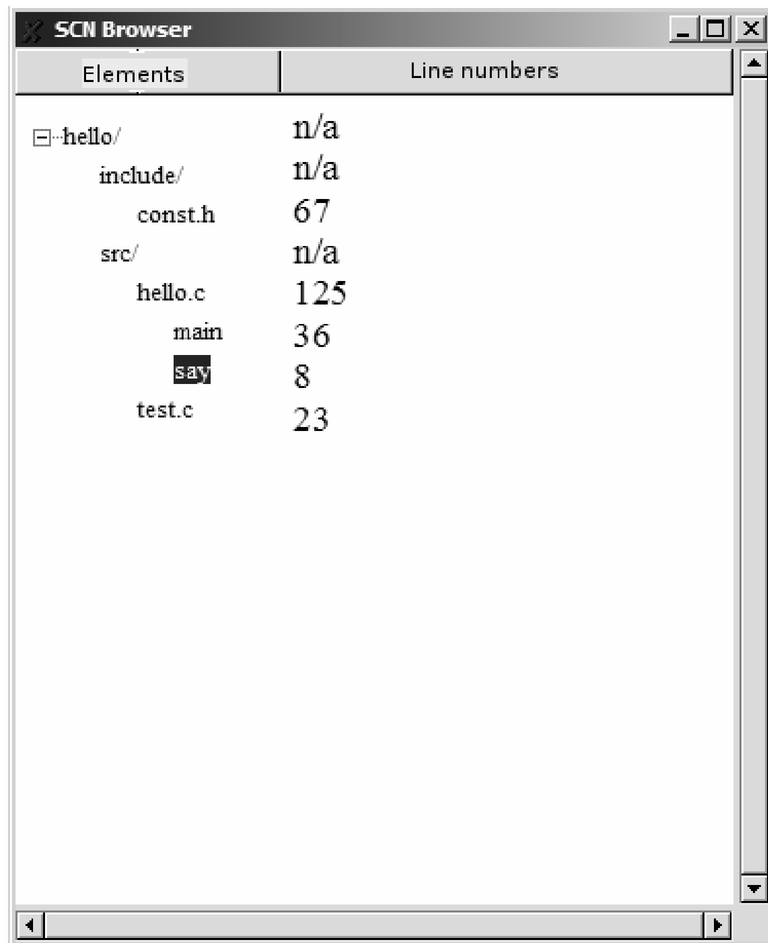

Fig. 4-2 SCN Browser Window

By double-clicking a node in the browser window, the corresponding source code is to be displayed in the viewer window.

4.1.2 Viewer Window

The viewer window shows source code and source-code-related information to users. By responding to user actions reasonably, the viewer window supports navigation at source code level.

The viewer window consists of following visible components which constitutes a widget hierarchy:

- A toolbar
    - Buttons, dropdown menus, text boxes for miscellaneous functions
- A multi-page window
    - Each page in the window displays one source file.
- A status bar

A viewer window is illustrated in Fig. 4.3.

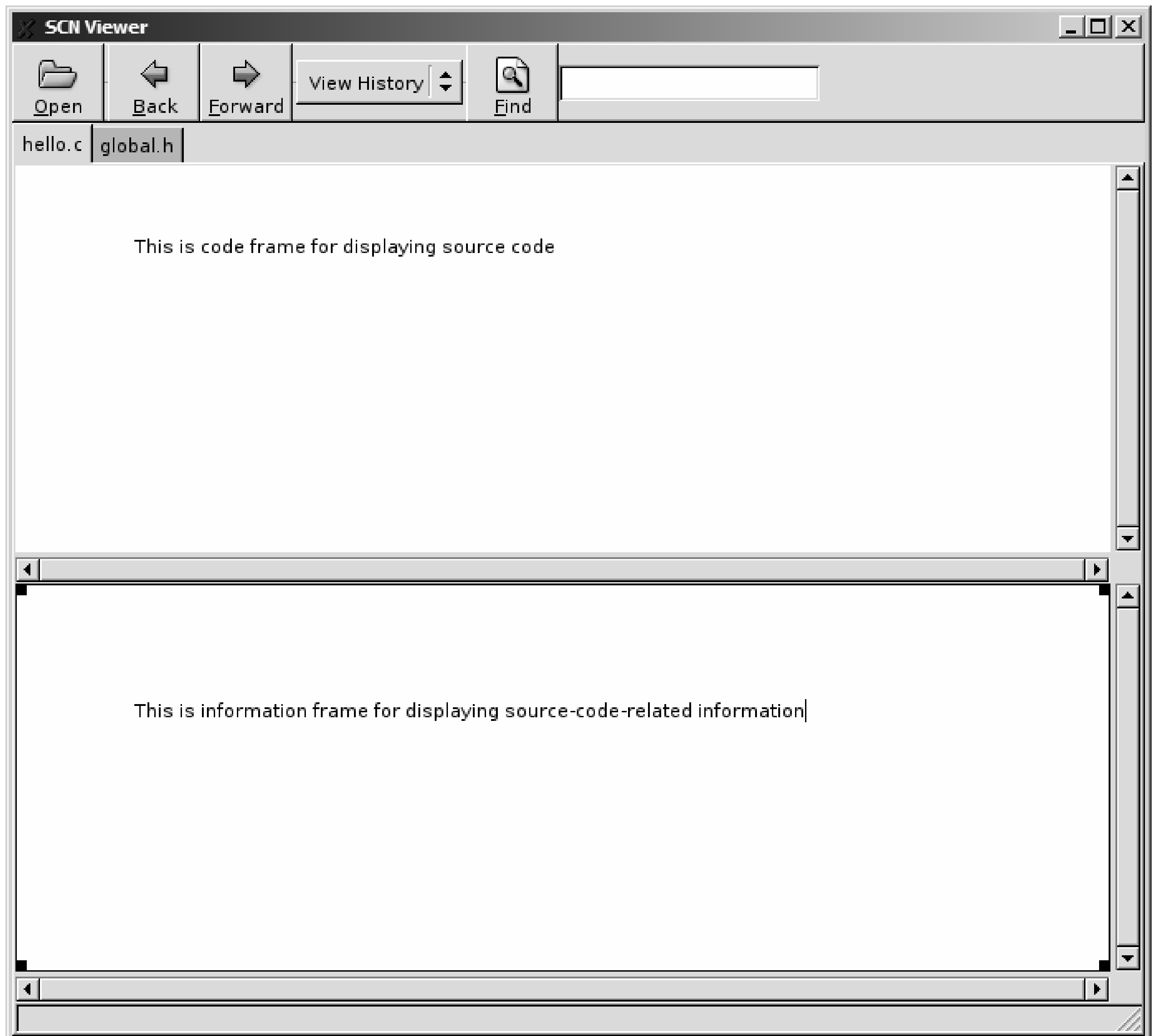

Fig. 4-3 SCN Viewer Window

On the toolbar, a button with label "Open" is for opening of source files. Buttons "Back" and "Forward" are used to go backward or forward in the view history. To jump to a previously investigated program point, the dropdown menu with label "History" may be more convenient than the "Back" and "Forward" buttons. Text patterns can be entered in the text box on the most right, then clicking the button "Find" causes searching in the background and jumping to the nearest matching string.

Because there is exactly one viewer window at any time of a running SCN session, a mechanism is required to display multiple files simultaneously.

The viewer window has a MDI (Multiple Document Interface) implemented by a multi-page window. Pages can be opened or closed independently. Each page is devoted to a source file and has a title which is the name of the corresponding file. At any time, exact one page is active and visible in the foreground. In the example above, the active page displays a file "hello.c", and there is another page for a file "global.h" which is currently inactive. Clicking a page title causes the clicked page to become active.

When users invoke SCN from Gravis for displaying some source code construct, but SCN has already been running, this source code construct will be opened in a new page and the page becomes active. So it is possible that several pages are devoted to a same file. But it doesn't matter because SCN doesn't support editing of text.

A new page can be created not only from Gravis, but also by using the button "Open" in the toolbar, or by SCN automatically during navigation.

A page for displaying source code is divided vertically into two frames:

- Code frame
- Information frame

These two kinds of frames are read-only text views.

Code frames display source code of files and information frames displays source-code-related information. Contents of the two frames in the same page shall be consistent.

Sizes of the two frames can be adjusted to increase the view space for source code or source-code-related information.

At the bottom of the viewer window is a status bar. It displays for example the line and column number of cursor in code frames.

4.2 User Actions

As a user interface tool, user actions shall be modeled. Only two types of user actions are recognized by the source code navigator:

- Follow: User follows a link.
- Query: User queries a program element.
- Select: User selects an option from a menu.

"Follow", "Query" and "Select" represent abstract user actions actually. That is, they can be implemented as different user operations on different platforms, or even customized by users. In the default implementation, mouse double left-clicking is interpreted as "Follow", mouse right-clicking as "Query" and mouse left-clicking as "Select". The interpretation of concrete user operations to abstract user actions is illustrated below.

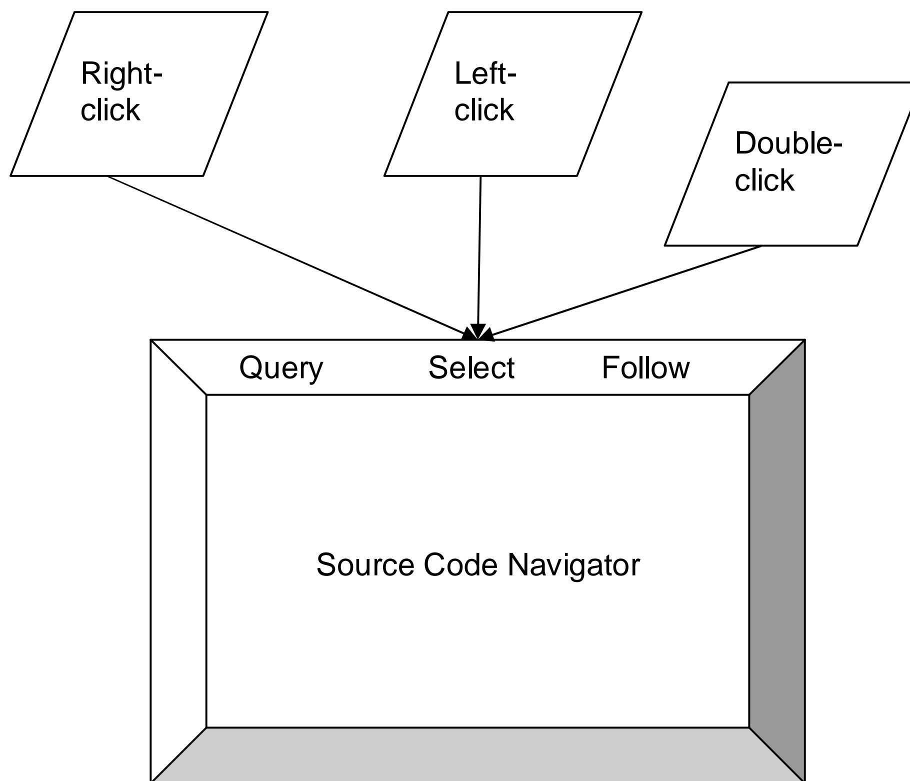

Fig. 4-4 User Actions

The result of “Follow” is a jump to the target position of the clicked link. It is common in web surfing.

“Query” of program element results in pup-up of a context menu constructed dynamically for the program element in question.

“Query” generally happens in the code frame, while “Follow” generally happens in the information Frame.

“Select” happens on context menus or toolbar. When users select an item in a context menu, the operation associated with the select item is invoked, and the result of the invocation is displayed.

4.3 Source Code Construct Selection

Concrete user operations on SCN, for example, mouse clicking, happen at a specific position in text, not at a range of text. But the targets of abstract user actions are source code constructs which normally expand upon a range of text.

The concrete user operations are easily interpreted to abstract user actions, but converting text position to text range isn't a trivial task, due to the fact that a text position normally belongs to several source code constructs.

Example:

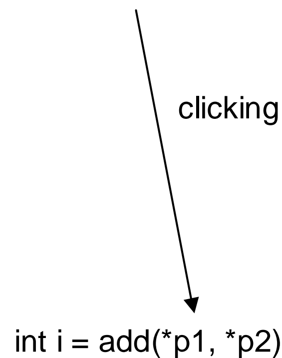

Fig. 4-5 Overlapping of Source Code Constructs

In this example, clicking on the character "`1`" implies several source constructs that are overlapped:

- The pointer "`p1`"
- The pointer target "`*p1`"
- The function call "`add(*p1, *p2)`"
- The whole assignment statement

Which one is the intended target construct by this clicking?

Another problem with source construct selection lies in that not all source code constructs have source-code-related information to be queried. Only those constructs relevant in Bauhaus is concerned in SCN. Invocation of wrong operations on source constructs must be prevented.

In conclusion, a mechanism is required to support users to select source code construct easily and consistently.

The method designed in this thesis is “incremental and interactive selection” of source code constructs, which is illustrated in the diagram as follows.

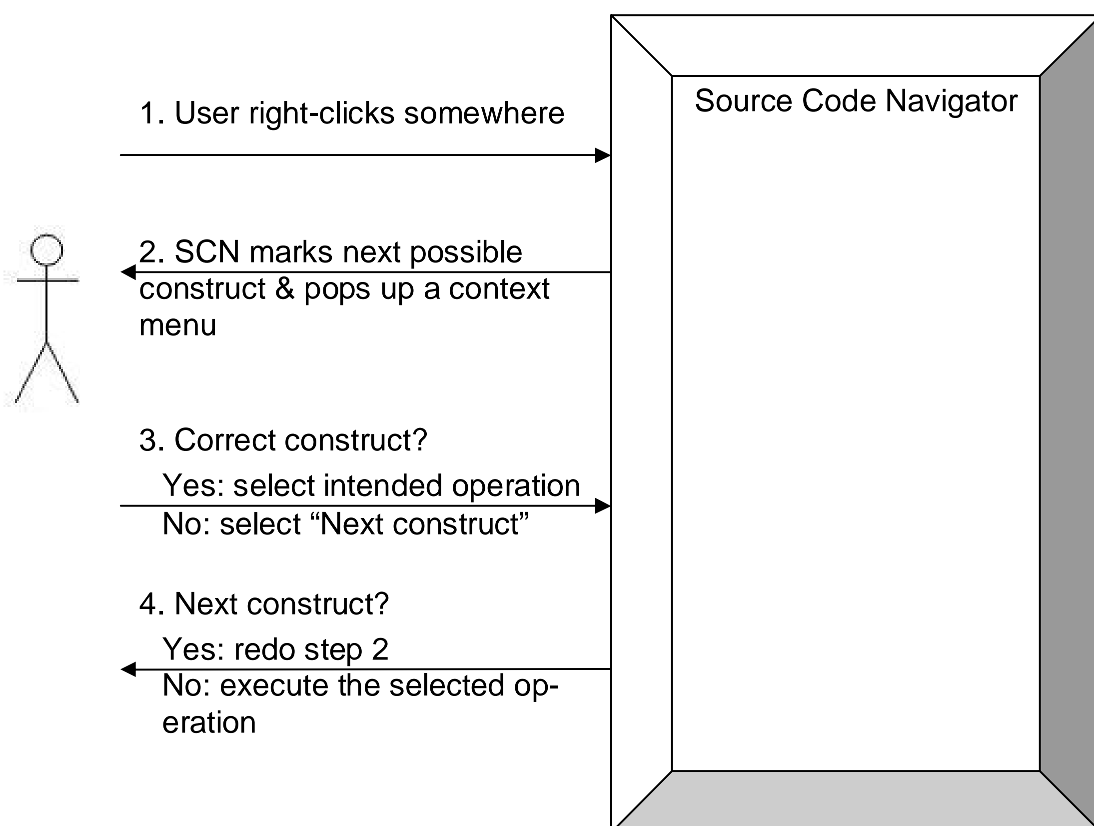

Fig. 4-6 Incremental and Interactive Selection of Source Code Constructs

The key idea behind this method is to let the user and SCN cooperate with each other to go towards the intended source construct incrementally.

The user starts a source construct selection process by right-clicking somewhere in code frame, which is modeled as step 1 in Fig. 4-6.

In step 2, SCN responds by marking the next possible range of source construct around the clicked character and popping up a context menu for the currently marked source construct. In the context menu there exists an item “Next construct”.

If the marked construct presented to the user is the intended one, the user select her/his intended operation on this source construct in the context menu, for example, to show its points-to information. If it’s not the intended source construct, the user selects “Next construct” from the context menu.

Obviously, SCN then executes the selected operation if the user gets her/his intended source construct finally; or redo step 2 to mark a larger range of text and provide a context menu containing “Next construct” to the user again.

The term “next possible construct” needs some explanations. There are several scenarios of “next possible construct”:

- The smallest range of text for a source construct around the clicked character, when the user just initiates a selection process
- A larger range of text for a source construct than the construct marked before, if it exists
- Nothing, if no larger range of text for a source code construct exists. In this case, the selection process is terminated without selection of any operation

4.4 Context Menu Construction

Notice that different types of source code construct have different set of applicable operations. For example, it is meaningless to request points-to information for an expression of non-pointer types. So context menus popped up to query requests are to be constructed dynamically, bases on the types of selected source construct. Only applicable operation for the construct in question will be listed in the context menu.

4.5 Source-code-related Information

4.5.1 Reference Information

A prototype for reference information is to be reviewed in this subsection. Reference information is accessible through context menu item “Refers to”. Assume that a user is interested in variable “*max*” at line 10 in file “hello.c”. She/he selects the identifier “*max*” through incremental and interactive source construct selection and raises a query of it.

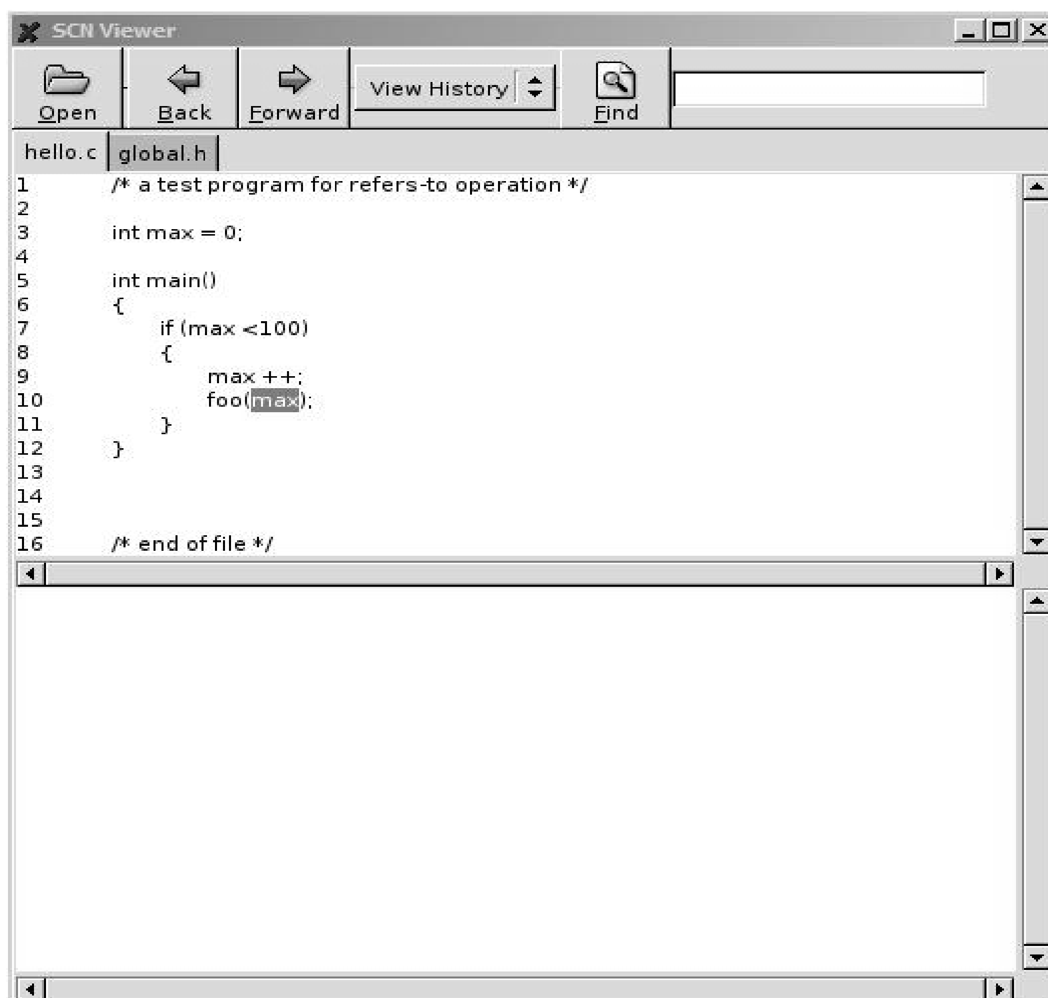

Fig. 4-7 Reference Information in SCN (a)

The query is replied with a context menu containing an item “Refers to”. The user selects this item, so SCN retrieves reference information for the variable “`max`“ from IML data and finds that it refers to the variable definition at line 3. So SCN navigates to the position where “`max`“ is defined.

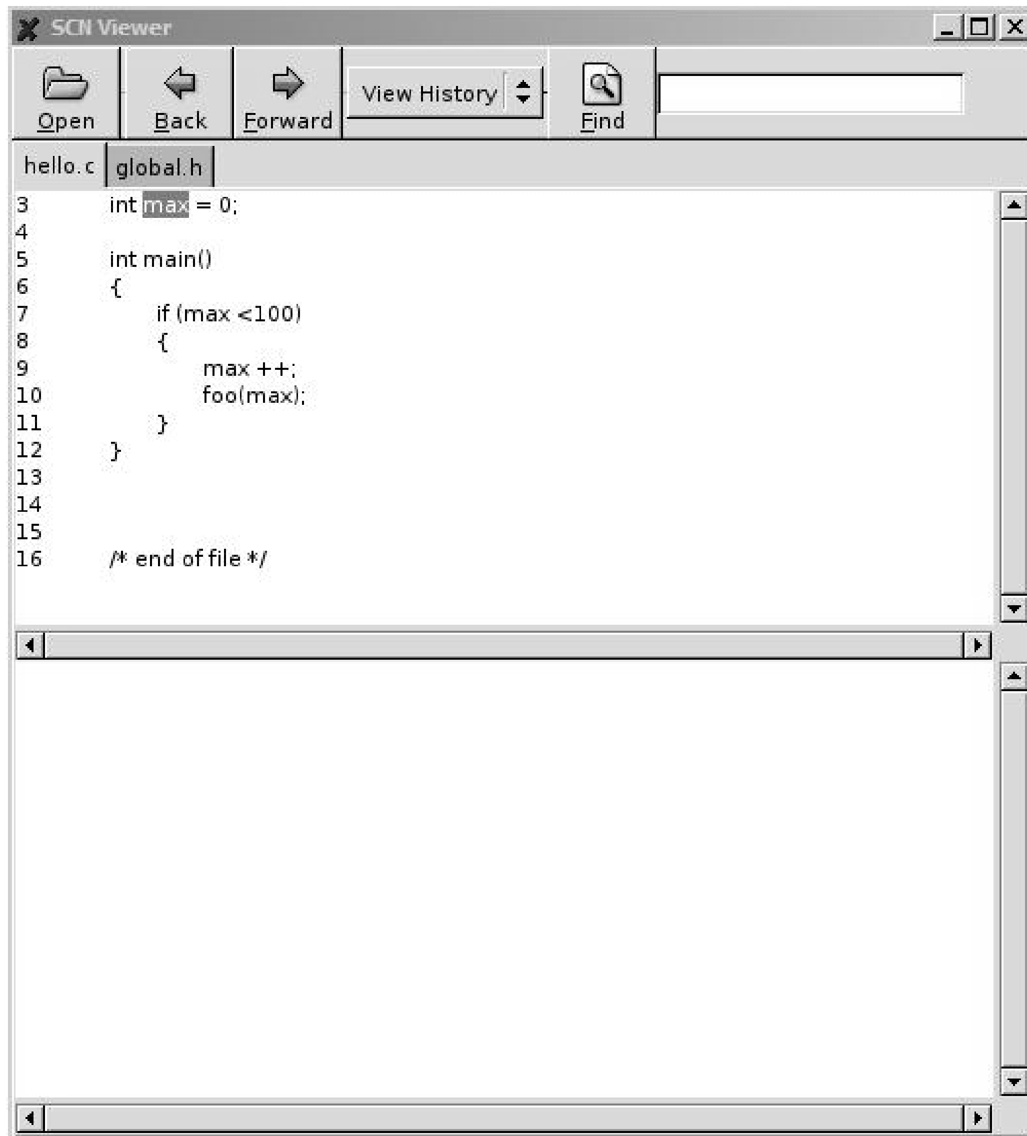

Fig. 4-8 Reference information in SCN (b)

In this very simple example, the definition and use of variable “`max`“ are very near to each other. But in real projects, the referred target construct normally is in

some other file among hundreds to thousands of files. So navigation in source along with references supported by SCN is quite useful.

For example, the function call of “*foo*“ in the example above may refer to a function definition in file “lib.c“. If the user raises a query of “*foo*“ instead of “*max*“ , then a new page for “lib.c“ will be created and become active.

4.5.2 Class Information

In contrast to reference information, visualization of class information doesn’t cause navigation in source directly.

Assume that a user queries which classes inherit class “*Triangle*” by selecting “Inherited by” in context menu. It results in the following window.

Fig. 4-9 Class Information in SCN

The inheritance relationship is visualized as text in the information frame. Subclass names in the information frame are implemented as links. Following such links the definition of the corresponding class will be displayed in the code frame of a new page.

This is one of the navigation rules of SCN that is worth an explanation. Navigation steps normally follow this sequence:

1. The user raises a query of some specific source construct;

2. SCN displays source-code-relation information in the information frame of the current page;

3. The user follows a link in the textual source-code-relation information;

4. SCN creates a new page and displays the source construct corresponding to the link in source frame of the new page

4.5.3 Calling Context

Based on the specification in chapter 2, calling context is visualized in strings of the following grammar in BNF notation:

Calling_Context ::= Function_Name

Calling_Context ::= Calling_Context → number Function_Name

An instance of calling context is:

main →27 f →90 g →20 k

This instance of calling context says that it starts from function "`main`", which calls function "*f*" at line 27, and "*f*" calls "*g*" in turn at line 90, finally "*g*" calls "*k*" at line 20.

When displayed in SCN, functions names in calling context strings are implemented as links to the corresponding function definitions, and numbers following arrows between function names are implemented as links to the corresponding function calls.

4.5.4 Points-to Information

Remember the analysis of pointers, pointer targets depend upon calling contexts. Pointer targets are organized according to calling contexts, as illustrated in the next figure.

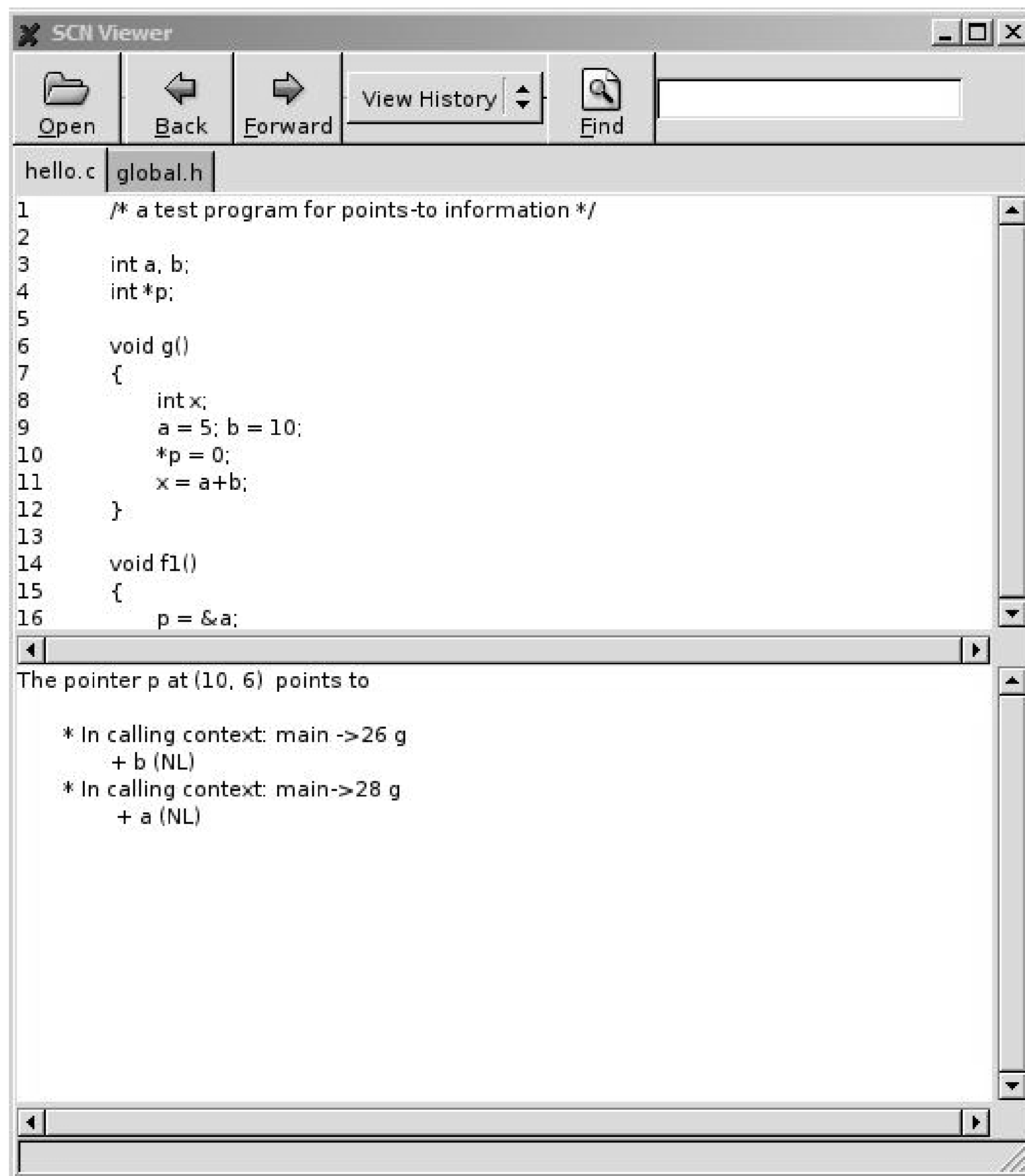

Fig. 4-10 Points-to Information in SCN

For query of the pointer “*p*“ at source position (10, 6), the pointer targets are listed in the information frame. In the calling context of “main →26 g”, “*p*“ points to the variable “*b*“; in the calling context of “main →28 g”, “*p*“ points to the variable “*a*“.

Except for links in calling context strings, pointer targets in the information frame are also implemented as links to the corresponding object declaration.

There could be more than one possible pointer targets in a same calling context. In this situation the list of pointer targets in the information frame is simply expanded. If the list of pointer targets under certain calling context is very long, the list can be folded or unfolded as required.

Also note that pointer targets may be represented by calling contexts. For such pointer targets, calling context strings are used.

4.5.5 Side-effects Information

Side-effects information depends on calling contexts, so it is organized in the same way as points-to information.

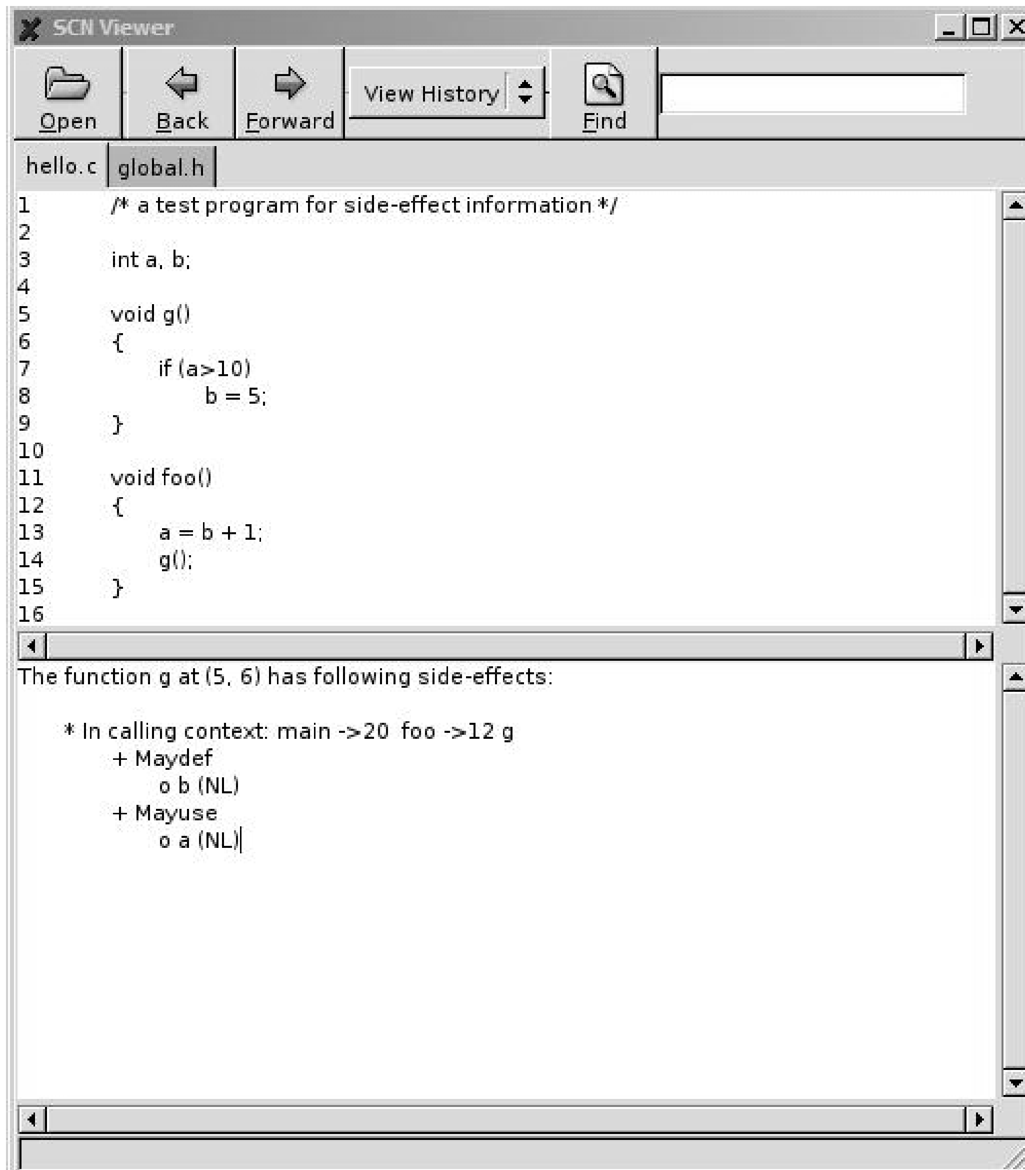

Fig. 4-11 Side-effects Information in SCN

Unlike points-to information, side effects are classified into three categories “Mayuse”, “Mustdef” and “Maydef”. So side effects in each calling context is further organized according to their classes.

If a list in the information frame is too long, it can be folded to save space and unfolded later for inspection.

4.5.6 Source Code Metrics

Source code metrics are visualized in the browser window. As an example only the metric of line number per function/file is illustrated below.

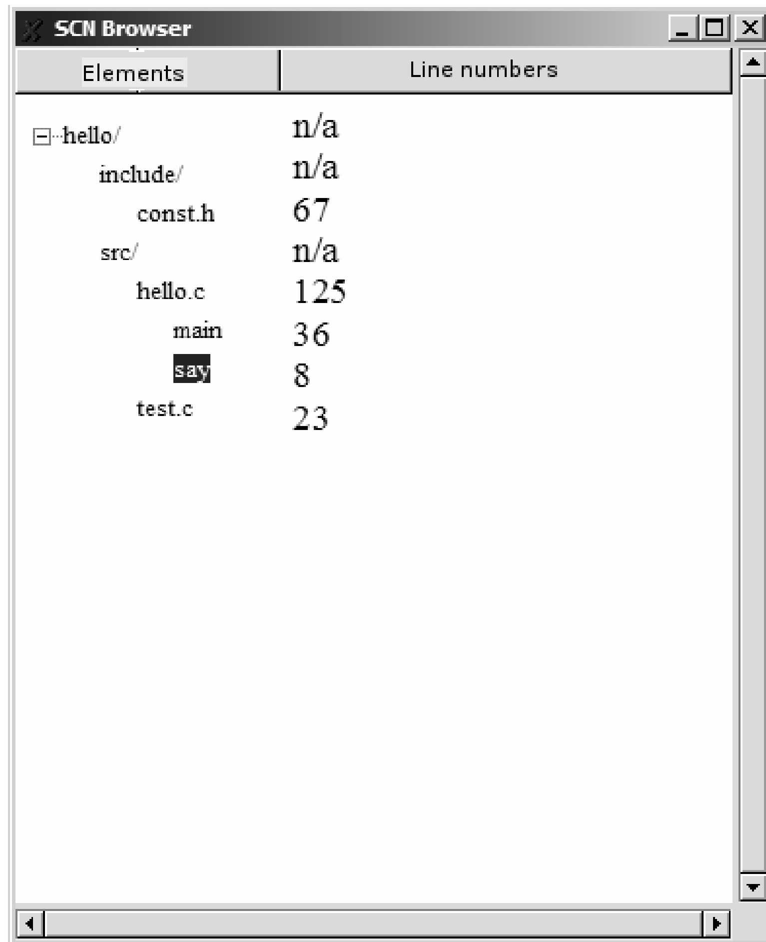

Fig. 4-12 Source Code Metrics in SCN

Note that source code metrics are also textually visualized, because SCN is designed as a pure text visualization tool.

4.5.7 Miscellaneous

Other source-code-related information such as program slicing is visualized in the same way introduced above under the framework of SCN. The policy of the design is to be as simple and intuitive as possible.

4.6 Navigation Semantics

4.6.1 Model of Navigation Process

Navigation semantics are dynamic aspects of program understanding from the user's view. To capture navigation semantics, the process of navigation is to be analyzed first. The following figure shows a simple deterministic finite state machine (DFA) as a model of navigation process.

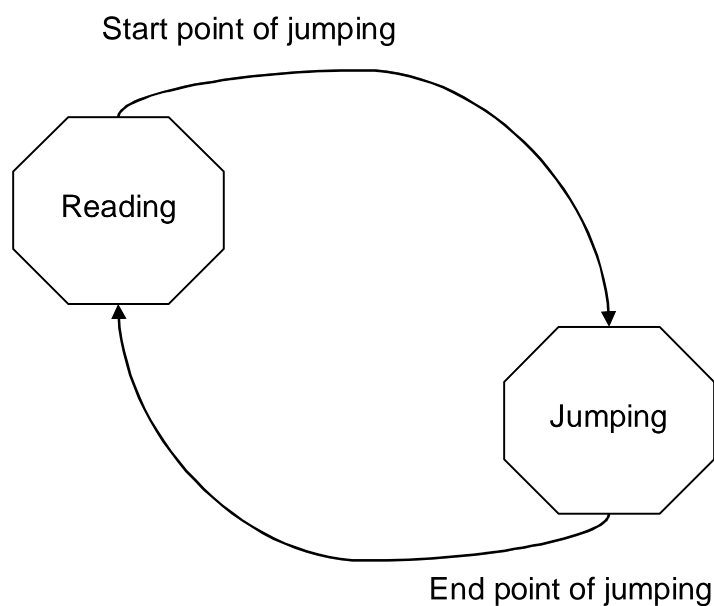

Fig. 4-13 Finite State Model of Navigation Process

At any time, users are either reading source code or jumping within source code. Obviously that reading takes much more time than jumping.

When reading source code, users may raise a query on some source code construct, which is named "Start point of jumping" in the finite state machines. Right after the query of the start construct, users enters the jumping state.

User leaves the jumping state instantly after that SCN visualize the requested source-code-related information or source code. Thus the transition from "jump-

ing" to "reading" is labeled with "End point of jumping", which represents the visualized source-code-related information or source code.

Start and end points of jumping can be identified by a string in the following format:

<Timestamp, File_Name, Symbol_Name, Source_Position, User_Action>

For instance,

<(2003.12.29, 9:20:23), "hello,c", "foo", (5, 10), Query_Refers-to>

This instance represents a start point of jumping, that is, a query of "*foo*" at (5, 10) in "hello.c" at 9:20:23 on 2003.12.29 for refers-to information.

Another example:

<(2003.12.29, 9:20:25), "lib.c", "foo", (100, 6), None>

It represents an end point of jumping, indicated by the special user action "None". It means that SCN jumps to "*foo*" at (100, 6) in "lib.c" at 9:20:25 on 2003.12.29.

Concatenation of these two examples means that a user raises a refers-to query when reading "*foo*" at (5, 10) in "hello.c" at 9:20:23 on 2003.12.29; after two seconds she/he gets replies and jumps to the referred "*foo*" at (100, 6) in "lib.c". Then both characterize a jumping during navigation.

Make use of this finite state machine model and the format for start/end point of jumping, navigation semantics is captured in two aspects:

- Navigation sequence
- Navigation map

4.6.2 Navigation Sequence and Navigation Map

Let SCN record each jumping in a list of start/end point of jumping, thus the navigation sequence is implemented and presented to users as back/forward and history functions (see section 4.1.2, Viewer Window). It is called the jumping list and can be stored in an XML application language.

The view history in SCN is implemented as a volatile jumping list per session of SCN.

To implement navigation maps, SCN just stores the jumping lists of each session permanently. Based on these lists, a graph can be constructed and information can be extracted from the graph, for instance, which source construct is visited when and how. Such information is quite valuable for navigation.

For example, from the navigation map the fact is revealed that the user has visited a specific function just 1 hour ago and 5 times at all in the past week. Some tool may notify the user: "You should have completely understood the `foo` function. You are not supposed to visit it again!"

Other kinds of facts may be revealed, too. For instance, jumping from occurrences of variable x to its definition is quite often. It may imply that x plays an important role but the name is far too simple to understand it purpose.

Applications of history map are another subject and won't be discussed further in the thesis.

5. Implementation of SCN

Implementation of SCN is briefly discussed in this chapter. Emphasis is placed on software architecture.

5.1 Software Architecture

The architecture of SCN is illustrated in the following UML diagram:

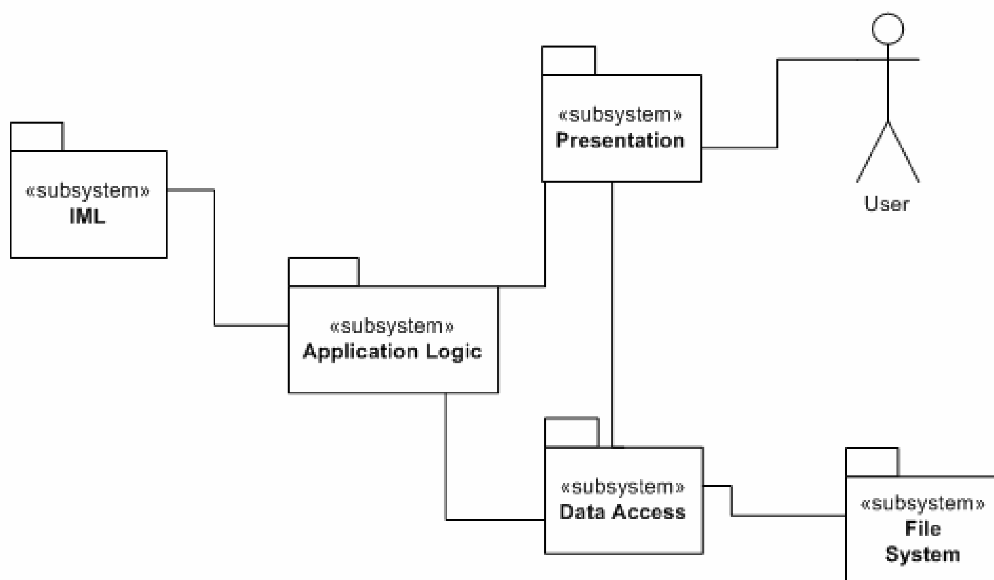

Fig. 5-1 Software Architecture of SCN

The source code navigator is implemented in the MVC (Model-View-Control) approach. The subsystem “Application Logic” is responsible for retrieving data from IML and providing it to users, so it plays the role of “Model”. The subsystem “Presentation” interacts with users and the subsystem “Data Access” gets source code from file system. Packages of the implementation are organized according to the global software architecture.

The source code navigator is implemented using the object-oriented approach.

5.2 Miscellaneous

Design patterns of object-oriented systems are put into use in the implementation. For example, to implement the multiple document interface of the view window, the view window is implemented as a singleton class (See [GHJV]).

The object-oriented approach and adoption of design patterns improve modularity, extensibility, reusability and maintainability of SCN.

Table of Definitions, Acronyms and Abbreviations

Ada95: The Ada 95 programming language

Bauhaus: The Bauhaus project

BNF: Backus-Naur Form, a grammar notation of programming languages

CodeSurfer: A source code analysis and navigation tool by GrammaTech, Inc.

Emacs: A powerful text editor under the GNU public license

Gnat: An Ada 95 compilation system by Ada Core Technologies

GNU: The Gnu project

Gravis: The Graph Visualization Editor in the Bauhaus toolkit

Gtk+: The Gimp Toolkit, a graphical programming toolkit

GtkAda: The Ada 95 graphical toolkit based on Gtk+

HTML: The HyperText Markup Language

IDE: Integrated Development Environment

IML: The Intermediate Language developed for Bauhaus

Mayuse/Maydef/Mustdef: Types of side effects of program expressions

MDI: Multiple Document Interface

MVC: Model-View-Control structure

Program Slicing: A technique for source code analysis

SCN: Source Code Navigator in the Bauhaus toolkit

Singleton: A design pattern for object-oriented systems

Source-Navigator: A source code editing and navigation by Red Hat, Inc.

SourceForge: A website for open source software development

UML: The Unified Modeling Language

XML: The Extensible Markup Language

Bibliography

[Ada95] Programming in Ada 95, 2nd Ed.

John Barnes

Addison Wesley, 1998

[Bauhaus] Bauhaus home page

http://www.bauhaus-stuttgat.de

[CS-Guide] CodeSurfer User Guide and Technical Reference

[GA-Web] GtkAda home page

http://libre.act-europe.fr/GtkAda/

[GA2.2] GtkAda 2.2 Reference Manual

[GHJV] Design Patterns: Elements of Reusable Object-Oriented Software

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Addison Wesley

[Gnat] Gnat home page

http://libre.act-europe.fr/gnat/

[GT-Web] GrammaTech, Inc. home page

http://www.grammatech.com

[Gtk+-Web] Gtk+ home page

http://www.gtk.org

[IEEE830-1998] IEEE Standard 830-1998: IEEE Recommended Practice for Software Requirements Specifications

[SF-Web] SourceForge home page

http://sourceforge.net

[SN-Web] Source-Navigator home page

http://sourcenav.sourceforge.net

[UML-OMG] UML home page

http://www.omg.org

[UML-RJB] The Unified Modelling Language Reference Manual, 1st Ed.

James Rumbauch, Ivar Jacobson, Grady Booth

Addison Wesley, 1999

[XML-Web] XML home page

http://www.w3c.org/XML/

Erklärung

Hiermit versichere ich, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

____

Leiqin Lu