Semi-automatic Generation of Parallelizable Patterns From Source Code Examples

Dejan Markovic Jack R. Hagemeister Cauligi S. Raghavendra
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164-2752

Sanjay Bhansali
Semantic Designs, 12636 Research Blvd. #C-214, Austin TX 78759-2200

Abstract

Generation of program patterns from source code is a difficult, time consuming and error-prone process when performed by programmers. We describe an implemented system which generates patterns from an abstract syntax tree with interaction by the user. Our approach is based on creating intermediate patterns by exploring data dependencies in the source code and allowing the user to change and/or eliminate parts of it in order to create a final pattern. We describe the architecture of our system as well as the pattern language used, and illustrate our approach with examples.

1. Introduction

Many software engineering tasks are being automated using program comprehension. One significant method of utilizing program comprehension uses program patterns to identify concepts of interest within source code. These patterns are written to allow a software engineering tool to assist a programmer in finding code segments of interest for maintenance [8] or within automatic parallelization tools [3]. A programmer may have to write a pattern at the time of search to locate code parts of interest. Additionally, a set of patterns may be collected in a knowledge library to be used to search a large set of programs for concepts of interest.

Writing such patterns is difficult and prone to programmer error. The syntax for constructing patterns can be difficult since the programmer may have to guess at the dependencies within code fragments and may not understand all the possible permutations of the implementation of a program concept of interest. We also believe that programmers will use examples from existing code to formulate the patterns for code fragments which are of interest to them. For these reasons, it is beneficial to have a tool to automatically generate appropriately generalized patterns from example code fragments.

We have implemented such a tool to assist with pattern development in the recognition of concepts for algorithmic code transformation for parallel computing. It uses an annotated abstract syntax tree to analyze code examples and user/programmer inputs to generate patterns which recognize parallelizable concepts in scientific FORTRAN programs. In this first experiment, we used programmer input to assist and guide the generation of patterns. In this paper we first briefly describe the SPUR project for automatic parallelization of existing sequential code, and then concentrate on one of its tools, the pattern-builder. The pattern language, the analysis of data dependencies and the system for generating patterns are described. We give examples of the generation of a pattern from source code and explain the analysis process.

2. SPUR project

One significant barrier inhibiting the use of parallel computing is the difficulty of writing parallel software. In addition to that, the scientific community has accumulated a large corpus of sequential programs to support their research efforts, and are unwilling to give up on these efficient and well tested programs. SPUR project proposes a solution to the problem of parallelization by building tools that would automatically (or semi-automatically) replace these sequential programs by equivalent, more efficient parallel programs from the library. The approach is based on a knowledge base of patterns that represent concepts in a specific domain that are amenable to a parallel solution.

An architectural overview of the system is shown on Figure 1. The first phase of the system consists of building the pattern library. The parser reads sequential FORTRAN source programs that represent base concepts we want to recognize and builds an annotated representation of the abstract syntax tree that is used by the pattern-builder to generate patterns and store them in the pattern library. In the second phase, the pattern matching engine performs a systematic search, under domain specific direction from a user, of the abstract syntax tree created from the program we want to parallelize for domain concepts that are amenable to parallelization. In the third phase, the code transformer utilizes a library of target replacement code for replacing matched patterns by the efficient parallel code. This phase will be interactive, with the system suggesting

1092-8138/97 $10.00 © 1997 IEEE

potential regions of code to be parallelized, and the user making the final decision on code replacement. A verification tool is planned to allow evaluation of known test cases to ensure that specification semantics are preserved after a code transformation .

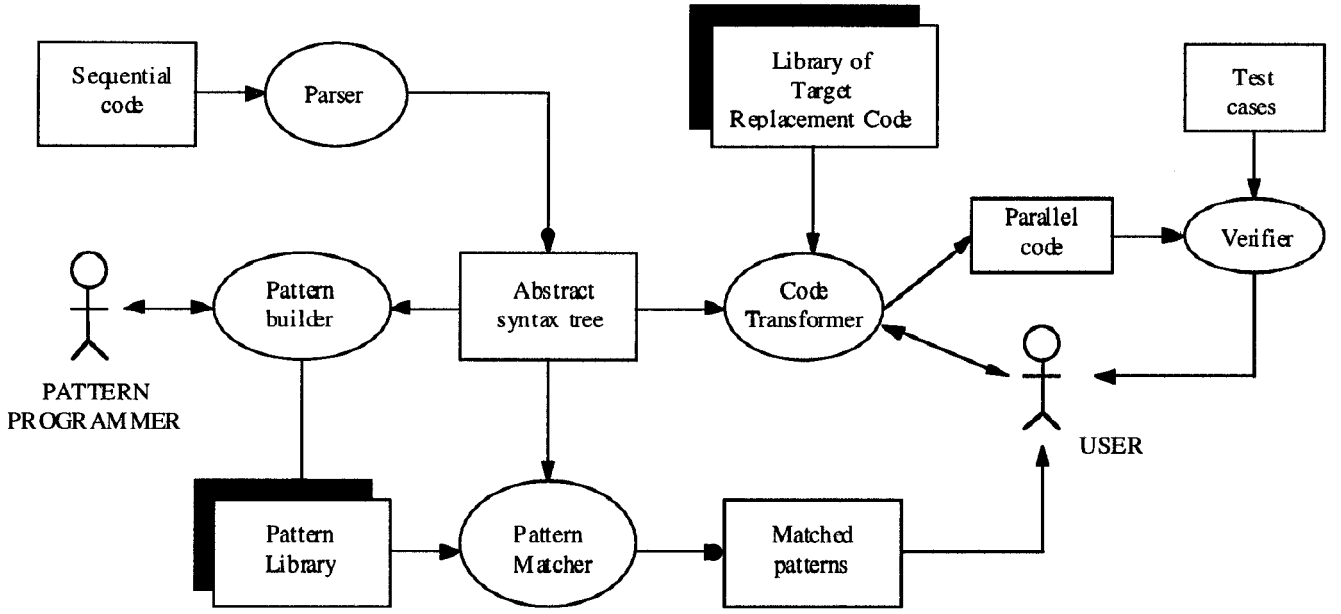

Figure 1. Architecture of an knowledge-based parallelizing tool

3. Pattern-building tool

Any system designed to automatically generate patterns from source code examples must rely on the syntax of the language and the data dependencies. Our system represents a way to gather semantic information for a program and instantiate a semantic fragment. In this section we first describe the pattern language used as the target of automatic pattern generation. We then discuss data dependency analysis and give brief examples. Next, we outline user involvement in the system and, finally, show how we combine user involvement and data analysis to generate valid patterns for semantic fragments from source code examples.

3.1. Pattern language

Our pattern language is designed to facilitate searching for complex application-specific concepts for which exist alternative parallel algorithms or system-specific library routines. The primary concern is that the pattern language will be expressive enough to describe *programming concepts* as well as *domain concepts* (as defined by Kozaczynski, Ning and Engberts (1992)) [5]. The pattern language should also be close to the syntax of the target programming language to aid in readability. Finally, it should be possible to search efficiently for patterns in the source code.

The pattern language uses a functional notation that is designed to facilitate efficient pattern matching. Since our target programming language is FORTRAN, the syntax of our pattern language parallels that of FORTRAN. Figure 2 shows part of the syntax of the pattern language.

The keyword SEQ indicates a sequence and the keyword SET indicates a set of statements following the keyword. Normally, patterns are written using the keyword SEQ. However, there are cases where we want to match statements and are not concerned with the order in which they occur. In the absence of the SET construct, we would have to write several different patterns, one for each possible permutation of the statement. By using the SET construct, we can write one pattern and leave the pattern matching engine to automatically try all possible combinations during matching. Examples of patterns using these constructs are given shortly.

In addition to the basic pattern language, we introduce a set of symbols that match different syntactic entities in the source code. These symbols can be thought of as typed wild cards that match different kinds of source code fragments. Providing such typed wild cards increases the readability of patterns, reduces errors in writing patterns, and increases the efficiency of the pattern matcher. The pattern symbols are shown on Figure 3.

```
pattern-def = (PATTERN pattern-name (args) pattern-body [constraints])
pattern-body = (SEQ pattern-body ) | (SET pattern-body) | Lstatements
Lstatements = Lstatements | Lstatement
Lstatement = (label statement)
label = (NO-LABEL) | (LABEL INTEGER)
statement = (ASSIGN identifier expression) |
            (ASSIGN array-term expression)
            (IF-THEN condition pattern-body) |
            (IF-THEN-ELSE condition pattern-body pattern-body) |
            (DOFOR label-num index start end step pattern-body) |
            (GOTO label-num) |
            (CONTINUE) |
            (CALL subroutine-call) |
            (RETURN expression) |
            (STOP)
subroutine-call = (SUBROUTINE symbol expressions)
array-term = (ARRAY identifier dimension array-args)
array-args = expression [expression ...]
expression = arithmetic-exp | boolean-exp | relational-exp | function-call
function-call = (FUNCTION symbol expressions)
```

Figure 2. Syntax of Pattern Language

Syntactic Entity	Pattern Symbol
variable	?v
array variable	?a[...]
functions	?f(...)
expression	?e
statement	?s
integer	?i
real	?r
type	?t
declaration	?d
label	?l

Figure 3. Symbols used for syntactic entities in source code.

The ellipses (...) in the array and function entries stand for a list of arguments that can themselves be other wild cards, identifiers, or constants. In addition we use the following notation to denote argument lists:

- $* = 0 or more arguments
- $+ = 1 or more arguments

For example, *?a[$*]* would match *X*, *X[i]*, *X[i,j]*, etc. and *?f($+)* would match *abs(i)*, *gcd(i,j)*, etc.

A * is also used to indicate optional entities and is essential with certain patterns. For example, a common practice in FORTRAN is to skip the step size of a DO loop (which defaults to 1). However, many programs will explicitly set the step size. If the step size is not important to the concept we are trying to recognize, we would have to write two patterns, one with the step size omitted and one with the step size explicitly given, although there would be no other difference in the two patterns. By using the optional wild card, *?e**, we can recognize both forms using one pattern and thus eliminate the redundancy in writing patterns.

All pattern symbols can be named, e.g. *?a_name* where *name* can be any symbol made of alphanumeric characters. The name is used to bind a variable so that it can be used in another part of the pattern. This allows us to express certain constraints and restrict the kinds of code fragments that are matched. We do not allow optional entities to be

named. However, argument lists to arrays and functions can be named, e.g. *?a_a[$+_1]*. To handle such cases, we found it necessary to augment the pattern language with a set of additional constraints. These constraints are implemented using a set of service routines after the pattern matcher completes a syntactic match and binds all pattern variables. Such service routines were also used by Kozaczynski, Ning et al.

Finally, we use the symbol *?P_pattern-name* to match other patterns. Here, *pattern-name* is the name of a pattern defined in the pattern library. (Note that unlike other wild card symbols, the name for the pattern is mandatory). With the pattern wild card we can compose patterns from sub-patterns in a hierarchical manner.

3.2. Data dependencies

One of the most important issues in generating patterns is the ability to determine whether parts of original code can be moved without changing the program's semantics. If we embed that information in our pattern, we will be able to use it to recognize pieces of code that essentially differ only in initial order of the statements, but have the same semantics. We show that this step in pattern generation can be completely automated using data dependency analysis.

The first step in our process consists of analysis of data dependencies and creation of a data dependency graph for each sequence of statements in our source code. We construct a separate graph for the main sequence and for each of the sequences contained within statements (bodies of if and do statements). We deal with dependencies from "inside" sequences as if they were generated by enclosing statements themselves in an outer data dependency graph. This approach prevents us from recognizing variations of code having unrolled loops for a few iterations or having statements inside both branches of 'if' with the same pattern. As we will discuss later, it is necessary to enhance pattern language itself to handle these situations.

Creation of a data dependency graph for every sequence is accomplished by creating a symbol table, finding where and how each symbol is used and, finally, using this information to create lists of statements-predecessors and statements-successors for each statement in the sequence. We look for three types of data dependencies:

- *read after write* (real data dependency) - if statement s writes variable x, statement t reads that variable , and statement s preceeds statement t in original code, then statement t is dependent on statement s.
- *write after read* (anti-dependency) - if statement s reads variable x, statement t writes that variable , and statement s preceeds statement t in original code, then statement t is dependent on statement s.
- *write after write* (output dependency) - if statement s writes variable x, statement t also writes that variable , and statement s preceeds statement t in original code, then statement t is dependent on statement s.

Write after read and *write after write* dependencies are not true dependencies because they can be eliminated by introducing extra variables into program. However, we still look for these dependencies in the source code because their existence limits freedom in the pattern.

The presence of arrays slightly complicates data dependency analysis. Arrays are mostly used in loops, therefore creating possibilities for existence of loop-carried dependencies (dependency between operation s in iteration i and operation t in iteration $i+k$). We decided not to implement analysis of loop-carried dependencies at this stage, because the pattern language and pattern matcher can not support pattern recognition based on information that can be obtained from such analysis. Instead, we decided to treat array as a single variable (therefore treating using or changing any element of the array the same way as using or changing the whole array). The only difference between arrays and scalar variables is that we treat blocks of successive read or write operations as a single read or write operation , making each read of the current block dependent on all the writes from the previous block (in the case of read after write dependency) or each write in the current block dependent on all the reads from the previous block. That prevents us from losing dependencies between a loop and several statements that change (or use) particular elements of the array before or after the loop.

After creating a data dependency graph, we compute the earliest and latest point for execution of each statement in it. This helps us to determine whether that statement has freedom to move within code. Following this step, we are able to identify pattern parameters as variables whose first use is reading.

In the second step of our algorithm we determine whether the data dependency graph for each sequence is connected. If it is not, we can represent the sequence as a set of (sub)sequences, and apply processing from the next step to each subsequence separately.

The final step involves optimization of the data dependency graph for each pattern. Since our pattern language cannot express all valid combinations of statements (see discussion below), our goal is to create patterns that would be able to represent most of the valid combinations. We are using greedy approach because:

- Code examples from which we generalize are typically not very large pieces of code, therefore do not contain very many statements.
- Patterns at which we are looking typically do not have very complicated data dependency graphs, so greedy approach will give a solution close to optimal most of the time.

Our approach includes sorting data dependency graphs topologically (level by level) and then, starting at topmost level in the graph, performing the following:

- If there is more than one statement on the level, for each pair of topmost statements find the statement that is their highest common (not necessarily immediate) successor in the data dependency graph. Choose the pair of statements whose highest common successor (HCS) is lowest of all and create the pattern as a sequence of three subpatterns. The first subpattern consists of all (not necessarily immediate) predecessors of the chosen HCS and all their successors that are not also successors HCS, the second subpattern consists only of HCS statement, and the third consists of all successors of HCS statement. After that we recursively apply the same procedure to the first and third subpatterns.
- If there is only one statement on a given level, create a (sub)pattern that is a sequence of two parts -- the statement, and the subpattern created from the rest of the graph (can again be a sequence, or a set of sequences), and apply the same procedure to the rest of the graph.

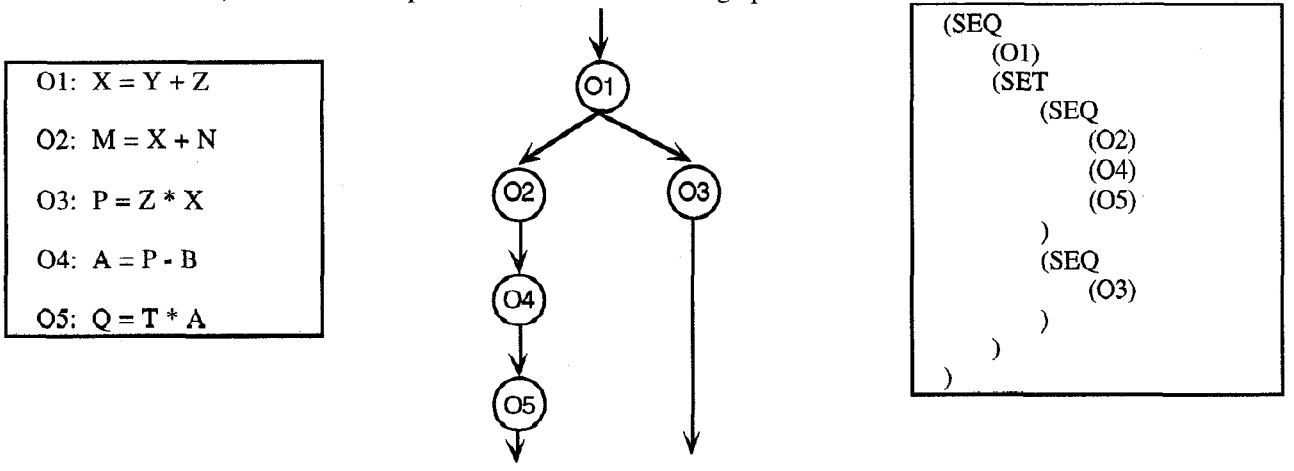

Figure 4 : Fragment of code, corresponding data dependency graph and pattern created

Figure 4 shows how we create pattern from a simple data dependency graph. In this case we first try to identify any disconnected parts of the graph. Since there are no disconnected parts, and there is only one statement (O1) on the topmost level, we create the sequence of statement O1 and the rest of graph. Then we apply the same procedure to the rest of the graph, where we find two independent parts, create a set of two sequences, and again apply the same procedure to each of the sequences in the set. The final result is outlined in the graphic above.

Note that we sometimes embed single statements in one-element sequences, and sometimes not, since both representations are recognized by the pattern matcher.

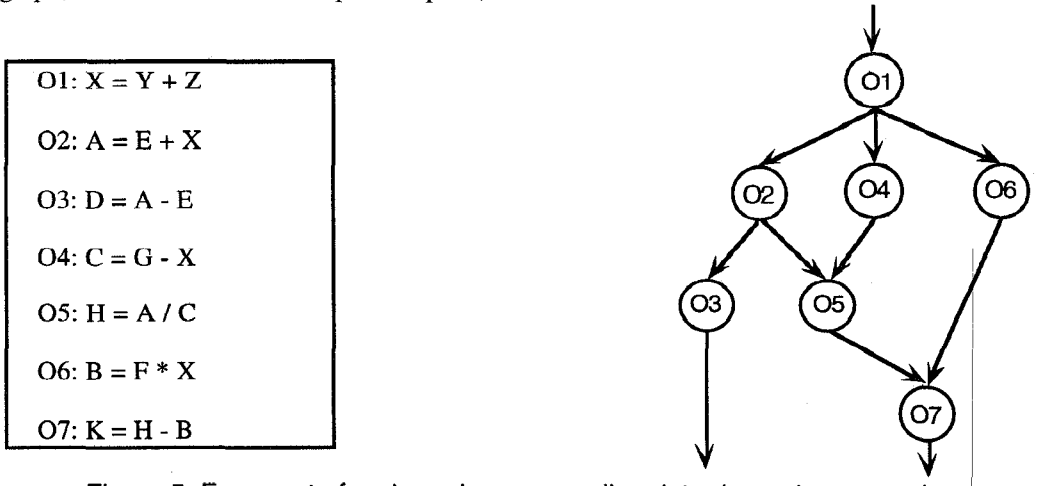

Figure 5: Fragment of code and corresponding data dependency graph

Our pattern language is able to represent pattern as either a sequence or set of elements that can themselves be statements, sequences or sets. In the Figure 5. we show a data dependency graph that cannot be completely represented

in our pattern language.

As in the previous example, we begin our analysis by determining whether there are any independent parts of the data dependency graph. Since there are none, we create a sequence of O1 and the rest of the graph, and apply the same procedure to the remainder of the graph. There, we find three non-independent statements on the top (O2, O4 and O6). We compute the highest common successor for all pairs of statements on the top level, and choose either pair (O4, O6) or (O2, O6) because they have lowest HCS (O7). Then, as we described earlier, we split the graph in three parts: all predecessors of O7 and their successors, O7, and successors of O7 (in this case we do not have this part). This way, we created artificial dependency from O3 to O7, because our three subpatterns are in sequence so any patterns thus generated would not be able to recognize any (valid) code in which O3 comes after O7. Our approach to minimization of the effects of this problem consists of creating a set of patterns that match different valid orders of statements. One method is to create the other pattern starting from statements at the bottom of the graph and looking for the lowest common predecessor for pairs of statements. In our example, we can cover all valid situations with two patterns - one created with a top-down approach, the other with a bottom-up approach. Both patterns are shown in Figure 6.

In our opinion this does not represent a serious drawback, because we expect patterns to be created from relatively small pieces of code, that do not have much freedom in data dependency graphs. Therefore, few situations would occur similarly to the example given in Figure 5.

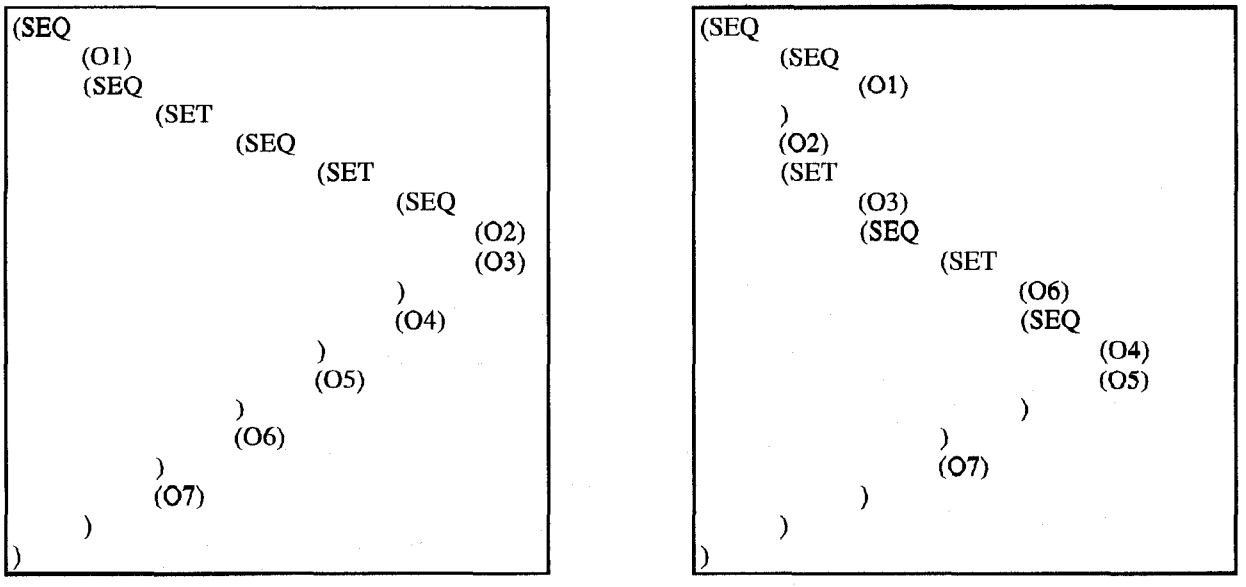

Figure 6: Patterns obtained using top-down (left) and bottom-up (right) approach

3.3. User interaction

In a data dependency graph, we can automatically create the pattern. However, parts of the code are often too specific to the particular application, and it is possible that code which should constitute our pattern is interleaved with a few statements that do not belong to our pattern, or that we need to find all the occurrences of code similar to the example (like double DO loop with arbitrary statements in the body). Sometimes, the parser needs declarations of arrays in the source code in order to distinguish between arrays and function calls in FORTRAN, but we do not want these declarations to be a part of our pattern. Also, there are parts of expressions that are not important for us (for example, we want to find all DO loops that go from 1 to N, with an arbitrary step). In these cases, the user can give us valuable information about what parts of the code we can ignore. Without user interaction (with only data dependency analysis) we would not be able to identify patterns for more general purposes.

Our pattern-generator has a graphical user interface that allows the user to perform the following:

- browse through the source code and select a part of the code from which a pattern will be created
- mark statements that are not important for our pattern (delete them) or generalize them into "statements"
- delete sequences or sets of statements
- remove unimportant expressions or parts of them
- generalize specific variables into general variables

Provided that the user may want to change some part of the pattern, a list of allowed options is provided.. For example, assume that the user wants to change the DO statement. The following set of options is given:

- generalize to "general statement" deleting body of the loop
- generalize to general DO statement, that is do statement with general variable for index, general expressions index boundaries (therefore removing its data dependencies from data dependency graph) and step and arbitrary body
- change name of index variable only
- change boundaries and/or step only
- go to loop body (another sequence or set) and change it
- insert a new statement after this statement

3.4 Combining data dependency analysis and user interaction

Our approach for combining data dependency analysis and user interaction is to perform data dependency analysis at the beginning by default since, once it is performed, the user receives information about which parts of code can be moved or what code creates data dependencies that are not important for that particular pattern, so that such code can be changed. This is the point at which user interaction can help create a pattern which is more suitable for some particular use.

Conversely, in some cases the user may want to first remove some statements which are known to be unimportant. Therefore, an option to change the pattern before performing data dependency analysis by bypassing default settings, is also provided.

Of course, after any change user made we must perform the data dependency analysis again, because user changes may affect data dependencies. This will occur automatically when the user finishes with changes. However, the option to perform it at any time will be added since the user may want to see how any previous changes affected the data dependency graph prior to continuing with other changes.

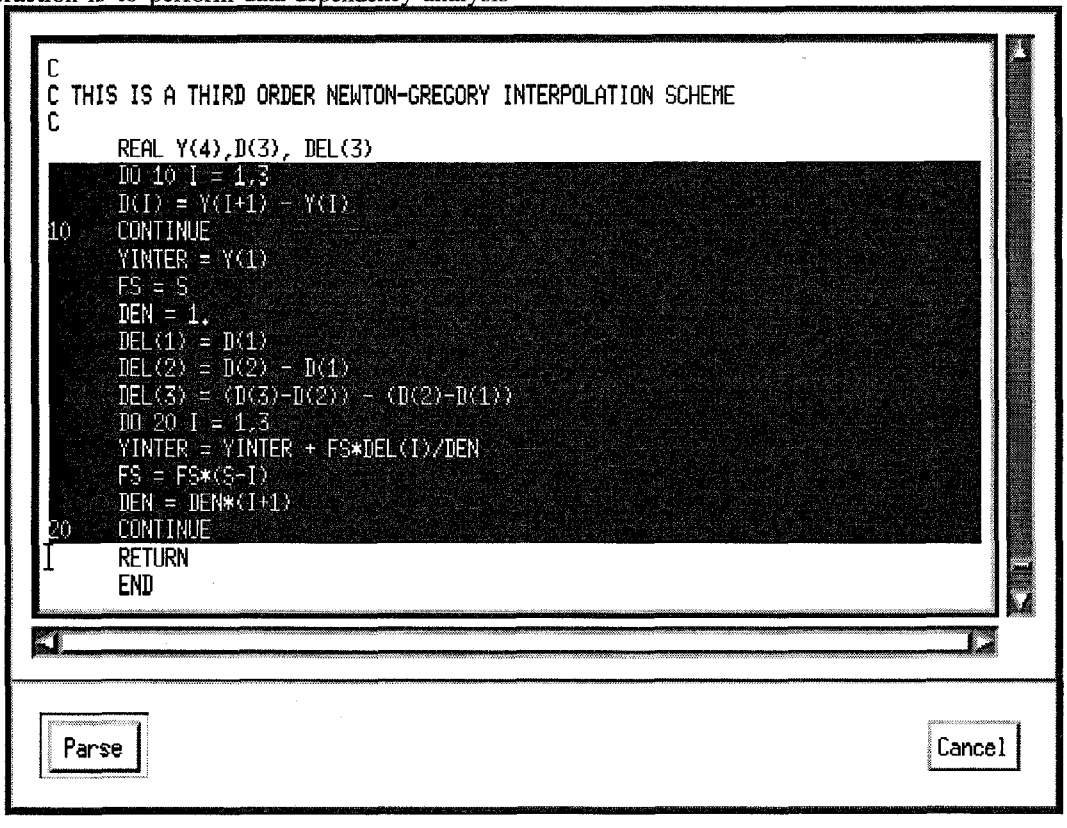

Figure 7. User can select piece of code from which pattern is to be created

4. A Complete Example

As an example of pattern generation from real world FORTRAN code, we present here generation of a pattern for a third order Newton-Gregory interpolation scheme. Figure 7 shows the initial screen where the user can select the part of code that is going to be used to create the pattern. When the user presses the "Parse" button, the parser transforms selected code to an abstract syntax tree,

which is then transformed to an initial pattern - sequence of statements in initial order.

After creating an initial pattern, the user can choose either to perform a dependency analysis or to modify the initial pattern. Since in this example we do not need any initial modifications, only the data dependency analysis is performed. The data dependency graph is given on Figure 8, and the pattern obtained is shown on Figure 9. Note that, for this data dependency graph, the same pattern is obtained using top-down and bottom up ananlysis

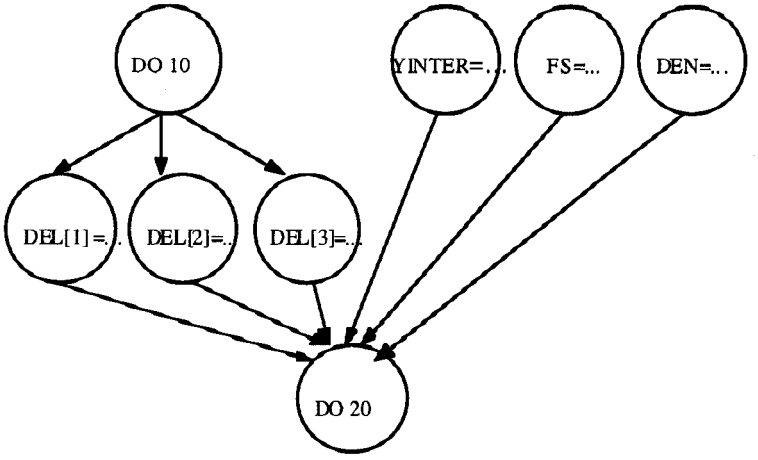

Figure 8. Top level data dependency graph for third order Newton-Gregory interpolation

```
(defpattern interp (?v_S  ?a_Y )
 (seq
  (seq
   (set
    (seq
     (lstatement ?L_1
      (dofor ?L_10 ?v_I (constant 1)(constant 3)
                        (constant 1)
        (set
         (lstatement ?L_2
          (assign ?a_D[?v_I ]
           (minus ?a_Y[(plus ?v_I (constant 1))]
                  ?a_Y[?v_I ] )))
         (lstatement ?L_10 (continue!))
     )))
     (set
      (lstatement ?L_3
      (assign ?a_DEL[(constant 1)] ?a_D[(constant 1)] ))
      (lstatement ?L_4
       (assign ?a_DEL[(constant 2)]
        (minus ?a_D[(constant 2)] ?a_D[(constant 1)] )))
      (lstatement ?L_5
       (assign ?a_DEL[(constant 3)]
          (minus
          (minus ?a_D[(constant 3)] ?a_D[(constant 2)])
          (minus ?a_D[(constant 2)] ?a_D[(constant 1)])))
      ))
    ))
    (lstatement ?L_6
     (assign ?v_YINTER ?a_Y[(constant 1)] ))
```

```
    (lstatement ?L_7
     (assign ?v_FS ?v_S ))
    (lstatement ?L_8
     (assign ?v_DEN (constant 1) ))
  )
  (lstatement ?L_9
   (dofor ?L_20 ?v_I (constant 1)(constant 3)
                     (constant 1)
     (set
      (lstatement ?L_11
       (assign ?v_YINTER
        (plus ?v_YINTER
         (divide
          (multiply ?v_FS ?a_DEL[?v_I ] )
          ?v_DEN )
      )))
      (lstatement ?L_12
       (assign ?v_FS
        (multiply ?v_FS
         (minus ?v_S ?v_I ))
      ))
      (lstatement ?L_13
       (assign ?v_DEN
        (multiply ?v_DEN
         (plus ?v_I (constant 1)))
      ))
      (lstatement ?L_20 (continue!))
   )))
)))
```

Figure 9. Pattern generated for third order Newton-Gregory interpolation

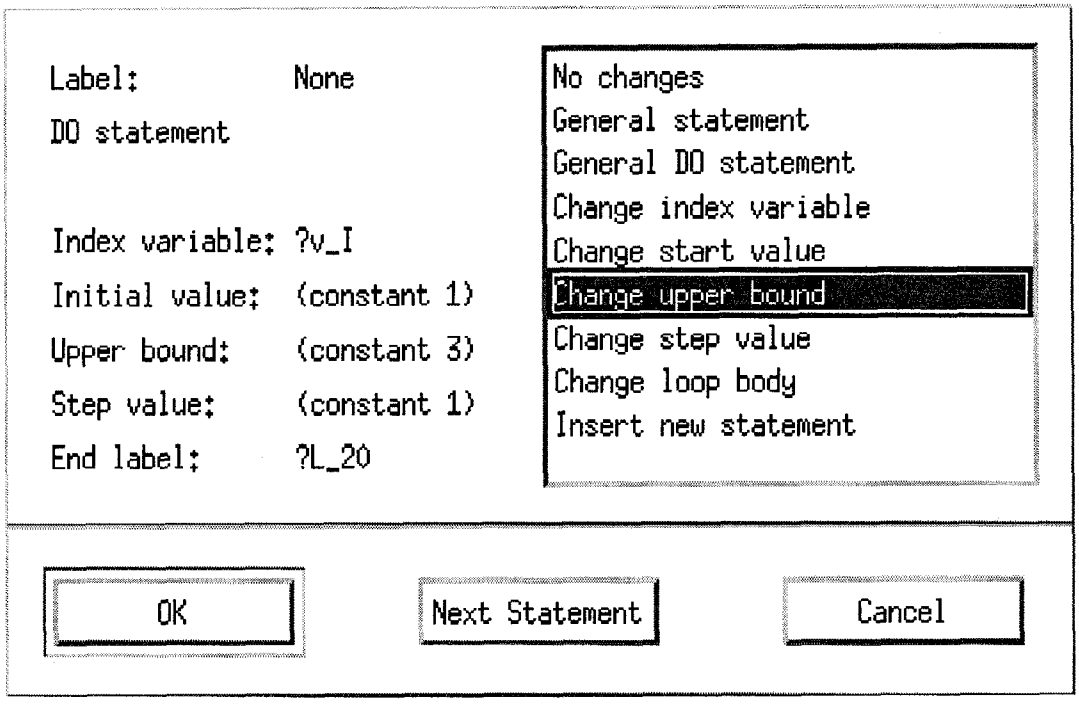

Figure 10. Example of dialog window for changing pattern for DO statement

After generating a pattern, we can change some statements. For example, if we want to make a pattern for Newton-Gregory interpolation of arbitrary order, we need to change the upper bound for DO loops and insert a general assign statement in the set of statements that assigns values to the DEL array with the constraint that DEL must be on the left side of the assignment. On Figure 10, we show an example of a dialog window for changing DO statements.

5. Conclusion

We have described an approach to pattern generation that is based on two basic techniques: data dependency analysis and user interaction. We have described details of both techniques and illustrated the application of our approach in creating patterns from parallelizable fragments of FORTRAN code by utilizing several examples.

We are currently finishing implementation of an X-windows-based graphical user interface for the system. Our next step will include testing the results of integration of this system with the pattern matcher from SPUR project. In the longer term, we have the following goals:

- To extend our system to be able to generate patterns that match several given examples of similar code.
- To improve the system by adding more methods to create the patterns from the data dependency graph, so that more patterns can be created from one example in cases where we have to deal with more complicated DDGs. The system is designed to be extensible in this respect.
- To explore possibilities for creating composite patterns (patterns that contain subpatterns) through user interaction and searching the pattern library.

References

[1] Banerjee, U. (1988) "An introduction to a formal theory of dependence analysis" *J. Supercomput. 2*, 2 (Oct.), 133-149

[2] Bhansali, S, J. Hagemeister (1995) "A Pattern-matching Approach for Reusing Parallel Software Libraries" *Proc. of the 1st Intnl. Workshop on Knowledge-based Systems for the (re-)use of program libraries*, Sophia Antipolis

[3] Bhansali, S., J. Hagemeister, et al. (1994). "Parallelizing sequential programs using algorithm-level transformations." *Proc. of the 3rd Workshop on Program Comprehension*, Washington D.C.

[4] Kessler, C. W. (1995). "Pattern-Driven Automatic Parallelization." *Workshop on Automatic Data Layout and Performance Prediction*, Rice University, Center for Research on Parallel Computing.

[5] Kozaczynski, W., J. Ning, et al. (1992). "Program Concept Recognition and Transformation." *IEEE Transactions on Software Engineering* **18**(12): 1065-1074.

[6] Martino, B. D. and G. Iannello (1994). "Towards Automated Code Parallelization Through Program Comprehension." *3rd Workshop on Program Comprehension*, Washington D.C.

[7] Paul, S. and A. Prakash (1994). "A framework for source code search using program patterns." *IEEE Transactions on Software Engineering* **20**(6): 463-475.

[8] Wills, L. M. (1990). "Automated Program Recognition: A Feasibility Demonstration." *Artificial Intelligence* **45**: 113-171.