Program Analysis for the Construction of Libraries of Programming Plans Applying Slicing

Gustavo Villavicencio
Universidad Catolica de Santiago del Estero
Santiago del Estero
Argentina

Abstract

Why the automatic understanding systems of programs based on plan libraries of programs are not of current application in the reverse engineering processes or in the ambient of re-engineering?. Even though the complexity of the algorithms that explore the libraries constitutes one of the principal problems to solve, the construction itself of the plan libraries represents a problem even more crucial. To mount a repository of this type, it is essential to have the professionals that have developed the systems or that accomplish maintenance in the interest area. To date, there exist no reports that describe some technique on how to accomplish analysis of programs oriented to the construction of libraries of programming plans. In the framework of the NEIGHBORS Project currently in execution, a technique based on the automatic comparison of slices is being investigate; this permits the analyst to focus his attention on a meaningful code for the design of program plans, liberating to a large extent the efficiency of the tasks of analysis from the previous knowledge of the applications and from the problems domain. The obtained results until now confirm the feasability of the approach.
Keywords: Reverse Engineering, automatic programs understanding, reuse, re-engineering, libraries of programming plans, slicing.

1 Introduction

The process of programs understanding for maintenance purposes, reuse, or re-engineering, can be seen as the construction of "mappings" between the knowledge of the professional (analyst) that accomplishes the process, and the implementation of the application software [WQ96]. The knowledge previously acquired by the analyst, can be general, on how the system has been built, or specific, on the typical algorithmical constructions.

The automatic understanding systems of programs intend to simulate the behavior of the analyst. The final objective consists of extracting programming plans and design objectives from the source code [QYW96]. The programming plans are algorithmical structures that the programmers have applied reiteratively during the implementation of the system and that are disperse in the source code[1] [RSW96]. Precisely the integration of such structures and its storage in automatic support (libraries of program plans), constitutes the "knowledge" with which it is attempted to "map" the source code object of

[1]Such code segments also *delocalized plans* are called.

analysis.

In general, the process consists of moving the source code of input in a intermediate representation, usually AST (Abstract Sintax Tree). Then, search mechanisms explore the libraries in search of index[2] [Q94] that permit to link some programming plan with components AST. The indices do not determine the existence of a plan, but simply the possibility of their existence. Starting from the detection of the instance of an index, the "checkup" of a series of pre-defined conditions is started. Such pre-conditions describe the architectural arrangement of the components and the relationships of data flow and control among them. This problem is known as the *constraint satisfaction problem* [WY95]. It has been demonstrated in [QYW98] that an approach based on the constraint satisfaction is adapted to the type of programs understanding in which interest lies, not thus the recognition algorithms applied to artificial intelligence.

Unfortunately, the application of this technology to real programs requires the construction of large libraries of programming plans. In this regard in [CQ96] it is argued: .

> *Unfortunately, there are several fundamental problems with trying to apply this approach to large, real-world legacy systems:*
>
> - *This approach requires enormous libraries of code patterns. Every domain has its own domain-specific design elements, each of which requires a set of domain-specific code patterns to represent its different implementations.*
> - ...
> - ...

Also in [QYW98] it is asserted:

> *One key problem is that the plan recognizer requires a library of program plans. Our simple example to illustrate the behavior of Kautzs algorithm showed that a relatively complex hierarchy is required to understand just a few lines of code. That implies that a significantly more complex hierarchy will be required to understand 10,000-lines modules. It's clear that to apply plan-based understanding to real-world systems we will need a cost-efective way to create plan hierarchies.*

Now then, what technique of programs analysis for the construction of library of programming plans is executed?. To date, there are no known reports that refer this topic. However, it is not difficult to notice that would be indispensable the participation of the professional that have developed the system or, at least, of those that maintain it, to extract from them and/or with them, the repetitive algorithmical structures. It si widely known the motives because of which, in practice, it is not possible to count on such professionals. It is interesting to emphasize, nevertheless, that such problem constitutes a decisive limitation for the proliferation of these understanding environments, inasmuch as for each problem domain experts on this field are required, and it can not be thought about professionals devoted to the assembly of libraries of programming plans in different domains.

The foregoing, implicitly discredits the alternative of the anticipated definition of the programming plans. That is to say, again it is required the participation of the experts in the problems domain. The technique that is described has been designed to be applied

[2]It is considered real predefined "hooks" through which a code segment is attempted to bind to some plan in the library.

by professional that do not have meticulous knowledge of the domain of application.

However, also in [QYW98], it is briefly described a possible approach with automatic support on how analysis for the construction of libraries of programs plans would be practiced; details about which this approach are scarce:

> *We have begun exploring an approach for helping programmers construct a plan library. The idea is to provide programmers with a tool that allows them to provide plans by example. In particular, the approach is to let them highlight existing code as an instance of a plan, provide them with a detailed view of the components and constraints present in this instance, and allow them to delete and/or generalize constraints and components. The systems can support this process by checking whether various cambinations of the components/constraints presents in this plan instance correspond to already-entered plans, and then automatically grouping and replacing them with previously-defined plans. The end result is a definition of the plan and links from it to other library entries. Given a sufficiently fast program understanding algorithm, the set of programs that may contain the user-provided plan can be immediately searched, and the user can adjust the plans definition based on the results.*
> *Besides the technical issues involved in constructing this tool, its an open, empirical question whether such a tool can be used to cost-effectively provide plan libraries. However, it does suggest one possible path toward addressing the problem of how the necessary plans are provided to program understanding system.*

One of the observations to this approach is that due to the fact that initially the plan library would be empty or in the best of the cases scarcely populated, the combinations of components/restrictions of the instances could not be compared with any plan and the process would fail. Other of the limitations that is observed, is that it is considered a strategy highly dependent on the professional or professionals, that is to say, of those that have participated in the development of the system or that are devoted to its maintenance. Their participation is indispensable, not only because in their minds the potential plans against which example plans will be initially checked are kept[3], but furthermore because they know the approximated location in the source code where such plans could be implemented, fundamental knowledge when a source code of great size is analyzed. Thus again emerging the problem already mentioned about the unavailability of these professionals.

In the framework of the NEIGHBORS Project[4], a alternative technique based on the automatic comparison of slices has begun to be developed. This technique has as objective:

- To filter those code segments potentially meaningful for substantially reducing the search space of the programming plans.
- To liberate to a large extent the efficiency of the task of the analyst from his previous knowledge of the source code object of study, supplying automatic tools and well defined tasks.

Below briefly the basic concepts needed to know about programs slicing will be checked. Then the general ideas that compose the technique will be introduced through a simple example, before entering to detail the components of it. Thereinafter its application through an example of source code of the real world will be shown. The work will end with the conclusions to which it has been arrived.

[3]Again, to be the empty plans library, the plans in the mind of the analyst would be the only one possibility of comparison.

[4]The Neighbors Project approved by resolution 164/99 of the Council Superior of our university has as purpose to develop a slicer (extracting of slices), and the same time, to explore new applications of they.

2 Background: Program Slicing

It is considered a technique developed by Weiser [W84] for the understanding and debugging of programs. A slice S(v,l)[5] of a program, on a variable v in a sentence number l brings all the previous sentences that "contribute" to affect the value of v in l (backward slicing). Inversely, given the slicing criteria S(v,l) the sentences which depend on it can be calculated (forward slicing) [HRB90], [HDS96].

The calculation of the sentences involved in a slice is accomplished automatically applying dependencies analysis on data flows and control flows [Tip94].

This technique that in principle has been applied to the program understanding and debugging, has also been adapted to other contexts such as maintenance [GL91], [KR97], [ADS93], [KS98], reuse [LV96], [LV97], programs specialization [RT96], architecture analysis of software [Z98], re-engineering systems OO [S98], [LH96], programs evaluation [DH99], etc. With the present work a new area of application of this technique is being explored.

For the purposes pursued, it is interesting to calculate complete (maximum) slices on a datum item in a program, that is to say, as of the last sentence where the variable that integrates the slicing criteria is employed. Which, however, does not guarantee that all the computations set associated with the variable is captured, as it is demonstrated through the program example of the figure 1 extracted from [GL91].

```
1 input a
2 input b
3 t=a+b
4 print t
5 t=a-b
6 print t
```

Figure 1: Example Program Fragment

The slice with criteria S(t,6) is composed by the sentences 1, 2, 5 and 6. The selection of the last sentence, is done with the simple purpose of attempting to extract the greatest possible quantity of sentences to, at the same time, increase the possibility of detecting similarities as it will be seen below. For the fixed objectives, the slices S(t,6) and S(t,4) will have to be considered independently one from the other.

A form of establishing if the slice is completed is to attempt to accomplish forward chaining (as calculating forward slicing) to depart from the sentence that integrates the slicing criteria. If more sentences are incorporated the slice calculated is incomplete. However, it is clear that automatically a complete slice can be calculated given any slicing criteria, without need of calculating first a backward slicing and then to verify if it is complete or not.

Nevertheless, since it is desirable to reduce the source code volume of lines to be analyzed, the semi-complete slices are those which take the greatest importance in practice. A semi-complete slice is defined in the same way Weiser did in [W84], that is to say, given a slicing criteria calculate the previous dependencies.

However, due to the fact that the specific sentence from which it is possible calculate the slice that contains the sentences that will be instances of the plan that is sought and is

[5]It is designated slicing criteria, and v is able to do reference to a variables set.

intended to design is not known a priori, usually it will be failed in specifying the correct slicing criteria. For this reason, the concept of *gradually calculated slice* is introduced. That is to say, to calculate a semi-complete slice and, starting from executing forward chaining, to add sentences in the measure that the analyst considers necessary. It is clear that applying this mechanism one can arrive to calculate a complete slice.

Other important concept for detecting recurrent algorithmical structures and due to the fact that they can be found delocalized, is that of *inter-procedural slice*, which is calculated applying dependencies analysis beyond the limits of the procedure to which the slicing criteria make reference [RHS95], [HRB90], [FG97].

On the other hand, the slices calculated from a same program can show different relationships mutually, according to the fact of sharing or not sentences. In this way, if there are slices with common sentences but at the same time with own sentences they will be called *slices with not empty intersection* or simply *slices in intersection* (figure 2.b). Also it is possible to calculate slices totally contained in others, in whose case they will be called*included slices* (figure 2.a). And finally it will be possible to have slices that do not possess common sentences which will be called *disjoined slices* (figure 2.c).

Figure 2: Relationships Between Slices

These relationships among slices are the mechanisms of filter of source code lines that will be applied to reduce the volume of sentences to be analyzed. It constitutes a subsequent investigation matter to analyze whether these relationships among slices can provide some clue on the arrangement of the programming plans in the libraries.

3 A Technique of Programs Analysis for the Construction of Programming Plans

3.1 *Slices and Programming Plans*

Until now, there are no reports that may have attempted to combine these two techniques. On what is the idea of linking the slices to the programming plans founded?. If a slice S on a specific datum item **d** is calculated, and there is a plan programming **P** where the computations abstracted in its components can be applied to the datum item **d**, then S contains the sentences that instance **P**. That is to say, a "well calculated" slice can contain a programming plan, and if this programming plan exists no abstract components without instancing will remain.

In other words, if a hypothetical programming plan is in mind, the key consists in calculating the slice that contains the instances of the abstract components of the plan. It is clear that to calculate the appropriate slice in programs that manipulate thousands of data items is not a simple task, since not only it would be necessary to know which the datum item to be specified in the slicing criteria, but furthermore, of what sector of the program it should be calculated so that it will contain the instances of the programming plan.

So far no one has ever developed mathematical bases that guarantee that if there is a programming plan then there has to exist a slice that contains the instances of its abstract components. However, it has been gathered empirical evidence that demonstrates this hypothesis.

These ideas will be analyzed with an example. It is assumed that there is intention to design a hierarchy of programming plans as the one shown in figure 1 of [QYW98], and that is the program fragment of the figure 3, extracted from the same work. If the slice S(sum,l+6) is calculated all the sentences that compose the fragment will be extracted, and this certifies a perfect instance of the program plans that integrate the hierarchy of the figure 1 of that work.

```
...
l       n=0;
l+1     while (scanf("%", &value) == 1)
l+2             a[n++] = value;
l+3     sum=0;
l+4     for(i=1; i<n; i++)
l+5             sum += a[i];
l+6     printf("%i\n", sum);
...
```

Figure 3: Program Fragment that Instance a Programming Plan

However, it is evident, that if the slice S(sum, l+3) is calculated the results that will be obtained (for this case) will be more specific (poor) with view to the construction of program plans (miscalculated slice), since it will be possible to design only some of the desired plans.

That is to say, it can occur that a correct programming plan is desired and that the data item or items that will constitute the slicing criteria have been selected correctly, but it has been failed in indicating the correct sector from which the slice would have to be calculated the slice. Under such circumstances, it can be supposed incorrectly that a slice on that data item does not permit to capture the sentences that instance the abstract components of the programming plan thought. An extreme solution would be to calculate a complete slice, that is to say, in all the program, but this would hinder the subsequent analysis incorporating perhaps unnecessary sentences and hindering the selection of the region from which to begin the analysis.

Even though this is not the case, it can also occur that the calculated slice is composed by other sentences that will not be instances of any abstract component of any programming plan. In this case, the slice will be "dirty" with irrelevant information for the pursued purpose what will hinder the detection of the plan hidden in it. This situation in program fragments COBOL will be below.

3.2 *A Introductory Example*

An very simple example is presented below as a way of illustrating the basic ideas that compose the technique that has been designed.

It is supposed the presence of the program fragment observed in figure 4 that a reproduction of [WY95], and from which the slices on the data items A, B, and C have been calculated.

It is clear that in case of being the complete program, perhaps the slices would be composed of additional sentences that provide nothing to the construction of the hierarchy intended to be designed. Even though the simplicity of the example makes a deep study to establish the similarities among slices unnecessary, a short analysis will proceed

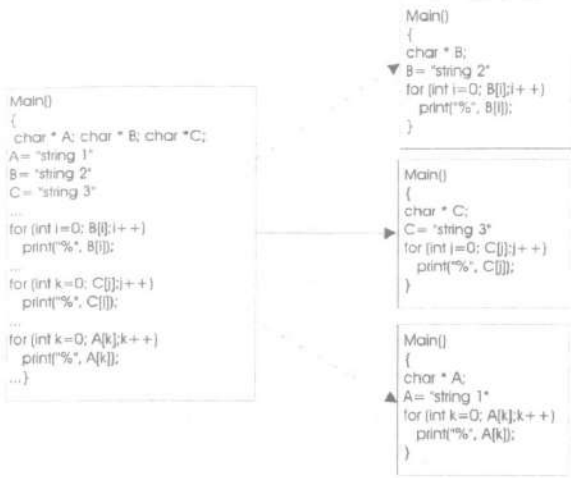

Figure 4: Calculated Slices

in order to delineate the general ideas of the technique.

At the beginning the two slices to the tool are proposed to accomplish the analysis, one of the them will enable to accomplish the observations (slice on B), while the other one will reflect the similarities with the first, if these are present (slice on C). In reality, it is perfectly possible to accomplish simultaneous comparisons between more than two slices.

Starting at the PRINT sentence in the first slice, it is observed that it has a dependency of control and data (by the variable index i) on the FOR....NEXT sentence; consequently, it is required the tool that confirms if the PRINT sentence in the second slice fulfills with identical restriction. The response is positive. Continuing with the observations in the first slice, it is also observed that the PRINT sentence has a data dependency on the assignment sentence previous to the FOR....NEXT sentence. Again to the tool is consulted to check, if the corresponding PRINT sentence in the second slice complies with the same restriction. In fact the response is positive. Finally, the FOR....NEXT sentence in the first slice has a data dependency on the assignment sentence. The tool confirms that in the second slice identical restriction is verified.

The fact that the restrictions or dependencies are fulfilled, implies that the sentences involved can be considered conceptually identical, and that both are potential instances of an abstract component of a hypothetical programming plan. This concept is very important, inasmuch as there is *no* seeking of syntactically identical sentences but of the concept that can be abstracted from them.

Figure 5 shows the sequence in which the components considered similar have appearing.

As shown below, based on the detected similarities, the task of the analyst is to create the abstract components that synthesize both instances. As a rule, and due to the fact that the two slices possess instances that can be considered conceptually identical and that the data and control dependencies demonstrate that the architectural arrangement of such components is the same in both slices, then the involved source code can hide a

Figure 5: Appearance Sequence of Similarities

programming plan and therefore deserves the attention of the analyst. Figure 6 shows a hierarchy of program plans that would be the result of the observations accomplished by an analyst in the previous fragments source code.

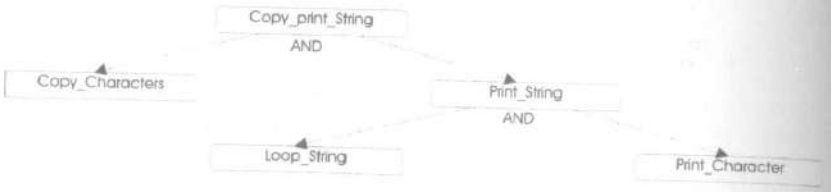

Figure 6: Designed Plan Base on the Observed Similarities

The connections AND and OR are employed, even though the latter is not present, as in [WQ96], to relate abstract components.

It is evident that if the analyst has good knowledge of the problem domains, and has a accurate mental representation of the programming plans as that showed the figure 3 of [WY95], the existence of a second slice would not be indispensable, since the analyst would compare his knowledge directly with the existing slice. However, in this last case, the comparison process would depend exclusively on the knowledge of the problems domain on the part analyst, not only to detect what is considered a plan of the domain, but also to know the approximated location in the code source where such plan would be implemented. However, this approach is not supported by automatic means like the technique that is being defined.

3.3 *Activities and Automatic Support*

Taking into account the previous example, it is possible to define the elements that compose the technique. As it was mentioned previously, the objective is to focus analyst's attention on code segments potentially relevant to the design of programming plans. It is clear that to apply this technique the analyst must have an approximated idea about the programming plan that he is seeking and trying to build, on one side; and on the other side about the source code, more specifically on the data items, that will constitute the slicing criteria. The last is important since the analyst will specify a appropriate slicing criteria to be able to capture the sentences that will be instances of the abstract components of the programming plan that he is seeking and trying to design.

What types of slices are particularly interesting for this purpose?. Since it is desirable to detect recurrent algorithmical structures comparing slices, included slices are of less importance for these objetives, since they are not useful at least by now, to compare exactly the same sentences[6]. On the contrary, the disjointed slices are of particular interest, the same as the code fragments that constitute the differences in the slices with

[6]This idea that in principle results trivial, has its importance from the perspective of the data that compose the slicing criteria. That is to say, the data item that composes the slicing criteria of the included slice, is conceptually included in the data item that integrates the slicing criteria of the inclusor slice. In case of existing programming plans hidden in such slices, this relationship of inclusion can provide clues on how the programming plans disposed will be in the library. In general, the relationships of disjoint and intersection of slices would also have to supply similar clue.

non empty intersection (shaded regions of the graphic 2). Consequently, the relationships among different and disjoined slice will be applied as mechanisms to reduce the volume of sentences to analyze.

It has been considered convenient to detail the technique the style of an algorithm as follows:

- 1. To calculate the pair of slices.
- 2. To detect relationships between slices (difference, inclusion or disjoint). If there is inclusion return to 1.
- 3. To select a sentence in a slice from which the analysis will be continued. If aren't any more sentence to consider go to 8.
- 4. To request the tool that detects identical sentences in the other slice (As of here, the second slice will try to reflect the dependencies that have been detected in the first and that the analyst has confirmed).
- 5. To accomplish backward chaining (or forward) to detect sentences that may have data or control dependencies.
- 6. Once detected the sentence in the first slice, observe if the calculated sentence in the second slice is conceptually identical to the one calculated in the first slice. In this case, accept correspondence between the calculated components. On the contrary return to 3.
- 7. With the new calculated component return to 5.
- 8. With the conceptually equivalent sentences and the structural relationships between them, design the programming plan.

Some explanations are necessary. In the first step, the calculated slices can be complete, semi-complete, or gradually calculated, evaluating the usefulness of the sentences that have been incorporated in order to design the programming plan. The gradually calculated slices is an other mechanism that is used to restrict even more the number of sentences to analyze.

Continuing with the step 1 and as a way of making the process more efficient, not necessarily the exploration is restricted to a pair of slices but it can be perfectly amplified to more than two slices.

In the step 3 it is necessary to indicate from what sentence the analysis will be continued. It has been observed that in certain cases the calculation of the dependencies "bring" sentences of which correspondence in the other slice is not found. In some opportunities this occurs due to the fact that the programming plans are delocatized, what makes necessary to continue the analysis from other sectors of the slice. Thereinafter it is sought to assemble the scattered segments through the corresponding dependencies.

In the step 5 any type of chaining can be applied indistinctively. Normally, one has been begun with backward chaining for example, then one should continue the analysis with the same, as long as the sector of the slice that is being analyzed has not changed. That is to say, it should be possible to apply different types of chaining in different sectors

of the slice, and then combine the results. It is clear that to accomplish this, the tool must preserve the track of the correspondence that has already been detected.

In the step 6 is where the participation of the analyst actually is required, since his observation is vital to know if the calculated sentences are or not equivalent, and consequently the calculation of the successive dependencies will continue or not from there. The fact that a correspondence between sentences is accepted or confirmed, implies that a link between them has been created and that the automatic support must maintain them.

With respect to the stage 8, actually it doesn't belong to the same technique, simply it has been introduced at the end to complete the ideas tried to transmit. There is not any information in this sense, the design and the incorporation programming plans in the libraries would be a subsequent stage within process of construction of a repository of this type. However, it can be glimpsed that once detected the segments of source code with potential programming plans, the analysis base on the technique here proposed, some of the extracted ideas from [QYW98] and reproduced in the section 1 would be applied accomplishing, since there would be availability of components and restrictions necessary to accomplish eliminations, generalizations, etc., activities impossible to accomplish previously.

According to what has been defined here, the designed technique requires an important automatic support. Among the automatic tasks it is emphasized:

- Calculation of slices and detection of relationships among them (difference, inclusion or disjoint).
- Simultaneous manipulation of two or more slices. This implies:
  - To execute backward chaining or forward chaining from different sectors of the slices.
  - To maintain the track of the links among the equivalent sentences considered by the analyst.
  - To permit to section the analysis on slices to enable the detection of delocalized plans.

The analysis by sectors of the slices implies that the correspondence found in different sectors of the slices should be preserved, to permit the subsequent assembly with the rest of the correspondence detected in other sectors.

With respect to calculation of the slices the tool must permit to calculate complete slices, semi-complete or to calculate slices gradually, as well as its storage and subsequent recovery.

To complete the environment, while the analysis is accomplished the plans of programming should be designed. Therefore, the system would have to permit not only to create, to modify, to eliminate, etc., the hierarchies of programming plans, but also, to preserve the instances from which its components have been abstracted. Moreover, the plans already entered would have to be able to be applied to the base of slices, as in the approach described briefly in [QYW98]. But this is out of the scope of the present work.

3.4 *Experimentals Results*

Below a more complex example applying the technique on code source COBOL is considered. The program sources that have been taken as example, used to integrate the system of control of academic management of the students of the university. This application is

composed by approximately 40 files .cbl, each one containing 1000 lines of code in average. The calculation of the slices has been accomplished on the file Alum05.cbl, of about 1500 lines. Due to space reasons, it is not reproduced here. As there is not a slicer, the calculation of slices has been executed manually.

It is supposed that the analyst is interested in designing a programming plan to have access and travel through indexed files, for which he has calculated semi-complete slices on the key item CLAVE-CURR[7] but from different sectors of the program source: the first from line 1388 and the second from line 1357. However, the semi-complete slices that there were calculated do not include the sentences of files reading. On the other hand, the analyst's knowledge indicates to him that a plan to have access and travel through files must have an abstract component that synthesizes that operation. Therefore, executing forward chaining starting from the sentence that has been specified in the slicing criteria to calculate the semi-complete slice, adds new sentences in the slice until finding the sentence that executes the reading of the file. Figure 7 shows the final slices that the analyst has calculated.

Firstly the tool detects what slices are considered in intersection, but as it has already been indicated previously, in this case the only interest are the similarities that could be observed in the sentences that integrate the differences in both slices.

As it is observed in line 1395 of the first slice and in line 1365 of the second, there is an operation of reading. Now the tool is requested to indicate what the dependencies of control of such sentence are. In the first of the slices such dependency leads to line 1392 while in the second to line 1359, both iteration sentences are therefore considered conceptually identical by the analyst. At the same time, the reading sentence depends on the location of the pointer that in the first slice is accomplished in sentence 1389 and in the second in sentence 1358. In the same way, the location of the pointer depends on the value on key whose initialization is produced in the first slice in line 1388, while in the second slice the "setting" is accomplished in line 579, which at the same time has multiple data dependencies on the previous sentences: 551, 562, 566, 570, 574 and 578. As a consequence, this set of sentences is conceptually equivalent to line 1388 of the first slice, since they also iniciate the key but applying different mechanisms.

Base on the detected equivalences the analyst can design a plan hierarchy as the one is shown in figure 8.

It is worth observing, that the component "*Open File*" is incorporated for achieving of completion of programming plan, and not as a consequence of the abstraction of sentences present in the differences of both slices, since the sentence that opens the files is common to both slices.

As it has already been observed, the purpose of the second slice is to have an other scheme of reference if the knowledge of the domain problem is limited. Regrettably, here there exists the problem that in "lagacy" programs of hundreds of thousands of lines of code source, there will have to be a whole slew of slices calculated, and it does not result very comfortable to analyze slices in pairs.

On the other hand, the fact that a slice does not show similar algorithm structures to the rest of the calculated slices, does not necessarily imply that does not contain an instance of some programming plan, if this exists. Simply can occur, that such plan is implemented only once in the current application, but there can be other instances in

[7]The key field CLAVE-CURR is defined as key of access and is composed by the sub-fields that appear in the line 1388. For reasons of space the Working Storage Section is not included.

```
420   PROCESO SECTION.
421   APER.
422       OPEN I-O PERSONAL CURRICULA PLAN.
          ...
424   COMIEN.
          ...
434       PERFORM MENU-PRIN UNTIL OPC = 13.
435   CIERRE.
436       CLOSE IMPRESORA PERSONAL CURRICULA PLAN.
          ...
438       STOP RUN.
      *** RUTINAS ***
440   OP.
          ...
          ACCEPT OPC LINE 24 POSITION 78 PROMPT ECHO TAB.
      PAUSA.
          ACCEPT SEGUIR LINE 24 POSITION 79 PROMPT.
          ...
468   MENU-PRIN.
469       MOVE 1 TO TDOC-IN.
          ...
471       PERFORM MENU-PRI2.
472   MENU-PRI2.
          ...
493       MOVE 1 TO B1.
494       PERFORM ING-OPC-2 UNTIL B1 = 0.
          ...
496       PERFORM DERIVA-1.
497   ING-OPC-2.
498       PERFORM OP.
499       IF OPC IS NUMERIC AND OPC < 14 MOVE ZERO TO B1.
500   DERIVA-1.
          ...
534       IF OPC = 11
535           PERFORM LISTACURR UNTIL TDOC-IN = 9
536       ELSE ...
          ...
1385  LISTACURR.
          ...
1388      MOVE 0 TO CPO-BLANCO TIP-DOC1 NRO-DOC1 COD-FAC-C...
1389      START CURRICULA KEY IS NOT < CLAVE-CURR...
1390      PERFORM LEER-CURR-22.
1391      MOVE 0 TO OPC.
1392      PERFORM LISTAR-1 UNTIL OPC = 1.
1393      MOVE 9 TO TDOC-IN.
1394  LEER-CURR-22.
1395      READ CURRICULA NEXT AT END MOVE 1 TO OPC...
1396  LISTAR-1.
          ...
1399      PERFORM LEER-CURR-22.
          ...
1581  END PROGRAM.
```

```
420   PROCESO SECTION.
421   APER.
422       OPEN I-O PERSONAL CURRICULA PLAN.
          ...
424   COMIEN.
          ...
434       PERFORM MENU-PRIN UNTIL OPC = 13.
435   CIERRE.
436       CLOSE IMPRESORA PERSONAL CURRICULA PLAN.
          ...
438       STOP RUN.
      *** RUTINAS ***
440   OP.
          ...
          ACCEPT OPC LINE 24 POSITION 78 PROMPT ECHO TAB.
      PAUSA.
          ACCEPT SEGUIR LINE 24 POSITION 79 PROMPT.
          ...
464   ING-OPC-1.
465       PERFORM OP.
          IF OPC = 1 OR OPC = 2
              MOVE 0 TO B1.
468   MENU-PRIN.
469       MOVE 1 TO TDOC-IN.
471       PERFORM MENU-PRI2.
472   MENU-PRI2.
493       MOVE 1 TO B1.
494       PERFORM ING-OPC-2 UNTIL B1 = 0.
495       MOVE 0 TO BANALTA.
496       PERFORM DERIVA-1.
497   ING-OPC-2.
498       PERFORM OP.
499       IF OPC IS NUMERIC AND OPC < 14 MOVE ZERO TO B1.
500   DERIVA-1.
530       IF OPC = 10
531           PERFORM CONSULMAS-CURR UNTIL TDOC-IN = 9
543   ING-CLAVE.
544       MOVE 0 TO TDOC-IN NDOC-IN CFAC-IN CCAR-IN ASIG-IN.
551       ACCEPT TDOC-IN LINE 4 POSITION 9 PROMPT ECHO TAB.
552       IF TDOC-IN NOT = 9
553           IF TDOC-IN = 0
554               MOVE T-DNI TO TDOC-IN
          ...
556           PERFORM RESTOCLAVE
557       ELSE
558           PERFORM RESTOCLAVE
559       ELSE
560       MOVE 0 TO B1.
561   RESTOCLAVE.
562       ACCEPT NDOC-IN LINE 4 POSITION 20 PROMPT ECHO TAB.
          ...
566       ACCEPT CFAC-IN LINE 4 POSITION 38 PROMPT ECHO TAB.
          ...
570       ACCEPT CCAR-IN LINE 4 POSITION 48 PROMPT ECHO TAB.
          ...
574       ACCEPT ASIG-IN LINE 4 POSITION 60 PROMPT ECHO TAB.
          IF ASIG-IN NOT > 999
              IF BANALTA = 1
                  GO TO RESTOCLAVE.
578       MOVE 0 TO B2 B1 BLANCO-IN.
579       MOVE CLAVE-IN TO CLAVE-CURR.
1349  CONSULMAS-CURR.
1354      PERFORM ING-CLAVE.
1355      IF TDOC-IN NOT = 9
1356      MOVE 0 TO OPC.
1357      MOVE CLAVE-IN TO CLAVE-CURR.
1358      START CURRICULA KEY NOT < CLAVE-CURR.
1359      PERFORM LEER-CURR-OTROS UNTIL OPC = 2.
1364  LEER-CURR-OTROS.
1365      READ CURRICULA NEXT AT END MOVE 2 TO OPC.
1366      IF OPC = 2
              NEXT SENTENCE
1581      END PROGRAM.
```

Figure 7: Calculated Slices on COBOL Program

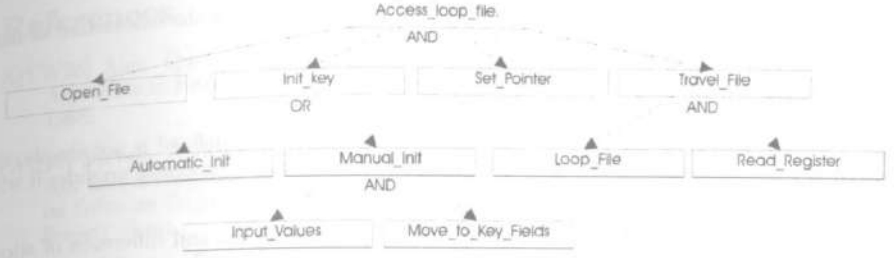

Figure 8: Programming Plan to Access and Travel Through Indexed File

calculated slices of other applications in the same domain[8].

4 Conclusions

It has been demonstrated experimentaly that a programming plan can be contained in a well calculated program slice. Starting at this point, a technique that permits to detect repetitious structures through the application of dependencies analysis of data and control has been started to be defined, that gradually define an architectural context that allows an analyst to synthesize abstract concepts from the observed instances.

Substantially, the technique permits to direct the attention of the analyst to meaningful segments' source code for the design of programming plans. The comparison of slices is highly automated reducing the intervention of the analyst to the observation of the calculated dependencies. The following stage, within a global construction process of a library of program plans, would be the same design of the plans.

Briefly, the novelties introduced by the present work are:

- Entailment slicing techniques to the programming plans.
- Application of slicing techniques as basis for the definition of a technique of programs analysis for purposes never explored before.

Observed weaknesses of the technique:

- Due to the great quantity of data items that an application can have, it can take some time to find slices that possess similar programming patterns. However, the relationships of disjoint and intersection of slices, already reduces, to a large extent, the volume of code lines to analyze.
- The establishment of the scopes of the slicing criteria turns to be an important factor since it can include irrelevant code segments in the slices or exclude useful sectors where the programming plans are present.
- Due to the fact that the programming plans can be found delocalized, even in a slice, not always will the last sentence be the starting point to begin the analysis. That is to say, it is perfectly possible to have program plans located in the intermediate zones of the slices.

[8]Here it is being dealt with the comparison of slices that belong to different applications in the same problem domain.

- Requirement of a complex automatic support that permits the calculation of data and control dependencies.

Detected strengths:

- Even though the automation of the tasks that have been defined is not simple, the fact that the technique is strongly supported by automatic processes provides it with solidity.
- The mechanisms of filter that have been defined (disjoint and difference of slices, plus the gradually calculated slices), reduce to a large extent the number of sentences to analyze.
- The automatic comparison of slices reduces to a large extent the knowledge a priori that the analyst must have to detect programming plans in an unknown source code.

On the other hand, some interesting directions to follow would be:

- To develop mathematical bases that support the hypothesis.
- To reduce the quantity of slices to be compared, an alternative would be to calculate slices base on criteria composed by more than one data item.
- To explore algorithms of slicing that permit to reduce even more the volume of sentences to analyze.
- To establish if the conceptual proximity of the data that compose the slicing criteria, have some role in the arrangement of the plans in the hierarchies, for which perhaps it will be necessary to apply techniques of clusters analysis or concepts analysis [DK99].
- These techniques also can be useful to organize a repository of slices composed by "regions" grouping "conceptually near" slices in each one, and applying the algorithms of searches in such regions.
- To experiment with algorithms based on the constraint satisfaction that explores not all the source code but a repository of slices.
- To explore the defined technique in the detection of reusable components.

With respect to the last aspect, some experiments has been accomplished, for space reasons of the results have not been detailed.

Regrettably, the lack of a slicer is a decisive limitation because the calculation and subsequent manual comparison of slices is an irksome task that considerably conditions the advance of the project and the obtained results.

In addition to the construction of the slicer that has already been begun, it is also open the construction of the rest of the tools that compose the environment that has been defined here.

References

[QYW98] Alex Quilici, Qiang Yang and Steven Woods. "Applying Plan Recognition Algoritms to Program Understanding", Journal of Automated Software Engineering, 1998.

[WQ96] Steven Woods and Qiang Yang. "The Program Understanding Problem: Analisys and a Heuristic Approach", Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, March 1996. Pages 6-15. IEEE Computer Society Press.

[WY95] Steven Woods and Qiang Yang. "Program Understanding as Constraint Satisfaction", In Proceedings of International Workshop Conference Reverse Engineering Toronto, Canada, 1995.

[CQ96] David Ching and Alex Quilici. "DECODE: A Cooperative Program Understanding Environment", Journal of Software Maintenance, 8(1):3-34, 1996.

[GL91] Keith Brian Gallagher and James R. Lyle. "Using Program Slicing in Software Maintenance", IEEE Transaction on Software Engineering, August 1991.

[LV96] Filippo Lanubile and Giuseppe Visaggio. "Extracting Reusable Functions By Program Slicing", Technical Report CS-TR-3594, University of Maryland, College Park, January 1996.

[LV97] Filippo Lanubile and Giuseppe Visaggio. "Extracting Reusable Functions By Program Slicing", IEEE Transactions on Software Engineering, 23(4):246-259, April 1997.

[Q94] Alex Quilici. "A Memory-Based Approach to Recognizing Programming Plans", Communications of the ACM, 37(5):84-93, 1994.

[W84] Mark Weiser. "Program Slicing", IEEE Transaction on Software Engineering, July 1984.

[ADS93] Hiralal Agrawal, RichardA. DeMillo, and EugeneH. Spafford. "Debugging With Dynamic Slicing And Backtracking", Software - Practive And Experience, 23(6):589-616, June 1993.

[FG97] Istvan Forgacs and Tibor Gyimthy. "An Efficient Interprocedural Slicing Method For Large Programs", In Proceedings of SEKE'97, pages 279-287, Madrid, Spain, 1997.

[HRB90] SusanB. Horwitz, ThomasW. Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs" , ACM Transactions on Programming Languages and Systems, 12(1):26-60, January 1990.

[KR97] Bogdan Korel and Jurgen Rilling. "Application Of Dynamic Slicing In Program Debugging", In Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97), Linkping, Sweden, May 1997.

[KS98] Jens Krinke and Gregor Snelting. "Validation Of Measurement Software As An Application Of Slicing And Constraint Solving", Information and Software Technology, 40(11-12):661-675, December 1998.

[QYW96] Alex Quilici, Qiang Yang and Steve Woods. "Applying Plan Recognition Algorithms to Program Understanding", Knowledge Based Software Engineering Conference. Syracuse, New York, September 1996. Pages 96-103.

[RHS95] ThomasW. Reps, SusanB. Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis Via Graph Reachability", In Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages, pages 49-61, San Francicso, CA, January 1995.

[RT96] ThomasW. Reps and Todd Turnidge. "Program Specialization Via Program Slicing", In O.Danvy, R.Glueck, and P.Thiemann, editors, Proceedings of the Dagstuhl Seminar on Partial Evaluation, Lecture Notes in Computer Science, Schloss Dagstuhl, Wadern, Germany, February 1996. Springer-Verlag.

[Z98] Jianjun Zhao. "Applying Slicing Techniques To Software Architectures", In Proc. 4th IEEE International Conference on Engineering of Complex Computer Systems, pages 87-98, 1998.

[Tip94] Frank Tip. "A Survey Of Program Slicing Techniques", Technical Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, July 1994.

[S98] Christoph Steindl. "Intermodular Slicing Of Object-Oriented Programs", In International Conference on Compiler Construction (CC'98), 1998.

[LH96] LorenD. Larsen and MaryJean Harrold. "Slicing Object-Oriented Software", In Proceedings of the 18th International Conference on Software Engineering, pages 495-505, Berlin, March 1996.

[DH99] MatthewB. Dwyer and John Hatcliff. "Slicing Software For Model Construction", In Proceedings ACM SIGPLAN Partial Evaluation and Program Manipulation, January 1999.

[HDS96] Mark Harman, Sebastian Danicic, and Yoga Sivagurunathan. "The Next 700 Slicing Criteria", 2nd UK Program Comprehension Workshop, Centre for Software Maintenance, University of Durham, July 1996.

[DK99] Arien van Deursen and Tobias Kuipers. "Identifying Objects Using Cluster and Concept Analysis", Proc. 21st. Internation Conference on Software Engineering. ACM Press 1999.

[RSW96] Spencer Rugaber, Kurt Stirwalt and Linda Wills. "Understanding Interleaving Code", Automated Software Engineering, June 1996.