Program Plan Recognition For Year 2000 Tools

Arie van Deursen [a], Alex Quilici [b], and Steve Woods [c]

[a] *Department of Software Engineering, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.*

[b] *Department of Electrical Engineering, University of Hawaii at Manoa, 2540 Dole St, Holmes 483, Honolulu, HI 96822, USA.*

[c] *Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 15213, USA.*

Abstract

There are many commercial tools that address various aspects of the Year 2000 problem. None of these tools, however, addresses the closely-related leap-year problem. In this paper, we provide experimental results that suggest that the leap-year problem can be addressed by plan-based techniques for automated concept recovery. In particular, we provide representative code fragments illustrating the leap year problem, and we show the results of an empirical study that provides evidence that a plan-based approach can efficiently recognize both correct and incorrect leap year computations and that the needed plan library is likely to be tractable in size. This paper furthermore argues that plan-based techniques are in fact mature enough to make a significant contribution to the Year 2000 problem itself, despite none of the existing tools making any documented use of these plan-based techniques.

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, primarily as a consequence of having used a two-digit representation of years. Y2K is one of the most severe problems the software industry has ever faced [14,29]. As a result, many tools have been developed to address the Y2K problem [20,23,40]. These tools deal with system inventory, impact analysis, project planning, code

[1] The authorship of this paper is alphabetical. The authors may be contacted at Arie.van.Deursen@cwi.nl, alex@wiliki.eng.hawaii.edu, and sgw@sei.cmu.edu, respectively.

remediation, testing, and so on, using existing parsing and pattern-matching technologies.

Perhaps surprisingly, none of these tools makes any apparent use of the results of research in using plan-based techniques for concept recovery [28,34,24,16,13,6,37]. A program plan describes common combinations of low-level program actions that implement higher-level design concepts (such as "verify within warranty period" or "determine day of year"). 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 and incorrect date-manipulating plans (such as determine-day-of-year, check-whether-leap-year, and so on). Given such a library, many Y2K infections could be located accurately, classified precisely, and potentially corrected automatically.

In this paper, we discuss how Y2K tools actually work, present a problem closely related to Y2K, explain why current Y2K tools can't address this problem, and then provide evidence that program plan recognition techniques can be successfully applied to it.

1.1 The Leap Year Problem

Our focus is on recognizing *l*eap year computations, such as the one shown in Figure 1.[2] This code fragment, taken from real-world legacy COBOL code, 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).

Ensuring that a program does its leap year calculations correctly is usually *not* considered part of the Y2K problem. However, it is closely related [7], as many programs fail to recognize the year 2000 as a leap year, considering it as a century year without recognizing it as a year divisible by 400 as well [27, Chapter 4]. This situation has arisen because the definition of a leap year is relatively complex and many programmers did not have a correct definition available while programming, resulting in their missing important cases and performing leap year computations incorrectly [27].

The costs related to a failed leap year computations may be substantial, making recognizing and repairing them a potentially economically significant problem. As

[2] 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 and 1900 and 2100 are not).

```
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
```

Fig. 1. Example COBOL code that contains a leap year bug.

just one example, when the control computers of a New Zealand aluminum smelter simultaneously went down because they could not deal with the 366th day of 1996 it resulted in $1,000,000 damage [32]. Another recent example occurred in one vendor's version of DCE, a key UNIX-based application, that couldn't function at all from February 29th through March 31st, 1996 [5].

1.2 Leap Year and Y2K Tools

The prevalance of incorrect leap year computations suggests that it is not sufficient for Y2K tools to carefully replace two-digit dates with four-digit dates. Instead, the ideal Y2K tool should identify our example code involving leap years 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]

Unfortunately, this example is problematic for the standard mechanisms used by existing Y2K tools and it gives rise to a need for alternative mechanisms to address the problem. The next section discusses the approach taken by existing Y2K tools and explains why extending them to address the leap year problem appears to require augmenting them with plan recognition technology.

[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 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
```

Fig. 2. A fixed version of the initial leap-year computation.

2 Current Y2K Tool Support

A variety of commerical tools are available to support a Y2K conversion, including Peritus AutoEnhancer/2000 [12,11], Reasoning Systems Reasoning/2000 [30,19,18,4], The Software Revolution's Revolution/2000 [23], Microfocus's SoftFactory/2000 and SmartFind/2000 [21], Legasys's LS/2000 [17], and Techforce's Cosmos/2000 [31].

Most existing Y2K tools concentrate on two areas:

- *Identifying Y2K-related code* by heuristically locating date-manipulating elements in source code and then identifying code that is dependent on those elements. [4]
- *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).

Overall, much of the process of locating Y2K code, and some of its modification, is automated, although it may require some assistance from the programmer. 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

[4] As well as identifying dependencies on control input, data dictionaries, screen definitions, and so on.

is to avoid false positives.

2.1 Identifying Date-Related Code

While there are many differences in the details, existing tools share the same general approach to identifying date-related code: hypothesizing that a particular piece of code is date-related, heuristically verifying that the code is date-related, and then using slicing to locate other code that must also be date-releated.

2.1.1 Hypothesizing Date-Relatedness

Hypothesizing that a particular piece of code is date related is usually done by a combination of seeding and pattern matching.

Seeding is the process of forming a library of likely date-related identifiers, such as `Year` or `Date`, and related data formats, such as COBOL pictures of the form MM-DD-YY. Most tools support a customizable library of seeds, where programmers can suggest program- or domain-specific candidate identifier names, such as `CONTRACT-SY`, that are not part of the standard set of known date-related identifiers.

Most tools use a pattern-based representation of seeds to allow more complex descriptions of date-related items, such as names ending in "Y". The result is that tools tend to support patterns that are lexically-based, which deal directly with the source code entities; syntax-based, which deal with the internal nodes of the abstract syntax tree (AST); or a combination of the two (e.g., looking for a particular combination of names and operators in a specified region in the tree).

The tools typically parse programs into an abstract syntax tree and then apply pattern-matching techniques to locate the places where various seeds appear. The result is an initial collection of candidate date-related identifiers. However, this heuristic approach can lead both to false positives and false negatives. For example, a complex lexical heuristic can lead to false positives, where an identifier is initially considered to be a date but isn't (e.g., assuming that names that end in "Y" are date-related could lead to hypothesizing that `SALARY` is date-related). Alternatively, using simpler patterns or a simple list of names can lead to false negatives, in which a date-related identifier is initially missed, such as not recognizing that `Contract-SY` is a year. As a result, an inference process is necessary to filter false positives and to minimize false negatives.

The end result of this seed-and-pattern-match process is an initial set of possibly date-related identifiers.

2.1.2 Heuristic Verification

Accurately verifying code as date-related requires examining a variety of factors. Most tools appear to use an inference process to do this verification.

These tools address the false negative problem (missed date-related code) by hypothesizing additional date-related identifiers based on how identifiers previously hypothesized as date-related are used. For example, if the tool notices a comparison between a date-related identifier and another, previously unknown identifier, it's likely that the other identifier is also a date.

These tools address the false-positive problem (code erroneously recognized as date-related) by examining how a possible date-related identifier is used to gather evidence that verifies that the identifier is, in fact, date-related. This task often involves checking whether an identifier is involved in expressions involving key constants such as 4, 28, 29, 100, 365, and 2000. This task is also heuristic in nature, as not every expression involving one or more of these constants is date-related. For example, dividing by 4 could not only be part of a leap year check, but could also be part of computing a `QUARTERLY-PAYMENT` from an `ANNUAL-PAYMENT`. Similarly, dividing by 100 is not only a part of leap year checks, but is also a common way to handle percentages. It's only the combination of a variable that represents a year being divided by 4 and 100 that's likely to be part of a leap-year computation.

The end result of the heuristic verification process is a suggested set of identifiers that the tool has concluded are date-related.

2.1.3 Locating Other Date-Related Code

Given a set of data-related identifiers, most tools use slicing [33] to determine all code that is data- or control-dependent on those identifiers. In particular, a backward slice from a given variable's use in a particular statement locates those statements that might influence the values of that variable, and a forward slice locates that statements that are influenced by that statement.

Assuming that the suggested set of date-related identifiers is accurate, the result of the slicing process is that all date-related code is located precisely.

2.2 Finding and Fixing Problematic Code

Once tools have isolated the parts of the program that may have Y2K-related problems, most tools have mechanisms that attempt to determine exactly which code is problematic. This involves searching for possibly problematic operations (e.g., incrementing a year, comparing a value with a two-digit constant, and so on), as

well as trying to filter out those expressions with "safe" calculations (e.g., comparing two date-related identifiers). The tools then either flag suspicious code to the user, or they make use of a variety of transformation-based techniques to perform appropriate substitutions.

3 Applying Existing Tools To The Leap Year Problem

Despite all of the effort currently focused on Y2K, there are as of yet no tools that claim to have successfully addressed the closely related problem of automatically recognizing and repairing incorrect leap-year calculations.

Our hypothesis is that this situation has arisen because most Y2K code can be remediated and repaired *without understanding* the underlying purpose of the code—but that this is not true for finding and fixing leap year problems. For example, given an expression that compares a two-digit variable with a two-digit constant, a tool need only recognize that the variable is a two-digit year, infer that the constant must therefore represent an offset from the year 1900, replace the year with a four digit year and add 1900 to the constant. There is no need, however, for the tool to understand the higher-level task this comparison is supporting, such as determining whether an input is in error.

In contrast, processing leap year examples requires recognizing that the purpose of a set of related code fragments is to perform a leap year calculation, and it requires modifying or replacing those fragments to perform the calculation correctly. This suggests that existing tools must be augmented to be able to recognize when and how various code fragments contribute to a leap-year calculation.

3.1 Using A Rule-Based Approach

Perhaps the most straightforward approach to use to recognize leap-year computations is to use rule-based techniques. The idea is to use rules to describe various properties and interrelationships of program components that must hold to have an instance of a leap year computation. This approach assumes that we can write specific rules to identify instances of common classes of correct and incorrect leap year computations, and it further assumes that we can then efficiently recognize leap year instances by applying a standard rule-based deductive inference engine.

As an example, Figure 3 is a rule that could be used to recognize the fragment of Figure 1. Paraphrased, this rule states that if there is a numeric variable, and its value is involved both in a division by 4 and a division by 100, and there are tests to determine whether the division by 4 is zero and whether the division by 100 is

```
IF      NumericVariable(?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)
```

Fig. 3. A rule to recognize a particular invalid leap year computation.

not zero, then we know the variable is a year and that we have an instance of an incorrect leap year computation.

At first glance, the rule-based approach seems sensible. 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 incorrect 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: *s*calability. In general, rule-based systems suffer scalability problems when they have large fact bases and many complex, interacting rules. In the leap year problem, the programs to be inspected are likely to be large, resulting in a large database of program facts (describing the program's components and the control-flow and data-flow relationships between them). Moreover, the rules are often complex, because each rule has potentially many antecedents describing the pieces of a leap-year computation and the relationships between those pieces. Last but not least, there may be many rules covering the many fundamentally different ways to implement correct and incorrect leap-year computations.

3.2 Addressing The Problems With The Rule-Based Approach

There are two ways to deal 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 pro-

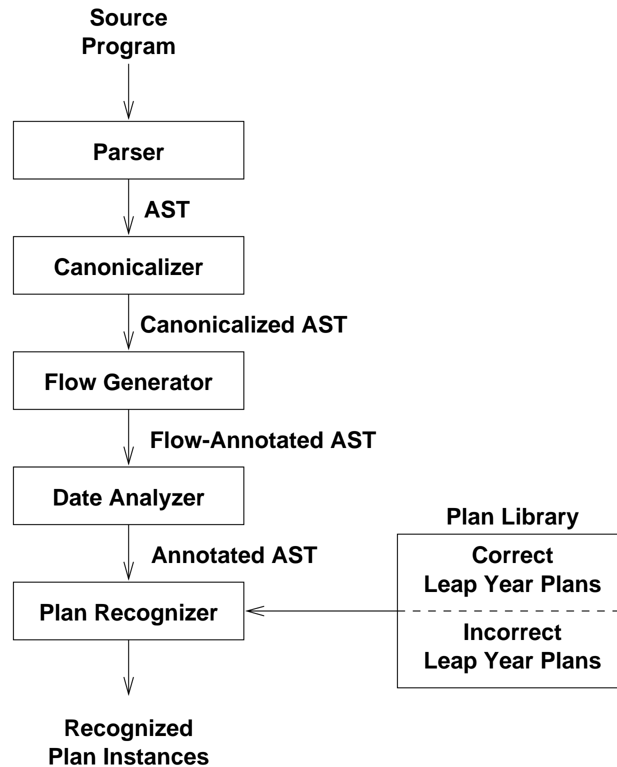

Fig. 4. An architecture for recognizing leap-year related program plans.

cessing a particular class of rules. This approach is more attractive, but can require considerable effort in finding an appropriate engine.

Plan-based techniques are an example of the latter option. These techniques can be thought of as combining a special class of rules (the plans) with a dedicated engine optimized for recognizing applications of rules from this class. Our earlier experiments provided some initial evidence that these plan-based techniques do, in fact, scale [26], which initially led us to believe that they might be useful for recognizing leap-year computations.

4 Plan-Based Recognition of Leap-Year Computations

Figure 4 shows an adaptation of a standard plan-based architecture to address the leap-year 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), which is essentially the “date-recognition” component of existing Y2K tools [12,1]. Effectively, the

```
plan Not-100th-Year(In: ?year, Out: ?out)
isa Incorrect-Leap-Year-Check-Plan

recognize
Not-100th-Year(Year: ?year, Status: ?out)
  components
    Divide1: REMAIN(Src1: ?year, Src2: ?divby1, Rem: ?rem1)
    Divide2: REMAIN(Src1: ?year, Src2: ?divby2, Rem: ?rem2)
    EqTest1: EQ(Src1: ?rem1, Src2: ?zero,  Dest: ?t1)
    EqTest2: NOT-EQ(Src1: ?rem2, Src2: ?zero, Dest: ?t2)
    EqTest3: LOGICAL-AND(Src1: ?t1, Src2: ?t2, Dest: ?out)
 constraints
    Numeric-Field(?year)
    Constant-Value(?divby1, 4)
    Constant-Value(?divby2, 100)
    Constant-Value(?zero, 0)
    DataDep(Divide1, EqTest1, ?rem1)
    DataDep(Divide2, EqTest2, ?rem2)
    DataDep(EqTest1, EqTest3, ?t1)
    DataDep(EqTest2, EqTest3, ?t2)
```

Fig. 5. A plan that recognizes our earlier incorrect leap year computation.

SDA phase associates date types with variables (without worrying about correct or incorrect constructs). The plan library can take advantage of these date types to reduce the search space when looking for these plans.

The plan recognizer is a special purpose engine that is given a library of leap-year plans and that tries to locate instances of them in the canonicalized AST 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 [6,16,34].

4.1 Representing Leap-Year Plans

Our particular approach to plan recognition represents plans as a combination of components and constraints (in the spirit of the Concept Recognizer [16] and DECODE [6]).

Figure 5 is an example plan in the component-and-constraint formalism. This plan is suitable for recognizing a leap-year-related code fragment similar in function to the one in Figure 1.

The components are syntax tree entities or sub-plans. This example specifies five components: two remainder computations, an equality test, an inequality test, and a logical AND of the results of the two tests. Any program containing these five

components matches the plan, provided it also meets the plan's constraints. Each plan component has attributes which correspond to variables or constants in the program or to the results of computations. For example, the `EqTest1` component corresponds to an equality test, the `Src1` attribute corresponds to the result of evaluating the operator's left operand, the `Src2` attribute corresponds to the result of evaluating the operator's right operand (in this case, 0), and the `Dest` attribute corresponds to a boolean that holds the operator's result (which may then itself appear in another expression).

The constraints can be restrictions on the component's attributes or on the relationships between components. Some examples of attribute constraints are that the `year` variable must be numeric and that the divisors must be constants with values 4 and 100. Alternatively, the `DataDep` constraint ties two components together by specifying that the value of the specific variable produced by one component is the same used by another component. In our example plan, this produces an implicit partial order on the divisions and the tests that combine them.

4.2 Recognizing Plans

Our approach is to treat program plan recognition as a constraint satisfaction problem (CSP) [36,39,38,25,35]. In particular, our recognition engine, *Layered MAP-CAP*, represents plan components as CSP variables and the possible values of these components as the variable domains. In addition, we represent the types of components and constraints on component attribute values as *n*ode-constraints and the inter-component constraints are *a*rc-constraints. We then apply a specialized constraint satisfaction engine to locate instances of plans in the code [36,26]. [5]

Our engine further relies on a hierarchically organized plan library and operates breadth first through the hierarchy, processing plans consisting only of AST nodes first, then the plans that involve the initial set of plans and so on. Our engine uses properties of the constraints and the information available in the AST and flowgraph to direct the actual plan-matching process and recognizes all possible instances of a given plan before moving on,

The details of this engine and its application to a variety of recognition problems have been presented elsewhere [36,26]. Our focus in this paper is on its specific application to the leap-year problem.

[5] This engine is specialized to the class of constraint satisfaction problems containing at least some constraints that can be evaluated as functions that return the particular values for which some relationship between variables holds true, rather than simply as functions evaluating the truth of individual relationships between values for those variables.

5 Empirically Evaluating The Plan-Recognition Approach

There are two key issues in applying plan-based techniques to the leap year problem:

- The feasibility of using plans to describe existing leap-year computations. That is, how many plans are needed?
- The scalability of the CSP-based algorithm for locating plan instances. That is, how quickly can this algorithm locate leap years?

Ideally, a "manageable" set of plans is all that is necessary to capture a significant fraction of actual leap year computations in code. It is certainly reasonable to construct a plan library containing the same number or fewer plans than the number of common synonyms and patterns used to recognize date-related identifiers in existing tools, as a pattern library of that size has proven to be constructable in practice for the general Y2K problem. Unfortunately, there are potentially a wide variety of different ways to compute leap years (both correct and incorrect). As a result, it's necessary to empirically determine how many leap year plans are needed.

Similarly, our recognition algorithm must run in a "reasonable" time. Since, in the worst case any algorithm addressing this problem is NP-complete [37], this suggests that we have to verify the performance of our algorithm empirically.

5.1 A Plan Library To Capture Leap Year Computations

We have examined a large amount of COBOL code (several hundred thousand lines worth) to determine how many plans would be needed to handle the leap year computations found in that code. This code contained 15 different correct and incorrect leap year computations that fell into the following six categories:

- *Every-Fourth-Year*: An incorrect computation (although it is correct until the year 2100) that computes a remainder divided by 4 and then tests it against zero.
- *Every-Fourth-Year-Except-2000*: An incorrect computation that is similar to *Every-Fourth-Year*, except that the year is erroneously also verified not to be 2000.
- *Check-Leap-Year-List*: An incorrect computation that compares a year against each element in a list of leap years (and that fails for the first leap year not in the list).
- *Not-100th-Year*: An incorrect computation that computes remainders divided by 4 and 100.
- *Not-400th-Year*: An incorrect computation that is very similar to *Not-100th-Year*, except that it computes remainders divided by 4 and 400 rather than 4 and 100.
- *Complete-Leap-Year*, which computes remainders divided by 4, 100, and 400, and does the appropriate tests.

```
...
COMPUTE Q = CONTRACT-SY / 4.
COMPUTE R-1 = CONTRACT-SY - (Q * 4).
...
MOVE 'F' TO LY
IF R-1 = 0
  COMPUTE Q = CONTRACT-SY / 100.
  COMPUTE R-2 = CONTRACT-SY - (Q * 100).
  IF R-2 NOT = 0
    MOVE 'T' TO LY
  END-IF
END-IF
...
IF LY = 'T'
  [leap year related code]
END-IF
```

Fig. 6. A variant of our initial buggy COBOL fragment (category: *Not-100th-Year*).

From this initial study, there appear to be only a relatively small set of categories of incorrect leap year computations, and these involve either forgetting one or more divisions (e.g., failing to divide by 100 or 400) or explicitly testing for specific years (e.g., explicitly checking whether the year is 92 or 96).

Each category can be thought of as placing some requirements on the components that must be present in a computation that falls into that category. For example, any computation in the category *Every-Fourth-Year-Except-2000* will be required to test divisibility by 4 and equality against 2000.

However, despite these requirements, there are several dimensions by which the individual computations within a particular category can vary. These include:

- The ordering and exact placement of specific computations and comparisons (e.g., whether the remainder of the division by 4, 100, or 400 happens first, as in Figure 1, or whether the divisions by 100 and 400 occur only after realizing the division by 4 has a remainder of zero, as in Figure 6).
- Whether logical operators, nested IF statements, or a combination of both are used (e.g., whether a single, complex expression is used to determine whether a year is a leap year, as in Figure 1, or whether a pair of IF statements is used, as in Figure 6).
- The specific constructs used to compute remainders (e.g., using DIVIDE-GIVING, as in Figure 1, integer division, as in Figure 6), or an alternative method such as storing the division's result in a variable that can store only two digits behind the decimal point).
- The specific logical operators used (e.g., testing whether the year is divisible by 4 and not divisible by 100, as in all our earlier examples, versus testing whether it's not true that the year is not divisible by 4 or divisible by 100), as in Figure 7.
- Whether the test is to determine whether a year is or is not a leap year (e.g., setting a flag to remember a year is a leap year, as in Figure 1, versus setting a flag to remember that a year is not a leap year, as in Figure 7).

```
DIVIDE CONTRACT-SY BY 4 GIVING Q REMAINDER R-1
DIVIDE CONTRACT-SY BY 100 GIVING Q REMAINDER R-2
...
MOVE 'T' TO LY
IF NOT (R-1 = 0 OR R-2 NOT = 0)
  MOVE 'F' TO LY
END-IF
...
IF LY = 'F'
  [code that applies to non-leap years]
ELSE
  [code that applies to a leap year]
END-IF
```

Fig. 7. Yet another COBOL fragment that contains a leap year bug (category: *Not-100th-Year*).

```
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
```

Fig. 8. Determining whether the current year or the last year is a leap year.

In addition, many categories have variants specific to that particular category. For example:

- With *Every-Fourth-Year-Except-2000*, the order of the tests can vary (e.g., whether the division involving 4 or the explicit test against 2000 is computed first).
- With *Check-Leap-Year-List*, the specific values that are tested can vary as well as the order in which they are tested (e.g., whether the years tested are 92 and 96, 96 and 00, or 96 and 92).

Finally, in addition to these variations, there were specialized calculations closely related to leap year computations. Figure 8 shows one example, with a computation that determines both whether the current year or the last year is a leap year. These specialized uses appear to be relatively rare and tend to include one of the basic leap-year plans within them. As a result, at least the basic leap year calculation will be recognized, drawing attention to that general area of code as potentially problematic.

It's clearly necessary to have at least one plan for each category. However, the precise number of plans needed and the completeness of the resulting plans depends significantly on how much hierarchical structure there is to the plan library and how much canonicalization is done before hand.

```
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
```

Fig. 9. Another Leap Year Example

5.1.1 Hierarchical Structure With Subplans

One way to reduce the number of plans needed to recognize high-level concepts, such as leap-years, is by providing supporting plans for recognizing low-level details. As we've seen, there are a number of ways to compute a remainder: "DIVIDE-GIVING", using integer division, and using a variable that can store only two digits behind the decimal point. We can capture these variants in three specific plans "Remainder-By-Divide-Giving", "Remainder-By-Integer-Division", and "Remainder-By-Fixed-Point", which are specific instances of the general plan "REMAIN".

The other place where subplans are useful with leap years is in recognizing a DIVISION-BY-100. In COBOL, there are a variety of different ways a value can be divided by 100 without using an explicit division. Figure 9 is an example of a correct leap year computation that 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.

These implicit divisions, however, can be recognized by fairly simple plans: any use of variable that's been redefined in a two digit field is an instance of a DIVISION-BY-100.

Having these subplans simplifies the higher-level plans corresponding to the various leap-year categories, and it results in the need for fewer plans to be able to

recognize computations with a particular category.

5.1.2 Canonicalization

Canonicalization as a complimentary technique to subplanning for reducing the number of plans we need to handle leap year variants. In general, the more powerful the canonicalization component, the fewer plans we need to recognize program-level variations.

The simplest form of canonicalization is to turn program source into an AST annotated with flow information, which allows our plans to ignore a variety of differences in variation between plans [34]. For leap year, this allows us to ignore the precise order in which the remainder calculations take place, as our plans can just verify that certain flow-based constraints hold between those calculations. In addition, it allows us to ignore differences in Cobol dialects, as these can presumably be handled by the canonicalization component.

Other forms of canonicalization are essentially a form of specialized plan recognition and transformation, where there are engines for recognition and transformation that are targetted to specific types of components. One such complex form of canonicalization involves techniques such as GOTO elimination and expansion of non-recursive procedures, which in general allow our plans to ignore meaningless variations in control flow.

Another similar form includes expression simplification and reordering techniques that allow us to ignore meaningless variations in expressions (e.g., always using LESS-THAN for comparisons rather than GREATER-THAN, rewriting expressions that involve logical negation without it, recomputing expressions in disjunctive normal form, simplifying negated conditions by switching the IF and ELSE branches, and so on. These allow plans to be written to check for only particular constructs, such as EQUALS and NOT-EQUALS, without worrying about the use of NOT.

The final place where cannonicalization is useful in our leap-year example is turning logical connectors (such as “AND” and “OR”) into sequences of IF statements (or doing the opposite and turning particular sequences of IF into ANDs and ORs). That’s because we have a low-level plan “LOGICAL-AND” that can be implemented in two different ways: either the language’s logical AND construct, or a pair of nested IFs, where one test is in the outer IF, and the other test is in the inner IF. By mapping one into the other, we have only to write plans to deal with the construct that remains.

Category	Needed Plans	Key Components And Constraints
Every-4th-Year	2	Variable is year (In)Divisibility-by-4 test
Every-4th-Year-Except-2000	4	(In)Divisibility-by-4 test (In)Equality-with-2000 test
Check-Leap-Year-List	2	Var is year Comparison(s) with constant
Not-100th-Year	4	(In)Divisibility-by-4 test (In)Divisibility-by-100 test
Not-400th-Year	4	(In)Divisibility-by-4 test (In)Divisibility-by-100 test
Complete-Leap-Year	8	(In)Divisibility-by-4 test (In)Divisibility-by-100 test (In)Divisibility-by-400 test

Fig. 10. Required Leap Year Plans

5.1.3 The Results

Figure 10 lists each of our leap-year categories and the number of required plans. Each plan describes a particular variation that cannot be easily handled with subplans or cannonicalization. The plans differ primarily in terms of exactly which tests occur. For example, the four plans in the category Every-4th-Year-Except-2000 capture the four different possible combinations of divisibility and indivisibility by 4 and equality or inequality with 2000.

Each of these plans is designed to recognize a block of code that is executed only if a particular variable is a leap year. There is also another, similar set of plans to recognize blocks of code that are executed only if the variable is not a leap year. The result is a set of approximately 50 plans in all, a number that is well within the manageable range. While there is no guarantee that this set of plans is complete, it covers all variations we have seen in the COBOL code we have examined, as well as many alternative ways those leap year tests could have been written.

5.2 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

```
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, Re-
sult: ?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, Re-
sult: ?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)
    Control-Flow(IfCond2, TRUE, Divide2)
```

Fig. 11. A second incorrect leap year plan.

recognizing the leap year plan shown in Figure 11. This plan is a more complicated version of Figure 5, using two nested if statements instead of the AND clause.

Our current experimental testbed is tied to C language programs, precluding an experiment recognizing this particular plan in COBOL code. As a result, 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 12 shows the results, along with comparisons to other plans we have searched for in programs of similar sizes (an array averaging plan and a simple plan to compute variance). 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

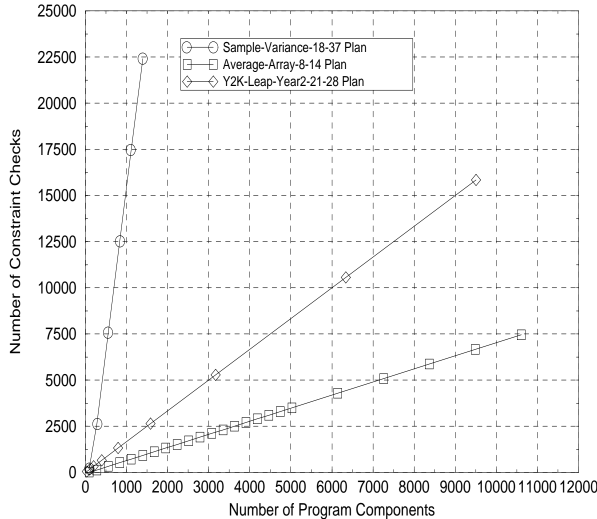

Fig. 12. The results of our search for our Y2K Leap Year plan.

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.

In this experiment, we have recognized all instances of a single leap year plan in a single piece of software. In general, we have to attempt to recognize instances of all leap year plans in the library. Assuming that our experimental results are similar across different the library contains 50 leap year plans, and that we recognize each of the 50 plans from scratch, this suggests that approximately 80 constraints need to be evaluated for each line of code. However, there are no dependencies between plans, so there is no theoretical reason why recognizing the individual leap year plans can't be done in parallel.

6 Future Work

We have obtained some initial empirical evidence of the scalability of our approach to locating leap-year related code. Our goal is to now more perform more realistic experiments that work with real-world code and that involve both leap-year related plans and other, more general plans.

6.1 An Environment For Plan-Recognition Experiments

We are currently developing an extensible, component-based environment to support realistic COBOL reverse engineering experiments. It consists of the compo-

nents outlined earlier in Figure 4:

- The COBOL parser is an instantiation of the ASF+SDF Meta-Environment [15,8] with a COBOL grammar [3].
- The cannonicalization step is done through a series of *transformation functions* on the abstract syntax trees produced by the parser, where these transformation functions are automatically generated from the grammar [2]. These primary use of these transformations is to map the full set of COBOL constructs to a smaller set of canonical ones.
- Control and dataflow analysis is done by means of the DHAL dataflow analysis framework [22]. It consists of an abstract representation, called DHAL (Dataflow High Abstraction Language), which covers exactly those program elements that affect control and dataflow. The framework provides a mapping from COBOL to DHAL, as well as control flow normalization, alias propagation, inter-procedural analysis, and definition-use chaining at the DHAL level. This environment supports the easy addition of additional analysis functions, allowing us to quickly provide dataflow dependencies in the format required by the plan recognizer.
- The static date analyzer is realized by means of a general COBOL type inference engine [10]. It essentially builds a hierarchy of groups of variables by analyzing assignments, relational operators, and constants used. The inferred groups closely correspond to types. Certain types (and hence all variables of that type) can be marked as date-related by means of standard lexical pattern matching on names, suspicious literals, or by recognizing typical record structures such as three two-digit fields.
- The plan recognizer uses our constraint-based approach [38,25,35,36], with the added benefit that the interplay between plans and the types recognized by the static date analyzer (such as year or month) may help to improve its performance. Type information can be used in the *n*ode consistency propagation phase of the constraint solver used in the recognition engine [36, Chapter 3], potentially reducing the engine's search space significantly.

This environment has only two components tied to date-related problems: the static date analyzer (which searches for date-related types) and the recognizer's plan library (which has so far been set up with leap-year related plans). As a result, the tool environment can easily be adapted for other sorts of plan recognition experiments, for example in the area of discovering typical Euro or currency related computations in COBOL code.

6.2 Specific Experiments

Our initial goal is to apply this environment to determine its performance results in searching for our leap year plans in [9] in a collection of real-world COBOL programs. The end result of this experiment will go a long way toward validat-

ing 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 the performance results hold up, we are planning to then recognize other date-related code plans, such as windowing-related Y2K code fragments. This involves providing a more general date-related plan library, which will be an excellent opportunity to determine how large libraries have to be to understand significant amounts of code within a particular domain, as well as giving us some insight into the effort necessary to construct and maintain a sizable plan library.

7 Conclusions

This paper argues that plan-based concept recovery can play an important role in addressing the real world problem of locating and classifying incorrect leap year computations.

In particular, we have discussed several problems with the pattern-based and rule-based approaches to locating potentially problematic leap year calculations, 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.

While our paper has focused on recognizing leap year computations, it is likely to be applicable to other date-related computations, such as recognizing computations that rely on date windows (and, more importantly, exactly what those computations are doing). This suggests that our work may prove useful to those maintaining code that has been automatically remediated using Y2K tools, since our recognizer can check for common date-related mistakes that might be introduced in the maintenance phase. More generally, our work is likely to be applicable to other problems that involve locating highly stereotypical computations within large programs (such as currency manipulations in banking systems).

Our work with leap years suggests that plan-based techniques are, in fact, applicable to real world problems, and they they should not 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 NSF Research Initiation Award #9309735.

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] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Working Conference on Reverse Engineering; WCRE97*, pages 144–155. IEEE Computer Society, 1997.

[3] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In A. Sellink, editor, *Theory and Practice of Algebraic Specifications; ASF+SDF'97*, Electronic Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[4] Jules Burn. Overview of Software Refinery: Product family for automated software analysis and transformation. Technical report, Reasoning Systems, Palo Alto, CA, 1992.

[5] W. R. Carithers. A major OS leap year glitch. *The Risks Digest*, 17(82), 1996. URL: `http://catless.ncl.ac.uk/Risks/17.82.html#subj4`.

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

[7] A. van Deursen. The leap year problem. *Year/2000 Journal*, 2(4), July/August 1998.

[8] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[9] A. van Deursen, P. Klint, and A. Sellink. Validating year 2000 compliance. Technical Report SEN-R9712, CWI, 1997.

[10] A. van Deursen and L. Moonen. Type inference for COBOL systems. Proceedings of the 1998 Working Conference on Reverse Engineering. Honolulu, HI, 1998.

[11] J. Hart. Identifying date-sensitive items: The essential step. URL: `http://www.peritus.com/wp-dsiid.html`, 1998. Peritus.

[12] 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`.

[13] 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/`.

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

[15] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

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

[17] Legasys. Legasys guide to the year 2000. URL: `http://www.qucis.queensu.ca/~legasys/LS2000_Info/technical.html`, 1998.

[18] L. Markosian, R. Brand, and G. Kotik. Customized software evaluation tools: Application of an enabling technology for re-engineering. In *Proceedings of the Fourth Systems Re-engineering Technology Workshop*, pages 248–255, Baltimore, MD, Feb 1994.

[19] 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.

[20] R. A. Martin. Dealing with dates: Solutions for the year 2000. *IEEE Computer*, 30(3):44–51, 1997.

[21] Microfocus. Softfactory/2000 white paper. URL: `http://www.microfocus.com/year2000/sf2wpap.htm`, 1998.

[22] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In A. Sellink, editor, *Theory and Practice of Algebraic Specifications; ASF+SDF'97*, Electronic Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

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

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

[25] A. Quilici and S. Woods. Toward a constraint-satisfaction framework for program understanding. *Journal of Automated Software Engineering*, 4(3):271–290, 1997.

[26] A. Quilici, S. Woods, and Y. Zhang. Program Plan Matching: Experiments With A Constraint-Based Approach In *Science of COmputer Programming*, this issue, 1999.

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

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

[29] 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.

[30] Reasoning Systems. Reasoning/2000 for cobol. URL: `http://www.reasoning.com/downloads/R2K_COBOL.pdf`, 1998.

[31] Techforce. Solutions for the year 2000. URL: `http://www.techforce2000.ca/solutions_2000.pdf`, 1998.

[32] J. Towler. Leap-year software bug gives "million-dollar glitch". *The Risks Digest*, 18(74), 1997. URL: `http://catless.ncl.ac.uk/Risks/18.74.html#subj5`.

[33] M. Weiser. Program slicing. *ACM Transactions on Software Engineering*, 10:352–357, 1984.

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

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

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

[37] 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.

[38] S. Woods and Q. Yang. Program understanding as constraint satisfaction: Representation and reasoning techniques. *Journal of Automated Software Engineering*, pages 114–123, 1997.

[39] Q. Yang A. Quilici and S. Woods. Applying plan recognition algorithms to program understanding. *Journal of Automated Software Engineering*, 5(3):1–26, 1998.

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