Native Patterns

Alex Sellink and Chris Verhoef
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
`alex@wins.uva.nl, x@wins.uva.nl`

Abstract

We generate a native pattern language from a context-free grammar. So if we have the underlying grammar of code that needs to be analyzed, or renovated the pattern language comes for free. We use native patterns for recognition and renovation of code. The pattern language is global in the sense that patterns can match entire programs. We illustrate native patterns by discussing a tool that remediates a notoriously difficult Year 2000 problem using native patterns.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.3.4. [**Processors**]: Parsing.

Additional Key Words and Phrases: Reengineering, System renovation, Patterns, Plans, Clichés, Native pattern language, Year 2000 problem, Leap year problem.

1 Introduction

Recognition of problematic code is a prerequisite to reengineering it. Therefore, much research has been carried out in the area of pattern recognition. See [25] for an overview. It is well-known that real world problems do not have the habit of residing in one location. This makes recognition using lexical technology not always an optimal methodology, but see [20] for generation of source model extractors from lexical specifications. A powerful approach towards recognizing real world problem patterns is known as program plan matching. A program plan is a set of components and a set of constraints on them. Each component represents a part of the problem to be recognized and each constraint describes the interrelations between the components. This approach is suitable for locating a problem without generating too many so-called false positives due to the constraints. Once the problems have been identified the next step is to solve them. Sometimes this can be done by hand, but for massive modifications like the Year 2000 problem or the Euro conversion problem, a manual approach is not feasible.

When massive amounts of code need to be processed, a factory approach is particularly suited [10]. Nevertheless, *someone* has to work in such a factory. The work consists of constructing patterns, constructing replacements, and carrying out transformations. The qualifications that many potential factory workers have, are that they know the languages in which the code has been written and that they can think of solutions for specific problems. For example, a COBOL programmer is perfectly capable of solving an individual Year 2000 problem when confronted with problematic code (possibly after a short introductory course in the problem area). In our opinion, the set-up of a COBOL renovation factory should be such that a COBOL programmer is able to work in it. Since the used programming language is a substantial factor for productivity [12], it is crucial to design the pattern language that should be used in this factory as carefully as possible.

As we all know, the Year 2000 problem comes in numerous languages—many with a myriad of dialects. One approach towards solving the Year 2000 problem is to translate all those dialects to an intermediate format and to define a pattern language on this format. The factory workers then have to learn the newly invented intermediate format and the pattern language that comes with it. This situation does not comply with the qualifications that many potential workers have. We propose in this paper an alternative solution: *native* patterns. Native means that for a COBOL renovation factory the patterns resemble normal COBOL programs. A normal COBOL program, for example, is also a pattern that can be used in the factory without a single change. This implies that the workers in our factory can grasp the pattern language very quickly. Moreover, there is no intermediate format that has to be learned. Of course, it is not feasible to design a native pattern language for every project—we stated in the beginning of this paragraph that there are so many dialects.

Our solution is that we generate the native pattern language from the grammar of the code that has to be reengineered. Of course, first one should obtain such a grammar. This is not a task for factory workers but for developers of software renovation factories. We refer to [4] for a method to build a grammar from legacy code. We refer to [2] for comparisons with other methods and an overview of the state of the art in parsing technology used in reengineering practice. Needless to say that a grammar is always necessary in order to parse the code for later processing. So our restriction that a grammar is necessary does not impose extra work.

Related work Native patterns have already been used in practice. We refer to [5] where an assembly line that normalizes the control flow of COBOL/CICS legacy systems is discussed. The transformations use native patterns. The native pattern language for the combined COBOL/CICS grammar was generated with our tools. In [18] intuitions are provided about patterns that are formalized in this paper.

In [18] REFINE [23] is used, while we use the ASF+SDF Meta-Environment [15]. Those systems are related.

Organization In Section 2 we describe the idea of native pattern languages. We give a formal definition of a pattern and a native pattern language. Moreover, we compare our definition of patterns with the literature. Next, we discuss in Section 3 the generation process for native pattern languages. We will also address the issue of generating documentation for the native pattern language and its usage with the aid of a generated text and structure editor that understands the native pattern language. In Section 4 we will discuss a difficult Year 2000 problem in order to illustrate how we use native patterns.

Acknowledgements We thank Johanna Persson (Reengineering Center, Ericsson Software Technology AB) for useful comments.

2 Native Pattern languages

In this section we explain our ideas behind patterns and we give a formal definition the notion of a native pattern language. Furthermore, we make a comparison between our native patterns and plans.

2.1 The idea behind Native Patterns

As we all know from elementary universal algebra [19], it is simple to inductively define (many-sorted) terms over a given signature. Suppose we have a signature Σ containing function symbols with their arity and constant symbols. Terms over this signature are defined as usual: we assume a (infinitely countable) set of variables X with typical elements $x, y, z \dots$. A constant symbol is a term, a variable is a term, if $t_1, \dots, t_n$ are terms and f is an n-ary function symbol, then $f(t_1, \dots, t_n)$ is a term. We give an example. We define a function symbol f with three arguments: $f : B \times S \times S \to S$. for terms b of sort B, and s, t of sort S, we can now define the term $f(b, s, t)$.

Now we define this function symbol again in a slightly different fashion. We depart the standard prefix notation and impose no restrictions on the form of the function symbols at all. Moreover, we start using descriptive names for the sorts. We define a function symbol with three arguments as `IF Boolean THEN Statement ELSE Statement END-IF -> Statement`. The name of the function symbol `IF _ THEN _ ELSE _ END-IF` is interspersed with its domain that would be `Boolean` × `Statement` × `Statement` in the standard notation. The form that we used here is known as distributed fix operators, distfix operators, or mixfix operators. These names are due to Peter Mosses and Joseph Goguen [9].

Terms over this signature are constructed as usual, so there exist variables `Boolean1` of sort `Boolean` and `Statement1` of sort `Statement` and we can build the term `IF Boolean1 THEN Statement1 ELSE Statement1 END-IF`. This term contains two variables; it represents all the COBOL `IF` statements that have the *same* statement in their conditional bodies. A closed term over this signature is a term without variables. For instance `IF X < 1 THEN CONTINUE ELSE MOVE X TO Y END-IF` is a closed term. The expression `X < 1` contains a variable, if we consider it a COBOL program. If we consider it to be a native pattern, it is a *closed* term of type `Boolean`. The statements `CONTINUE` and `MOVE X TO Y` are closed terms of type `Statement`. In this way, we build the closed terms over our distfix signature. We stress that every real COBOL code fragment is in fact a closed term over the above signature. The syntactic freedom gave us the possibility of representing a term as a COBOL statement containing variables. For instance, the term `Program1` is a variable of type `Program` and matches any complete COBOL program. The term `Data-division1` is a variable of type `Data-division` and matches an arbitrary COBOL `DATA DIVISION`. We give a formal definition of a native pattern language.

Definition 2.1. *The native pattern language $P(L)$ over a give language L is the set of (many-sorted) terms that can be inductively defined in the standard way [19] over the subsequent signature and variables. The signature of the native pattern language consists of the distfix function symbols that define the language L. For each sort name S used in the distfix function symbols, we define three types of variables: variables of sort S, variables of sort $S+$, and variables of sort $S*$. The notation $S+$ stands for a list of one or more variables of type S; $S*$ is similar but then a list containing zero or more elements.*

In [26] it is stated that 'The concept of programming clichés is a pre-theoretic notion with no precise formalization.' From [18] we quote 'A pattern is an abstract description of code that can be thought of as a "program template" that contains variables.' We believe that the above definition is a sound formalization of the notions that are described more informal in other papers. In [18], it is also stated that a transformation is a process of rewriting one program into another program by application of a set of transformation rules possibly containing constraints. Although the authors of [18] have the intention to formalize this, a definition of a pattern lacks, a definition of the form of the constraints lacks, and a definition of an object in a program is not present. Apart from this, we believe that their intuitions have a strong connection with our definition. For instance, their *named single value* and *named multi-value* variables are analogous to our three types of variables. In fact, by our formalization of a pattern, we enter the situation that we can formalize transformations as well. In our opinion, a transformation is a conditional term rewriting system [16, 13]. This notion stems from universal algebra [19] and from the theory of abstract data types, as implementations of specifications with positive conditions. In our situation such systems may contain negative conditions, as well. It is not straightforward to give such systems a sound semantics [14, 8]. Such formal treatments do have implications for the every day practice in reengineering: in [8] a theorem is proved stating that *combining* transformations does not influence the behavior of the individual transformations if certain easy-to-check syntactic criteria are satisfied for the individual transformations. This theorem implies that a tool factory can be constructed from components. Therefore, in [8], a formal definition of a software renovation factory can be found.

We stress that despite of all the theory, we use native pattern languages to solve real-world problems. From the definition, it becomes clear why we call this a native pattern language: real COBOL programs are closed terms in the native pattern language over COBOL. They are the most precise patterns that exist: they only match themselves and nothing more. Open terms are extensions of real COBOL. Open terms can express constraints, control flow, and context of a program. Next, we make some abstract comparisons to explain this.

2.2 Native patterns versus plans

Let us compare our native patterns with plans. In the introduction, we already indicated that a plan is a set of components together with a set of constraints on those components. In order to specify plans for Year 2000 recognition, in [7] some special syntax has been developed. This syntax should be transformed into a Lisp representation that can be fed to a plan recognizer. The source code will be parsed, resulting in an AST. The AST is annotated with control flow and data flow information and is fed to the recognizer as well. Then the engine will output recognized plan instances.

The control flow information gives an idea of the order in which the code is executed. Constraints in a program plan use this information to indicate such dependencies. The control flow in a native pattern is defined by the term itself: we write down the source code, possibly using variables, so the order is relevant. The data flow part is taken care of by variables that we use. Compare this to $f(x, x)$: it is clear from using variable x twice that there is a data dependency described. Using the plan based approach this is expressed using a constraint. Something that is not present with plans is the context information of a pattern. We can express this information using a native pattern. This is crucial information if we also wish to change code automatically. In Section 4 we treat an example that we found in [7] to make our comparisons more concrete.

The advantage of plans is that in principle the ordering of the components is not important. So its possible to detect many variations of the same idea. The disadvantage is that false positives can distort the analysis. This problem is not too severe: the false positives can be filtered by a knowledge expert. When dealing with the desire to change code automatically, it is rather problematic when a pattern matches where it should not. Therefore, it is a good property for native patterns that the order is important in principle. We see both approaches as complementary: using, for instance, the plan based approach of [7] it is possible to make an inventory of the problematic Year 2000 code fragments, and using native patterns for transformations they can be matched exactly, implemented conveniently by factory workers, and automatically transformed into Year 2000 compliant code.

In the next section we explain how to obtain a native pattern language for a language.

3 Generating Native Patterns

In this section we discuss an implementation of the generation process of a native pattern language as defined in definition 2.1. With this implementation we can generate a native pattern language for an arbitrary context-free grammar. The input format of our tool is SDF. SDF stands for Syntax Definition Formalism and stems from [11]. Since it is possible to define variables in SDF the native pattern language is also in SDF format. We use a COBOL grammar in SDF as a running example, but we stress that the process we describe applies to an arbitrary context-free grammar. This grammar has been discussed in [4].

A common way of defining a grammar is to use the Backus-Naur Formalism. Many tools use this formalism as input format. If we, for instance, wish to define the COBOL `IF` statement we write `<Statement> ::= IF <Boolean> THEN <Statement> ELSE <Statement> END-IF`. Let us define the above BNF rule in SDF notation: `"IF" Boolean "THEN" Statement "ELSE" Statement "END-IF" -> Statement`. Note that this is similar to the distributed fix operator that we discussed earlier. Such constructions are also called context-free functions. From the viewpoint to see a program as a term, like in definition 2.1, it seems more natural to define the context-free functions. Defining grammars in SDF implies that SDF immediately provides the closed terms over the signature. So we only miss the variables of definition 2.1 to complete the native pattern language. Since SDF supports the definition of variables, it is not hard to generate the pattern language. Let us first explain SDF in more detail.

3.1 The Syntax Definition Formalism

An SDF module consists of a number of sections. There are three types of sections that may occur (more than once) in an SDF module: an `imports` section, an `exports` section and a `hiddens` section. An `imports` section contains names of other SDF modules, an `exports` section contains syntax information that can be exported to other SDF modules, and a `hiddens` is the same as an `exports` section except that the syntax is local to the module. The `exports` and `hiddens` sections may contain the following paragraphs. There can be `sorts` paragraphs, a `lexical syntax` paragraphs, a `context-free syntax` paragraphs, a `priorities` paragraphs, and a `variables` paragraphs. A `sorts` paragraph contains sort declarations, lexical syntax can be defined in `lexical syntax` paragraphs, and context-free syntax is located in `context-free syntax` paragraphs. In a `priorities` paragraph we can take care of precedence of operators. In `variables` paragraphs we can define any variable we need later on over the sorts that have been declared in the `sorts` paragraph. We depict a schematic example of an SDF module:

```
imports ...
exports
  sorts ...
  lexical syntax ...
  context-free syntax ...
  priorities ...
  variables ...
```

```
hiddens
  sorts ...
  lexical syntax ...
  context-free syntax ...
  priorities ...
  variables ...
```

The imports section enables the modular construction of grammars. In [2] it is argued that this is an important feature for grammar construction in the area of reengineering. SDF allows us to define syntax and variables in a very powerful way so that we can construct terms. We refer to [11] for more details on SDF. Important for now is that we can construct a native pattern language using this formalism by adding the variables of definition 2.1. This step can be automated.

3.2 The Generation Process

We explain how we generate the native pattern language. In fact, a grammar is itself a term. In our case it is an SDF term: it is a term over the signature that defines SDF. We generate the variables for a grammar with the aid of a transformation on SDF terms. The generation of the native pattern language is not difficult: we implement a transformation on SDF terms that generates for each declared sort three types of variables in a separate exports section containing a single variables paragraph. Let us give an example to make the idea clear. Suppose we have an SDF module that defines the context-free function defining the COBOL conditional (in fact this definition is in reality not so easy, but for the explanation of the generation process this complexity would distract. See [3] for a realistic definition of the COBOL IF in SDF):

```
exports
  sorts Boolean Statement
  context-free syntax
    "IF" Boolean "THEN" Statement "ELSE"
         Statement "END-IF" -> Statement
```

In accordance with definition 2.1 we need to generate the following module:

```
exports
  sorts Boolean Statement
  context-free syntax
    "IF" Boolean "THEN" Statement "ELSE"
         Statement "END-IF" -> Statement
exports
  variables
    Boolean[0-9]+           -> Boolean
    Boolean[0-9]+    "+" -> Boolean+
    Boolean[0-9]+    "*" -> Boolean*
    Statement[0-9]+      -> Statement
    Statement[0-9]+ "+" -> Statement+
    Statement[0-9]+ "*" -> Statement*
```

First, we discuss the notation of the variables. We chosed to give variables their sort name followed by a natural number. We believe that in this way the patterns resemble real code as much as possible, provided that in the grammar descriptive names are chosed for the sorts. We stress that it is possible to give other names as well. This is just our choice. The notation [0-9]+ stands for one or more digits. So Boolean1 is a variable of type Boolean. We can postfix a sort name with a + or a *. This is a built-in list matching feature of the support environment that understands SDF. A variable Statement1+ is of type Statement+. The variable represents a list of one or more COBOL statements. Similarly Statement1* is a list representing zero or more COBOL statements. This is in compliance with definition 2.1.

The above example modules are closed terms over the SDF signature. The generation process is a transformation that turns the first SDF module into the second SDF module. In order to make this transformation, we actually need a native pattern language over SDF to construct it. Actually we need the tool to create the tool. What we do is, we create a restricted native pattern language by hand: we only incorporate the variables that we really need in order to construct the transformation. Then we can use this transformation to create the unrestricted native pattern language for SDF so that we can make other tools as well. See [24] where we employ this strategy to develop an environment for developing tool factories. Next, we define an SDF transformation using the restricted native pattern language and this results in the desired generation process for native pattern languages.

We split the SDF transformation into two parts: first we collect the sort names, and then we add the variables. We do this since we need the separate functionality also in other circumstances (for instance, for tools to support the development of grammars). We implemented these transformations in an environment for generating programming environments. It is called the ASF+SDF Meta-Environment, see [15]. ASF and SDF are the supporting formalisms. ASF stands for Algebraic Specification Formalism (ASF) [1] and it can be used to describe semantic issues. As we've seen SDF is used to describe syntax. The syntax of our transformations has been defined in SDF and their semantics in ASF. The ASF+SDF Meta-Environment combines an SDF module and an ASF module in an ASF+SDF module. So syntax (an SDF module) can be accompanied with its semantics (an ASF module using the syntax of the SDF module). For more details we refer to [15]. We display the syntax of the GIVE-SORTS module. Note that this is an SDF term.

```
imports SDF
exports
context-free syntax
 "GS_SDF-Module"    "(" SDF-Module    ")" -> SDF-NonTerminal*
 "GS_SDF-Section"   "(" SDF-Section   ")" -> SDF-NonTerminal*
 "GS_SDF-Paragraph" "(" SDF-Paragraph ")" -> SDF-NonTerminal*
variables
 Module-name[0-9] "+"     -> Module-name+
 SDF-NonTerminal[0-9] "+" -> SDF-NonTerminal+
```

Note that this module imports a module called SDF. This module describes SDF formalism in SDF. It contains a grammar that describes the syntax of SDF itself. We extended this grammar with variables, but only those that we really need for defining the transformation that adds variables for every sort. In fact, this module is a restricted native pattern language over SDF. The sort names and the variables describing SDF constructions are declared in that module. The context-free syntax defines three functions: one with as argument a complete SDF-module, one with as argument an SDF-section and one with as argument an SDF-paragraph. `GS` is short for `Give-Sorts`, we used it just to fit two column format. We subscripted `GS` with the sort name of its argument.

The ASF formalism has a simple form, it is an implementation of positive/negative conditional term rewriting systems [14, 8]. There can only be `equations` sections. The form of the equations is also simple: a tag to name an equation also called (rewrite) rule followed by an equation. An equation consists of two terms over the signature that is defined in SDF and an equality sign in between them. In this way we describe the semantics of terms. For instance, the semantics of the above functions is in ASF:

```
equations
[1] GS_SDF-Module(SDF-Section1 SDF-Section1+) =
    GS_SDF-Section(SDF-Section1)
    GS_SDF-Module(SDF-Section1+)
[2] GS_SDF-Module(SDF-Section1) =
    GS_SDF-Section(SDF-Section1)
[default-3] GS_SDF-Section(SDF-Section1) =
[4] GS_SDF-Section(exports SDF-Paragraph1 SDF-Paragraph1+) =
    GS_SDF-Paragraph(SDF-Paragraph1)
    GS_SDF-Section(exports SDF-Paragraph1+)
[5] GS_SDF-Section(exports SDF-Paragraph1) =
    GS_SDF-Paragraph(SDF-Paragraph1)
[default-6] GS_SDF-Paragraph(SDF-Paragraph1) =
[7] GS_SDF-Paragraph(sorts SDF-NonTerminal1+) = SDF-NonTerminal1+
```

We explain the functionality of this ASF-module. For a given SDF module, we have to traverse its tree and at the paragraph level we need to return nothing unless it is a `sorts` paragraph. In that case we return a list of non terminals. Equations `[1]` and `[2]` are traversal cases: The first one treats the case that an SDF module consists of two or more SDF sections, and equation `[2]` describes the single section case. Since we are only interested in the `exports` section, there is no need to traverse the other sections. This is expressed in equation `[default-3]`. A default equation is a built-in abbreviation mechanism that saves us the time and space to specify all the cases minus a few. We just give an equation so-called default status by providing a tag name starting with the word `default`. The special behavior is taken care of in equations `[4]` and `[5]`: there we traverse from `exports` section level to the paragraph level. Since we are only interested in the `sorts` paragraph, we return default the empty list. This is expressed in equation `[default-6]`. In equation `[7]` we return the list of sort names. So `GS_SDF-Module` is a function that for any given SDF-module returns its set of sort names.

Now we give the syntax for the `ADD-VARS` module in SDF:

```
imports GIVE-SORTS
exports
 context-free syntax
  "AV" "(" SDF-Module                      ")" -> SDF-Module
  "AV" "(" SDF-Module "," SDF-NonTerminal* ")" -> SDF-Module
  "VD" "(" SDF-NonTerminal* ")"                -> SDF-Paragraph
 variables
  Var [0-9]+ "+" -> Var+
```

Note that we import the above discussed `GIVE-SORTS` module. The import implies that the functions and their semantics are known in this module, in particular, we may use SDF syntax, since that has been imported by the `GIVE-SORTS` module. There are two functions `AV`, short for `Add-vars`, one with a single SDF-module as argument, and one taking an additional argument: a list of zero or more identifiers (this kind of overloading is harmless in the ASF+SDF Meta-Environment). Moreover, there is a function `VD` (short for `Var-decls`) that turns a list of non terminals into a paragraph (containing the desired variables). We give the semantics of these functions that we implemented in an ASF-module below.

```
equations
[1] AV(SDF-Module1) =
    AV(SDF-Module1,GS_SDF-Module(SDF-Module1))
[2] AV(SDF-Section1+, ) = SDF-Section1+
[3] AV(SDF-Section1+, SDF-NonTerminal1+) =
    SDF-Section1+ exports VD(SDF-NonTerminal1+)
[4] VD(SDF-NonTerminal1) =
  variables
    SDF-NonTerminal1 [0-9]+     -> SDF-NonTerminal1
    SDF-NonTerminal1 [0-9]+ "+" -> SDF-NonTerminal1 +
    SDF-NonTerminal1 [0-9]+ "*" -> SDF-NonTerminal1 *
[5] VD( SDF-NonTerminal1+ ) = variables Var1+
    ========================================
    VD( SDF-NonTerminal1 SDF-NonTerminal1+ ) =
  variables
    SDF-NonTerminal1 [0-9]+     -> SDF-NonTerminal1
    SDF-NonTerminal1 [0-9]+ "+" -> SDF-NonTerminal1 +
    SDF-NonTerminal1 [0-9]+ "*" -> SDF-NonTerminal1 *
    Var1+
```

We discuss this implementation. Rule `[1]` distributes the SDF-module to the `GS_SDF-Module` function so that the result is the SDF-module as first argument and a list its sort names as a second argument. Rule `[2]` takes care of the case that there were no sorts declared: in that case the SDF-module itself is returned and we are ready. If one or more sort names are returned, we return in equation `[3]` the SDF-module extended with a new `exports` section that will contain a new `variables` paragraph. This has been taken care of in the last two equations. In case there is just one sort in the list, we create the desired variables. Equation `[5]` is a conditional equation. The rules above the double line are called conditions and the rule below the line, the conclusion, is said to fire when the conditions are satisfied. So, if there are one or more sort names in the variable `SDF-NonTerminal1+` and they constitute already a `variables` paragraph with list of variables `Var1+`, the rule below the line fires. This one makes the variables for the variable `SDF-NonTerminal1` and adds the list `Var1+` that we had already to the end. So the function `AV` with an SDF-module as its argument returns the same SDF-module extended with the desired variables. We can do this for one module, so we can do it for a complete grammar consisting of possibly more modules. After this automatic generation process the native pattern language is complete.

3.3 Generating Support

When factory workers have to operate the factory by developing patterns for analysis or transformations, all support is welcome. In this section we briefly mention two support options: documentation and editors. Both are generated from the grammar. First of all a grammar needs to be developed. We do this in a modular way, and we provide comments in the grammar that explain certain issues important to factory workers. Using technology discussed in [6] we automatically generate a LaTeX document that contains the entire native pattern language in type setted form. Typically, for COBOL this generates a 25 page document with a generated table of contents, sections representing top modules and subsections representing lower modules. Cross references are generated as well. The generated documentation is in compliance with the so-called book paradigm. In [21] it has been argued that the book paradigm improves productivity. So this is an important feature: we have a fully documented native pattern language at our disposal. Both documentation and the pattern language are generated. For more information on the generation of formatters that enables us to generate documentation we refer to [6]. In [2] it is argued that grammars for reengineering purposes are not stable, so generated documentation and native pattern languages is important, since after modifying a grammar we can generated up-to-date documentation and native patterns.

The ASF+SDF Meta-Environment supports a so-called generic text and structure editor [17]. From a context-free grammar a structured editor is generated that understands the grammar of the language. This is important for factory workers: they can implement patterns using the generated structured editor as a guide. This feature is also important when developing factories: the generic structured editor is incremental. So as soon as we change the grammar, the editor understands this structure. For more information on generic structured editors we refer to [17].

4 Using Native Patterns

We apply the use of a native pattern language for COBOL. Note that we used native patterns in [5] before. The example transformation that we will implement is known as a notoriously difficult Year 2000 problem and is reported on in [7]. We first discuss the problem, then the in and output pattern and the transformation. Then we address comments in a transformation and we mention an assembly line that has been implemented to automatically solve the Year 2000 problem.

4.1 The Problem

In this section we discuss a Year 2000 problem that looks Year 2000 compliant, since it uses four digits for the century. The problem resides in the incorrect leap year calculation. We depict the code containing the problem. We took this code from [7]:

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

Let us first explain the problem of this code: the leap year calculation is incorrect. The simple leap year algorithm of the past (every fourth year is a leap year) implied an error of about 0.01 day per year. So, over a period of 1500 years the calendar was no longer in sync with the seasons. Since the calendar was used to calculate the date of Easter, this was seen as an annoying error. Pope Gregorius XIII concluded that the calculation thusfar should be changed and he decided that the following algorithm should be applied in the future: if the a year is divisible by 4, it is a leap year, unless it is divisible by 100; in that case it is no leap year, unless it is divisible by 400; then it is a leap year after all. According to this algorithm, the year 2000 is a leap year. Since the so-called Gregorian calendar was introduced after 1600 in many countries, the year 2000 is the very first time that the third part of the algorithm applies. In [22] it is claimed that the leap year calculation is frequently missed, since it is relatively complex and the algorithm was not available during programming.

4.2 Recognition Pattern

In [7] it is stated "The ideal Y2K tool should identify this chunk of code as Y2K related (despite its using a four digit date), identify the pair of divisions and remainder tests as being an incorrect check for whether we have a leap year, and automatically transform that portion of the code to correctly test for leap years". We think that the ideal Y2K tool does not exist, but we will use our native pattern language to construct a tool that recognizes the above code, and changes it automatically to the corrected code that is also provided in [7]. In this section we start with the input pattern. For comparisons we recall a rule from [7] that describes the above code fragment:

```
IF Numeric-Variable(?V)
   Exists(Division(?V,4,?Q,?R1),?Div-1)
   Exists(Equality-Test(?R1,0),?Test-1)
   Data-Dependency(?Test-1,?Div-1,?R1)
   Exists(Division(?V,100,?Q,?R2),?Div-2)
   Exists(Inequality-Test(?R2,0),?Test-2)
   Data-Dependency(?Test-2,?Div-2,?R2)
   Same-Data(?Div1,?Div-2,?V)
THEN Is-Year(?V)
     Recognized(Invalid-Leap-Year-1)
```

We see here that it is checked that `V` is a numeric variable. It is tested that there are two `DIVIDE` calculations as in the code fragment and that there are (in)equality tests as in the code fragment. These are the components. The constraints are that there must be a data dependency between the various components: the variable `R-1` should be the same in the divisions and in the tests. Furthermore the divisions should use the same data. The context of the code fragments in the example are denoted by ellipses. In the above plan they are not present. This is indeed not too important when we only wish to detect this sort of code. However, we also wish to make a change automatically. When making automatic changes it is undesirable to detect false-positives. So this degree of freedom is not desirable when we both wish to recognize and change.

Let us provide a native pattern that recognizes this code fragment. There are a few possibilities to construct this pattern. Since we operate in a completely automated fashion, it is wise to pretreat code before we start with the main operation. We do this to reduce the number of cases we have to distinguish for main operations. We first add possibly missing scope terminators, e.g., `END-IF` or `END-DIVIDE`. This makes it possible to assume that future patterns contain these constructs. The leap year pattern can occur in one sentence, or in two, or in three, etc. We preprocess the code by removing possibly occurring separator periods in COBOL paragraphs except the last one. Separator periods are *implicit* scope terminators, so we can remove them since we added all the *explicit* scope terminators. Then we normalize boolean conditions, so `X NOT = 0`, `NOT (X = 0)`, and so on, are transformed into, say, `X <> 0`. In IBM COBOL dialects, a `PROCEDURE DIVISION` does not need to start with a `SECTION`. Moreover, the first paragraph does not need to be labeled. We take care of this aspect as well. We preprocess the code by adding a temporary first `SECTION` and/or a temporary label to the first paragraph if these were not present. During postprocessing we remove the possibly added first dummy `SECTION` and/or paragraph label. Furthermore, for this particular leap year problem we do some special preprocessing. Both the order of the `DIVIDE` statements and the order of the test `Data-name3 = 0 AND Data-name5 <> 0` are irrelevant. There are a few options to deal with this. First we can make more patterns. In this case the total is four, which is acceptable. If we have 5 possibilities this would become 32 patterns, which is not desirable anymore. Another way of dealing with this, is to write an auxiliary function that takes care of the ordering. An example could be to abstract in the `DIVIDE` statements from the numbers `4` and `100` and use variables. Then we can implement a condition for the transformation we wish to perform. We opt for yet another solution. We preprocess the code with two very simple transformations by choosing one particular ordering out of the four possibilities. Then we can assume in the leap year pattern that this ordering is always present. In Section 4.4 we will treat this preprocessing transformation. In [5] we elaborately discussed an assembly line using similar preprocessing phases for other purposes.

After preprocessing we assume that the code fragment containing the erroneous leap year calculation is contained in a single COBOL sentence, that boolean conditions are in normal form, that explicit scope terminators are present, that the code starts with a `SECTION`, and that the `DIVIDE` statements and the tests are in the order as in the pattern below. The following pattern now recognizes the incorrect leap year calculation:

```
DIVIDE Data-name1 BY   4 GIVING
  Data-name2 REMAINDER Data-name3
END-DIVIDE
DIVIDE Data-name1 BY 100 GIVING
  Data-name4 REMAINDER Data-name5
END-DIVIDE
Stat1*
IF Data-name3 = 0 AND Data-name5 <> 0
  Stat1+
END-IF
```

We will explain this pattern. First of all, this is an open term: it contains variables. We discuss their purpose. First we recognize a `DIVIDE BY 4` statement. Then follows a `DIVIDE BY 100` statement. We also demand that both divisions use the same `Data-name1`. So in the example this would match the `CONTRACT-SY`. The variables `Data-name1` – `Data-name5` are of type `Data-name`. A `Data-name` is a variable that makes a unique reference to a data item. This can be of the form `A IN B` or `A OF B` if `B` is a subrecord of `A`. Note that in the plan of [7] it is checked that the variable that we call `Data-name1` is numeric. Maybe this is not always correct since it can also be a `Data-name` like `A IN B`. More importantly, using a `Data-name` variable is correct. Then zero or more statements follow. This is expressed with the list variable `Stat1*`. In the code fragment this is expressed with an ellipse. Since we know that this code fragment is in a single sentence, we also know that the ellipse can only represent zero or more COBOL statements. Note that we have to check that the `Stat1*` statements are innocent with respect to the variables `Data-name3` and `Data-name5`. This has been implemented, see Section 4.6. Then the `IF` statement containing both tests on the remainders `Data-name3` and `Data-name5` follows. Note that the variables that are present in the divisions are the same as in the conditional. We see here that the use of variables in a natural way expresses the data dependencies that are present. In the plan-based approach this is expressed with data-dependencies as a constraint.

4.3 Replacement Pattern

Now that we have an input pattern, we wish to change the code. This will impose extensions of the input pattern since the change is global. So we revise our input pattern and provide its replacement pattern. The reason why the input pattern has to change can be found in the solution that is given in [7]:

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

Let us first comment upon this solution. First of all, this will not compile unless also the data items `Q-3` and `R-3` are declared. This implies that the change is global: in the PROCEDURE DIVISION we have to make changes to the code and in the DATA DIVISION we have to add fresh variables. This is the reason why the input pattern is affected: we need to recognize the DATA DIVISION, as well. Moreover, we have to test whether the variables Q-3 and R-3 are fresh. We do not focus on these issues here since we wish to illustrate the use of native pattern languages (but it has been implemented, see Section 4.6). Below, we provide a pattern that recognizes the erroneous code:

```
Ident-div1
Env-div1
DATA DIVISION.
File-sec1
WORKING-STORAGE SECTION.
Data-desc1*
Link-sec1
PROCEDURE DIVISION Using1.
Decl1*
Section1*
Lab1 SECTION.
Paragraph1*
Lab2.
  Stat1*
  DIVIDE Data-name1 BY   4 GIVING
    Data-name2 REMAINDER Data-name3
  END-DIVIDE
  DIVIDE Data-name1 BY 100 GIVING
    Data-name4 REMAINDER Data-name5
  END-DIVIDE
  Stat2*
  IF Data-name3 = 0 AND Data-name5 <> 0
    Stat1+
  END-IF
  Stat3*.
Paragraph2*
Section2*
```

The heart of the pattern is the same as before. We just added the necessary context so that we can insert all the new code as proposed in [7]. So we only need to discuss the new parts. Ident-div1 is a variable that matches a complete IDENTIFICATION DIVISION. Similarly, Env-div1 matches an ENVIRONMENT DIVISION. Note that the latter division is optional. This is not a problem, since an empty ENVIRONMENT DIVISION is also matched by Env-div1. Since we have to change the DATA DIVISION, we give as much detail as necessary in the pattern to make the change. We can expect three different sections in this division. Only in the second section we wish to make changes. Therefore, the first section and the last are matched by variables: File-sec1 matches an entire FILE SECTION if it is present, and Link-sec1 matches an optional LINKAGE SECTION. The second section, the WORKING-STORAGE SECTION, contains the data descriptors. They are matched by the variable Data-desc1*. This variable stands for zero or more data descriptors. We wish to add two data items to this list in the output pattern, so therefore we need this variable. Then we turn to the PROCEDURE DIVISION. This is the most detailed part of the pattern, since we have to make a few changes in it: on one location we need to add a DIVIDE statement and on another location we extend a Boolean test. The Using1 variable matches an optional USING phrase. Decl1* matches the optional DECLARATIVES part. The code in the DECLARATIVES part, is executed only in case certain errors in the procedure part occur. This way of preventing abnormal termination of a program is comparable with the try/catch/finalize mechanism in Java. So, it is unlikely that the pattern does occur in the DECLARATIVES part. Since we preprocessed the code we are sure that the rest of the program consists of a list of COBOL sections. In one such section we will find the incorrect leap year pattern. To express this we use the variables Section1* and Section2*. Section1* matches zero or more sections, then the leap year section, and then Section2* zero or more sections. Let us focus on the section that contains the leap year pattern. Lab1 matches the name of the SECTION. On this level we repeat the trick: a section consists of a number of paragraphs and in one such paragraph we can find the leap year calculation. Therefore, we use the variables Paragraph1* and Paragraph2*. Let us focus on the 'middle' paragraph. A paragraph starts with a label Lab2 and always contains one sentence. A sentence is zero or more statements, and some of them represent the leap year calculation. Therefore we have the variables Stat1*, Stat2*, and Stat3*. Note that these statements represent ellipses in the original code fragment that we took from [7]. The rest of the pattern was already explained before.

Now let us present the output pattern:

```
Ident-div1
Env-div1
DATA DIVISION.
File-sec1
WORKING-STORAGE SECTION.
Data-desc1*
  01 LEAP-YEAR-CORRECTION.
     03 Q-3 PIC S9(9) COMP VALUE +0.
     03 R-3 PIC S9(4) COMP VALUE +0.
Link-sec1
PROCEDURE DIVISION Using1.
Decl1*
Section1*
Lab1 SECTION.
Paragraph1*
Lab2.
  Stat1*
  DIVIDE Data-name1 BY   4 GIVING
    Data-name2 REMAINDER Data-name3
  END-DIVIDE
  DIVIDE Data-name1 BY 100 GIVING
    Data-name4 REMAINDER Data-name5
  END-DIVIDE
  DIVIDE Data-name1 BY 100 GIVING
    Q-3 REMAINDER R-3
  END-DIVIDE
  Stat2*
  IF (Data-name3 = 0 AND
      Data-name5 <> 0) OR R-3 = 0
    Stat1+
  END-IF
  Stat3*.
Paragraph2*
Section2*
```

As can be seen, we added in the DATA DIVISION a LEAP-YEAR-CORRECTION record containing the variables Q-3 and R-3. In the PROCEDURE DIVISION we added the code that our customers van Deursen, Woods, and Quilici asked for: the extra DIVIDE statement and the extra test R-3 = 0 in the IF. Note that the matching takes care of the fact that in the DIVIDE statement, the constant CONTRACT-SY will be substituted for the variable Data-name1.

4.4 The Transformation

Now that we have seen the input and output patterns, let us explain how to make the actual transformation. Before we do this let us first illustrate the preprocessing transformation that takes care of consistent ordering of the DIVIDE statements and the conditionals. We call this special purpose tool Leap-year-order. Its syntax is as follows in SDF:

```
imports Tool-basis
exports
  context-free syntax
    "Leap-year-order"  -> TRANSFORM
```

Let us explain the nonobvious parts of this module. First there is the import of the module Tool-basis. If we would draw the import graph of this module, it would result in a graph containing over 120 nodes (see [2] for the import graph). The lower parts of this graph contain the complete definition of the native pattern language. From this pattern language we also generated two sets of modules containing generated generic transformations and analysis functions (see [3] for details on this generative technology). We hide all these modules for the factory worker. It is only necessary to import the Tool-basis in order to have access to the native pattern language and a framework that deals with the tedious parts of transformations. Examples of tedious parts can be found in Section 3: we specified traversal manually. Such functionality has been generated for factory workers, see [3] for an elaborate discussion. The simple statement that Leap-year-order is of type TRANSFORM gives access to the generated functionality. It enables the factory worker to use Leap-year-order for each sort that is available in (the documentation of) the native pattern language. There is only need for defining nondefault behavior of the tool we called Leap-year-order. When we match the two DIVIDE statements in the wrong order, it switches the order. If we match a Boolean Data-name1 <> 0 AND Data-name2 = 0, we switch it. So for the sorts Boolean and Stat* we specify the nondefault behavior of Leap-year-order. The rest is taken care of by the generated functionality.

```
equations
[1] Leap-year-order_Boolean(
    Data-name1 <> 0 AND Data-name2 = 0)=
    Data-name2 = 0 AND Data-name1 <> 0
[2] Leap-year-order_Stat*(
    DIVIDE Data-name1 BY 100 GIVING
      Data-name2 REMAINDER Data-name3
    END-DIVIDE
    DIVIDE Data-name1 BY   4 GIVING
      Data-name4 REMAINDER Data-name5
    END-DIVIDE) =
    DIVIDE Data-name1 BY   4 GIVING
      Data-name4 REMAINDER Data-name5
    END-DIVIDE
    DIVIDE Data-name1 BY 100 GIVING
      Data-name2 REMAINDER Data-name3
    END-DIVIDE
```

The notation that we generated in [3] is that the name of a function can be subscripted with the sort name. This can be changed without effort, it was just our choice. So Leap-year-order_Boolean is the function on the Boolean level and Leap-year-order_Stat is Leap-year-order on the statement level.

Now we construct the Leap-year-corrector transformation. First we give the syntax of this tool in SDF.

```
imports Tool-basis
exports
  context-free syntax
    "Leap-year-corrector"  -> TRANSFORM
```

The only difference with the above tool is that we give it a different name. Let us take a look at the semantics of this tool. This is described in the following ASF module:

```
equations
[1] Leap-year-corrector_Program(
      <input pattern>) = <output pattern>
```

For reasons of space we did not reiterate the input and output pattern we discussed in Section 4.3. We used placeholders <input pattern> and <output pattern> instead. Since the patterns cover a complete program, we subscripted Leap-year-corrector with the sort Program.

4.5 Comments and Transformations

In real-world COBOL programs there can be a lot of comment. When correcting a leap year calculation, it is undesirable to remove comments. So we treat comment as part of the native pattern language: comments have a sort of their own. We have experienced that comments can be inserted on virtually every location in a program. Therefore, after every terminal, zero or more comments may occur. When developing a huge grammar it is a difficult and tedious job to handle comments in the grammar consistently. So, we made a grammar development tool that generates comment variables on the appropriate locations. Below, we present the earlier displayed SDF-module for the COBOL IF statement for which comments were generated.

```
exports
 sorts Boolean Statement
 context-free syntax
  "IF" Comment* Boolean "THEN" Comment* Statements "ELSE"
       Comment* Statement "END-IF" Comment* -> Statement
```

The input and output patterns that we discussed earlier have to be extended with comment variables in order to match real world code. Since comment possibilities are present in the grammar, they have been included in many variables already. For instance, if we match an arbitrary statement, it can be seen from the above grammar fragment that comments are taken care of. So, it is only necessary to explicitly insert comment variables after terminals that are present in the pattern. Although our grammar is able to recognize comments everywhere, it is not necessary to insert comment variables everywhere. We display a typical fragment of our input pattern:

```
Lab2. Comment1*
  Stat1*
  DIVIDE Data-name1 BY   4 GIVING
    Data-name2 REMAINDER Data-name3
  END-DIVIDE Comment2*
```

Note that our pattern does not match if comments were inserted in the original code directly after DIVIDE, BY, GIVING, and REMAINDER. We refrained from inserting comments since we never experienced these cases. After a paragraph label we often see comments and sometimes after statements. Hence the two comment variables Comment1* and Comment2*. We recall that in Stat1* the possible comments are taken into account so there is no need to specify it separately. After possible comment insertion the transformation is complete.

4.6 An Assembly Line

A PhD student, Eggie van Buiten, has implemented an assembly line that takes care of all the preprocessing, the leap year correction, and the necessary postprocessing to solve the leap year problem that we discussed in this paper. He used the transformations described here, and he could reuse all the pre and postprocessing tools described in [5]. The assembly line consists of: addition of scope terminators, removal of unnecessary separator periods, normalization of conditional expressions, addition of dummy sections and/or paragraphs, the Leap-year-order transformation, the Leap-Lear-corrector transformation, and removal of a possibly created dummy section and/or paragraph label. Furthermore, a fresh variable adder is used for insertion of variables, and a check is made if the statements in the context (Stat2* in the input pattern of Section 4.3) are innocent.

5 Conclusions

In this paper we explained a method to generate a native pattern language from a context-free grammar. We implemented this idea and showed with an elaborate example its usage in solving real-world problems. We think that the presence of native pattern languages enables the use of software renovation factories by end-users, since they are presumably knowledge-experts in the problem area rather than tool experts.

References

[1] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[2] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. Technical Report P9719, University of Amsterdam, Programming Research Group, 1997. To be presented at the 6th Reengineering Forum. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[3] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *proceedings of the fourth working conference on reverse engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[4] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer verlag, 1997. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html.

[5] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In *proceedings of the second euromicro conference on maintenance and reengineering*, 1998. To appear. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[6] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[7] A. van Deursen, S. Woods, and A. Quilici. Program plan recognition for year 2000 tools. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings 4th Working Conference on Reverse Engineering*, pages 124–133, 1997.

[8] W.J. Fokkink and C. Verhoef. Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories. Technical Report P9802, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/~x/cade/cade.ps.

[9] J. Goguen. Personal Communication, January 1993.

[10] B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*, page 14. Gartner Group, 1996.

[11] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[12] C. Jones. *Programming Productivity*. McGraw-Hill, 1986.

[13] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175–193, 1984.

[14] S. Kaplan. Positive/negative conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 129–143. Springer-Verlag, 1988.

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

[16] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume II*, pages 1–116. Oxford University Press, 1992.

[17] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Diets, editor, *Computing Science in the Netherlands (CSN'92)*, SION, pages 168–177, 1992.

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

[19] K. Meinke and J.V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science, Volume I*, pages 189–411. Oxford University Press, 1993.

[20] G.C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[21] P.W. Oman and C.R. Cook. The book paradigm for improved maintenance. *IEEE Software*, 7(1):39–45, 1990.

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

[23] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

[24] M.P.A. Sellink and C. Verhoef. A development environment for a tool factory. Technical Report P980?, University of Amsterdam, Programming Research Group, 1998.

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

[26] L.M. Wills. Using attributed flow graph parsing to recognize cliches in programs. In *Proceedings of the International Workshop on Graph Grammars and Their Application to Computer Science*, pages 101–106, 1994.