Program Plan Recognition
For Year 2000 Tools

Arie van Deursen
Dept. of Software Engineering
CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
http://www.cwi.nl/∼arie/, arie@cwi.nl

Steve Woods
Dept. of Electrical Engineering
University of Hawaii at Manoa
Honolulu, HI 96822, USA
sgwoods@spectra.eng.hawaii.edu

Alex Quilici
Dept. of Electrical Engineering
University of Hawaii at Manoa
Honolulu, HI 96822, USA
alex@spectra.eng.hawaii.edu

ABSTRACT

There are many commercial tools that address various aspects of the Year 2000 problem. None of these tools, however, make any documented use of plan-based techniques for automated concept recovery. This implies a general perception that plan-based techniques is not useful for this problem. This paper argues that this perception is incorrect and these techniques are in fact mature enough to make a significant contribution. In particular, we show representative code fragments illustrating "Year 2000" problems, discuss the problems inherent in recognizing the higher level concepts these fragments implement using pattern-based and rule-based techniques, demonstrate that they can be represented in a programming plan framework, and present some initial experimental evidence that suggests that current algorithms can locate these plans in linear time. Finally, we discuss several ways to integrate plan-based techniques with existing Year 2000 tools.

1991 Computing Reviews Classification System: D.2.2, D.2.3, D.2.7., D.3.4, F.3.1, I.2.2.

Keywords and Phrases: Software maintenance, program understanding, plan-based concept recovery, COBOL, Y2K.

Note: To appear in *Proceedings of the 4th IEEE Working Conference on Reverse Engineering*, October, 1997

Note: Work carried out under project SEN-1.1, *Software Renovation*.

1 Introduction

The Year 2000 problem (generally abbreviated Y2K) is that many existing software systems that manipulate dates will behave incorrectly at the turn of the millennium. Y2K is one of the most severe problems the software industry has ever faced [7, 16]. As a result, many tools have been developed to address the Y2K problem [10, 24]. These tools deal with system inventory, impact analysis, project planning, code remediation, testing, and so on, using existing technology such as lexical pattern matching (grep-like facilities), repositories, parsing, and attribute grammars [1].

Surprisingly, however, none of these tools makes any apparent use of the results of research in using plan-based techniques for concept recovery [15, 18, 11, 8, 5, 2, 22]. A program plan describes common combinations of low-level program actions that implement higher-level design concepts (such as "traverse a list" or "read a file line by line"). A plan-based approach recovers design concepts by taking a library of program plans and automatically identifying the pieces of source code that actually implement such plans. An obvious application of this approach to Y2K is to construct a library consisting of typical correct date-manipulating plans (such as incrementing or comparing years, checking leap years, and so on). Furthermore, a list of typical, often encountered errors can be represented as plans in this library. Given such a library, many Y2K infections could be located, classified, and potentially corrected automatically.

In this paper, we study what plan-based techniques for concept recovery have to offer Y2K tools. In particular, we present examples of representative Y2K-related code fragments, discuss existing Y2K technology and some notable shortcomings, describe available plan-based technology and its relationship to existing Y2K tools, and discuss a scalability experiment in recognizing Y2K program plans.

Our focus is on recognizing *leap year computations*. Although incorrect leap year computations are just one aspect of the millennium problem, the result we present can easily be generalized to other Y2K-related computations, such as recognizing computations relying on date windows. In addition, leap year problems can be substantial. Many programs fail to recognize the year 2000 as a leap year[1], considering it as a century year without recognizing it as a year divisible by 400 as well [14, Chapter 4]. An example of the cost that might be involved is the $1,000,000 damage caused by the fact that the control computers of a New Zealand aluminum smelter simultaneously went down as they could not deal with February 29th 1996 [17].

2 Current Y2K Tool Support

Various tools are available to support a Y2K conversion [9, 4, 10]. Most of the existing Y2K tools are focused on two areas:

- *Locating Y2K related code* by identifying date-manipulating elements in source code and then using slicing techniques to identify dependent code.[2] This identification is done by examining variable declarations (e.g., noting date-related identifiers such as `Year` or `Date` and related data formats such as Cobol pictures of the form MM-DD-YY) and expressions and statements (e.g., noting expressions involving key constants such as 4, 28, 29, 100, 365, 2000, and so on).

- *Supporting Y2K code changes* by identifying suspicious expressions and statements within the code (e.g., year increments and comparisons involving date elements) and making some automatic repairs (e.g., widening year fields to four digits).

Much of the process of locating Y2K code, and some of its modification, is automated, although it may require some assistance from the programmer (such as suggesting program-specific candidate identifier names). However, the heuristic recognition of Y2K code leaves open the possibility of both false positives (recognizing code as potentially date-related when it is not) and false negatives (failing to recognize code as date-related when it is). It’s easy to avoid recognizing false negatives simply by considering everything to be date-related, but at a cost of having more false positives. Therefore, the main challenge of Y2K tools is to avoid false positives.

2.1 A Y2K Leap-Year Example

Figure 1 is an example Y2K fragment (taken from real-world legacy COBOL code) that correctly uses a four-digit date, rather than a two-digit date, but incorrectly tests whether the variable `CONTRACT-SY` is a leap year. This means that when processing dates after February 28th, 2000, errors may occur in computations involving the number of days (e.g., interest payments) or the day of the week (e.g., determining weekend days for time locks).

```
01 CONTRACT-INFO
   ...
   05 CONTRACT-SM  PIC 99.
   05 CONTRACT-SD  PIC 99.
   05 CONTRACT-SY  PIC 9999.
...
DIVIDE CONTRACT-SY BY 4 GIVING Q REMAINDER R-1
DIVIDE CONTRACT-SY BY 100 GIVING Q REMAINDER R-2
...
MOVE ’F’ TO LY
IF R-1 = 0 AND R-2 NOT = 0
  MOVE ’T’ TO LY
END-IF
...
IF LY = ’T’
  [leap year related code]
END-IF
```

Figure 1: Example of non-compliant Y2K code.

Because the definition of a leap year is relatively complex and many programmers did not have a correct definition available while programming, leap year computations are often done incorrectly and cases are frequently missed [14]. This is a big problem for Y2K tools, since it provides further evidence that it is not sufficient to carefully replace two-digit dates with four-digit dates.

The ideal Y2K tool should identify this chunk of code as Y2K related (despite its using a four digit date), identify the pair of divisions and remainder tests as being an incorrect check for whether we have a leap year, and automatically transform that portion of the code to correctly test for leap years, as shown in Figure 2.[3] However, this example illustrates several problems with current approaches to the Y2K problem.

2.2 Pattern-Based Techniques

One approach to locating Y2K-relevant code is to write simple patterns, either lexically-based (dealing directly with the source code entities), AST-based (dealing with the internal nodes of the abstract syntax tree), or a combination of the two (looking for names in a particular place in the tree). This approach suffers from three problems.

First of all, it is difficult for pattern-based techniques to accurately recognize Y2K instances, without admitting many false positives. Straightforward lexical searches for standard identifiers such as `YEAR` will fail to flag the fragment of Figure 1. Extending them to try more complex

[1] Leap years are those years that are divisible by 4 but not by 100, unless they are divisible by 400 (so 1996 and 2000 are leap years, 1900 and 2100 are not).

[2] As well as to identify dependencies on control input, data dictionaries, screen definitions, and so on.

[3] This example shows a simple change that fixes the problem solely through an insertion of new code.

```
01 CONTRACT-INFO
   ...
   05 CONTRACT-SM  PIC 99.
   05 CONTRACT-SD  PIC 99.
   05 CONTRACT-SY  PIC 9999.
...
DIVIDE CONTRACT-SY BY 4 GIVING Q REMAINDER R-1
DIVIDE CONTRACT-SY BY 100 GIVING Q REMAINDER R-2
DIVIDE CONTRACT-SY BY 400
  GIVING Q-3 REMAINDER R-3
...
MOVE ’F’ TO LY
IF (R-1 = 0 AND R-2 NOT = 0 ) OR R-3 = 0
  MOVE ’T’ TO LY
END-IF
...
IF LY = ’T’
  [leap year related code]
END-IF
```

Figure 2: A fixed version of the Y2K code example.

lexical heuristics (such as assuming that variables ending in Y are date-related) will succeed for our example—at the cost of false positives (such as hypothesizing that SALARY is date-related). The obvious alternatives, such as examining the AST for expressions that involve dividing by 4 and storing the remainder, will also suggest CONTRACT-SY as a possible year, at the cost of other false positives (such as hypothesizing that computing a QUARTERLY-PAYMENT from an ANNUAL-PAYMENT is doing a date-related computation). Another seemingly good idea, checking for division by 100, is just as bad, as that is a common way to handle percentages.

In fact, accurately identifying this code as date-related involves looking at interrelationships between code fragments. In particular, at a minimum it requires noting that the same value is being divided by 4 and 100 and the remainders computed in the division are later compared against zero.

Secondly, it is difficult for pattern-based tools to accurately determine the specific code at the heart of a Y2K problem. Thus, even if the above heuristics could be refined slightly to identify this example, it’s still necessary to identify the source of the problem to the user. While it’s possible to provide the user with the entire data slice related to this code as potentially problematic, that is essentially a false positive for most of the code in that slice. Alternatively, simply tagging the divisions as suspicious is also insufficient, as the set of suspicious code should include the related IF that tests the remainders. In fixing the code, however, it’s necessary to do one of two things: either ensure that R-1 and R-2 have appropriate values after the leap-year computation or modify the test in the subsequent IF.

Finally, it is difficult for pattern-based tools to verify that a particular piece of code is Y2K compliant. Obviously, not all Y2K-related code is in error. Figure 3 is a correct leap year computation that the programmer has conveniently written using nice clear names, so that it is easy to flag as being date-related. However, simply providing the user with this slice of code and suggesting that it might be problematic is not particularly useful. The tools should be able to distinguish correct from incorrect code.

```
01 DATE.
   02 DAY       PIC   99.
   02 MONTH     PIC   99.
   02 YEAR      PIC   9999.
   02 CCYY REDEFINES YEAR
      03 CC     PIC   99.
      03 YY     PIC   99.
01 LEAP         PIC   X.
...
MOVE ’F’ TO LEAP.
DIVIDE YEAR BY 4 GIVING Q REMAINDER R-1.
IF R-1 = 0
  IF YY = 0
    DIVIDE YEAR BY 400 GIVING Q REMAINDER R-2
    IF R-2 = 0
      MOVE ’T’ TO LEAP.
    END-IF.
  ELSE
    MOVE ’T’ TO LEAP.
  END-IF
END-IF
...
IF LEAP = ’T’ THEN
  [Leap year-related code]
END-IF
```

Figure 3: Another Leap Year Example

2.3 Rule-Based Techniques

One alternative to a pattern-based approach is a rule-based approach. Rule-based techniques examine collections of program components and their relationships. The assumption is that problematic code fragments can be described by rules operating on the abstract syntax tree and efficiently recognized by a deductive rule-based inference engine. In particular, the assumption is that we can effectively write specific rules to identify known correct and incorrect date examples. As an example of the potential application to Y2K, Figure 4 is a rule for recognizing the fragment of Figure 1.

At first sight, the rule-based approach seems to address many of the problems with the simpler, pattern-based approach. The rule antecedents take care of verifying that particular program entities exist and that certain relationships hold between them (e.g., that there is a division by 4, that there’s an equality test on the result of that division, and so on). The rule consequences are responsible for notifying us about which particular correct or in-

```
IF
    Numeric-Variable(?V)
    Exists(Division(?V, 4, ?Q, ?R1), ?Div-1)
    Exists(Equality-Test(?R1, 0), ?Test-1)
    Data-Dependency(?Test-1, ?Div-1, ?R1)
    Exists(Division(?V, 100, ?Q, ?R2), ?Div-2)
    Exists(Inequality-Test(?R2, 0), ?Test-2)
    Data-Dependency(?Test-2, ?Div-2, ?R2)
    Same-Data(?Div-1, ?Div-2, ?V)
THEN
    Is-Year(?V)
    Recognized(Invalid-Leap-Year-1)
```

Figure 4: A rule to recognize a particular invalid leap year computation.

correct date-manipulation was detected, what variables in that code were date-related, and possibly what transformation can be used to correct the code if an erroneous date-manipulation is detected.

Unfortunately there is one important problem with the use of general rules in combination with a deductive rule-based inference engine: *scalability*. In general, rule-based systems suffer scalability problems when they have large fact bases and many complex, interacting rules. In a Y2K setting, the programs to be inspected will be large, resulting in a large database of program facts (describing the program's components, control flow, and data flow). Moreover, there will be many rules covering the many fundamentally different ways to implement various Y2K-related computations. Last but not least, the rules will be complex, because each rule has potentially many antecedents describing the pieces of a Y2K-related computation and the relationships between those pieces.

There are two approaches to dealing with scalability problems in rule-based systems. One approach is to modify the rules with additional information about how they are used (exactly when each should be applied, the order to use to process antecedents, and so on). The drawback to this approach is that placing this control information into rules makes them complex, hard-to-maintain, and difficult to debug. The other is to try to provide a special-purpose engine that is targeted toward efficiently processing a particular class of rules. This approach is more attractive, but can require considerable effort in finding an appropriate engine.

We have taken the latter option, using plan-based techniques to overcome these scalability problems. These techniques can be thought of as combining a special class of rules (the plans) with with a dedicated engine optimized for recognizing applications of rules from this class. Recent experiments have provided some initial evidence that these plan-based techniques do, in fact, scale [13].

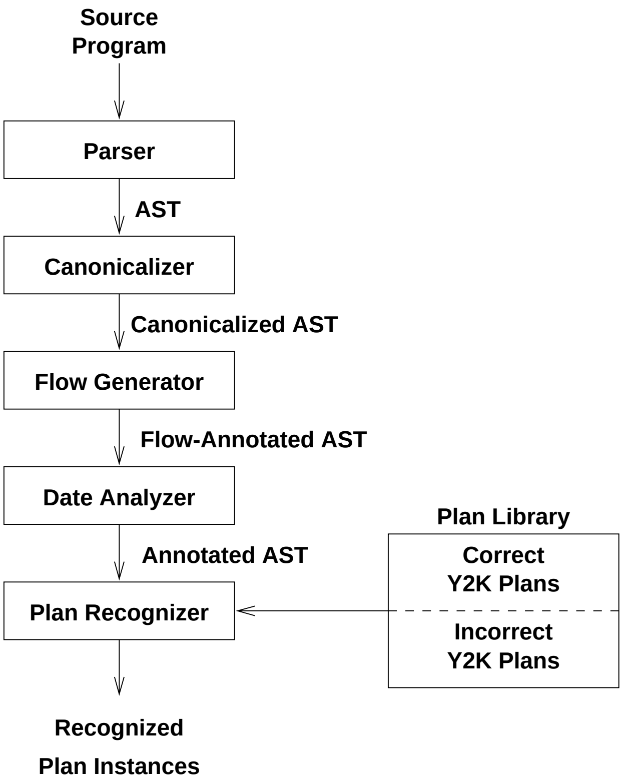

Figure 5: A straightforward architecture for recognizing Y2K program plans.

3 Plan-Based Concept Recovery for Y2K

Figure 5 shows an adaptation of a standard plan-based architecture to address the Y2K problem. The source program is fed into a parser for building an abstract syntax tree (AST), which is then passed to a canonicalization tool that handles tasks such as regularizing expressions in the AST (e.g., modifying comparisons to use only a subset of the relational operators) and to static analysis tools that produce control-flow and data-dependency graphs.

In addition, the source is fed into a Static Date Analyzer (SDA). The SDA contains "datedness" inference technology currently available in Y2K tools [4, 1]. Its tasks are to find initial date seeds based on lexical pattern matching on variable names, PIC clauses, and so on, and to propagate these date seeds according to the data flow. Effectively, the SDA phase associates date types with variables. These date types can be used in the plan library, and to reduce the search space when looking for these plans. Typically, the SDA phases only marks variables as date related or not; it does not yet decide about correct/incorrect constructs.

The plan recognizer is a special purpose engine that is given a library of Y2K plans and that tries to locate instances of them in the canonicalized AST [8, 2, 18]. Particular plan recognition engines differ in the details, but they all describe plans in terms of syntactic, data, and control flow dependencies, and view plan recognition as an explicit process of matching this description.

3.1 Our Approach To Plan Recognition

Our particular approach to plan recognition represents plans as a combination of combination of components and constraints (in the spirit of the Concept Recognizer [8] and DECODE [2]) and treats recognizing a program plan as a constraint satisfaction problem [23, 12, 20, 21].

In particular, plan components are variables, the types of components and some constraints on component attribute values are node constraints, and the inter-component constraints are arc constraints. We then use a modified version of a standard forward-checking with dynamic rearrangement constraint satisfaction engine to locate instances of plans in the code [13].[4] The engine operates breadth first, checking all possible applications of a given rule before moving on, and using properties of the constraints and the information available in the AST and flow-graph to direct the actual rule-matching process.

3.2 An Example Plan

Figure 6 is an example plan in the component-and-constraint representation. This plan is suitable for recognizing a Y2K code fragment similar in function to the one in Figure 1.

The components are syntax tree entities or sub-plans. This example specifies six components: two remainder computations, an IF statement, an equality test, an inequality test, and a logical AND. Any program containing these six components matches the plan, provided it also meets the plan's constraints.

These constraints can be restrictions on particular attributes of the components. For example, the `year` variable must be numeric, and the divisors must be constants with values 4 and 100. Alternatively, the constraints can tie the components together. For example, the `SharedDep` constraint specifies that the same value is divided in the two divisions. Similarly, the data dependencies connect the results of the divisions to the tests and the test results to the AND that combines them.

```
plan Incorrect-Leap-Year-1(In: ?year, Out: ?out)
isa Incorrect-Leap-Year-Plan

recognize
Incorrect-Leap-Year-1(Year: ?year, Status: ?out)
  components
    Divide1:
      REMAIN(Src1: ?year, Src2: ?divby1, Rem: ?rem1)
    Divide2:
      REMAIN(Src1: ?year, Src2: ?divby2, Rem: ?rem2)
    IfCond:
      IF(Cond: ?test3,
         Then: ?stmt-then, Else: ?stmt-else)
    EqTest1:
      EQ(Src1: ?rem1, Src2: ?zero,  Dest: ?t1)
    EqTest2:
      NOT-EQ(Src1: ?rem2, Src2: ?zero, Dest: ?t2)
    EqTest3:
      AND(Src1: ?test1, Src2: ?test2, Dest: ?out)
 constraints
    Numeric-Field(?year)
    Constant-Value(?divby1, 4)
    Constant-Value(?divby2, 100)
    Constant-Value(?zero, 0)
    SharedDep(Divide1, Divide2, ?year)
    DataDep(Divide1, EqTest1, ?rem1)
    DataDep(Divide2, EqTest2, ?rem2)
    DataDep(EqTest1, EqTest3, ?t1)
    DataDep(EqTest2, EqTest3, ?t2)
    DataDep(EqTest3, IfCond,  ?out)
```

Figure 6: A plan that recognizes our earlier leap year computation.

3.3 The Y2K Plan Library

The full Y2K plan library must have plans that cover the typical computations involving dates.[5]

It can be organized according to the following traits:

- The overall scheme for representing years (e.g., a four-digit year, a two-digit (sliding) window, or a two-digit encryption/encoding).
- The type of date representation used (e.g., YYDDD, YYYYMMDD, DDMMYYYY, and so on).
- The overall purpose of the plan (e.g., leap year detection, day-of-the-week determination, field-format determination, date-ordering, duration computation, and so on).

The library will not be able to contain *all* correct or incorrect plans. However, the library can contain plans that capture *typical* correct and incorrect fragments and can grow over time as more programs are examined.

[4] The algorithm underlying this engine improves upon earlier work by carefully exploiting particular similarities in the graph structures of source programs and program plans.

[5] See [6, p.1-2] for a list of Y2K exposures that can be used as a starting point for finding such typical computations.

4 Applying Plan-Based Techniques to Y2K

There are two key issues in applying plan-based techniques to Y2K: the feasibility of capturing many existing leap year computations in plans, and the scalability of the algorithm for locating plan instances. In this section, we use leap year examples to illustrate how these two issues can be addressed.

4.1 Using Plans to Capture Leap Year Computations

Ideally, a small set of plans is all that is necessary to capture a significant fraction of actual leap year computations in code. We have examined a large set of COBOL code (several hundred thousand lines) to determine whether this may indeed by the case. We have found 15 different correct and incorrect leap year computations in this set of code, and these appear to be recognizable using 5-10 plans, depending on what assumptions are made about the overall architecture of the plan recognition system.

The number of plans needed and the completeness of the resulting plans depends on the recognition technology used. Simply using an AST annotated with flow-representation allows us to ignore differences in variation in the order of divisions and tests [19]. In addition, simple expression simplification and reordering techniques (e.g., always using less than for comparisons rather than greater than, treating nested IFs as ANDs, simplifying negated conditions by switching the IF and ELSE branches, and so on), allow us to ignore many other variations. And finally, restructuring techniques such as GOTO elimination and expansion of non-recursive procedures allow us to have plans that deal with relatively simple control flow.

In general, the more powerful the canonicalization component, the fewer plans we need to recognize higher-level variations. With leap years, for example, one important issue is that there are several different ways a value can be divided by 100 within COBOL without using an explicit division. Figure 3 (our earlier example of a correct leap computation) takes advantage of the implicit division that results from using REDEFINE clauses. It redefines the date as a century field and a year field, and it then checks whether the two-digit YY sub-field equals zero instead of testing whether the remainder of dividing the four-digit field YEAR by 100 is zero.

Implicit divisions, however, can be handled by postprocessing the AST before the plan recognition phase begins. In particular, whenever there is an assignment to a redefined field, it's necessary to insert appropriate divisions or remainders for the pieces of that field. Assuming that has been done, our example plan will recognize the above leap year computation as well, without change.

```
COMPUTE Q = YEAR / 4.
COMPUTE R = YEAR - (Q * 4).
```

(a) Computing a remainder using integer division

```
  05 TMP PIC V9(02).
...
COMPUTE TMP = YY / 4.
```

(b) Computing a remainder using a variable that can store only two digits behind the decimal point.

Figure 7: Several different ways to compute COBOL remainders.

We can reduce the number of plans needed to recognize high-level concepts, such as leap-years, by providing supporting plans for recognizing low-level details. For example, using a "DIVIDE GIVING" construct is only one way to compute a remainder. Figure 7 shows two alternatives. However, both of these can be recognized by simple plans, allowing our original plan to recognize computations with the same high-level structure.

A factor that increases the number of plans is the need to detect incorrect date computations, such as locating leap year computations where the code does not correspond to the correct definition of a leap year. However, it appears that there are only a few categories of these incorrect computations, and these involve either forgetting one or more divisions (e.g., failing to divide by 400) or explicitly testing for specific years (e.g., explicitly checking whether the year is 92 or 96). The first category can be addressed by having different plans for different combinations of divisions, although these are similiar in structure to our example plan. The second category can be addressed by a single, general plan which an explicit comparison as a component, and which requires through constraints that the comparison involve a year and a numeric value. The "year" constraint assumes that certain variables have been pre-labeled as years, either by a human or by the SDA component.

The other factor that increases the number of plans is the sheer volume of relatively specialized uses of dates. Figure 8 shows one example, with a computation that determines whether the current year and the next year are both leap years.

4.2 A Scalability Experiment

The other important factor in the application of plan-based techniques to Y2K technology is the speed of the plan recognition engine. We performed an experiment in recognizing the leap year plan shown in Figure 9. This plan is a more complicated version of Figure 6, using two nested

```
MOVE 'F' TO LEAP-THIS-YEAR
MOVE 'F' TO LEAP-LAST-YEAR
DIVIDE YEAR BY 4 GIVING Q REMAINDER R.
IF R EQUAL 0
  MOVE 'T' TO LEAP-THIS-YEAR
ELSE
  IF R EQUAL 1
    MOVE 'T' TO LEAP-LAST-YEAR
  END-IF
END-IF
```

Figure 8: A computation that determines whether the current or previous year is a leap year.

```
plan Incorrect-Leap-Year-2(In: ?year, Out: ?out)
isa Incorrect-Leap-Year-Plan

recognize
Incorrect-Leap-Year-2(Year: ?year, Status: ?out)
  components
    Divide1:
      REMAIN(Src1: ?year, Src2: ?divby1, Result: ?rem1)
    EqTest1:
      EQUAL(Src1: ?rem1, Src2: ?zero, Dest: ?t1)
    IfCond1:
      IF(Cond: ?test1,
         Then: ?stmt-then, Else: ?stmt-else)
    EqTest2:
      EQUAL(Src1: ?year, Src2: ?zero, Dest: ?t2)
    IfCond2:
      IF(Cond: ?test2,
         Then: ?stmt-then, Else: ?stmt-else)
    Divide2:
      REMAIN(Src1: ?year, Src2: ?divby2, Result: ?rem2)
    EqTest3:
      EQUAL(Src1: ?rem2, Src2: ?zero, Dest: ?out)
  constraints
    Numeric-Field( ?year )
    Constant-Value( ?zero, 0 )
    Constant-Value( ?divby1, 4 )
    Constant-Value( ?divby2, 400 )
    DataDep( Divide1, EqTest1, ?rem1 )
    DataDep( EqTest1, IfCond1, ?t1 )
    Control-Flow( IfCond1, TRUE, IfCond2 )
    DataDep( EqTest2, IfCond2, ?t2 )
    Control-Flow( IfCond2, TRUE, Divide2 )
    SharedDep( Divide2, IfCond2, ?year )
    Control-Flow( IfCond2, TRUE, Divide2 )
```

Figure 9: A second incorrect leap year plan.

if statements instead of the AND clause.

Our current experimental testbed is tied to C language programs, precluding COBOL experiments at this stage. In our experiment, we first translated this plan into a lower-level representation tied to our AST representation for C programs. The result is a plan with 21 components and 28 constraints. We then constructed C programs of varying sizes, from 100 to 10,000 lines, containing one instance of this plan within each 100 lines of code. We didn't just use random C programs because we wanted to be able to have some control over how many instances were present in programs of different sizes. Finally, we used constraints checked as our measure of effort.

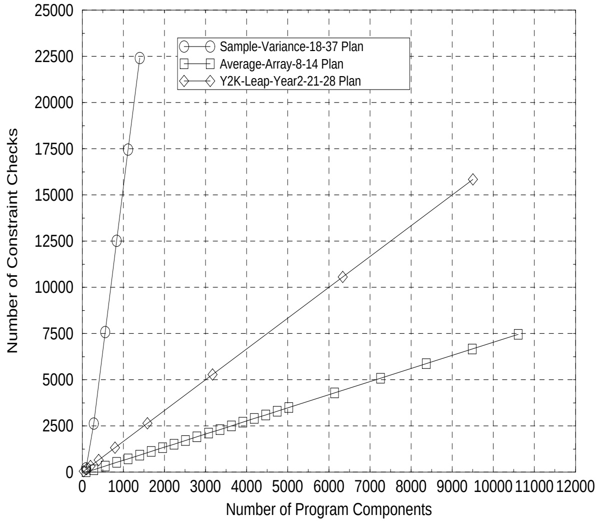

Figure 10: The results of our search for our Y2K Leap Year plan.

Figure 10 shows the results, along with comparisons to other plans we have searched for in programs of similar sizes. It takes linear effort (of about 1.7 evaluated constraints per line of code) to recognize instances of this plan. It took approximately 30 seconds to locate and find all instances of this plan in the 10,000 line program, using an unoptimized Lisp implementation of our plan recognition algorithm running on a Sun workstation.

4.3 Modifying Y2K Tools

Assuming our performance results hold up, we can significantly improve Y2K environments by using plan-based techniques to recognize design concepts.

In particular, the recognition of correct plans helps us eliminate areas of date-related slices from further user consideration, and also allows us to highlight the areas of those slices that were not recognized as part of a plan. This narrows down the part of the program that must be examined by the user. Along the same lines, recognizing incorrect plans helps us point out areas that must be fixed, while indicating precisely what the problem with that code is.

The reverse is true as well. Hooking the plan recognition engine into current Y2K environments offers the chance to heuristically improve its performance. If, for example, we know that a particular variable is a year (perhaps because its name is YEAR), we can reduce the amount of effort necessary to recognize date-related plans. In particular, with the plan Incorrect-Leap-Year-2 we can reduce the sets of candidates for the EqTest1 and EqTest3 components (the EQUAL tests on the remainders) by eliminating any EQUAL involving YEAR. That's because those tests involve variables that are known not to be years. Similarly, if we know that a variable is not a year (perhaps because the user has determined that for us), we can further reduce the candidate sets for components. In Incorrect-Leap-Year-2, for example, we can reduce the relevant REMAINs to only those whose numerand is a year, eliminating the rest.

These are only several of the more obvious ways that an effective plan engine can fits into the overall Y2K architecture. There are undoubtably others.

5 Future Work

Now that we have some initial empirical evidence of the scalability of our approach to locating Y2K-related plans, we are planning on connecting our recognition engine up to the output of a COBOL parser and flow-analyzer and determining its performance on recognizing a number of instances of Y2K code fragments discussed in [3] in a collection of real-world COBOL programs. These code fragments include a number of different examples of leap year and windowing-related Y2K code fragments. The result of this experiment will go a long way toward validating the apparent linear performance of our plan recognizer.

Along the same lines, we are also planning to perform experiments that measure the performance improvements possible within our recognizer when we have determined in advance (through heuristic means) that a particular variable is actually a particular type of date-related value.

Assuming that our recognizer's performance is validated, we are planning on exploring how the plans we recognize can be hooked to slices to display only the relevant part of the slice that requires changes, as well as to transformations to automatically repair Y2K code fragments when they have been recognized.

Last but not least, applying reconnition technology to Y2K will be an excellent opportunity to experiment with building up a plan library. While experimenting, we will be able to evaluate the library's completeness, its effectiveness in detecting real-world Y2K problems, and the effort necessary in maintaining the plan library.

6 Conclusions

This paper argues that plan-based concept recovery can play an important role as a key component of real-world tools addressing Y2K issues.

In particular, we have discussed several problems with the pattern-based and rule-based approaches to locating potentially problematic Y2K code fragments, and we have demonstrated that our plan-based approach addresses these drawbacks. In addition, we have shown how to represent leap-year plans and provided experimental evidence that they can be recognized in time that is linear with the size of the program. Finally, we have indicated several ways plan-based concept recovery and Y2K environments fit together.

A Y2K tool encompassing the plan-based techniques as outlined in this paper would have several advantages.

- It would significantly increases the level of automation for the Y2K analysis phase. The successful recognition of date-related design concepts has the potential to greatly reduce the number of false positives that must be examined and discarded by hand.

- It would allow for the automatic location and modification of negative cases, even when a four-digit date is used. Incorrect Y2K plans can be augmented with accurate transformation rules for automatic repair.

- It would allow for the the *validation* of the Y2K process by *explicit inspection* of the list of examples. Many tools hide their technology out of fear of competition. Large users may be unwilling to hand over mission critical software to a tool whose validity cannot be assessed. Plans provide an explicit list of cases covered that help users assess tool quality and applicability.

- It would support analyzing code that is already Y2K compliant. In particular, it would support *regression analysis*: Verifying that software that was made Y2K compliant in an early stage but had to undergo regular maintenance afterwards is still Y2K compliant.

Plan-based techniques can do a lot to help address the Y2K problem. They should not be ignored due to an incorrect perception that they don't scale.

Acknowledgements

Arie van Deursen was sponsored in part by bank ABN Amro, software house DPFinance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 "SOS Resolver". Steve Woods was sponsored in part by the Natural Sciences Engineering Research Council of Canada (NSERC). Alex Quilici is sponsored in part by an NSF Research Initiation Award.

We thank the anonymous reviewers of WCRE97 for their helpful comments.

References

[1] S. A. Bohner and R.S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[2] D. Chin and A. Quilici. DECODE: A cooperative program understanding environment. *Software Maintenance: Research and Practice*, 8:3–33, 1996.

[3] A. van Deursen, P. Klint, and A. Sellink. Validating year 2000 compliance. Technical report, CWI – Centrum voor Wiskunde en Informatica, Department of Software Technology, 1997. To appear. Also Chapter 2 of *Program Analysis for System Renovation – Resolver Release I*.

[4] J. Hart and A. Pizzarello. A scaleable, automated process for year 2000 system correction. In *Proceedings of the 18th International Conference on Software Engineering ICSE-18*, pages 475–484. IEEE, 1996. URL: `http://www.peritus.com/1c1d.htm`.

[5] J. Hartman. Workshop technical introduction. In *Workshop Notes, AAAI Workshop on AI and Automated Program Understanding, 10th National Conference on Artificial Intelligence*, 1992. URL: `http://www.cis.ohio-state.edu/~hartman/`.

[6] IBM. The year 2000 and 2-digit dates; a guide for planning and implementation, 1996. URL: `http://www.software.ibm.com/year2000/`.

[7] C. Jones. The global economic impact of the year 2000 software problem. URL: `http://www.spr.com/library/y2k00.htm`, 1997. Software Productivity Research, Inc.

[8] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, 1992.

[9] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, 1994. Special issue on reverse engineering.

[10] Ph. H. Newcomb and M. Scott. Requirements for advanced year 2000 maintenance tools. *IEEE Computer*, 30(3):52–57, 1997.

[11] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.

[12] A. Quilici and S. Woods. Toward a constraint-satisfaction framework for program understanding. *Journal of Automated Software Engineering*, 4(3), 1997.

[13] A. Quilici, S. Woods, and Y. Zhang. Some new experiements in program plan recognition. In *Proc. 4th IEEE Working Conference on Reverse Engineering*. IEEE Computer Society, 1997. To appear.

[14] B. Ragland. *The Year 2000 Problem Solver: A Five Step Disaster Prevention Plan*. McGraw-Hill, 1997.

[15] C. Rich and R. Waters. *The Programmer's Apprentice*. Frontier Series. ACM Press, Addison-Wesley, 1990.

[16] D. Smith, H. Müller, and S. Tilley. The year 2000 problem: Issues and implications. Technical Report CMU-SEI-97-TR-002, Software Engineering Institute, 1997.

[17] J. Towler. Leap-year software bug gives "million-dollar glitch". *The Risks Digest*, 18(74), 1996. URL: `http://catless.ncl.ac.uk/Risks/18.74.html#subj5`.

[18] L. M. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1–2):113–171, September 1990.

[19] L. M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, 1992.

[20] S. Woods. *A Method of Program Understanding using Constraint Satisfaction for Software Reverse Engineering*. PhD thesis, University of Waterloo, 1996.

[21] S. Woods, A. Quilici, and Q. Yang. *Constraint-based Design Recovery for Software Reengineering: Theory and Experiments*. Kluwer Academic Publishers, 1997. To appear.

[22] S. Woods and Q. Yang. The program understanding problem: Analysis and a heuristic approach. In *Proceedings of the 18th International Conference on Software Engineerig, ICSE-18*, pages 6–15. IEEE Computer Society, 1996.

[23] S. Woods and Q. Yang. Program understanding as constraint satisfaction: Representation and reasoning techniques. *Journal of Automated Software Engineering*, 1997. To appear.

[24] N. Zvegintzov. A resource guide to year 2000 tools. *IEEE Computer*, 30(3):58–63, 1997.