Potential Programming Plan and Domain Concept Detection Supported by Slicing Technique

Gustavo Villavicencio

Facultad de Matemática Aplicada

Universidad Católica de Santiago del Estero

4200 Campus de la UCSE, Santiago del Estero, Argentina

Abstract

Little has been written about the component provision problem (programming plans and domain concepts) for the repositories of knowledge in program understanding systems and reports that specifically address this issue are unknown. Model-driven and plan-driven approaches use repositories where construction and evolution are informal and depend on the experts. In domains where the expert is not available the only valid source of information is the source code. But, domain concept design and programming plans from the source code by non-expert professionals is not a trivial task.

In this paper, a hybrid top-down/bottom-up approach based on algorithmic pattern matching and slicing techniques has begun to be defined in order to provide plans or concepts (termed activities in the model-driven approaches) to the knowledge base. Initially, the exploration area is restricted by slicing techniques, and then, software inspection tools are used to further limit the relevant areas. Finally, the initially hypothesized pattern is matched with these segments.

Keywords: Program understanding, programming plans, program slicing, domain model, constraint satisfaction problems, partial constraint satisfaction problems.

1 Introduction

The construction and evolution of the knowledge repositories of the plan-driven [29, 7] and model-driven [11, 12] approaches have always been dependent on the software engineer's experience on the domain problem. Regarding this topic, [13] describes the problems in the model-driven approach; and the situation isn't much better in the plan-driven approach as it is argued in [21].

So, both model-driven and plan-driven approaches have the same problem: their main knowledge repository is constructed and evolves in an intuitive manner. Therefore, we have a great problem in our hands: The main component of the system understanding is constructed and maintained in a deficient way; what can we expect on effectiveness, precision, etc.?. The situation is even worse in areas where the expert is not available. Must we resign the use of these technologies in those cases?

Here we are providing a more pragmatic approach to the potential programming plan (PPP hereafter) detection problem than those in [25, 26]. It is a combination of those described briefly in [21] and in [25, 26]. Besides, this technique has a strong automatic support that enormously increases process agility.

The proposed technique allows engineers, maintainers, etc. to explore the source code for detecting PPPs and domain concepts (DCs hereafter)[1] in the domain. The scope of the source code to analyze is reduced by applying slicing technique [28]. The detected PP instance provides the information to scale-up to the DC.

The present work is continuation of [25, 26] where slice-to-slice matching [2] is performed to design PPPs from the source code. However, the number of slices calculated in real-world systems makes the technique very hard to apply. So, a more pragmatic approach is required.

On the other hand, [13] relate DCs to *Executable Concept Slicing* (ECS). Essentially, the information in the domain model is used to construct high level slicing criteria to obtain the ECS (top-down approach).

The present work tries to combine both top-down and bottom-up approaches to program understanding. On one side, the imprecise knowledge of the domain problem and the applications in it, is expressed as PPP (top-down). On the other side, a useful program slice is calculated (bottom-up). The PPP is matched with the slice and, starting from here, the PPP is improved and the involved DC detected. A graphical view is shown in fig. 1.

The contributions of this paper are:

1. According to the bibliographical information available, no other technique has been proposed to "feed" PPs and/or *action concepts* (activities) [11, 12] to the repositories of knowledge.

2. It is the first attempt of combining concept assignment and PPs with slicing technique.

3. Slicing techniques are used for a purpose they have never been used before.

4. In [20, 21] et al. a code-driven approach is presented as an improvement of the library-driven approach [15] et al. The present technique, would lead a *plan/slice-driven approach* to recover programming plans and also action concepts.

---

[1]In our context DC will be synonymous to action concept [11, 12] or domain activities.
[2]The matching process is executed by *partial constraint satisfaction algorithms* [27].
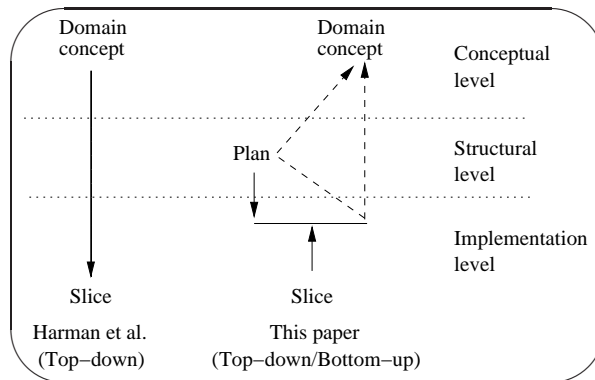
Figure 1: Approaches to relate DCs to slices

The rest of this paper is organized as follows: Section 2 gives a very quick overview on PP concepts and its relation to DC. Section 3 analyzes how the slicing technique is used in this approach. The strategy is presented in section 4. Then, an example is developed in section 5. Finally, future directions and conclusions are presented in sections 6 and 7 respectively.

2   Background

2.1   Plan Recognition

In general, there is a uniform criterion among the authors in the program understanding area about what concepts like *cliché*, *PP*, *pattern*, and *program template* mean. So, in [19] a pattern is defined as solutions to recurring problems at different levels of abstraction (e.g. code pattern, design pattern, and architectural pattern). In [30], the clichés are defined as commonly used data structures and algorithms that implement higher level abstractions. In [8] a plan denotes a description or representation of a computational structure that the designer has proposed as a way of achieving some purpose or goal in a program. But, a plan is not necessarily stereotypical or repeatedly used; it may be novel or idiosyncratic [8].

Therefore, a PP represents a scheme in which a problem can usually be solved by computational means. The scheme is mainly represented in terms of structural information, data and control flow. The PPs are stored in libraries. Pattern matching algorithms explore the libraries in order to detect instances of PPs in the source code. A pattern can be structural if it is only based on syntactical structure, or behavioral if its components have restrictions based on control and data flow. Here we are interested in the last kind of PPs, i.e. based on *constrain satisfaction algorithms* [21, 20] et al.

In [5], Biggerstaff claims that programming plan technologies cannot be applied to real-world systems. However, [20] claim the opposite.

2.2   PPs and DCs

[23] provides a domain concept definition focused on activities.

(. . .) Such concepts are encoded as plans, which are abstract representations embodying the knowledge about program concept components and constraints.

Based on this evidence, we can argue that the PPPs can supply *structural evidence* for the design of concepts (activities) at the domain model. We also argue that, in order to scale-up to action concepts, a great part of the information can be obtained from the detected PPP, or at most, from the surrounding code (the slice).

But, can we include slicing technique in the previous context? We argue that it is possible. On one side, [13] relate DCs to ECS. This is a refined idea of that presented in [7] which talks about *conceptual slices*. On the other side, [25, 26] relate slicing technique to PPs. In this paper we are trying to "close the circle" arguing that PPs provide structural evidence for constructing a DC in a domain model. However, the same programming plan can be shared by more than one DC. Therefore, further analyses of the surrounding source code (the slice) would have to be carried out to find the concept to which the plan is linked.

Thus, the importance of detecting a PP first is that it defines the surrounding source code from which to obtain evidence to scale-up to the DC.

3   Slicing Technique Role

A *static slice* is calculated with respect to a *slicing criterion* which includes a program location (usualy a line number) and a variable of interest. Then, a sub-program is computed performing a dependency analysis (data and control flow analyses). It includes all sentences to which the variable in the slicing criterion is related. For an overview on this technique see [6] et al.

In our approach, we will try to exploit the **r**educing power of the slicing technique to limit the space of the PPP search. To achieve this, the calculation of a useful slice will be necessary. But [13], talking about how to detect the segment that executes the *mortgage calculation* concept, said

Unfortunately, pure slicing cannot help unless the engineer knows which variables are important for this computation.

We agree that it is so, but we cannot ignore the informal information utility [2, 3, 4] et al. which combined with a *string pattern matching* technique, can quickly solve this problem. We must also consider that a program understanding process cannot be possible without any pre-existing knowledge on the subject program and domain problem as it is argued in [22].

4   Detecting PPPs and DCs

The strategy that will be described in this section has an important automatic support, not only to inspect the source code but also the specifically calculated slice. The matching phase also has automatic means.

Figure 2 shows the graphic to keep in mind during the description. For a more precise illustration of the process SADT notation was used.
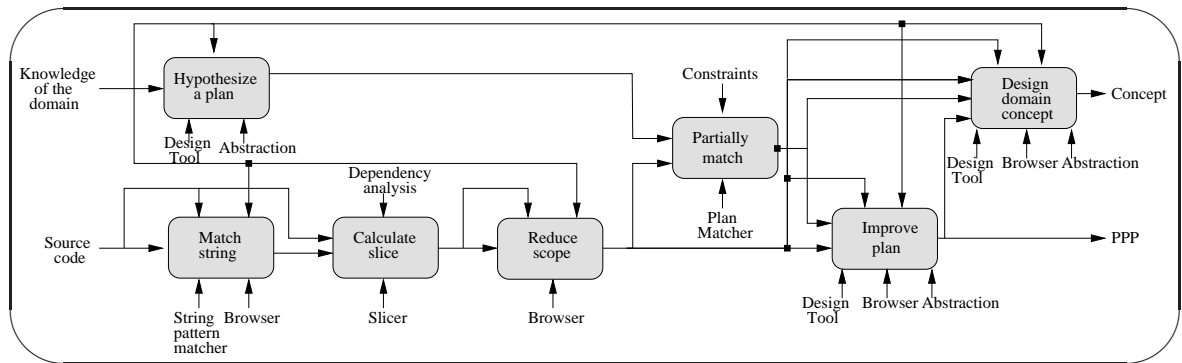
Figure 2: The process in SADT format

4.1 Hypothesized Plan

When a DC is searched, the reverse engineer has in mind, among other things, some kind of *abstract algorithmical morphology*, which should be present in the concept. But if this ideal representation does not have a concrete existence, its utility decreases.

Thus, based on the pre-existing knowledge of the domain and the applications in it, the reverse engineer proposes a PPP related to the DC. This activity is similar to that proposed in [21]. But in our case, the PPP leaves the programmer's mind and has a real existence. This plan expresses the imprecise knowledge the reverse engineer has regarding the pattern searched. It will only be composed of basic components and some restrictions among them. In order to adopt the terminology used in [21], the PPP designed here will be termed *plan example* or *hypothesized plan*.

We use this kind of imprecise knowledge because we propose an automatic support to handle it: Algorithms of *partial constraint satisfaction problems* (PCSP) [10, 27].

4.2 String Matching

Based on the limited knowledge about the subject application, the reverse engineer searches for significant character sequences in the source code. The aim here is to focus the reverse engineer's attention on the significant segments. The tools used can be diverse, simple like *grep* or more complex like ESPaRT [19]. Actually, this last tool can detect very complex syntactical patterns. However, *CodeSurfer* [1] and *Program Slice Browser* (PSB) [9] can also provide string searching mechanisms.

Commonly, the request is focused on the data, but it will not always be so. Some cases will require to explore comments; others will require typical sentences, procedure names, etc. These alternative heuristic searches can be used to approximate us to the sentences we are interested in and study the surrounding code. The aim is to gather evidence on a specific data item; that which is the output of the *plan example*. This kind of variables are called *principal variables* in [13] where they are used to detect the *key statements* of the concept by slicing technique. Similarly, here we are trying to detect the principal variables, and then, by means of slicing technique, gather the sentences that instance a plan.

The fact that the data item is the output of the *plan example* will guarantee then that the calculated slice will bring all the computations that are plan component instances.

4.3 Slicing

Once relevant data items are detected, a slice is calculated based on them. Obviously, the smaller the slice, the better. So, it is important to specify the *slicing criterion* based on correct data.

But several cases can happen when the data item in the slicing criterion is not the required one:

- Case 1: The specified data item has no relation with the data item it is looking for. So, the calculated slice will not bring the plan instance we are interested in.

- Case 2: The computations on the specified data item request computations on the relevant data item. Therefore, the calculated slice will be greater than necessary.

- Case 3: The computations on the specified data item request some computations on the relevant data item. In this case, the calculated slice can only bring some or none instances (computations) of the plan we are interested in.
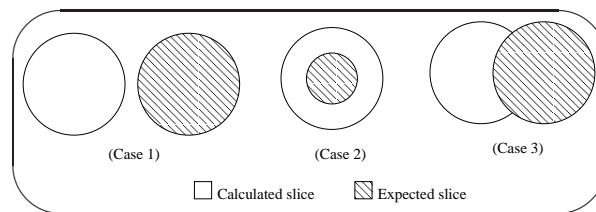

Figure 3: Relation between slices

Figure 3 depicts these situations. Obviously, the second case is the most usual and there are different levels of *closeness* between calculated and expected slices. However, the third case (fig. 3) can also be successful but it depends on the presence of all the relevant computations in the intersection region.

A slice is calculated at the end of the program because it is not possible to know *a priori* in which segments of the slice the hypothesized PP components are. They can also be delocalized [16].

4.4 Reduced Scope

The main target of this phase is detecting those segments in the slice, usually procedures or functions, where the presence of PP components is expected. In fact, this activity can be omitted depending on the size of the calculated slice. If the slice has a moderate size, the next task of *Partial Matching* can be executed successfully. At this point, what was previously said in section 1 acquires importance. That is, the approach we have taken can also be used to define a *plan/slice-driven approach* based on constraint satisfaction algorithms.

However, when the resulting slice is very large, it might contain more than one programming plan; that in which we are interested as well as others. The calculated slice does not only recover

sentences that affect directly the data items in the slicing criterion, but also those sentences that modify variables that affect the items in the criterion and so on. Therefore, the calculated slice can contain algorithmical structures indirectly related to the PP that has been searched, and, consequently, are secondary for our purposes. Therefore, the next task of *Partial Matching* can have problems.

The slice inspection (or *Reduced Scope* phase) can be driven by the informal information in the source code. Thus, for example, the function or procedure names are very significant [2] and can help us detect relevant segments. Obviously, here we are considering *inter-procedural slicing* [14], which, in our context, is related to *delocalized PPs* [16].

However, in other cases, the informal information is not sufficient and further analyses would be needed to detect relevant segments. A heuristic to use in this situation is to obtain those segments where the variables in the slicing criterion have been assigned or printed (*key statements* in the context of [13]). Other segments where the variables are only used would be less relevant because they would not have a direct relation with the components and restrictions of the hypothesized PP. This task can be supported by a tool like *The Finder* in *CodeSurfer*.

Once these sentences are found, the relations with other sentences are analyzed. The target is to detect some clues of the components and restriction of the hypothesized PP presence. These observations must be confirmed afterward in the *Partial Matching* phase.

Both, *PSB* and *CodeSurfer* provide tools for supporting this slice inspection task.

Up to a degree, during this phase, the reverse engineer obtains a detailed view of the components and constraints present in the instance as described by [21] (recall section 1), but differently; here we have a slice in the background!.

4.5 Partial Matching

At this point, the source code obtained would have been reduced. The *plan example* would have been confronted with the critical areas selected in the previous *Reduced Scope* activity. The target of this phase is not "understanding" the slice segments obtained by reduction, but "marking" the zone where there is evidence of the hypothesized PP presence.

Depending on the precision with which the hypothesized PP was described, algorithms of *constraint satisfaction problems* (CSP) [20] et al. or PCSP can be applied. The first kind of algorithms can be used when the reverse engineer relies on a highly accurate plan example. However, the imprecision introduced during the hypothesizing phase can make the problem insoluble by CSP algorithms. Therefore, a PCSP algorithm can also be applied. See a formal description and application example of this last algorithm in [26].

We can say that at this phase we are trying out an "imprecise understanding" as suggested by [21], but in an entirely different way.

4.6 Improving PP

Since the hypothesized PP is still incomplete, many components and restrictions which were not considered before can become evident at this moment.

Here, the reverse engineer focuses his attention on algorithmical structures and tries to leave aside the relations among them, and with those components that already exist in the hypothesized PP. The exploration area in the slice is restricted to those identified in the *Reduced Scope* activity.

A browser combined with abstraction mechanisms supplied by a reverse engineer is intensely used at this phase. Cross reference capacities are applied to detect the relations among relevant sentences. In PSB also, irrelevant segments in the slice can be encapsulated in blocks to reduce the information shown. Emphasizing the important components (sentences and relations among them) and hiding the irrelevant ones would be very useful capacities at this phase.

4.7 Domain Concept Design

As in the previous activity, the search area for evidence has been restricted by the *Reduced scope* activity.

Because the main target at this phase is to find action concepts, the reverse engineer's attention would be focused on sentences. Specifically, the sentences that instance the PP where the concept indicators are expected to be found. Usually, there is an object on which the action is performed. It would be extracted from the data items that handle the sentences that instance the PPP. Again, the main "tool" at this phase is the reverse engineer's abstraction.

These last two activities are entirely manual and there is not any way to replace the reverse engineer's abstraction capacity. In an approach as the one just described, technology can only help to guide and focus the reverse engineer's attention towards significant segments.

5 An Example

This case study has been achieved on a small, but real-world program. Due to lack of space, the program source code is not listed. Only the calculated static slice is shown in appendix A. In order to save space, all the calculated slice commentaries have been omitted.

Suppose, for instance, that we have some source code analyzing tools in the software documentation area. Specifically, we have a program that extracts and counts the number of comments from C and C++ programs.

Let us suppose, in addition, that we are interested in the pattern that *extracts a C comment*. The final output of this PP would be a comment on the standard output. We suppose a vague knowledge about the components of the PP and the relations among them. This knowledge can be depicted as shown in fig. 4.
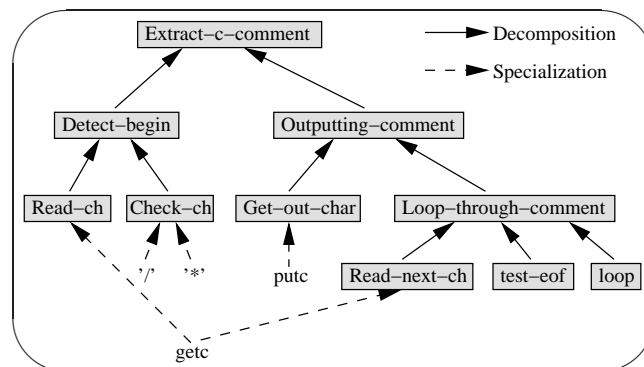

Figure 4: Vague *comment extraction pattern*

The main restrictions in the plan example are:

1. The OUTPUTTING-COMMENT component has a control dependency on the DETECT-BEGIN component.

2. The GET-OUT-CHAR component has a control dependency on the LOOP-THROUGH-COMMENT component.

3. The GET-OUT-CHAR component has a data dependency with READ-NEXT-CH component.

4. The READ-NEXT-CHAR component has a data dependency on the TEST-EOF component.

Now, we will try to extract the slice related to this PP. First, we can execute some kind of string pattern matching to detect those source code segments where the output variable of the activity modeled by the PP is defined or used. However, we have no evidence on which is the string to search on the data items. But, we know that two macros may be applied on it: *putchar* or *putc*. The search of *putchar* by means of **grep putchar GETCMT.C** is negative. But, positive result is obtained by means of **grep putc GETCMT.C**.

Specifically, analyzing the context of the sentence

```
putc(chi,stdout)
```

(executed sentences, comments, etc.) we conclude that the *chi* variable is the data item which we are searching, the principal variable (recall section 4.2). Therefore, we use this variable in the *slicing criterion* to calculate the slice where we hope to find an instance of COMMENT-EXTRACTION pattern. At the end of this paper we show the calculated slice.

Although the slice has an acceptable size executing *Partial Matching* successfully, we assumed it was not so, and achieved some detailed observations. If we were using *CodeSurfer* we would have many ways (using the System Dependency Graph, for instance) to see that both $putc$ sentences in *inside_c_cmts* have a control dependency on the $while()$ sentence (perhaps restriction 2). The second $getc$ sentence has a data dependency on the conditional in the $while()$ sentence (perhaps restrictions 4). The $putc()$ sentence has a data dependency on both $getc$ in the $while()$ sentence (perhaps restriction 3). All these sentences are executed because *inside_c_cmt* is called by *extract_c_cmts* where there are sentences that detect the beginning of a comment (perhaps restriction 1).

Based on this evidence, we can propose *extract_c_cmts* and *inside_c_cmt* as relevant segments on which to execute the *Partial Matching* task.

Once the exploration space is reduced, the process continues in the *Partial Matching task*. Here, the hypothesized plan is confronted with the sentences in *extract_c_cmt* and *inside_c_cmt*. The result of the matching is shown in fig. 5.

This is a nice view of the results obtained by the partial matching activity. However, in larger slices this representation can be impractical. For solving this problem, those sentences in the source code that instance a component in the plan can be annotated with the component name.

Based on this evidence we can now improve the *plan example*. This is a completely manual activity. Analyzing the surrounding source code to each detected instance, the reverse engineer completes the hypothesized PP. In this case, the main contribution of these segments to the

Plan diagram boxes:

Extract–comment

Detect–begin Outputting–comment

Read–ch Check–ch Get–out–char Loop–through–comment

Read–next–ch Test–eof Loop

```
int extract_c_cmts()
{
  register int chi, cht;
  chi = getc(infile);
  while(chi != EOF)
  {
    if (chi == '/')
    {
      cht = getc(infile);
      if(cht == EOF)
        return();
      if(cht == '*' || cht == '/')
        inside_c_cmt(cht);
      else
        ungetc(cht,infile);
    }
    chi = getc(infile);
  }
  return;
}
```

```
void inside_c_cmt(int ch)
{
  register int chi, cht;
  if(ch == '/')
    ch = '\n';
  chi = getc(infile);
  while(chi != EOF)
  {
    if(chi == ch)
    {
      if(ch == '\n')
        return;
      cht = getc(infile);
      if(cht == '/')
        return;
      else
      {
        ungetc(cht, infile);
        putc(chi, stdout);
      }
    }
    else
      putc(chi, stdout);
    chi = getc(infile);
  }
  return;
}
```
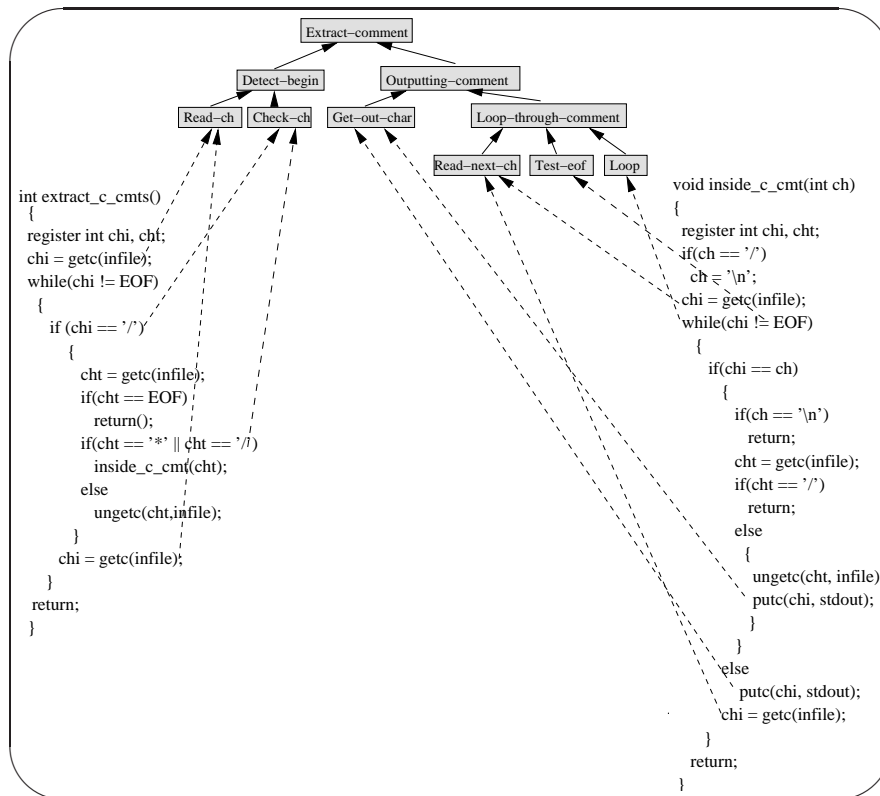
Figure 5: Matching result of the program plan example and the selected segments in the slice

library in fig. 4, is that of providing evidence of a plan that extracts C++ comments. Therefore, the library must be reestructured to introduce this *specialisation*. At the same time, there are algorithmical components that are shared by both programming plans that must also be modeled in the library. The final result is depicted in fig. 6.

Finally, based on the structural evidence provided by the PP, plus the detailed information in the surrounding source code, we design the corresponding segment of the domain model as shown in fig. 7. The notation used in this fig. was taken from [13].

Like the *plan example* improvement task, the design of concepts is also a manual activity. The reverse engineer must observe, detect, and abstract from the surrounding code the elements composing the concepts and how they will be arranged in the library.

Therefore, from the instance of the PP components the reverse engineer obtains the *indicators* [3] which are evidence of the presence of a concept (action concept in this case) in the source code. The detected specialisation of *Comment* concept in *C-comment* and *C++-comment* is a typical result of an abstraction task executed considering the surrounding source code.

This is a small example. But, a more complex PPP and DC can be designed from a medium-large COBOL slice as shown in [25].

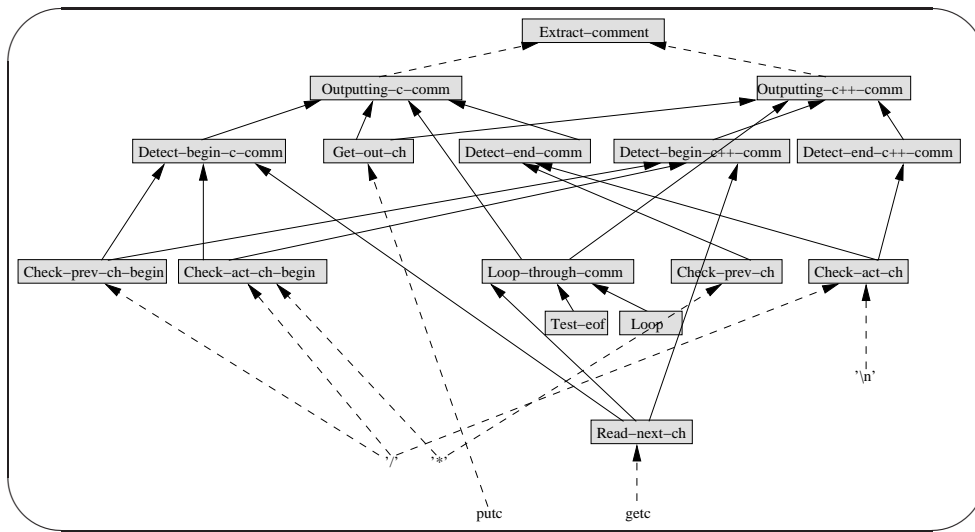[3]Following the terminology used by [11, 12, 13].
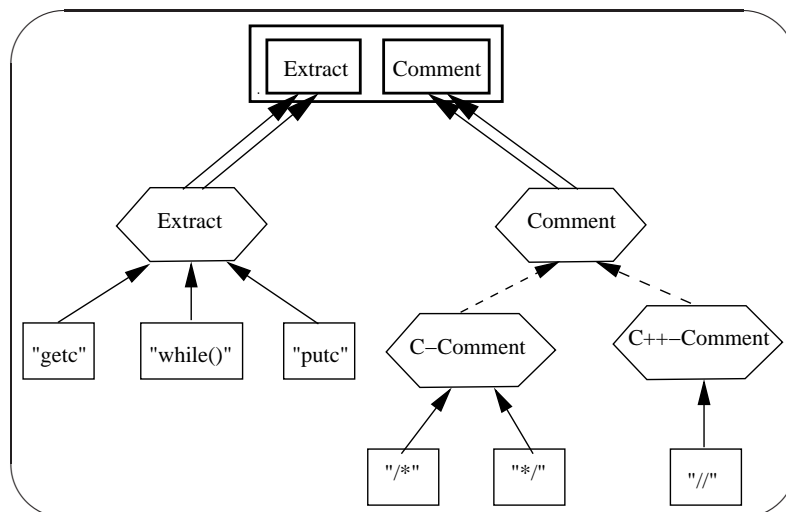
Figure 6: Improved *hypothesized plan*

Figure 7: Domain model fragment designed from the detected PP and surrounding source code

6 Future Work

The construction of repositories of knowledge for program understanding systems (plan-driven and model-driven approaches) from the source code is not a trivial activity. The critical question: where to begin this process, has not been treated here. And we do not have any answer either. However, some clues on where to follow once a plan is detected are provided. In [25] it has already been suggested that the closeness of the plans in the libraries has relation to the proximity of the data items used to specify the slicing criterion [4]. Thus, *cluster analysis tech-*

---

[4]That is, if two slices are calculated on two related data items, and each slice has a programming plan instance, then these plans are related in the library.

niques [18, 24] can be applied to detect the related PPs. We have no experimental results about this. However, [13] also propose the use of cluster analysis to detect related concepts.

On the other hand, the combinational explosion that occurs when the constraint satisfaction algorithms are applied on real source code, has limited the plan-driven technology to the laboratory environment. But, the doubt imposed by Biggerstaff et al. on the applicability of this technology to real software systems, and partially cleared by [20], could be definitely solved by a variation of the present approach. In [15] et al. a library-driven approach is taken to plan recognition. In [21, 20] et al. a code-driven approach is used to detect concepts. This just presented technique can become the base to a more complex one: The *plan/slice-driven approach* to plan recognition. In this context, a string analysis as suggested by [2, 3] for instance, would be performed. Based on this analysis, a *catalog of slicing criteria* would be constructed. Thus, the matching process would be carried out selecting a slicing criterion from the catalog and a plan from the library. In this way, the matching algorithms search for plan instances on specific areas in the source code, in contrast to the whole source code as up to now. Maybe this research path is the most exciting finding of this paper.

It would also be very interesting to compare the previously described approach with that in [17], where the plan library is pruned using a signature-based technique to reduce the search area.

7   Summary and Conclusions

Up to date, the construction of PP libraries or concept libraries in a knowledge-based concept assignment approach has been made by domain experts. But, we do not have any solution for domains where the expert is not available, nor any technique to help the expert provide components to the library. So, the construction and maintenance of these libraries is rather erratic [13].

We have outlined a technique that allows to detect plans for plan-driven approaches and/or provide structural evidence to model activities (*action concepts* in [11] et al.) in model-driven approaches. The theoretic support for the approach is based on [25, 26] (a slice can implement a PP) and [13] (a slice can implement a DC). Therefore, the key activity underlying this technique is program slicing. Its reducing power is applied to limit the search space. A set of automatic tools is used throughout the process to reduce the set of domain values (slice sentences) that instance the variables (PP components).

A key component of the strategy is the use of the PCSP algorithms which allow an imprecise understanding from where to scale up during the *Improvement Plan* and *Design Domain Concept* phases.

A Calculated slice

```
#include <stdlib.h>                        int extract_c_cmts()
#include <stdio.h>                          {
#include <ctype.h>                            register int chi, cht;
#include <string.h>                           chi = getc(infile);
                                              while(chi != EOF)
#define LOOKS_GREAT 1                         {
#define LESS_FILLING 0                            if(chi == '/')
                                                  {
int extract_c_cmts(void);                           cht = getc(infile);
void inside_c_cmt(int);                             if(cht == EOF)
                                                        return();
FILE *infile = stdin;                               if(cht == '*' || cht == '/')
int  show_nos = 0;                                  {
                                                        inside_c_cmt(cht);
main(int argc, char *argv[])                        }
{                                                   else
 register int i;                                        ungetc(cht, infile);
 const char *hype =                             }
   "\nGETCMT v1.1 -                             chi = getc(infile);
   GET CoMmenTs\nby Byte_Magic Software\n";    }
 const char *oops =                            return();
   "\a*** GETCMT - Can't open input file ";  }
 fputs(hype, stderr);
                                            void inside_c_cmt(int ch)
 if (1 < argc)                              {
 {                                            register int chi, cht;
  for (i = 1; i < argc; ++i)                  if(ch == '/')
    {                                             ch = '\n';
     if ('/' == *argv[i])                     chi = getc(infile);
       {                                      while(chi != EOF)
        if ('l' == tolower(argv[i][1]))       {
        else                                      if(chi == ch)
          {                                       {
           int ercode;                                if(ch == '\n')
           ercode = ('?' == argv[i][1]) ? 0 : -1;        return;
           if (ercode)                                cht = getc(infile);
              putc('\a', stderr);                    if(cht == '/')
              return(ercode);                            return;
          }                                           else
       }                                              {
     else                                                ungetc(cht, infile);
       {                                                 putc(chi, stdout);
        infile = fopen(argv[i], "r");                 }
        if (!infile)                              }
          {                                       else
           fputs(oops, stderr);                       putc(chi, stdout);
           fputs(argv[i], stderr);               chi = getc(infile);
          }                                       }
       }                                        return;
    }                                         }
 }
 i = extract_c_cmts();
 putc('\n', stdout);
 return(i);
}
```

References

[1] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *1st Workshop on Inspection in Software Engineering*, Paris, France, Jul. 2001.

[2] N. Anquetil. Characterizing the informal knowledge contained in systems. In *Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2-5 2001. IEEE CS Press.

[3] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON'98*, Toronto, Canada, 1998.

[4] N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:201–221, 1999.

[5] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993. IEEE CS Press.

[6] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 2004. To appear.

[7] David Ching and Alex Quilici. Decode: A cooperative program understanding environment. *Journal of Software Maintenance*, 8(1):3–34, 1996.

[8] Richard Clayton, Spencer Rugaber, and Linda Wills. On the knowledge required to understand a program. In *Fifth IEEE Working Conference on Reverse Engineering*, Honolulu, Hawaii, October 1998.

[9] Yungo Deng, Suraj Kothari, and Yogy Namara. Program slice browser. In *9th IEEE International Workshop on Program Comprehension*, Toronto, Canada, May 12-13 2001.

[10] E. Freuder and R. Wallace. Partial constraint satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan, USA*, 1994.

[11] N. Gold. *Hypothesis-Based Concept Assignment to Support Software Maintenance*. PhD thesis, Department of Computer Science, University of Durham, 2000.

[12] N. Gold. Hypothesis-based concept assignment to support software maintenance. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 545–548, Florence, Italy, November 2001. IEEE CS Press, Los Alamitos, California, USA.

[13] Mark Harman, Nicolas Gold, Rob Hierons, and David Binkley. Code extraction algorithms which unify slicing and concept assignment. In *IEEE Working Conference on Reverse Engineering*, Richmond, Virginia, USA, October 2002.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[15] V. Kozaczynski and J. Ning. Automated program understanding by concept recognition. *Automated Software Engineering 1*, 1:61–78, March 1994.

[16] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, May 1986.

[17] Y. Limpiyakorn and I. Burnstain. Applaying the signature concept to plan-based program understanding. In *IEEE International Conference on Software Maintenance*, Amsterdam, The Netherlands, September 22-26 2003. IEEE Press.

[18] S. Mancoridis, B. Mitchell, C. Rorres, and Y. Chen. Using automatic clustering to produce high-level system organization of source code. In *International Workshop on Program Comprehension*, Ischia, Italy, June 1998.

[19] Martin Pinzger and Harald Gall. Pattern-supported architecture recovery. In *10th International Workshop on Program Comprehension*, pages 53–61, IEEE CS Press, Paris, France, June 2002.

[20] Alex Quilici, Steven Woods, and Yongjun Zhang. Program plan matching: Experiments with a constraint-based approach. *Science of Computer Programming*, 36:285–302, 2000.

[21] Alex Quilici, Qiang Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. *Journal of Automated Software Engineering*, 5(3):347–372, 1998.

[22] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *International Workshop on Program Comprehension*, pages 271–278, Paris, France, June 2002.

[23] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo. Augmenting pattern-based architectural recovery with flow analysis. In *Third Working Conference on Reverse Engineering*, Monterrey, California, USA, November 8-10 1996.

[24] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. *ICSE'99, ACM*, 1999.

[25] G. Villavicencio. Program analysis for the construction of libraries of programming plans applying slicing. In *XIV Brazilian Symposium on Software Engineering*, João Pessoa, Paraíba, Brasil, October 2000.

[26] G. Villavicencio. Program analysis for the automatic detection of programming plans applying slicing. In *5th European Conference on Software Maintenance and Reengineering*. IEEE CS Press, Lisbon, Portugal, March 2001.

[27] C. Voudouris and E. Tsang. The tunneling algorithm for partial csps and combinatorial optimization problems. Technical report, University of Essex, September 1994.

[28] Mark Weiser. Program slicing. In *Fifth International Conference on Software Engineering, San Diego*, California, March 1981.

[29] L. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, AI Lab, Massachesetts Institute of Technology, 1992.

[30] Linda Wills and Charles Rich. Recognizing a program design: A graph parsing approach. *IEEE Software*, pages 82–89, Jun. 1990.