Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Validating Year 2000 Compliance

A. van Deursen, P. Klint, A. Sellink

Software Engineering (SEN)

SEN-R9713 July 31, 1997

Report SEN-R9713
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Validating Year 2000 Compliance

Arie van Deursen[1], Paul Klint[1,2], Alex Sellink[2]

arie@cwi.nl, paulk@cwi.nl, alex@wins.uva.nl

[1] *CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

[2] *WINS, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

ABSTRACT

Validating year 2000 compliance involves the assessment of the correctness and quality of a year 2000 conversion. This entails inspecting both the quality of the conversion *process* followed, and of the *result* obtained, i.e., the converted system. This document provides an overview of the techniques that can be used to validate year 2000 compliance. It includes typical code fragments, and a discussion of existing technology, impact analysis, solution strategies, code correction, testing, and tools.

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

Keywords and Phrases: Software maintenance, programming language technology, COBOL, Y2K.

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

Note: The authors were sponsored by bank ABN Amro, software house DPFinance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 "SOS Resolver".

Note: The report has appeared in A. van Deursen, P. Klint, and G. M. Wijers, eds., *Program Analysis for System Renovation: Resolver Release I*, revised edition, Amsterdam, July 1997.

Acknowledgments: The authors would like to thank the reviewers of Resolver Release I for their intensive reading and useful comments, Jasper Kamperman and Rudolf van Laatum for reading earlier drafts of this document, and the tool providers who were willing to spend half a day answering the Resolver questionnaire.

Executive Summary Validating year 2000 compliance involves the assessment of the correctness and quality of a year 2000 conversion. This entails inspecting both the quality of the conversion *process* followed, and of the *result* obtained, i.e., the converted system. This document provides an overview of the techniques that can be used to validate year 2000 compliance. After giving a problem statement and discussing related literature, this document covers:

- A characterization of a series of typical COBOL date operations (such as leap year computations, date subtraction, century computation, ...).
- A discussion of existing technology from the areas of compiler construction and reverse engineering that can be used in a year 2000 conversion. Of particular interest are scanning, parsing, dataflow analysis and program slicing.
- A presentation of impact analysis, which finds date-infected variables and statements based on initial seeds and propagation rules. The theoretical limitations of static analysis methods include that it is impossible to predict value ranges of variables, or to find the *minimal* number of statements to be modified.
- A description of three correction strategies, field-widening, windowing, and compression. If the impact analysis is sufficiently detailed, automatic modification may be possible. Validation requires that the modification rules are explicitly available and (formally) verifiable.
- Issues related to testing year 2000 conversions (test plan, selection of test paths based on impact analysis, comparison of test or analysis results, etc.)
- A discussion of state-of-the-art commercial tools, such as Refine/2000, AutoEnhancer/2000, ARCdrive, COBOL Analyst, and Date Analyzer.

The document emphasizes on COBOL year 2000 migrations. All techniques discussed, however, can be directly used for other languages and environments, and are applicable to other types of conversions as well.

Contents

1 Introduction **2–5**
1.1 Overview . . . 2–5
1.2 Problem Statement . . . 2–6
1.2.1 Available Definitions . . . 2–6
1.2.2 Year 2000 Compliance . . . 2–7
1.3 Steps in a Year 2000 Conversion . . . 2–8
1.4 Scope . . . 2–9
1.5 Further Publications and Activities . . . 2–9
1.5.1 The IBM Guide . . . 2–9
1.5.2 The IEEE Millennium Task Force . . . 2–10
1.5.3 Remaining Pointers . . . 2–10

2 Example Code Fragments **2–10**
2.1 Date Formats . . . 2–11
2.2 Code Fragments with Date Manipulation . . . 2–14
2.2.1 Interpreting Date Values . . . 2–14
2.2.2 Computations with Dates . . . 2–16
2.2.3 Alteration of dates . . . 2–17
2.2.4 Determining Leap Years . . . 2–20
2.2.5 Date Transport . . . 2–22
2.2.6 Exception handling . . . 2–24

3 Existing Technology **2–26**
3.1 Lexical analysis . . . 2–26
3.2 Syntactic analysis . . . 2–26
3.3 Dataflow analysis . . . 2–27
3.4 Program slicing . . . 2–27
3.5 Abstract interpretation/type inference . . . 2–27
3.6 Program understanding . . . 2–28
3.7 Database analysis . . . 2–28

4 Impact Analysis **2–28**
4.1 Year 2000 Exposures . . . 2–29
4.2 Analysis Results . . . 2–29
4.2.1 Erroneous Results . . . 2–29
4.2.2 Result classifications . . . 2–30
4.3 Detecting Infections . . . 2–30
4.3.1 Initial Seeds . . . 2–30
4.3.2 Seed Propagation . . . 2–31
4.4 Theoretical Limitations . . . 2–31
4.5 Related Work . . . 2–32

5 Year 2000 Solutions **2–33**
5.1 Widen the data . . . 2–33
5.2 Windowing . . . 2–34
5.3 Encoding or Compression . . . 2–35

6 Choice of Strategy **2–35**
6.1 Which Solution to Choose? . . . 2–35
6.2 Clustering . . . 2–36
6.3 Bridge Programs . . . 2–36

7 Code Corrections **2–37**

8 Testing **2–38**
8.1 Controlling Testing Costs . . . 2–38
8.2 The Year 2000 Test Plan . . . 2–40
8.3 Preparing Test Sets . . . 2–41
8.3.1 Selection of Test Path . . . 2–41
8.3.2 Partitioning of Test Values . . . 2–41
8.3.3 Time Travelling . . . 2–42
8.4 Back-to-back Testing . . . 2–42
8.5 Safety-critical Programs . . . 2–43

9 Tools **2–43**
9.1 Tool Classification . . . 2–43
9.2 Example Tools . . . 2–43
9.2.1 Software Refinery . . . 2–43
9.2.2 Peritus . . . 2–46
9.2.3 SEEC . . . 2–47
9.2.4 ARCdrive . . . 2–48
9.3 Limitations and Research Directions . . . 2–48

10 Concluding remarks **2–49**
10.1 Summary of compliance-related issues . . . 2–49
10.2 Research issues . . . 2–50
10.3 Closing remarks . . . 2–51

1 Introduction

1.1 Overview

The software crisis caused by the turn of the century (the "millennium meltdown") is attracting more and more attention. Still, for many organizations, and particularly very large ones, the year 2000 software problem is overwhelming: It is often impossible to determine the costs involved, the effort required, or the effectiveness of the methods chosen.

In this document, we consider the year 2000 problem from the *validation* point of view. Validating year 2000 compliance involves the assessment of the correctness and quality of a year 2000 migration.

In validation, one can generally take two points of view: The first is to inspect the quality of the *process* involved: which steps are taken, which analysis techniques are used, etc. The other approach is to assess the quality of the *result*: does the result meet the expected criteria? For a year 2000 conversion, both forms of quality assurance are relevant. For example, process quality inspection could involve verifying the correctness of rules that were used for automatic code modifications. Likewise, the confidence in the resulting code can be increased by inspecting the result carefully (i.e., by selectively applying further analysis methods to it, such as accurate methods not used during the actual conversion because they were thought too expensive or time consuming).

In this document, we pave the way for a systematic approach to year 2000 compliance validation. A sine qua non is a thorough understanding of the technology one can use to analyse, modify and test software systems suffering from the year 2000 disease. It is the aim of this document to provide this understanding.

For the purpose of this document, the following sources of information have been used:

- (Scientific) publications (one example is [IBM96]; see also the references at the end of the document);
- Code fragments from programs performing date manipulations.
- Commercial tools and year 2000 solutions.

This first section indicates what is part of this document and what is out of scope. To that end, we define the year 2000 problem, identify phases in a typical year 2000 conversion, and indicate which steps are within the scope of this document.

Then, Section 2 discusses and classifies a large number of COBOL code fragments for manipulating dates as used in practice. Section 3 describes which standard techniques (taken, e.g., from compiler construction) can be used to address the year 2000 problem. Section 4 covers *impact analysis*, the technique for identifying year 2000 related variables and their propagated exposures. Section 5 discusses what date representations can be used in converted systems — which of these to choose and in

what order to convert the systems is the topic of Section 6. Section 7 describes how systems can be corrected automatically. Section 8 covers the test plan, year 2000 test sets, and testing of safety critical system. Section 9 summarizes the functionality of a number of state of the art commercial tools for year 2000 conversion. The last section, finally, summarizes the issues that are relevant for validating year 2000 compliance, and identifies areas for further research.

1.2 Problem Statement

1.2.1 Available Definitions

The year 2000 problem has been described as "the largest problem without a clear problem statement." There are some definitions, though, which mainly come from US guidelines for government year 2000 compliance contracts. The State of Minnesota [Nor96] provides the following definition of year 2000 compliance:

> "Year 2000 compliance" means that information resources meet the following criteria and/or perform as described:
>
> - Data structures (databases, data files, etc.) provide 4-digit date century recognition. Example: '1996' provides "date century recognition"; '96' does not.
> - Stored data contains date century recognition, including (but not limited to) data stored in databases and hardware / device internal system dates.
> - Calculations and program logic accommodate both same century and multi-century formulas and date values. Calculations and logic include (but are not limited to) sort algorithms, calendar generation, event recognition, and all processing actions that use or produce date values.
> - Interfaces (to and from other systems or organizations) prevent non-compliant dates and data from entering any state system.
> - User interfaces (i.e., screens; reports; etc.) accurately show 4 digit years.
> - Year 2000 is correctly treated as a leap year within all calculation and calendar logic.

Unfortunately, this definition is problematic: it is unclear what "date century recognition" is (if we know which "cut off year" to choose, a two-digit year has a unique century which can be recognized). Moreover, it prescribes one solution, widening to four digits, which in many cases need not be the best solution.

A much shorter definition is from US Federal agencies, to be used in their solicitation and contracts for year 2000 compliant systems [GSA96]:

> (...) each hardware, software, and firmware product (...) shall be able to accurately process date data (including, but not limited to, calculating, comparing, and sequencing) from, into, and between the twentieth and twenty-first centuries, including leap year calculations (..)

1.2.2 Year 2000 Compliance

A simple modification of the previous characterization of year 2000 compliance leads to the definitions below:

Definition 1.1, "error": An *error* is a discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition [IEE83].

Definition 1.2, "year 2000 error": A *year 2000 error* is an error caused by the inaccurate processing of date data (including, but not limited to, calculating, comparing, and sequencing) from, into, and between the twentieth and twenty-first centuries, including leap year calculations.

Definition 1.3, "year 2000 compliance": A system is *year 2000 compliant* if it does not contain year 2000 errors.

These definitions define the "ideal" case, where every year 2000 error is to be corrected. In practice, a more pragmatic approach will be necessary, in order to minimize the cost and changes to be made in a year 2000 conversion. For each application, the following questions need to be addressed:

- What is the scope of year 2000 errors?

  As an example: many leap year calculations just divide the year by four, thus considering 2100 as a leap year, which is not correct. Should a year 2000 project repair such an error which will not occur within the next 100 years?

  Depending on the nature of the application (mortgages, for example, involve dates 30 years ahead from now) and the expected life-time of the application, a decision has to be made what the horizon of year 2000 errors is. One possibility is to decide to correct only errors that will manifest themselves in, say, the next 10 years.[1]

- What severity levels can we attach to the various year 2000 errors? For example,[2] we can categorize into (1) error causes the system to crash (ABEND); (2)

[1] In this document, we take a horizon of 50 years.

[2] These categories are used by programmers from Microsoft [CS96, p.352]. The IBM year 2000 document [IBM96, p.2-3] distinguishes between *fatal* (ABEND), *critical* (produces incorrect result), and *marginal* (cosmetic).

a feature is inoperable, and there is no work-around; (3) a feature is inoperable, but there is a work-around; (4) cosmetic, minor error.

Depending on the nature of the application and on the cost the organization is willing to invest in that application, year 2000 compliance can be re-defined by some maximum number of errors for each category, a maximum fraction per lines of code for each category, a maximum number of errors per feature for each category, a weighted sum of these, etc.

The answers to the two questions raised above will be application-specific. For an actual year 2000 project it is important that answers to these questions are given: managers, programmers, external contractors, all have to agree on whether a certain error should be corrected in the year 2000 project or not.

1.3 Steps in a Year 2000 Conversion

The steps in an actual year 2000 conversion project will depend on a number of factors. Ragland [Rag97], suggest the following five steps: 1. Awareness: Make sure everyone in an organization with a potential year 2000 problem is aware of the risks, potential solutions, and possible costs; 2. Assessment: Build an inventory of the systems involved; 3. Renovation: Carry out the year 2000 corrections; 4. Validation: Test the corrected systems for year 2000 compliance. 5. Implementation: Transfer the corrected system to the run-time environment.

A similar organization is given by Chavan [Cha96], who emphasizes a geographical distribution of work, for example between the customer site in Europe and a software house in India.

In this document, we focus on the technical aspects of year 2000 conversion. We pay attention to the following steps:

System Inventory Which software systems, modules, executables, databases, copy books, utilities, data dictionaries, JCL scripts, ... are used in the system? How can these be grouped (clustered) into separate subsystems?

One important result of this step is the *source model*, which lists all artifacts and their inter-dependences needed to build the run-time system. Another is the *run-time model*, which lists the executables and databases needed to run the system.

Impact Analysis Where in the system are the date exposures? Which system components do depend on dates? How do date dependencies propagate between programs and databases?

Choice of Strategy Which systems need to be adapted in what order? Should field expansion, windowing or another technique be used? Which window values will be used, what will be done with screens, keyboard entry of dates, etc.

Code Modification On basis of the date infections found and the strategies chosen, modify the source code, and remedy the date problems. If the impact analysis results are sufficiently detailed, automated modification may be possible.

Testing Test that the system behaves correctly within the determined year 2000 horizon, and that the code modifications did not alter program behavior in an unintended way.

The steps will not in all cases be entirely sequential; a more detailed impact analysis may affect the modification strategy, for example. In terms of the Resolver renovation methodology model (see [DKW97]), the first three of these fit in the *inventory* phase, the last two in the *conversion* phase.

1.4 Scope

In this document, we emphasize the technical aspects of COBOL/MVS year 2000 conversions. The observations and techniques, however, are language-independent, and can also be used for other languages. Moreover, many of them can in fact be used for other renovations than year 2000 conversions as well.

Of the steps mentioned in the previous section, we deal most intensively with impact analysis, code modification, choice of strategy, and testing. Out of the scope of this document are:

- Inventory of year 2000 compliance of commercial off-the-shelf products (e.g., hardware, operating system, Excel, Lotus-123, Windows 95, ...);
- Cost estimates (e.g., Gartner's expectation of $1,- per executable line of code; rules of thumb like "modification requires one day per programmer per average program", assuming a good impact analysis has been performed, etc.).

1.5 Further Publications and Activities

1.5.1 The IBM Guide

IBM has published a document entitled *The Year 2000 and 2-Digit Dates; A Guide for Planning and Implementation* [IBM96]. This document covers planning (Chapter 2, 6 pages), identification of 2-digit year exposures (Chapter 3, 4 pages), ways for reformatting of year-date notations (Chapter 4, 13 pages), testing techniques (Chapter 5, 3 pages), actual code migrations (Chapter 6, 4 pages), and a discussion of tool categories (Chapter 7, 6 pages). The document concludes with a 12 page bibliography (including several WWW sites), a 2-page glossary, and a 5-page index.

Moreover, the document provides testing techniques for IBM platforms, explaining how to change date and time for testing on various IBM machines (Chapter 5, 2 pages). In addition to that, it gives an extensive description of tool support provided by IBM (for MVS, VM, VSE, AS/400, PCs, etc.) (Chapter 7, 70 pages), consulting

services offered by IBM (Chapter 8, 2 pages), and a list describing which products of IBM are at this stage "Year 2000 ready" (Appendix A, 36 pages).

1.5.2 The IEEE Millennium Task Force

The IEEE Computer Society (The Institute of Electrical and Electronics Engineers), in particular its Technical Council on Software Engineering, has set up a task force dealing with the year 2000 problem. It organizes "year 2000 summit meetings". Its aims include to coordinate year 2000 activities of researchers, vendors, and industrial, corporate and governmental organizations, stimulating every one involved to learn from each other's experiences.

1.5.3 Remaining Pointers

Other pointers are discussed throughout this document; a list of the papers used is given in the References section.

One pointer of particular interest is the Gartner Group, which has a series of publications on year 2000 related issues, such as [Gar96]. Their publications typically cover the risks, costs, and organizational consequences of year 2000 migrations. A paper emphasizing the global costs of the millenium problem is by Jones [Jon97].

Two internet sites of interest are the year 2000 home page at URL `http://www.year2000.com/` (which also has up to date pointers to a range of commercial year 2000 solution providers) and the usenet newsgroup `comp.software.year-2000`.

More and more books become available on the year 2000 problem: examples are [MM96, Rag97, Keo97, UH97]. Also, computer science journals such as the *Communications of the ACM* (the May 1997 issue) and *IEEE Computer* (the March 1997 issue) are paying more and more attention to the year 2000 problem.

2 Example Code Fragments

In this section a list of COBOL source code fragments is provided. The list contains code fragments of all kinds of date manipulation. We do not restrict to code fragments that might cause a date problem, but consider date manipulating code in general. This is done because not only erroneous code but also correct (i.e., year 2000 compliant) code fragments are interesting, for renovation tools will have to be able to distinguish between these two. In particular code fragments with unexpected, but correct, date manipulations might be damaged by naive renovation methods.

This section consists of two parts. In the first we discuss a list of *date formats* that are used in practice. The second contains examples of *code fragments* where dates are manipulated.

2.1 Date Formats

Contrary to many other programming languages COBOL does not have a date format as part of the language. As a result of that a large variety of different formats are used in COBOL sources to represent dates. In this section we give an overview of these different date formats. Rather than giving an exhaustive enumeration of all the possible date formats we tried to give a list of formats that can be qualified as a 'representative overview' of the date formats used in practice.

In all examples of this section we presented the date formats as `01` declarations. That is to say, all date formats are presented as separate entities. One should realize, however, that all these date formats can also occur as substructures of another structure. This is quite natural and occurs rather frequently. For instance, a structure `DELIVERY` can have a date (date of delivery) as one of its substructures.

We start the list of date formats with the Conventional 6-digit format (2.1), consisting of two bytes for the day, two for the month and two bits for the year. Particularly in older source code this format is frequently used.

```
01 DATE.                          01 DATE.                          (2.1)
   02 DAY     PIC 99.                02 YEAR   PIC 99.
   02 MONTH   PIC 99.                02 MONTH  PIC 99.
   02 YEAR    PIC 99.                02 DAY    PIC 99.
```

The right-hand version of the two declarations in (2.1) has the advantage that the chronology of two dates (i.e. which date comes first) can be determined at once, simply by comparing the values of `DATE`. E.g. July 11, 1997 comes after October 16, 1962 because `621016` $<$ `970711`. In case the left declaration is used the chronology is sometimes determined by first computing

```
DAY + 100 * MONTH + 10000 * YEAR
```

which interchanges the values of `DAY` and `YEAR`. Of course, the fact that these 6-digit date formats do not contain century information does not necessarily mean that they give problems in the year 2000. For instance, if windowing techniques are used these formats are adequate.

In most of the more recently developed code, however, either a century variable is present or the year variable can contain four digits. In either case, century information is explicitly stored in the dates. Sometimes, the century variable is instantiated with `19`. In (2.2) we give two examples of date formats with explicit century information.

```
01 DATE.                                01 DATE.                    (2.2)
   02 DAY      PIC 99.                     02 DAY      PIC   99.
   02 MONTH    PIC 99.                     02 MONTH    PIC   99.
   02 YEAR     PIC 99.                     02 YEAR     PIC 9999.
   02 CENTURY  PIC 99  VALUE 19.
```

Until now, in all declarations, the contents of dates have been numeric. In many cases, however, dates are considered to be alpha-numeric values, i.e., `"96"` instead of `96` etc. Furthermore, the century and the year are often interchanged (with the same motivation as before, namely to allow for easy determination of chronology by means of a simple comparison of the contents of `DATE`). The left example of (2.3) is a declaration of an alpha-numeric date. The right example of (2.3) is numeric and orders the sub-fields corresponding to the Dutch `DISPLAY` convention (e.g. `28-07-1997`).

```
01 DATE.                                01 DATE.                          (2.3)
   02 CENTURY  PIC XX  VALUE "19".         02 DAY      PIC 99.
   02 YEAR     PIC XX.                      02 MONTH    PIC 99.
   02 MONTH    PIC XX.                      02 CENTURY  PIC 99  VALUE 19.
   02 DAY      PIC XX.                      02 YEAR     PIC 99.
```

Instead of using a century variable it is also possible keep track of the century information by means of a predicate. The level-number `88` of `20TH-CENTURY` in (2.4) indicates that `20TH-CENTURY` is a predicate on `YEAR`. The predicate holds if the value of `YEAR` is in the given range `62 THRU 99`. Typically, the range contains those year values that are supposed to refer to the 20^{th} century.

```
01 DATE.                                                              (2.4)
   02 YEAR              PIC 99.
      88 20TH-CENTURY            VALUE 62 THRU 99.
   02 MONTH             PIC 99.
   02 DAY               PIC 99.
```

The smallest year-value which is interpreted as a year of the current century is called the *break year* (or *cut-off year*). In the example above the break year is `62`. Usually, the break year refers to the production year of the source code in question. In practice we see that a wide variety of different break years is used.

Another variation on the format of dates is the presence of separation-symbols (like dashes or slashes) between day, month and year. A declaration with dashes as separation symbol could look like this:

```
01 DATE.                                                              (2.5)
   02 DAY     PIC   99.
   02 FILLER  PIC    X  VALUE "-".
   02 MONTH   PIC   99.
   02 FILLER  PIC    X  VALUE "-".
   02 YEAR    PIC   99.
```

Earlier in this section we emphasized that dates can also be contained in a larger structure. We mentioned the example of a structure `DELIVERY` that contains a date

of delivery as one of its substructures. Conversly, it is also possible that a date can consist of more than just the minimal information necessary to uniquely determine a day. As an example we give format (2.6) that does not only store a date but also administrates whether or not this date is member of a leap year.

The one-digit variable `FLAG` in (2.6) is supposed to store the result of a computation `YEAR` *mod* `4`. The predicate `LEAP-YEAR` on `FLAG` succeeds if the result of this computation equals `0`. Note that this is an erroneous way of detecting a leap year. Nevertheless, such computations are frequently used. We come back to this later.

```
01 DATE.                                                    (2.6)
   02 DAY           PIC   99.
   02 MONTH         PIC   99.
   02 YEAR          PIC 9999.
   02 FLAG          PIC    9.
      88 LEAP-YEAR             VALUE 0.
```

Instead of using a month variable one can also decide to count only the days of each year.[3] In that case the days are declared with a 3-digit format:

```
01 DATE.                              01 DATE.                (2.7)
   02 DAY       PIC  999.                02 DAY      PIC  999.
   02 YEAR      PIC   99.                02 YEAR     PIC 9999.
```

Finally, we mention the use of the `REDEFINES`-clause which introduces so-called *aliases* (different variables referring to the same memory location).

```
01 DATE.                                                    (2.8)
   02 DAY     PIC   XX.
   02 MONTH   PIC   XX.
   02 YEAR    PIC XXXX.
01 CEN-DATE REDEFINES DATE.
   02 FILLER  PIC XXXX.
   02 CENTURY PIC   XX.
   02 FILLER  PIC   XX.
```

In (2.8), a variable `CENTURY` is added to a date declaration. The value of `CENTURY` is always equal to the first two digits of the value of `YEAR` because both variables share the same memory location (aliasing).

Another frequently applied use of the `REDEFINES`-clause is the splitting of a 4-digit year variable into a year and a century variable, of which we give an example in (2.9).

[3] In case of a `DDDYY` pattern, i.e., the left format in (2.7), this is called the *industrial date* or *Julian date*.

```
01 DATE.                                    (2.9)
   02 DD    PIC   99.
   02 MM    PIC   99.
   02 YYYY  PIC 9999.
   02 CCYY REDEFINES YYYY.
      03 CC PIC   99.
      03 YY PIC   99.
```

2.2 Code Fragments with Date Manipulation

In the previous section we gave a representative overview of different date formats as found in data divisions of COBOL source code. In this section we show how these different formats are actually used in the procedure division. Thus, the code fragments are not 'artificial' but derived from COBOL sources used in actual business critical software systems. In order to emphasize the core of the fragments we replaced some pieces of code by boxes in which we phrase only the relevant aspects of the omitted code. Furthermore we made use of parameters in cases where the actual value of a variable is not relevant for the example. For instance if a break year is used we abstracted from the actual value of the break year by using a parameter instead. Parameters are written in *italics*.

We distinguish code fragments that are year 2000 compliant[4] from those that are not by means of labeling. We labeled each fragment with + or − respectively. A third label ∼ is used for code fragments that are correct in the sense that they do not yield false information as soon as dates pass the turn of the century, but that are nevertheless in some sense unsatisfactory with respect to the date problem.

The date fragments are grouped in different subsections, according to their functionality. We start with code fragments that *interpret* a date. That is to say, code fragments in which the contents of a date variable is evaluated in order to determine which physical date is represented.

2.2.1 Interpreting Date Values

This subsection contains three examples of date interpretation. The first example (2.10) is an example of using a windowing technique. The interpretation of the contents of `YEAR` makes use of a value n that determines the century of the date. Parameter n is an arbitrary fixed value in the range of $00, \ldots, 99$.

```
01 DATE-1.                    01 DATE-2.    (2.10)
```

[4] By year 2000 compliant we mean that the code behaves correct as long as dates do not exceed the year 2050. This choice is motivated by the fact that many correction strategies (e.g., windowing techniques) actually postpone the problem to the second half of the next century.

```
   02 DAY    PIC 99.              02 DAY    PIC 99.
   02 MONTH  PIC 99.              02 MONTH  PIC 99.                +
   02 YEAR   PIC 99.              02 YEAR   PIC 99.                +
                                                                   +
IF YEAR IN DATE-1 >  n AND YEAR IN DATE-2 >  n                     +
OR YEAR IN DATE-1 <= n AND YEAR IN DATE-2 <= n                     +
  IF DATE-1 < DATE-2                                               +
    DATE-1  refers to an earlier date than DATE-2                  +
  ELSE                                                             +
    DATE-2  refers to an earlier date than DATE-1                  +
  END-IF                                                           +
ELSE                                                               +
  IF DATE-1 < DATE-2                                               +
    DATE-2  refers to an earlier date than DATE-1                  +
  ELSE                                                             +
    DATE-1  refers to an earlier date than DATE-2                  +
  END-IF                                                           +
END-IF                                                             +
```

Note that (2.10) needs revision as soon as dates pass $n + 2000$. Since the break year is in practice never below 50 we marked this code fragment with +. In code fragment (2.11) also a 2-digit year variable is compared with a break year value n in the range of `00`, ..., `99`. However, contrary to situation in (2.10) dates of the 21st century are rejected which makes implies that the code is correct but nevertheless useless in the next century, because its functionality is reduced to displaying error messages (which is of course still better than reasoning with wrong dates without complaining).

```
01 DATE.                                                           (2.11)
   02 DAY    PIC 99.
   02 MONTH  PIC 99.                                                ~
   02 YEAR   PIC 99.                                                ~
                                                                    ~
IF YEAR > n                                                         ~
  YEAR  refers to a year in the 20th century                        ~
ELSE                                                                ~
  Display an error message and terminate the application            ~
END-IF                                                              ~
```

Rejection of 21st century dates can also occur in code fragments where 4-digit years are used. In those cases the rejection can, for instance, be necessary because the date is transported to another code fragment where century values are ignored. In (2.12) parameter n ranges over `0000`, ..., `9999`.

```
01 DATE.                                                          (2.12)
   02 DAY    PIC   99.
   02 MONTH  PIC   99.                                              ~
   02 YEAR   PIC 9999.                                              ~
IF YEAR < 2000 AND YEAR > n                                         ~
   YEAR  refers to a date in the 20th century.                      ~
ELSE                                                                ~
   Display an error message and terminate the application           ~
END-IF                                                              ~
```

2.2.2 Computations with Dates

This subsection contains examples of computations where dates are involved. None of the examples in this subsection are correct. Note that (2.13) and (2.14) only make sense if the values of `H-MONTH` and `H-DAY` respectively are signed, because the intermediate results can be negative. Parameter d refers to an arbitrary distance between two dates (duration).

```
01 DATE-1.                      01 DATE-2.                        (2.13)
   02 DAY    PIC 99.               02 DAY    PIC 99.
   02 MONTH  PIC 99.               02 MONTH  PIC 99.                 –
   02 YEAR   PIC 99.               02 YEAR   PIC 99.                 –
                                                                     –
01 H-MONTH  PIC S9(4).                                               –
                                                                     –
DATE-1  refers to an earlier date than DATE-2                        –
COMPUTE H-MONTH = (YEAR  IN DATE-2 - YEAR  IN DATE-1)*12  +          –
                  (MONTH IN DATE-2 - MONTH IN DATE-1)                –
IF H-MONTH > d                                                       –
   DATE-2  is more than d months later than DATE-1                   –
END-IF                                                               –
```

Fragment (2.14) is based on the same idea as fragment (2.13). A difference with (2.13), however, is that days are counted whereas the number of years is required. This 'detour' is used to enforce rounding down of the number of years.[5]

```
01 DATE-1.                      01 DATE-2.                        (2.14)
   02 DAY    PIC 99.               02 DAY    PIC 99.
   02 MONTH  PIC 99.               02 MONTH  PIC 99.                 –
   02 YEAR   PIC 99.               02 YEAR   PIC 99.                 –
```

[5]Note that the assumption that years have 12 months consisting of 30 days each (giving years of 360 days) implies that for instance the number of years between 01-01-1996 and 31-12-1996 is equal to 1, whereas the number of years between 02-01-1996 and 01-01-1997 (i.e., both dates increased with one day) is equal to 0.

```
                                                                    –
01 H-DAY  PIC S9(5).                                                –
01 H-YEAR PIC S9(3).                                                –
                                                                    –
DATE-1  refers to an earlier date than DATE-2                       –
COMPUTE H-DAY = (YEAR  IN DATE-2 - YEAR  IN DATE-1) * 360  +        –
                (MONTH IN DATE-2 - MONTH IN DATE-1) * 30   +        –
                (DAY   IN DATE-2 - DAY   IN DATE-1 )                –
H-YEAR = H-DAY / 360                                                –
IF H-YEAR >= d                                                      –
   DATE-2  is more than d years later than DATE-1                   –
END-IF                                                              –
```

We conclude this subsection with a computation that can be used to 'link' the values of a date record to one value. Note that the century information in `H-DATE` is set to `19` in this example.

```
01 DATE.                                                            (2.15)
   02 DAY      PIC 99.
   02 MONTH    PIC 99.                                              –
   02 YEAR     PIC 99.                                              –
                                                                    –
01 H-DATE  PIC 9(8).                                                –
                                                                    –
COMPUTE H-DATE = DAY + 100 * MONTH + 10000 * YEAR + 19000000        –
```

2.2.3 Alteration of dates

In this subsection we discuss some examples of code fragments that introduce an alteration in a date. For instance 'adding 1 year' or 'subtracting 2 months' are examples of such alterations. As we can see from the examples, dates cannot savely be recognized by inspecting whether the contents are within some fixed range (`MONTH` $\leq$ 12, `DAY` ≤ 31 etc.)

In code fragment (2.16) one year is added to a date. Parameter n refers to the break year again. This fragment illustrates that the 8-digit format using two century digits (i.e. CC YY MM DD) is less natural than the 8-digit format containing a 4-digit year (i.e. YYYY MM DD). Technically spoken, this is so the latter format satisfies the property that each split is at the left of a sub-structure that do not range over its full domain, whereas the first format does have a split (between CC and YY) at the left of a sub-structure that ranges over its full domain (YY ranges over $\{00, \ldots, 99\}$). This split between CC and YY introduces some extra arithmatic on dates necessary to "glue" the split.

```
01 DATE.                                                    (2.16)
   02 CENTURY  PIC 99.
   02 YEAR     PIC 99.                                          +
   02 MONTH    PIC 99.                                          +
   02 DAY      PIC 99.                                          +
                                                               +
ADD 1 TO YEAR                                                  +
IF YEAR < n                                                    +
  MOVE 20 TO CENTURY                                           +
ELSE                                                           +
  MOVE 19 TO CENTURY                                           +
END-IF                                                         +
```

Syntax-based techniques to detect manipulations like the one given in example (2.16) might search for patterns consisting of an if–then–else construction on a date comparison. Example (2.17) — which is a minor variation on (2.16) — illustrates that it can be difficult to compose a list of patterns that covers all instances of a given functionality.

```
01 DATE.                                                    (2.17)
   02 CENTURY  PIC 99.
   02 YEAR     PIC 99.                                          +
   02 MONTH    PIC 99.                                          +
   02 DAY      PIC 99.                                          +
                                                               +
ADD 1 TO YEAR                                                  +
MOVE 19 TO CENTURY                                             +
IF YEAR < n                                                    +
  MOVE 20 TO CENTURY                                           +
END-IF                                                         +
```

If the negative result of a computation is stored in an unsigned variable then the sign is simply ignored. Thus, the result of subtracting `1` from the year `00` is the year `01`. Therefore, code fragment (2.18), that is supposed to subtract one year, is wrong.

```
01 DATE.                                                    (2.18)
   02 CENTURY  PIC 99.
   02 YEAR     PIC 99.                                          −
   02 MONTH    PIC 99.                                          −
   02 DAY      PIC 99.                                          −
                                                               −
SUBTRACT 1 FROM YEAR                                           −
IF YEAR < 0                                                    −
```

```
  SUBTRACT 1 FROM CENTURY                                          -
  MOVE 99 TO YEAR                                                  -
END-IF                                                             -
```

Erroneous additions often lead to a mismatch of exactly 100 years. Erroneous subtractions, however, often lead to dates that hardly show any correspondence to the date that should have been the result of the subtraction.

For instance code fragment (2.19), which is supposed to subtract one month from a date, returns `0112`xy if the contents of `DATE` was `0001`xy, i.e., a mismatch of 24 months (assuming that the year values `00` and `01` will be interpreted as dates in the same century).

```
01 DATE.                                                           (2.19)
   02 YEAR     PIC 99.
   02 MONTH    PIC 99.                                             -
   02 DAY      PIC 99.                                             -
                                                                  -
IF MONTH < 2                                                       -
  MOVE 12 TO MONTH                                                 -
  SUBTRACT 1 FROM YEAR                                             -
ELSE                                                               -
  SUBTRACT 1 FROM MONTH                                            -
END-IF                                                             -
```

In code fragment (2.20) `H-MONTH` months are added to a date.

```
01 DATE.                                                           (2.20)
   02 CENTURY  PIC 99.
   02 YEAR     PIC 99.                                             +
   02 MONTH    PIC 99.                                             +
   02 DAY      PIC 99.                                             +
                                                                  +
01 H-MONTH     PIC 99.                                             +
                                                                  +
H-MONTH  contains a value that is less than 13                     +
ADD H-MONTH TO MONTH                                               +
IF MONTH > 12                                                      +
  SUBTRACT 12 FROM MONTH                                           +
  IF YEAR = 99                                                     +
    MOVE 00 TO YEAR                                                +
    MOVE 20 TO CENTURY                                             +
  ELSE                                                             +
    ADD 1 TO YEAR                                                  +
  END-IF                                                           +
END-IF                                                             +
```

In code fragment (2.21) m days are subtracted from a date. The code makes use of a paragraph `LEAP-YEAR-TEST` which is supposed to move the result of a computation `YEAR` mod `4` to `FLAG`.

```
01 DATE.                                                        (2.21)
   02 DAY           PIC   999.
   02 YEAR          PIC  9999.                                     +
   02 FLAG          PIC     9.                                     +
      88 LEAP-YEAR              VALUE 0.                          +
                                                                  +
IF DAY > m                                                        +
  SUBTRACT m FROM DAY                                             +
ELSE                                                              +
  SUBTRACT 1 FROM YEAR                                            +
  PERFORM LEAP-YEAR-TEST                                          +
  IF LEAP-YEAR                                                    +
    ADD 366 - m TO DAY                                            +
  ELSE                                                            +
    ADD 365 - m TO DAY                                            +
  END-IF                                                          +
END-IF                                                            +
```

Sometimes the `COMPUTE` statement is used instead of arithmetic statements, e.g., `COMPUTE` x `=` x `-` y instead of `SUBTRACT` y `FROM` x and `COMPUTE` x `=` x `+` y instead of `ADD` y `TO` x. In the following example, where the year 2000 problem is completely ignored, we used `COMPUTE` instead of `SUBTRACT` for a change.

```
01 DATE-1.                    01 DATE-2.                        (2.22)
   02 YEAR   PIC 99.             02 YEAR   PIC 99.
   02 MONTH  PIC 99.             02 MONTH  PIC 99.                 –
   02 DAY    PIC 99.             02 DAY    PIC 99.                 –
                                                                  –
01 d  PIC S9(6).                                                  –
                                                                  –
COMPUTE d = DATE-1 - DATE-2                                       –
IF d < 0                                                          –
  DATE-1  refers to an earlier date than  DATE-2                  –
END-IF                                                            –
```

2.2.4 Determining Leap Years

Many code fragments where dates are involved have to do with the leap year phenomenon. Leap years are those years that are divisible by 400 and those years that are divisible by 4 but not by 100. However, in some code fragments leap years are

simply defined as years that can be divided by 4. Since 2000 is indeed a leap year, this will not give problems before the year 2100. Note that our convention on the meaning of year 2000 compliant means that this simplified leap year definition is regarded correct.

```
01 DATE.                                                          (2.23)
   02 YEAR     PIC 99.
   02 MONTH    PIC 99.                                             +
   02 DAY      PIC 99.                                             +
                                                                  +
01 Q  PIC 9(4).                                                   +
01 R  PIC 9(1).                                                   +
                                                                  +
DIVIDE YEAR BY 4 GIVING Q REMAINDER R.                            +
IF R = 0                                                          +
   YEAR  refers to a leap year                                    +
END-IF                                                            +
```

In some cases the quotient `Q` and the remainder `R` are constructed by hand, e.g. compute `COMPUTE Q = YEAR / 4` and `COMPUTE R = YEAR - (Q * 4)` instead of `DIVIDE YEAR BY 4 GIVING Q REMAINDER R`.

Other examples of leap year determination are "less general":

```
01 DATE.                                                          (2.24)
   02 YEAR     PIC XX.
   02 MONTH    PIC XX.                                             –
   02 DAY      PIC XX.                                             –
                                                                  –
IF YEAR = "92" OR "96"                                            –
   YEAR  refers to a leap year                                    –
END-IF                                                            –
```

Code fragment (2.25) is an example of 'unexpected code'. A possible explanation for the strange condition `AND YEAR NOT = 2000` is that it written by a programmer with a defective knowledge about leap years (divisible by 4 and not by 100), and who added the extra condition because he would not be put up with source code giving false information in the near future. The result was the following code fragment:

```
01 DATE.                                                          (2.25)
   02 YEAR     PIC 9999.
   02 MONTH    PIC   99.                                           –
   02 DAY      PIC   99.                                           –
                                                                  –
01 Q  PIC 9(4).                                                   –
```

```
01 R  PIC 9.                                                    −
                                                                −
DIVIDE YEAR BY 4 GIVING Q REMAINDER R.                          −
IF R = 0 AND YEAR NOT = 2000                                    −
  YEAR  refers to a leap year                                   −
END-IF                                                          −
```

2.2.5 Date Transport

This subsection consists of some examples where dates are transported from one variable to another. Parameter n refers to the break year.

```
01 DATE-1.                    01 DATE-2.                          (2.26)
   02 DAY    PIC 99.             02 DAY    PIC   99.
   02 MONTH  PIC 99.             02 MONTH  PIC   99.              +
   02 YEAR   PIC 99.             02 YEAR   PIC 9999.              +
                                                                  +
MOVE YEAR IN DATE-1 TO YEAR IN DATE-2                             +
IF YEAR IN DATE-2 > n                                             +
  ADD 1900 TO YEAR IN DATE-2                                      +
ELSE                                                              +
  ADD 2000 TO YEAR IN DATE-2                                      +
END-IF                                                            +
```

If the 4 year digits are split into 2 digits for the year and 2 digits for the century, we get:

```
01 DATE-1.                    01 DATE-2.                          (2.27)
   02 DAY    PIC 99.             02 DAY       PIC 99.
   02 MONTH  PIC 99.             02 MONTH     PIC 99.             +
   02 YEAR   PIC 99.             02 YEAR      PIC 99.             +
                                 02 CENTURY   PIC 99.             +
                                                                  +
IF YEAR IN DATE-1 > n                                             +
  MOVE 19 TO CENTURY IN DATE-2                                    +
ELSE                                                              +
  MOVE 20 TO CENTURY IN DATE-2                                    +
END-IF                                                            +
```

However, if the break year test is missing the resulting code is of course not year 2000 compliant anymore:

```
01 DATE-1.                    01 DATE-2.                          (2.28)
   02 DAY    PIC 99.             02 DAY       PIC 99.
```

```
   02 MONTH  PIC 99.            02 MONTH     PIC 99.              –
   02 YEAR   PIC 99.            02 YEAR      PIC 99.              –
                                02 CENTURY   PIC 99.              –
                                                                  –
MOVE DAY   IN DATE-1 TO DAY     IN DATE-2                         –
MOVE MONTH IN DATE-1 TO MONTH   IN DATE-2                         –
MOVE YEAR  IN DATE-1 TO YEAR    IN DATE-2                         –
MOVE 19              TO CENTURY IN DATE-2                         –
```

Also not year 2000 compliant are the following `STRING`-versions of date transport:

```
01 DATE.                                                          (2.29)
   02 DAY    PIC XX.
   02 MONTH  PIC XX.                                              –
   02 YEAR   PIC XX.                                              –
                                                                  –
01 H-DATE-1  PIC X(08).                                           –
01 H-DATE-2  PIC X(10).                                           –
                                                                  –
STRING DAY MONTH "19" YEAR INTO H-DATE-1                          –
STRING DAY "-" MONTH "-19" YEAR INTO H-DATE-2.                    –
```

Sometimes the contents of a date is coming from a `VALUE`-clause in the declaration of the data record. Manipulations also can be performed on the level of records.

```
01 DATE-1  PIC X(6)  VALUE  x1 ... x6.                            (2.30)
01 DATE-2  PIC X(8).
                                                                  +
MOVE DATE-1 TO DATE-2                                             +
IF DATE > n * 10000                                               +
  ADD 19000000 TO DATE                                            +
ELSE                                                              +
  ADD 20000000 TO DATE                                            +
END-IF                                                            +
```

We end this subsection with an example of a code fragment that makes use of a predicate.

```
01 DATE-1.                                 01 DATE-2.             (2.31)
   02 DAY    PIC 99.                          02 DAY      PIC 99.
   02 MONTH  PIC 99.                          02 MONTH    PIC 99.    +
   02 YEAR   PIC 99.                          02 YEAR     PIC 99.    +
      88 20TH-CENTURY VALUE n THRU 99.         02 CENTURY  PIC 99.    +
                                                                     +
```

```
MOVE DAY   IN DATE-1 TO DAY   IN DATE-2                    +
MOVE MONTH IN DATE-1 TO MONTH IN DATE-2                    +
MOVE YEAR  IN DATE-1 TO YEAR  IN DATE-2                    +
IF 20TH-CENTURY                                            +
  MOVE 19 TO CENTURY IN DATE-2                             +
ELSE                                                       +
  MOVE 20 TO CENTURY IN DATE-2                             +
END-IF                                                     +
```

2.2.6 Exception handling

A technique that is frequently used in COBOL source code and even occurs in tutorials on COBOL programming, is the use of 'extreme values' (like `xxxxxx` or `??????`) as an exception code. In the case of date variables this typically means that the contents `999999` does not refer to a date. At first sight this may seem harmless because the 99th day of the 99th month does not exist. However, the consequences of such a strategy can be difficult to predict. For instance a leap year computation might identify an exception code `000000` as a leap year without complaining.

Incorrect source code with errors that are not related to the turn of the century are beyond the scope of this document. Therefore, we will not reject (mark with '−') those cases of using exception code that are in our opinion dangerous but unrelated to the year 2000 problem.

The following example shows that the contents `0` in a century variable can occur. This value does, of course, not refer to a date in the roman period. Obviously, this can be dangerous (e.g., in a correct leap year computation applied to a date in the year 1900 or 2100, this would introduce an error).

```
01 DATE-1.                    01 DATE-2.                          (2.32)
   02 DAY      PIC 99.           02 DAY      PIC 99.
   02 MONTH    PIC 99.           02 MONTH    PIC 99.              +
   02 YEAR     PIC 99.           02 YEAR     PIC 99.              +
   02 CENTURY  PIC 99.           02 CENTURY  PIC 99.              +
                                                                 +
IF CENTURY IN DATE-1 = 0                                         +
  MOVE 19 TO CENTURY IN DATE-1                                   +
END-IF                                                           +
MOVE DATE-1 TO DATE-2                                            +
```

Even more dangerous is the use of 1-1-2000 as an exception code. In the example below the contents `01012000` is used as an exception code which indicates that the format of `YEAR`, `MONTH` or `DAY` does not satisfy the predefined predicate `NUMERIC`[6].

[6]The format of a variable satisfies the predicate `NUMERIC` if and only if it is of the form `S9...9V9...9`.

We label this fragment with question marks because it is not clear from the context whether this code fragment is year 2000 compliant or not.

```
IF YEAR  IS NUMERIC AND                                    (2.33)
   MONTH IS NUMERIC AND
   DAY   IS NUMERIC                                          ?
  STRING DAY MONTH YEAR INTO DATE                            ?
ELSE                                                         ?
  MOVE 01012000 TO DATE                                      ?
END-IF                                                       ?
```

The following examples of exception handling — using a sequence of `9`'s as exception code — are probably the most standard one in COBOL sources. The first example assigns an exception code `99` to a century variable. One could imagine that also year variables are filled with exception code `99`. However, we have not found this latter form of exception handling in any COBOL source file yet.

```
01 DATE.                                                   (2.34)
   02 DAY      PIC 99.
   02 MONTH    PIC 99.                                       +
   02 YEAR     PIC 99.                                       +
   02 CENTURY  PIC 99.                                       +
                                                             +
IF CENTURY = 99                                              +
   Exception handling code                                   +
END-IF                                                       +
```

The following variation is perhaps a bit less dangerous because the exception code is more specific. That is, only `999999` rather than all strings of the form $x_1x_2x_3x_4$`99` is exception code.

```
01 DATE.                                                   (2.35)
   02 DAY      PIC 99.
   02 MONTH    PIC 99.                                       +
   02 YEAR     PIC 99.                                       +
   02 CENTURY  PIC 99.                                       +
                                                             +
IF DATE = 99999999                                           +
   Exception handling code                                   +
END-IF                                                       +
```

Finally, a rather obscure example of using dates in the (far) future for exception handling.

```
01 DATE.                                                    (2.36)
   02 DAY      PIC 99.
   02 MONTH    PIC 99.                                          +
   02 YEAR     PIC 99.                                          +
   02 CENTURY  PIC 99.                                          +
                                                                +
IF CENTURY = 79 OR 89 OR 99                                     +
  MOVE 19 TO CENTURY                                            +
END-IF                                                          +
```

3 Existing Technology

For a large part, compiler construction technology, such as described for example in [ASU86], is directly applicable to year 2000 problems. Moreover, there are a number of useful techniques from reverse engineering (such as program understanding), for which a series of references are discussed in [BKV97a, BKV97b].

3.1 Lexical analysis

Lexical analysis (also known as lexical scanning) amounts to purely textual analysis of a program's source text. In its simplest form, one can search for literal strings such as variables `date` or `year`, date-related constants such as `19` or `1996`, or data declarations such as `PIC 9(02)`. Usually, *regular expressions* are used to describe these textual patterns. A major shortcoming of lexical analysis is that it is purely textual and does not take into account the deeper syntactic structure of a program. Knowledge of this syntactic structure is mandatory for performing more sophisticated forms of analysis (see below).

3.2 Syntactic analysis

Syntax analysis (also known as parsing) amounts to analyzing the deeper syntactic structure of a program's source text. The result of this analysis is usually a *syntax tree* (parse tree, abstract syntax tree) that can be used as starting point for further analysis. A syntax tree describes the structural decomposition of a program text and defines, among others, *contains* relations between enclosing program parts and their components, for instance, between a complete if-statement and a statement in its then-branch. Note that such structural relationships can only be established by means of syntactic analysis and not by lexical analysis.

Syntactic analysis is an essential intermediate step for all forms of analysis that are described below.

3.3 Dataflow analysis

Dataflow analysis is a technique for the static analysis of programs and is primarily used to find *definitions* and *uses* of variables. A variable definition is a program statement that assigns (directly or indirectly) a value to a variable. A variable use is a program statement that uses the previously defined value of a variable. One can distinguish between forward dataflow analysis (starting from a variable definition find all corresponding uses) and backward dataflow analysis (starting from a specific variable use find the corresponding definition). Major complications for dataflow analysis are unstructured flow-of-control (e.g., goto statements) and the aliasing of variables (due to redefinitions, pointers, and the like).

Given some "seeds" (e.g., a set of date-related variables or date-returning system calls) dataflow analysis can be used to find all statements that can be affected by these seeds. Dataflow analysis is a form of static program analysis i.e., the program is analyzed without executing it. As a consequence, the results of dataflow analysis have to be a conservative estimation and may hence be imprecise.

3.4 Program slicing

Program slicing is a more refined form of analysis than dataflow analysis and amounts to determining those parts of a program (the *program slice*) that affect the values of program variables computed at some point of interest in the program. Program slicing can be *static* (not considering a program's input) or *dynamic* (considering a specific set of input values).

Consider, for instance, a program point where a date field is written to a database. Typically, program slicing can then be used to determine all parts of the program that are responsible for computing that date value. In general, program slicing will give more precise results thus yielding a smaller set of "infections" in the program.

The computation of program slices is expensive. As a result, various restricted forms of slicing are being developed that are easier to compute at the expense of some loss in precision.

3.5 Abstract interpretation/type inference

Abstract interpretation can be used to infer the type of variables by looking at their use rather than their declaration. Typically, a date-related variable will be declared as a field of 6 or 8 characters, e.g., `PIC 9(08)`. However, this declaration does not determine the actual format in which dates will be stored, for instance, in the order year-month-day versus day-month-year. From this perspective, the type of a date-related variable is left unspecified by its declaration and should be inferred from the use of that variable in the program. It may also be the case that a subfield of a variable is used in a date-related manner, while this cannot be seen from its name or declaration.

The standard approach in abstract interpretation is to interpret (execute) a program in a non-standard way: instead of using actual values one uses more abstract descriptions of these values. For instance, in the addition `3 + 4` the constants `3` and `4` will both be represented by the new constant `INT` that stands for any integer value. The abstract interpretation of `INT + INT` will then again yield `INT`. Abstract interpretation of `3 + 4` will thus yield `INT` while the standard interpretation would yield `7`. In a similar, but more involved, manner abstract interpretation can be used to infer date-related types of variables. In this case, the type descriptions will be sequences of year (in 2 or 4 digits), month, and day.

At the unavoidable expense of loosing some precision (i.e., computing the approximation `INT` instead of the precise value `7`), abstract interpretation can be used for static analysis such as the inference of the types of program variables.

3.6 Program understanding

Program understanding is a general name for all activities (including the ones already discussed above) aiming at gaining insight in a program's behavior. One of the more successful approaches is to use *program clichés* that describe certain code patterns of interest. A typical cliché could be "computing an age by subtracting a birth date from the current date". The cliché will be formulated as a pattern that is used to find all instances of the cliché during a traversal of a program. Information used to recognize such clichés include the program's syntax tree, control flow, and data flow graph. Clichés can also be used as starting point for code correction. More information on constructing cliché libraries and recognizing clichés automatically can be found in [RW90, KNE92, Har92].

3.7 Database analysis

The database can also be used as starting point for analysis as opposed to the other methods mentioned above that perform analysis on the source text of a program. The criteria for potential date information are easily verified given the actual contents of a database. The results of this analysis are the positions in each record where date information is stored in which particular format. Database analysis can complement (or refute) other forms of analysis discussed above.

4 Impact Analysis

Software change impact analysis is a general technique for identifying the consequences or ripple effects of proposed software changes [Arn95, BBE+95, BA96, Boh96]. It is one of the most important ingredients of a year 2000 conversion project; given a list of initial date infected variables or language constructs, impact analysis finds out which other fields are dependent on these date infections. A high-quality impact

analysis significantly simplifies the year 2000 correction phase, to the extent that it may even be possible to do the corrections automatically. Last but not least, impact analysis can help to decrease testing costs: the analysis results can be used to determine which execution paths are affected by a certain modification and hence need testing.

Impact analysis is very closely related to the (compiler construction) areas of data flow analysis; we refer to Section 3 and to [Hec77, Moo96] for a detailed description of various generic data flow analysis techniques.

4.1 Year 2000 Exposures

Year 2000 impact analysis aims at determining the impact of the incorrect use of year representations and year computations. The following are typical exposures (see also [IBM96, p.1-2]):

- Use of fixed century constants, such as 19.
- Use of 00 or 99 as exception code
- Incorrect field format determination (e.g., to determine whether the format is MMDDYY or YYDDMM check against ranges such as 1–31 or 1–12).
- Arithmetic underflow $(00 - 02)$ or overflow $(99 + 2)$.
- Incorrect relational operators for comparison or sorting.
- Illegal merge of 2-digit and 4-digit years.
- Leap year failures: 2000 is a leap year. This also affects computations involving day counts, such as the day of the year (March 1st 2000 is a Tuesday).

4.2 Analysis Results

4.2.1 Erroneous Results

Like medical diagnosis, impact analysis may yield wrong results. Two cases can be distinguished:

- *false negative*: The analysis yields a negative result for a particular variable or statement ("no date problem"), whereas in reality there is a date infection.
- *false positive*: The analysis yields a positive result for a particular variable or statement ("yes, a date problem"), whereas in reality this variable or statement is harmless.

Both false cases are to be avoided; in practice the difficult part is to avoid the false positives (false negatives can be avoided by not being very selective, i.e., by marking everything slightly suspicious as year 2000 infected).

For validation purposes it is important to have an impression of the percentage of false positives and false negatives a particular tool or method yields. These percentages are difficult to obtain; they are relative to reality, but for the year 2000 problem we only know the analysis results, not reality. An estimate can be obtained by:

- Comparing the results of different tools, marking different results as either false positive or false negative, and equal results as correct;
- Comparing the results with the results of a human inspection;
- Comparing the results with test data (e.g., by lifting the year to 2000)

4.2.2 Result classifications

An analysis will not only yield a list of infected statements or variables, it will also group them into categories. Example categories are:

- Format of picture clause, e.g., DD-MM-YY, YY-DDD, YY-MM-DD, etc.
- Whether date values are exported (via databases, forms printed, screens, etc.)
- The likelihood that a certain variable is date related (e.g., a date system call certainly returns a date, a variable called DT may or may not be a date).
- The sort of expression the variable is used in, if any (comparison, addition, subtraction).
- Offset in the data division, in order to deal with renamings (renamed bytes have the same offset).

4.3 Detecting Infections

4.3.1 Initial Seeds

Dates can be imported into or exported from a program in the following ways:

- From library routines or system calls returning dates;
- As input of functions that can be called by external programs (such inputs can be given, e.g., by JCL scripts);
- In screens or printed reports;
- As the result of database transactions.

Using a lexical search, suspicious variable names, constant values, or procedure names can be recognized. Example names (MONTH, MDY, JULIAN, ...) are listed in [IBM96, p.3-1]. The actual list of names will differ per program and language (German, Dutch, ...) used to write comments or choose variable names. With each keyword a confidence level (likelihood) can be attached (YEAR is more suspicious than BEGIN).

To reduce the number of mismatches, certain variables which are known not to be dates in a particular program (e.g., a program dealing with currency, where CURR does not refer to the current-date), can be listed as negative patterns.

4.3.2 Seed Propagation

A contaminated variable can infect variables in a program via assignments (MOVE or COMPUTE statements) and procedure calls in which variables are passed. Dataflow analysis is the technique to find such propagations of infected variables.

Some year-2000 tools have problems with properly propagating infections for variables that are used for multiple purposes. For example, a variable TEMP which is used to contain both dates and telephone numbers, or a variable OUTPUT-LINE to which a series of output lines is moved which is then written on an output file by a special routine. Such variables are problematic if the propagation rules used are "if a date is moved into a variable X then X is a date as well", combined with "if a variable X is used to compute the value of a date-related variable Y then X is date-related as well". In such a setting, a series of moves of $X_1, ..., X_n$ into date related variable Y makes each of $X_1, ..., X_n$ date infected. The proper solution is to acknowledge that infection propagations are not symmetrical.

Language constructs that are problematic for doing minimal data flow analysis, i.e., finding only the data dependencies that are essential (avoiding false positives) include arbitrary control flow (gotos), aliases, computed offsets in arrays, self-modifying programs (assembly, ALTER statement).

To extend intra-program dataflow dependencies to inter-program dependencies, the intra-program analysis collects exported data items for every program involved, and then uses this information to find dependencies between data items from different programs, repeating this process if necessary.

4.4 Theoretical Limitations

To minimize the amount of testing that has to be done after a conversion, it is attractive to find the *minimal* number of statements that needs to be corrected. It is easy to see that this is an undecidable problem. Consider the (COBOL-like) program below:

```
01 YY PIC 99.
01 YEAR PIC 9999.
```

```
P1.
  PERFORM P2
  COMPUTE YEAR = 1900 + YY.
```

The last statement needs to be corrected if and only if performing `P2` terminates; i.e., we need to solve the undecidable *halting problem* before we can solve finding the minimal number of statements that requires year 2000 correction.[7] It is widely known that this problem cannot be solved, therefore it is impossible to find the minimal number of statements that need to be corrected in a year 2000 conversion.

There are, in fact, many other relevant year 2000 properties that can only be determined at run time, and which therefore are undecidable as well. For example, constants 1900, 2000, 400, etc. are suspicious. But variables holding such constants as values are equally suspicious, e.g., $X + Y$ is a potential date problem if X gets the value 1900. Unfortunately, it is undecidable in general to check whether a particular variable ever gets a particular value. Likewise, it is undecidable whether an addition will result in a number greater than 100.

The conclusion of this is that it is theoretically impossible to find an analysis, correction or validation method that can be guaranteed to find all cases. It may well be possible to find methods that for code as used in practice covers 99% of the cases, but not 100%. In other words, for every such method, a set of programs can be constructed which causes the method to fail.

4.5 Related Work

Bohner and Arnold [BA96] provide a tutorial on impact analysis. Arnold [Arn95] and Bohner [Boh96] discuss the potential of impact analysis for the year 2000 problem. The latter mainly discusses a software maintenance process model, and the role impact analysis plays in each phase. Bohner particularly emphasizes how impact analysis supports release planning by identifying software life-cycle objects (SLOs) that are likely to change for each software change proposed in a set of change requests. Some element of impact determination is present in each of the following software change activities:

- understanding software with respect to the change;
- implementing the change within the existing system;
- re-testing the newly modified system.

Barros *et al.* [BBE+95] describe an object-oriented approach to impact analysis. The result of a system inventory is stored in an object oriented repository. Links

[7] The *halting problem* is the impossibility to write a program H which inspects a program p and returns true if it can be concluded that p always terminates, and false if p can loop for ever (see, e.g., [GL88, Chapter 3]).

between objects are relations such as "called-by". To predict the impact of a change, various types of modifications are distinguished, such as "interface-change", "body-change", etc. Propagation is driven by a collection of *propagation-rules*. Propagations are given a "virtuality number", which effectively counts the number of propagation steps required to derive a given dependency. Their technique is implemented in an "Impact Analysis System" (IAS), which helps a software engineer to

- define a high level specification of the intended changes;
- interactively analyse (browse through a visual representation of) the impacts of the intended modifications;
- to dynamically adjust the initial modifications and validate the results of the impact propagation performed (to filter out false positives).

5 Year 2000 Solutions

There are three well-known ways to handle year 2000 problems: widen-the-data, windowing, and encoding. The decision whether to choose for a widening, windowing, or encoding strategy is a crucial one, and should be taken with care. Below we summarize each method, and discuss advantages and disadvantages. A more comprehensive description can be found in [IBM96, Chapter 4].

5.1 Widen the data

In the widen-the-data approach, all date data are extended such that they use four digits to store the century and the year. This has the advantages that it is

- intuitively correct, and easy to maintain;
- a permanent solution;
- Can ease your migration if you selectively ignore "cosmetic-only" situations.

The disadvantages, however, are that

- it requires more space (mainly a problem for large-volume data);
- if the program uses databases, these databases must be converted as well. This can be a tremendous effort, affecting many other programs also using these databases;
- if the program communicates date data with other programs, the interfaces with these programs have to be adapted;

- might impose a performance penalty due to increased time in processing and data access.

Because in many cases it is not desirable to touch or change the large volume database unless it is strictly necessary, the widen-the-data approach is generally not used when two-digit dates are stored in databases. Or, as formulated by [Rag97, p. 70]: "Widening is the preferred method for solving the *Year 2000 Problem*. However there are very few, if any, organizations that are actually using this method. Many organizations feel that they cannot possibly be completed with their Year 2000 tasks on time if they use the widening solution."

5.2 Windowing

Windowing is a technique where every two-digit year less than a certain *break year*, say 80, is assumed to be in the 21st century, and every year greater or equal than the break year is assumed to be in the 20th century. This has the advantages that:

- 2-digit year data stored in databases need not be modified;
- century information can be derived from the two-digit years.

Its disadvantages are:

- finding out which century a year is in will be more time consuming;
- it only works if the time range of the program is less than 100 years;
- different programs exchanging dates should be aware of the break year to be used;
- program modifications are necessary to correct comparisons, indexing and sorting procedures;
- the program modifications increase the complexity of the code, and complicate future maintenance;
- choosing a fixed break year will require annual maintenance, or maintenance as the break year approaches.

Within one organization, various applications may have to use different break years — but programs exchanging date data should be aware of the break year to be used (in practice it will be a constant in some library or copybook).

Alternatively, a "sliding window" can be used, which increases every year.

5.3 Encoding or Compression

A last technique proposed is to encode or compress dates with 4-digit years into representations originally devised to contain only 2-digit years. Examples are the use of a COBOL COMP-3 format to pack a CC-YY-MM-DD date into 6 bytes, the use of a flagged Julian format C-YY-DDD, or to convert the numbering scheme from decimal to hexadecimal. There also encodings possible where existing 20th century dates are represented as they were, and new 21st century dates are encoded in

The benefits of such encodings are (see also [IBM96, p.4-8]):

- There is no need to expand databases.
- It is possible to distinguish years from different centuries.
- If the representation of old dates does not change, there is no need to adapt the databases.

Their disadvantages are:

- The scheme can be applied only to a limited date range (which depends on the encoding implemented).
- Programs will have to be modified to perform the encoding / decoding.
- If the representation of old dates does change then
  - stored data in databases will have to be modified.
  - all programs using these dates will have to be modified simultaneously.
- Due to data conversion you may experience a performance impact.
- Encoded dates require conversion whenever you work with that data. Therefore, the presence of encoded dates will add another layer of complexity.

As observed by [IBM96, p. 4-6], encoding / compression is "considered the least desirable approach and should only be used if absolutely necessary."

6 Choice of Strategy

6.1 Which Solution to Choose?

As said before, deciding which strategy to follow requires careful thought. However, the general approach taken seems to be[8] to

[8] At this moment, there is no published evidence. The remarks above are partly based on presentations by Royal Dutch / Shell and KLM Royal Dutch Airlines at a year 2000 seminar in Amsterdam, May 28th 1997.

- Leave systems currently using four digits as they are, carefully checking for correct date exports, imports, leap years, hard century assumptions, etc.;
- Leave systems currently using two digits as they are, adapting the date logic to use windowing;
- Leave interfaces containing dates as they are (two or four digits).

The rationale behind this strategy is that if a system used two digits, its data probably fit into a 100 year range anyway, and that adapting the interface can require a cascade of modifications preventing incremental modification.

Obviously, in individual cases there will be good reasons to deviate from this strategy, in which case the pros and cons of windowing, widening or compression as discussed in the previous sections will have to be evaluated for the application at hand.

6.2 Clustering

A large software portfolio cannot be made year 2000 compliant in a single step. Instead, the programs will have to be corrected in series of more or less independent clusters. Relevant issues are:

- Programs that are (almost) independent, i.e., have no or a very small common interface, are to be placed in different clusters; programs that are highly dependent on each other are to be put in the same cluster.
- Once the clusters are determined, the dependencies determine an order in which the individual clusters can be migrated. Independent clusters can be migrated in parallel. Clusters with dependencies may have to be processed in a particular order (for example, if one cluster provides functionality used by the other). Bridge technology (see below) can be used to solve interface inconsistencies if new date formats are introduced.
- The order imposed by interface dependencies may conflict with business priorities.
- In the literature several publications have appeared that describe techniques that can be used to determine proposals for clustering automatically, such as [MNB+94]. A general discussion, covering more references on applications in the software re-modularization domain, is given by Wiggerts [Wig97, Section 4].

6.3 Bridge Programs

To make heterogeneous situations possible, where one system is corrected using windowing, and another using widening, *bridge* programs can be used. A bridge program

can convert data from one record format into another. A typical bridge program covers [IBM96, p.4-10]

- Input date format and encoding method;
- Output date format and encoding format;
- Logic that converts the data from input format to output format based on their encoding methods.

Bridge programs allow the gradual conversion of program and/or data, while still maintaining the compatibility between different data formats.

There exist automatic code modification tools, such as Peritus [HP96], which generates bridges and bridge calls in order to support incremental database conversions.

7 Code Corrections

Automating the year 2000 modifications is attractive, since it avoids boring work, is less costly, and yields more predictable results. Automation is only possible if the impact analysis results are sufficiently detailed and accurate.

Most techniques for automatic modifications heavily rely on code generation and transformation techniques taken from the area of compiler construction. The most suitable technique is to parse the sources, build an abstract syntax tree, annotate the tree with attributes representing date information, use these annotations to decide which modifications on the tree need to be made, and finally pretty print the modified tree. There also exist modification methods which rely on lexical scanning only, i.e., which are not aware of the tree structure and do a textual replacement only. This is equivalent to doing a manual "find and replace", which is dangerous since this gives no guarantee that the modified sources are syntactically correct.

If the code is modified automatically, the rules that govern these modifications should be explicitly available, thus supporting validation by human experts. Each rule should describe:

- The conditions under which it is applicable;
- What sort of modification is being made;
- An explanation why this rule is correct;
- Suggestions for ways to test the given modification.

An issue of concern is the future maintainability of the converted code. The least thing that should be done is proper commenting of the modifications made automatically. Since the comment is to be maintained in the future as well, the generated comment should be as concise as possible, while still being understandable to the future maintainer.

The highest maintainability is achieved if all date manipulation functions are available in a library, the actual date representation used is encapsulated, and whether two-digit years with a break year or four-digit years are used is not visible outside of this library. The library also provides functions for reading dates from files in a given format, writing dates to a particular format. An example date library is the IBM COMUDAS collection of date routines [IBM96, p.7-26].

When using such a library, a year 2000 conversion will replace existing (two-)digit operations by library calls. In practice this may require more code modifications than just doing in-line checks related to the break year (which for some organizations — if they have a policy of absolutely minimizing the number of changes in order to reduce the testing effort — may be a reason not to adopt this approach).

Validation will consist of:

- Checking the applicability of the solution (widening, windowing) chosen for the given system;
- Inspecting the library of date functions to be used (if available);
- Inspecting the rules used for automatic modification.

8 Testing

The testing phase of a year 2000 conversion is generally considered as the most expensive phase, which is expected to consume up to 50% of the migration effort. Much of the year 2000 test phase will be similar to regular testing activities, so organizations having well-established testing procedures will have a competitive advantage. A recent book covering all aspects of testing, throughout the entire software life cycle, is by Perry [Per95]. For year 2000 testing, the chapters on maintenance, installing software changes and testing tools [Per95, Chapters 11, 12 and 18] will be most relevant. A recent publication dealing specifically with a millennium testing factory is by Feord [Feo97].

In this section, we try to cover those aspects of testing that are specific to year 2000 migrations.

8.1 Controlling Testing Costs

Jones [Jon97] estimates that the millennium testing costs may consist of "testing the year 2000 repair" which covers 10%–30% of the total costs involved in making a system year 2000 compliant, and "regression testing the portfolio", which may involve 20%–50%. Thus, testing costs can be substantial, and everything that can be done to reduce these will be welcome.

First of all, millennium testing confronts an organization with testing obligations it will probably have never experienced before. All, or at least a great deal of its

software portfolio will have to be tested in a very limited amount of time. Moreover, possibilities to off-shore support during the test phase will be more restricted than during the analysis or correction phases: major parts of the testing will have to be done in-house. This calls for a well-organized testing factory, which can repeatedly perform massive numbers of program tests. Feord [Feo97] discusses issues related to setting up a millennium test factory.

It is of utmost importance to start testing in a very early phase of the millennium migration (i.e., start now). Testing for millennium compliance will involve two aspects: correct treatment of the year 2000, and unmodified behavior of non-date related issues. For the first, test cases covering dates will have to be carefully constructed; for the second a properly designed baseline test will have to be devised and run, so that future regression tests are possible. Both sets of test cases can be constructed at the beginning of a millennium conversion project. In fact, early testing may have an additional advantage: For organizations still having problems with year 2000 awareness, running sample tests may convince the sceptic that the software will crash indeed in the next millennium.

The amount of testing required, and hence the costs involved, is directly related to the quality of the tools used during the analysis and modification phases. Highly automated tools will be able to deal with standard situations correctly, without introducing errors. Thus, testing costs may be significantly lower if

- the software to be converted uses a relatively normal date representation and processes dates in a more or less standard way;
- the software is written in a widely used language (say COBOL/MVS) for which extensive tools exist;
- the tools used for conversion have been run on a substantial number of lines (hundreds of millions) in earlier projects.

In such a situation, the correction phase can be considered safe, and only minimal tests need to be run.

The amount of testing can further be reduced by carefully selecting the test cases (see also below). If proper impact analysis tools are used, the analysis results can be used to determine test data. The better the analysis results, the more specific the test cases can be. Depending on the analysis, it may even be possible to generate test cases automatically. To build in an extra layer of independence, one may consider using a different analysis tool for deriving test cases than the one used during the impact analysis used for code conversion.

Last but not least, testing costs can be controlled by using proper testing tools. Testing generates large amounts of information, necessitates numerous computer executions, and requires coordination and communication between workers. Typical tool support involves (see also [Per95, Chapter 18]):

- Time travelling and facilities for simulating new dates.

- Capture and playback.
- Result comparators.
- Test path coverage, which provides an indication of the completeness of the test run; it should preferably be able to express coverage information in terms of potential infections found during impact analysis.
- Testware libraries.
- Version control.

8.2 The Year 2000 Test Plan

The year 2000 test plan should at least contain the following:

- Overall description of the major phases of the testing process. This includes a description of each phase such as its purpose, desired input and output, dependencies on other phases, available tools, required expertise of personnel performing this phase, and the like.
- Overview of the software items to be tested. This includes all programs, copy books, JCL scripts, data sets, and the like. Also the physical formats in which the software items will be delivered for testing have to be established.
- Overview of all date-related topics that have to be tested. In some cases, certain "cosmetic" occurrences of dates are not converted. Typically, two-digit dates on interactive screens or printouts may be left unchanged. An inventory of such date occurrences should be made.
- A testing schedule for determining when software items are needed and for resource allocation within the testing team.
- Recording procedures for saving the results of test runs in order to make test runs reproducible and to enable auditing. The former is important when testing reveals that certain date-related variables have been missed during the conversion. Parts of the conversion should then be repeated and the outcome should be tested anew. The latter is important for quality assurance of the testing process as a whole.
- Overall description of the required test sets, and a schedule for test set preparation. This should include an inventory of the available test sets for the original program that could be reused or adapted.
- Hardware and software requirements. Typically, actual conversion will be done in another hardware/software environment than the production environment in

which the original program is normally used. This implies that the requirements for the testing environment should be made explicit, including possibilities for time travelling (see Section 8.3.3, below).

- Constraints that may affect the testing process, which include:
  - Availability of the software items to be converted.
  - Availability and expertise of test personnel.
  - Availability of the test environment. This includes access to the system on which the testing will be done as well as release dates of year 2000 compliant versions of necessary system software.
  - Allocated resources.
  - End date of test phase.

8.3 Preparing Test Sets

8.3.1 Selection of Test Path

In general, it may be too expensive to completely (re)test the converted code. However, the test cases used should at least execute the following statements in the converted code:

- All statements that have been changed or have been introduced by the conversion (to be determined by a file difference between the original and the converted code).
- All statements that refer to date-related variables (as determined by the impact analysis).
- All statements that execute date-related system calls.

8.3.2 Partitioning of Test Values

As usual, for each specific testing activity, there exist certain classes of values that crucially exercise the converted code. In the case of year 2000 conversion a number of test ranges can be defined. Some important ones include (see also [IBM96, p.5-7]):

- $year < 1980$
- $year = 1997, 1998, 1999$
- $year = 2000$
- $year = 2001, 2002, 2003$

- $2010 < year < B - 2$, where B is the break year used if a windowing solution is chosen.
- $year = B - 1, B, B + 1$
- Roll-over from 1999/12/31 to 2000/01/01
- Treatment of 2000/02/29 (should be recognized as valid)
- Periodical (daily, weekly, monthly ...) actions.

8.3.3 Time Travelling

In order to exercise date-related system calls, it is necessary to test the converted code in a simulated environment in which the system date can be set to the year 2000 and beyond.

One of the complications here is that not all system software that is needed during such a simulation is already year 2000 compliant.

8.4 Back-to-back Testing

The original code and the converted code have the same functionality, except for the treatment of dates. This creates the opportunity to apply so-called *back-to-back testing* to both systems. The necessary steps are as follows:

- Use the *same* set of test data for testing both the original program and the converted one. The cases where the original program generates erroneous output should be explicitly marked as such in the test set. In addition, expected date values for these cases should be given in the test set as well.
- Execute the original program and the converted one, while saving all generated output.
- Apply a result comparator to both outputs. The outputs should be identical, except for date values, for which the following cases can be distinguished:
  - Original program worked correctly: original date values and converted (e.g., expanded, windowed) new date values should be equal.
  - Original program worked incorrectly: new date values should match the expected values as given in the test set.

8.5 Safety-critical Programs

When the original program is safety-critical, one may consider the following additional measures to improve the quality of the program after year 2000 conversion:

- *Multiple impact analysis before conversion.* In order to maximize the amount of date-related problems that are found during impact analysis, one can apply *different impact analysis tools on the same program* and merge the results.

- *Impact analysis after conversion.* The converted program should no longer contain date-related infections. One way of determining this is by performing a complete *impact analysis after the conversion*, i.e., on the converted code. Preferably, different tools for impact analysis should be used for the first and the second analysis. It is obvious that no date-related problems should be discovered during the analysis of the converted code.

- *Multiple conversions.* One can also perform multiple conversions (using different tools) and compare the quality of the converted programs. Discrepancies between the output of the various converted programs may reveal bugs. In this way the reliability of each version can be assessed enabling the selection of the "best" version. Theoretically, one may even consider to use all converted versions of the original program in the production environment and use a majority voting mechanism to determine which program results should be used. In practice, this approach is mostly limited to programs that do not make modifications of global data sets.

9 Tools

9.1 Tool Classification

For all steps in a year 2000 conversion (see Section 1) tool support exists or is conceivable and tools can be classified accordingly. In the document *The Resolver Questionnaire for Renovation Tools* [DK97] a first attempt has been made to understand and assess commercially available tools.

In Section 9.2 we describe examples of tools we have studied so far. We conclude the section with a discussion of the limitations of these tools (Section 9.3).

9.2 Example Tools

9.2.1 Software Refinery

The Software Refinery environment, provided by Reasoning Systems, Palo Alto, is a framework for building re-engineering tools. In the last decade, it has been used in numerous successful projects involving the migration or renovation of legacy software

systems. As an example, Markosian et. al describe the use of Refinery for the automatic modularization of a 40,000 lines of code COBOL application from Boeing [MNB+94].

The discussion below describes the Refine/2000 solution which extends Refine/-COBOL, the suite of COBOL reengineering tools built using Software Refinery. The discussion is based on a 20-minute demonstration and the Software Refinery "Year 2000 white paper". The Refine/2000 solution consists of the following steps:

Building the system model: The system model describes all modules, datafiles, data bases, and jobs that comprise the organization's software portfolio, as well as the interfaces and data flow between these. The system model is used to identify subsystems that can be processed in isolation.

Most of the system model is automatically generated by Refine/Cobol after it has parsed and analyzed the JCL and COBOL programs. The system model is presented as an interactive graph where different icons represent the elements of the model, and links represent the relationship between the elements.

Analysis: The analysis phase determines *type information* about the data elements, and identifies those that actually might need year 2000 correction. Using a "data-flow-directed inference process" this information is propagated through the program logic.

The type information of a data type is a collection of properties that characterizes the values that can be stored in the data element. The sort of information inferred includes:

- The units of the data element (years, quarters, months, days, etc.);
- Whether the data element is an absolute date or a time interval;
- The range of values that the data element will store;
- Related data elements that serve as higher-order or lower digits.

The initial type clues are referred to as *seeds*, and are given a *confidence level*. The most reliable seeds come from known date manipulating or calendar routines. A less reliable source is pattern matching the names of data elements against a library of names such as `YYMMDD`, `YEAR`, `ANNIVERSARY`, etc.

Analysis is performed both at the intra- and inter-module level. When an intra-module analysis is complete, type information about externalized date-related data elements is communicated to the inter-module analysis. These are then propagated, using the system model, to other modules that read or write the same data elements. Intra- and inter-module analyses are alternated until all type information has been propagated.

Program correction: Infected programs can be corrected using the *widen-the-data* approach, which changes centuries to four digits. First the data division is

corrected, carefully taking care of aliases (RENAMES, REDEFINES). The procedure division must be corrected to cater for number constants such as "97". Moreover, the user is warned about screen maps (which may not have sufficient space for four digits), and JCL scripts which might have to be adapted (e.g., if they contain constants).

As an example of how to specify such a program correction in the Refine language, consider the following [BSM96]:

```
rule CORRECT-DATE-DIFFERENCES
  a = 'COMPUTE @RESULT = @D1 - @D2'
  & date(D1)
  & date(D2)
-->
  a = 'CALL COMPUTE-INTERVAL USING @D1 @D2'
```

This is a Refine *transform* rule. The rule recognizes that the node a matches the surface syntax for a COMPUTE statement involving subtraction. Moreover, the two conditions inspect the results of the analysis stored in the `date` attribute, indicating that both arguments are indeed dates. It then changes the node a to call a library routine `COMPUTE-INTERVAL` for computing date differences correctly.

Data correction: In the widen-the-data approach, the years stored in two digits in databases need to be expanded. Refine/2000 offers the options of a *batch conversion* or an *on-the-fly conversion*.

Recent Refine/2000 releases also support other correction schemes, such as (sliding) windows.

On the Use of Slicing Refine/2000 uses *program slicing* [Wei84, HR92, Tip95] during the analysis phase, as discussed in Reasoning's presentation [BSM96]. Slicing is a technique for determining those parts of a program that are responsible for giving a certain variable a particular value.

During the analysis phase, Refine/2000 first detects which variables are potential dates. Then it uses *forward* slicing, which finds all the lines of code impacted by the value of such a variable. This information is used to find other, related date fields. In [BSM96], the following example is given:

```
01 ...
  05 YEAR            PIC 99.
  05 ANV             PIC 99.
01 ...
  05 PARTS-PER-ORDER PIC 99.
```

```
  05 PARTS-COUNT     PIC 99.
01 ...
01 TEMP              PIC 99.
P1.
  MOVE YEAR TO TEMP.
  ...
  MOVE TEMP TO ANV.
P2.
  MOVE PARTS-PER-ORDER TO TEMP.
  ...
  MOVE TEMP TO PART-COUNT.
```

Assuming that the slicer knows that `YEAR` is a date field (derived, e.g., from the name), it finds out that within its forward slice (paragraph `P1`), the variables `ANV` and `TEMP` are used as dates. Outside this slice (in paragraph `P2`), it does *not* assume that `TEMP` or `ANV` are dates. In particular, Refine/2000 does not conclude from the last statement `MOVE TEMP TO PART-COUNT` that `PART-COUNT` must be a date; This statement is out of the scope of the forward slice of `YEAR`, and hence in that statement it is not assumed that `TEMP` is a date field, and therefore it is not concluded either that `PART-COUNT` is a date field.

9.2.2 Peritus

Peritus [Har95, HP96] is a generic environment. Its core ingredients are a LALR parser generator (based on Bison), and the *Peritus Intermediate Language*, a formalism based on Dijkstra's guarded commands [Dij76].

[HP96] covers the year 2000 tool from Peritus, AutoEnhancer/2000 in full detail. It describes a translation of dates to the following format:

```
01 ISSUE-DATE PIC 9(8).
01 ISSUE-DATE-YMD REDEFINED DATE.
   02 ISSUE-YYYY  PIC 9999.
   02 ISSUE-MM    PIC 99.
   02 ISSUE-DD    PIC 99.
```

Now `19440212` refers to February 12, 1944.

Peritus uses a syntax-directed attribute evaluation to compute a date *format* for variables involved. A format consists of ordered sequences of elements from {`C`, `Y`, `M`, `D`, `Z`} (where `Z` represents unknown). For variables, formats can be extended with their offset with respect to the first PIC byte defined, thus catering for redefines. For example, `ISSUE-DATE` above has format $\langle$`YYYYMMDD`$, 0\rangle$, `ISSUE-YYYY` has format $\langle$`YYYY`$, 0\rangle$ (i.e., the same offset, but shorter), `ISSUE-MM` has format $\langle$`MM`$, 3\rangle$. Constants have no offset, but do have a *base* value.

Propagation of the format information gradually builds a relation L between formats that depend on each other. These dependencies are derived from assignments,

I/O statements, procedure calls, and renamings. They are initialized by *seeds*, which have to be indicated manually. Moreover, whenever a variable is truncated such that its YY part is lost, a relation between the variables is removed from L.

In addition to the local relation L, a global, inter-program relation G is constructed (the dependencies are derived from the JCL code). The transitive closure of L and G gives all variables that need to be changed.

The code correction process is governed by a set of *correction rules*. Each rule consists of a name, an informal explanation, one or more examples of the use of the rule, a proof that the rule is correct, and the actual replacement formalized in the AutoEnhancer/2000 rule language. Currently AutoEnhancer provides *isolation rules*, which look at the formats and mark certain variables as date-sensitive; *data division correction rules*, primarily merging year and century fields and widening fields to accommodate 4-digit years; and *procedure division rules*, mainly for adjusting constants and removing century indicator logic.

Data correction can be done in batch mode, or by generating *wrappers*, which are subroutines used for dynamic, runtime access, performing data correction between two and four-digit representations. Special care can be taken for data fields that are used for date and non-date information.

For the testing process, AutoEnhancer automatically generates test data for the modified code. A *test data generator* processes a *correction report* as well as the corrected code. This results in a *test data report*, listing test data points along with the reasoning that led to the generation of that data. The *data formatter* then uses the test data reports to actually insert test data points into valid transactions and database records.

9.2.3 SEEC

The COBOL Analyst and the Date Analyzer are products of SEEC Inc.

COBOL Analyst is a PC-based conversion and maintenance tool for most COBOL dialects, JCL, and other COBOL extensions usually found in a mainframe environment such as, IMS/DB, CICS, ADABASE, and others. The tool is primarily intended for surveying and analyzing complete applications. In addition to lexical scanning and parsing, the tool uses a limited, heuristics-based, form of dataflow analysis for discovering dependencies between statements.

The results of analysis are stored in a repository and can be visualized in a flexible manner. Conversion is based on predefined templates and can best be characterized as machine-assisted editing in the form of (generated) editor-macro's. Testing is supported by a test case generator that uses (a limited form of) program slicing to find the path that have to be tested.

Date Analyzer is a year 2000 conversion tool. It uses lexical scanning to discover date-related variables. Starting with an initial set of regular expressions that represent

string patterns that may occur in date-related variables, a first list of date-related variables is produced. After manual pruning and extension of this list, it is further extended through dataflow analysis as provided by the COBOL Analyst. After several iterations, a stable list of date-related variables is reached which forms the basis for actual conversion. All usual year 2000 conversion scheme's are supported.

9.2.4 ARCdrive

ARCdrive is a tool developed and used by CAP Gemini.

ARCdrive is a year 2000 conversion tool for MVS COBOL. First, the application is parsed and placed in a repository. Analysis starts by defining (in the form of an Excel spreadsheet) the various string patterns that may occur in names of date-related variables. The definitions make a distinction between many different date formats (i.e., length and ordering of the elements in a date), intended use of variables (i.e, pure date variables, print lines, temporaries, and the like), and the best conversion rules to be applied. Using this initial information, a Prolog-based algorithm for the propagation of uncertain information is applied with as result a type assignment for all global variables including a certainty factor describing the plausibility of the inferred type.

This inferred information has to be confirmed (and sometimes adjusted) by an application expert. After this confirmation, a fully automatic conversion is done that applies built-in conversion rules. Due to the high level of automation, it is possible to repeat the complete conversion in case the source programs have been changed in the mean time (due to maintenance in the production environment).

9.3 Limitations and Research Directions

Based on our (admittedly limited) experience with commercially available tools, we come to the conclusion that various aspects of tool support for year 2000 conversion are still weak and merit further improvement.

Support for multiple languages and dialects is important. Many tools use a language-specific approach and cannot easily be adapted to other source languages. However, even for more generic tools many language definitions for commonly occurring languages or dialects are not available.

From a research perspective, it seems important to try to achieve generally available (and readable) formal language definitions for languages such as COBOL and PL/I.

Program analysis techniques used are nearly always limited in scope for efficiency reasons. This is the case for the various forms of dataflow analysis and also for the more advanced forms of program slicing. Currently, we do not know what the effect

(both on the speed and the quality of the analysis) would be of using more general and sophisticated techniques.

Automatic conversion is the key to improving the quality of conversions. In most cases, the translation rules are unknown and are thus not accessible for independent validation. This is unfortunate, since automatic, validated conversion rules will reduce the costs for testing.

Workflow Although most tools are embedded in an informal renovation method, tool support for managing the workflow of renovation processes seems to be missing.

Testing is an essential step before accepting the outcome of a conversion, as discussed in Section 8. Some of the tools discussed above give limited support for testing. So far, we have not yet studied the merits of general purpose testing tools to validate year 2000 conversions.

10 Concluding remarks

10.1 Summary of compliance-related issues

Based on the analysis given in this document, we can conclude that the following issues are important for determining year 2000 compliance as well as the quality of conversions in general. Here, we do not explicitly address economic aspects of year 2000 conversions.

Quality of the conversion process

- Does it cover all aspects of a year 2000 conversion, including:
  - System inventory.
  - Impact analysis.
  - Choice of strategy.
  - Code modification.
  - Testing.
  - After Care and Implementation.
- Is the whole conversion process reproducible?
- Are there explicit quality measures for each step?

Quality of the conversion and testing tools

- Quality of impact analysis (percentage of false negatives, i.e., missed date problems in the code).
- Quality of code conversion (do the rules for code conversion handle the date problems correctly?)
- Reproducibility of analysis and conversion (can the conversion be done in an iterative manner such that new information gathered in one iteration can be used to improve the quality of the conversion in the next iteration?)
- Are the testing strategy and testing tools sufficient to determine the quality of the converted system?

Quality of the converted systems

- Does the converted system deal correctly with all date-related issues?
- Does the converted system otherwise exhibit the same behavior as the unconverted system?

10.2 Research issues

We have identified the following research issues that are relevant for year 2000 conversions and merit further investigation.

- *Techniques for finding date-related patterns.* A prerequisite for automatic conversion is the ability to find not only all date-related variables but also common patterns of their usage. The more sophisticated these patterns, the better conversion rules can be defined. One possible technique is to further elaborate the notion of "program cliche's" — a first step is described in [DQW97]. Another interesting idea is to use dynamic techniques and run time information as well — Reps *et al* [RBDL97] discuss *program profiling* (running a program with pre- and post-2000 dates and comparing the paths covered) and its applications to finding date infections.
- *Techniques for the validation of conversion rules.* The amount of testing can be reduced by a better validation of the rules that are being applied during a conversion. Manual conversion requires full testing of the converted code, while a completely automatic conversion based on validated rules only requires minimal testing.
- *General testing techniques.* Testing of general software is of utmost importance in practice, but does not get the academic attention it deserves. We propose to start building up expertise in this area by formulating and exploring several basic research questions related to testing.

- *Year 2000-specific testing techniques.* While there is an abundance of general testing tools, there does not yet seem to be a lot of support for date-related testing. Particularly important is the interplay between the validation of conversion rules just mentioned and the testing techniques that can/should be applied.

- *Benchmarks for conversion tools.* One approach to assess the quality of conversion tools is to gradually build benchmarks — based, for example, on the code fragments listed in Section 2 — for classes of conversion tools (e.g., COBOL-specific, PL/I-specific, etc.). These benchmark should contain a variety of common programming practices for date manipulation as well as more exotic constructs that are encountered in practice. Since the metrics of these benchmarks are know beforehand, a detailed, quantitative, comparison of existing tools becomes possible.

10.3 Closing remarks

In this document, we have compiled an extensive overview of the technology that can be used to analyze, modify and test software systems with potential year 2000 problems. The resulting document complements other publications which emphasize the — also important — awareness and project management aspects.

There are several important conclusions that can be drawn based on this technology overview.

First of all, fully automatic migration with a 100% guarantee for all cases, as well as fully automatic validation of migrated code, is impossible to achieve. As discussed in Section 4.4, many of the relevant properties, such as will the run time value of a variable be 1900, are *undecidable*, and therefore impossible to determine, whatever technology is used. As a consequence, automatic conversion or validation of strategic systems will always have to be complemented with manual inspection and testing.

The fact that tools cannot be guaranteed to cover 100% in all cases, certainly does not mean they are useless: they are tuned towards software as used in practice, for which they may achieve a correction coverage of up to 99%. As discussed in Section 8.1, systems using dates in a standard way, written in a common language, using automatic conversion tools heavily used in earlier conversions, will require less testing and are therefore less expensive to make year 2000 compliant.

Tool developers have, for understandable reasons, exactly focused on this category of systems: written in widely used languages such as COBOL, and covering all standard date representations. Far less developed are tools that can be easily adapted to both new languages and deviating date representations. The tools that are available, are generally based on more superficial (lexical) analysis techniques, and consequently significantly less reliable during automatic conversion. Moreover, such tools may run into trouble when the lexical information, such as names of variables, is either misleading or missing, which for example is the case for systems for which the original source code is lost and conversion has to be done on sources recovered

from binaries. Further tool limitations are discussed in Section 9.3.

Several research initiatives have been taken in order to improve the state of the art of year 2000 tool technology — see the previous section for a discussion of open questions. Within the Resolver project, the DHAL (Dataflow High Abstraction Language) research program aims at developing a generic, flexible and accurate framework for carrying out data flow analysis, which is at the heart of all impact analysis techniques [Moo96].

References

[Arn95] R. S. Arnold. Millennium now: Solutions for century data change impact. *Application Development Trends*, pages 60–66, January 1995.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BA96] S. A. Bohner and R.S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[BBE+95] S. Barros, Th. Bodhuin, A. Escudié, J. P. Queille, and J. F. Voidrot. Supporting impact analysis: A semi-automated technique and associated tool. In *Proceedings International Conference on Software Maintenance ICSM'95*, pages 42–51. IEEE Computer Society Press, 1995.

[BKV97a] M. G. J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997. Also Chapter 6 in A. van Deursen et al., editors, *Program Analysis for System Renovation; Resolver Release I.*

[BKV97b] M. G. J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. *ACM Software Engineering Notes*, 22(1):57–68, 1997. Also Chapter 5 in A. van Deursen et al., editors, *Program Analysis for System Renovation; Resolver Release I.*

[Boh96] S. A. Bohner. Impact analysis in the software change process: A year 2000 perspective. In *Proceedings International Conference on Software Maintenance ICSM'96*, pages 42–51. IEEE Computer Society Press, November 1996. Monterey, CA.

[BSM96] W. A. Brew, K. Schimpf, and L. Z. Markosian. *Application of Program Slicing and Program Transformation to Solving the Year 2000 Problem*. Reasoning Systems, Palo Alto, California, 1996. Presentation at the 5th Reengineering Forum. 17 Slides.

[Cha96] S. Chavan. The year 2000 date conversion process. *Software Process–Improvement and Practice*, 2:111–122, 1996.

[CS96] M. A. Cusumano and R. W. Selby. *Microsoft Secrets*. HapperColllins, 1996.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DK97] A. van Deursen and P. Klint. The resolver queationnaire for renovation tools. In A. van Deursen, P. Klint, and G. Wijers, editors, *Program Analysis for System Renovation – Resolver Release I*, chapter 9. CWI, Amsterdam, 1997.

[DKW97] A. van Deursen, P. Klint, and G. Wijers. An overview of system renovation. In A. van Deursen, P. Klint, and G. Wijers, editors, *Program Analysis for System Renovation – Resolver Release I*, chapter 1. CWI, Amsterdam, 1997.

[DQW97] A. van Deursen, A. Quilici, and S. Woods. Program plan recognition for year 2000 tools. In *Proceedings 4th Working Conference on Reverse Engineering*. IEEE Computer Society, 1997. To appear.

[Feo97] M. Feord. Testing for millenium risk management. *IEEE Software*, pages 126–131, May/June 1997.

[Gar96] Gartner Group. *Year 2000 Date Crisis*, 1996. 18 page Conference Presentation.

[GL88] L. Goldschlager and A. Lister. *Computer Science, A Modern Introdution*. Prentice Hall, 1988.

[GSA96] GSA General Service Administration. Recommended year 2000 contract language, June 11 1996. URL: `http://www.itpolicy.gsa.gov:80/library/yr2000/y2kfnl.htm`.

[Har92] J. Hartman. Technical introduction to the first workshop on ai and automated program understanding. 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/`.

[Har95] J. M. Hart. Experience with logical code analysis in software maintenance. *Software – Practice and Experience*, 25(11):1243–1262, nov 1995.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, 1977.

[HP96] 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`.

[HR92] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering ICSE-14*. IEEE, 1992.

[IBM96] IBM. The year 2000 and 2-digit dates; a guide for planning and implementation, 1996. URL: `http://www.software.ibm.com/year2000/`.

[IEE83] IEEE standard glossary of software engineering terminology, 1983. ANSI/IEEE Standard 729-1983, IEEE Computer Society.

[Jon97] 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.

[Keo97] J. Keogh. *Solving the Year 2000 Problem*. Academic Press, 1997.

[KNE92] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, 1992.

[MM96] J. T. Murray and M. J. Murray. *The Year 2000 Computing Crisis — A Millenium Date Conversion Plan*. McGraw-Hill, 1996.

[MNB+94] 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.

[Moo96] L. Moonen. Data flow analysis for reverse engineering. Master's thesis, Programming Research Group, Universtity of Amsterdam, 1996. Technical Report P9613; Extended abstract as Chapter 11 of A. van Deursen et al., editors, *Program Analysis for System Renovation; Resolver Release I*.

[Nor96] Minnesota Government Information Services North Star. Year 2000 compliance: Information resource performance standards; irm standard 14, version 1, May 15 1996. URL: `http://www.state.mn.us/ebranch/admin/ipo/hb/document/std14-1.html`.

[Per95] W. E. Perry. *Effective Methods for Software Testing*. John Wiley & Sons, 1995.

[Rag97] B. Ragland. *The Year 2000 Problem Solver: A Five Step Disaster Prevention Plan*. McGraw-Hill, 1997.

[RBDL97] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of ESEC/FSE'97*, LNCS. Springer-Verlag, 1997.

[RW90] C. Rich and R. Waters. *The Programmer's Apprentice*. Frontier Series. ACM Press, Addison-Wesley, 1990.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[UH97] W. Ulrich and I. Hayes. *The Year 2000 Software Crisis — Challenge of the Century*. IEEE Computer Society Press, 1997.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[Wig97] T. Wiggerts. Using clustering algorithms in lecagy systems remodularization. In A. van Deursen, P. Klint, and G. Wijers, editors, *Program Analysis for System Renovation – Resolver Release I*, chapter 12. CWI, Amsterdam, 1997. Extended abstract to appear in proceedings of the *4th Working Conference on Reverse Engineering*.