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

One key task in reengineering software is the recognition of problematic code. For example, for Year 2000 related remediation, it is necessary to accurately identify exposures to faulty Year 2000 reasoning in large software systems. The need to recognize such code has resulted in an ongoing interest in software pattern recognition in both the academic [54, 55, 57] and industrial reengineering communities. See [23, 31] for an overview of the important players in the Year 2000 arena, for instance, TechForce or Reasoning. 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 [39] 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 [23, 31]. A factory is a set of tools that is operated by a vendor (who usually owns the factory). The vendor's employees operate the technology, either by setting up the factory on-site or at a central facility [23, 31]. Another characterization of the factory approach is given in [28, loc. cit. p. 608]: the software factory concept envisions software being produced like a manufactured product more or less following the assembly line technique. Obviously, *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 [27], 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 [8] for a method to build a grammar from legacy code. 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 for generating a pattern language does not impose extra work.

Applications of our Results Although we make an effort of illustrating that native patterns are useful, it is difficult to prove this in a single paper. Of course, we illustrate the use of native patterns in Section 5. The technology is also used outside this paper. We provide a short list so that the reader obtains an idea of the application range of native patterns.

- In [9] native patterns are used to implement an assembly line that performs control flow normalization of COBOL/CICS legacy systems.
- In [50] native patterns are used to implement an assembly line that performs several maintenance transformations in order to restructure COBOL/SQL systems.
- In [51] native patterns are used to implement computer aided language engineering tools supporting the rapid development of grammars needed for reengineering.
- In [15] native patterns are used to implement cluster analysis technology to detect classes in legacy code.
- In [16] native patterns are used to implement type inference for COBOL systems. Rewrite rules using COBOL concrete syntax extract type information about variables occurring in a COBOL system. This information is then used to infer type equivalences, subtype relations, literal elements contained in types, and enumeration types. For more information we refer to [16].

Related Work In many systems native facilities are used. In all lexical tools for instance. In program understanding and/or transformation tools, like TXL [13], Draco [40, 41] and REFINE [47], native facilities are used to ease implementation of the patterns. We compare these systems more elaborately in Section 3.1. We use the ASF+SDF Meta-Environment to implement the generation of native pattern languages and to use native patterns. REFINE and the ASF+SDF Meta-Environment are related: they both stem from the early eighties, they are both implemented in Lisp, etc. In [14] an elaborate comparison (41 pages) is made between the ASF+SDF Meta-Environment and REFINE. A full suite of tools for a toy-language called CALC is implemented in both REFINE and the ASF+SDF Meta-Environment. One of the remarks in this report is that a closer cooperation between Reasoning and the ASF+SDF group may be well beneficial for both parties, since there are many areas of mutual interest. For another comparison containing historical remarks on both systems and many other related systems we refer to [6]. In [6] some differences and similarities are summarized that were provided by experienced users of both systems.

Organization In Section 2 we discuss the rationale behind native patterns. In Section 3 we discuss native facilities versus native patterns, we compare native patterns with plans, and we characterize native pattern languages. We briefly discuss complexity issues, as well. Next, we illustrate in Section 4 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 5 we will discuss a difficult Year 2000 problem in order to illustrate the use of native patterns in the case of COBOL.

Acknowledgements We thank Johanna Persson (Ericsson Reengineering Center) for useful comments. We thank Steven Woods (SEI) for many suggestions for improvements of the explanations. We thank Jan Boef (ABN AMRO) and Han van der Loo (Roccade Finance) for their permission to disclose Chapter 9 of the SOS Resolver Release I deliverable [14]. We thank Arie van Deursen (CWI) for comments on a previous version and Arie van Deursen and Jan Heering (CWI) for discussions on the complexity of pattern matching in the ASF+SDF Meta-Environment. We thank the reviewers for their useful and elaborate comments and questions (800+ lines). We thank Mike Blaha and Alex Quilici for allowing us some extra space to address those comments and questions.

2 Rationale Behind Native Patterns

Reengineering is an area where technology exchange with the IT industry is crucial. Tools and methods that are developed in academics should, ideally, be adopted and used in the IT industry. For, in IT industry the real reengineering problems reside. The real problems that are known in the IT industry should become available to academics so they can help in finding solutions. This will only happen when IT industry benefits from it.

Sometimes, academic viewpoints on reengineering are in our opinion in direct conflict with the demands of the IT industry when it comes to tool support. We give two examples. As a first example we mention that researchers often advocate solutions that are based on moving code to other representations, whereas the IT industry experiences this as damage to their code and investment. We think that the fact that *"ordinary programming languages are just not designed to be easy to transform"* can not be solved by moving to another representation. Difficulties cannot just vaporize. Consequently, we do for instance not believe in the wide-spectrum language approach. The shortcomings of the wide-spectrum language approach are also observed by Neigbors. In [41, loc. cit. p. 301] problems with the wide-spectrum transformational approach are discussed. Our viewpoint is orthogonal to wide spectrum ideas: for each language a corresponding native pattern language is generated.

Let us give another example. According to the research community reengineering research should be independent of the used language and be applicable to more than one isolated case. In the IT industry, however, software engineers are interested in very specific language dependent tools. For instance, let us think of tool support to reengineer switching system software. People working for DeTeWekommunikationssysteme (Berlin) use Chill, a telecom language. They want tool support for Chill. People working for Ericsson use PLEX [22], also a telecom language, and they want tools for PLEX. People working for Lucent use C and want tool support for C. People working for Mitel, use Mitel-Pascal, also a telecom language, and want tool support for Mitel-Pascal. People working for Alcatel probably need tool support for ESPL/1 (Electronic Switching Programming Language/1), a proprietary language that was used when Alcatel was ITT. Now Chill uses concurrency, PLEX uses signals, Mitel-Pascal uses compiler directives, C we all know, ESPL/1 we all don't know. Presumably, they all use (their own) embedded assembler. We do not believe that there is a uniform language that covers the abovementioned telecom languages. Moreover, we do not believe it to be feasible to construct one tool that is applicable to all these languages, even if the application domain is fixed. The languages are too far apart, the mind set of the people working with the languages is disjoint, the problems that these people have are so specific and detailed that it is not at all clear how tools should be developed at all. Let alone that one tool can be made to serve the purpose of DeTeWekommunikationssysteme, Ericsson, Lucent, Mitel, and Alcatel at the same time. Our idea is to take away the drawbacks of the language specific approach (by means of developping language parametrized tools) rather than choosing a uniform approach. The native pattern generator is an example of such a language parametrized tool.

The conflicting views on tool development discussed above illustrate that it is far from trivial to develop tools that are useful to others. This fact was also established by Capers Jones[1]: nearly all organizations with more than 1000 programmers have internal tool groups. Many have proprietary analysis and design tools that are superior to those on the market [27].

Jones also gives more detailed information about the experiences that companies have with tools they bought on the market. Organizations that have nonformal software tool acquisition methods waste about 50 percent of all their money they spend on tools [28, loc. cit. p. 493]. Fifty percent. A formal acquiring strategy can drop the wastage to about 10 percent. Some common problems that Jones identifies with the acquisition of tools are: the software tool market claims more than they can do; earlier bad tool experiences preclude interest in the acquisition of more beneficial tools; 30 percent of the acquired tools do not meet enough user needs to be effective; 10 percent is shelfware directly after acquisition; lack of training causes 25 percent of the tools to be used not to their full extent; 15 percent are seriously incompatible with existing tools and need modification to fit into the intended environment. Let us give a few examples by way of anecdotal evidence. James Cross told us that the GRASP tool was used in 25 percent of the cases as a pretty printer for Java, obviously GRASP is not used to its full extent by those users. In [49] people from Mitre tried to modify a particular reverse engineering tool, which was so difficult that they published a paper on the subject, explaining that functionality should be better separated in such tools. So this is an example of serious incompatibility of tools. For more elaborate information on tool acquisition we refer to [28].

Question. Is a company going to use tools that have a high learning curve, that have no manuals, that are not at all geared towards their environment, tools that have no support organization, in other words: tools that come from academia? The question that we pose stems from IT Industry. We are advising various branches in IT industry on evaluation, acquisition, (ab)use, development, and design of tools (large organizations and reengineering companies). As far as we know, there is no overwhelming use of reengineering tools that come from academia. Moreover, we have not seen many papers discussing tools that are shipped to IT industry. On the contrary, in [17] we read "Surprisingly, none of these tools [from IT industry] makes any apparent use of the results of research in using plan-based techniques for concept recovery". We do not know whether this is true, but we are not surprised: the information on tools that Jones gives in his books is confirmed by our own experience.

We think that reengineering research cannot evolve without real problems that come from the IT industry. Our approach towards achieving a symbiosis is to speak the language of the IT industry. This means that we try to absorb their problem domain, we try to learn their language, and we try to discuss issues using examples in their language. Also their computer language. From our experience with tools and industry we learned the lesson that the more specific your tools are for their environment, the more likely they will benefit somehow from those tools. Of course, the tools should also solve a problem for them that is quite concrete. Starting from that position, we try to make progress. This means that we do not want to implement isolated language specific and problem specific tools that only one company, or even one person can use. A small step in this difficult process is the usage of what we call native patterns. We see the careful design of pattern languages as an implicit requirement for IT industry. In the next section we explain them in detail.

3 Native Pattern languages

Much of the source-based recognition literature (and commercial tools) use patterns that contain source material and variables. Our goal is that the patterns only contain source material, either code or pseudo-code that the programmers have already seen in the language reference manual. So they are familiar with native patterns before they have seen them. In this section we explain native patterns. In Section 3.1 we compare native patterns to approaches that contain native facilities but are in our opinion not truly native. In Section 3.2 we discuss native patterns, the more abstract plansi, and their possible combination. In Section 3.3 we characterize native pattern languages. We will very briefly discuss complexity issues as well.

3.1 Native Facilities and Native Patterns

Suppose we want a list of the authors of a set of COBOL programs. Then we can invoke the Unix command: `grep AUTHOR. * | less`. COBOL has a native expression for the name of the author: the keyword `AUTHOR`. As we can see, the Unix command makes use of native parts of COBOL in

[1]Capers Jones has one of the largest knowledge bases: in 1997 it was about 7000 projects from 600 enterprises in 22 countries from 50 industries [29, 30]. His knowledge base is growing with a rate of 50 to 70 projects a month.

order to make the analysis. We call this quite common use of the surface syntax of a programming language a native facility. One could argue that this is a tool that uses a native pattern: a piece of native code that is the pattern. So every lexical tool makes use of native facilities.

TXL is a programming language and rapid prototyping system specifically designed to support transformational programming. A commercial version is available via Legasys Corporation, Canada. The programming language TXL uses native facilities as well. In [12] we can see that the use of native facilities is more sophisticated than with lexical tools: in a TXL program you specify the grammar of the native parts of the program that you want to transform and then you can use those. Due to the availability of the grammar inside the TXL program the textual presence of the patterns are interpreted under the surface as trees. For instance, to change `blue fish` into `red apples` in the language of `words` one can write the following TXL rule [12]:

```
rule foo
   replace [words]
      blue fish
   by
      red apples
end rule
```

as can be seen, native facilities are used in TXL. Of course, this is not a complete TXL program. In [12, loc. cit. p. 19] we read that every TXL program must contain a definition for the nonterminal program, which is always the name of the goal nonterminal of the TXL program, the type as which all inputs to the program must be parsed. The rule set must contain a definition for the rule or function mainRule, which is automatically applied to the parse tree of the input.

The Draco [40] system also uses native facilities. Draco is an approach to the construction of software systems from reusable components. The Draco program transformations use syntactic patterns. We give three examples:

```
ADDx0: ?x+0 => ?x
MPYx0: ?x*0 => 0
IFTRUE: IF TRUE THEN ?s1 ELSE ?s2 => ?s1
```

They all use native facilities since the syntax without the question marks is the same as the syntax of the language that is being transformed. We can read in [40] that the transformations only state rules of exchange between statements in a domain language and statements in that same domain language. This is not a problem for the Draco approach since they only use transformations to optimize a domain language program. We mention that this very restricted use is not at all sufficient for reengineering purposes, where transformations from one language to another are common and where it is crucial to transform arbitrary syntactic structures and not only statements. Our approach handles these issues. We note that in [41], where possible changes to Draco are discussed, the source to source transformations are now called optimizations, but are not subject to redesign.

Another well-known environment and tool for transformations is REFINE [47]. The accompanying language is also called REFINE. In REFINE it is also possible to use native facilities. The idea of REFINE is similar to that of TXL. The grammar is specified separately. In [46] we can read that a grammar compiler produces a parser, pretty printer, and a pattern matcher for grammars. The pattern matcher simplifies object base queries by allowing you to phrase the query as a template that the query must match. The templates resemble fragments from the language. So also here we are dealing with native facilities. The surface syntax is placed in quotes, and is also called the pattern quotation construct. Noteworthy, is that in [47, loc. cit. p.3-198] it is stated that the pattern quotation construct is shorter and easier to understand than its non-quotation equivalent. Patterns are one of the most useful features of the REFINE language, and you are encouraged to utilize them whenever possible in programs that manipulate the object base [47] (we agree with that of course). Let us take a look at a code fragment that we found in [34], where REFINE was used to implement transformations on COBOL.

```
plan: LOOP-UNTIL (CONDITION: @cond)
consists of
    label: PARAGRAPH-NAME(PATTERN: "@par-name.")
    if-stmt: IF-STATEMENT(PATTERN: "IF NOT
                          @cond THEN ..")
    goto-stmt: GO-TO-STATEMENT(PATTERN:
                               "GO TO @par-name")
such that
   CONTROL-FLOW(label, if-stmt)
   CONTROL-DEP(goto-stmt, if-stmt, TRUE)
```

Also here native facilities are used. Indeed inside the quotations we can see fragments of the language: the separator period (`.`), `IF NOT`, `THEN`, and `GO TO`. The `@par-name` is a named single-value variable, the double dot `..` is an unnamed multi-value variable. So we see that in the above code fragment the program is interspersed with native code, and variables that match certain parts of a COBOL program.

What all these approaches have in common is that like in the case of `grep` parts of the language itself are used in an otherwise fixed language. Some approaches use lexical matching of the native code fragments and some use better matching technology since they rely on a grammar. The Draco approach even restricts to statement transformations. Furthermore, the naming of variables in REFINE and the Draco system makes use of special characters to indicate variables. We do not need such variable markers. This makes the patterns also more native.

As noted in [49], grammars are often structured to permit ease of parsing not ease of conceptual understanding. In many cases the grammar may take an unnatural form to fit the algorithm that the parser generator assumes (to accommodate one token look-ahead, and avoid reduce-reduce and shift-reduce conflicts [37]) [49]. The conclusion that one can draw from these (well-known and very true) observations is that it is a very bad idea to use such a grammar for constructing other functionality like, e.g., native facilities as used in TXL or REFINE, or even the generation of a native pattern language, as proposed in this paper. For, if the concepts of a language as expressed in the grammar are interspersed with the restrictions of some parser generator algorithm, it will be very unnatural to use any grammar based structure since the parser generator algorithm will be spoiling conceptual understanding. In [49] the unnatural form of the grammar was severely hampering the modification of a reverse engineering tool. We note that TXL uses recursive descent parsing, which imposes restrictions on the form of the grammar rules (no left recursive rules are allowed). The Draco system uses a conventional BNF notation augmented with control mechanisms such as parser backtracking [40]. This implies that probably recursive descent parsing is used as well, which forbids the use of left-recursive grammar rules. We also note that from grammar specifications in Reasoning/Dialect [46] a dialect of Lex and Yacc [36, 26] is generated, which also imposes restrictions on the form of the grammar rules (they indeed have to accommodate one token look-ahead, and avoid reduce-reduce and shift-reduce conflicts). In the publications that we have seen from TXL, Draco or REFINE the problem that is mentioned in [49] is not addressed.

So, how can we still advocate the generation of native patterns? Generation of native pattern languages should be an asset rather than an albatross around your neck. We explain why we do not have the abovementioned problems. The answer is simple: we do not use parser generator technology that imposes such restrictions on the production rules. Note that this implies that even if the surface syntax of the other approaches looks exactly the same as in our approach, the underlying tree structure is different: in their case it is biased by the parser generator algorithm. The parser generator that we use, implements an algorithm from the early seventies that is used in *natural* language processing [53] and is due to [35]. Rekers [48] was the first who implemented the GLR parsing algorithm as part of a system intended for practical applications (the ASF+SDF Meta-Environment). Many years of experience with GLR parser technology at CWI and the University of Amsterdam have convinced us that superior parser generator technology is a requirement for industrial reengineering. At the moment we see a trend among reengineering companies who are all implementing or discussing the use of GLR parser generators. We mention TechForce, Reasoning, and Semantic Designs. We refer to [4, 10] for discussions of the connection between parser generator technology and generic language technology and reengineering.

Let us compare syntax facilities of the above approaches to our approach. Let us depict a native COBOL pattern:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TEST-1.
WORKING-STORAGE SECTION.
  01 X PIC 9 VALUE 2.
PROCEDURE DIVISION.
PAR-1.
  IF NOT X < 1 THEN
    ADD 1 TO X
    GO TO PAR-1.
```

Note that we can compile this native pattern with our MicroFocus COBOL compiler. So the code is a real COBOL program. This pattern is native COBOL. Let us now compare our approach with the approach taken in [34] and express their plan as a native pattern:

```
Procedure-name-1.
  IF NOT Condition-1 THEN
     Statement-1+
     GO TO Procedure-name-1.
```

This code is completely conform the way code is explained in COBOL manuals or in the COBOL standard [1]. In the glossary of terms using in the COBOL standard [1, p. III-18] we read that a Procedure-name is a user-defined word which is used to name a paragraph or section in the Procedure division. It consists of a paragraph-name (which may be qualified), or a section-name. Page III-5 of the same glossary explains conditions: A status of a program at execution time for which a truth value can be determined. Where the term 'condition' (condition-1, condition-2, ...) and so on. In fact, the above code fragment is pseudo-code that a real COBOL programmer is familiar with. It may have been displayed in any textbook on COBOL. We call this a native pattern as well. For COBOL programmers there is nothing secret going on here. In the next section we will show that this native pattern expresses also the constraints in the plan that [34] provide. So we explained the native part about the pattern. Now we explain the pattern itself. One might get the feeling that this is a large regular expression that is used for lexical matching. This is not at all the case. The concrete syntax and the abstract syntax are closely integrated on the implementation platform that we use: there is a standard mapping between the textual (concrete) representation of actual syntax and there (abstract) representation as a tree. As a consequence, the user sees concrete syntax and the implementation sees the tree [25]. We refer to Section 8.1 of [6] where we discuss the dangers of using lexical techniques for making system wide changes. Now we explain the notation used in the native pattern in detail. `Procedure-name-1` is a variable of type `Procedure-name` and it matches exactly one real `Procedure-name` in real COBOL programs (so exactly as in COBOL manuals). The same holds for the `Condition-1` variable: its type is that of a `Condition` and it matches exactly one condition. The `Statement-1+` is a list variable. It matches a list of `Statements`.

We give an example of a native pattern for Java:

```
do Statement while (Expression);
```

This pattern is fully explained in the Java language specification [21, loc. cit. p. 448]. This pattern is also part of a Java renovation factory is constructed by Eggie van Buiten, a PhD student at the University of Amsterdam. It uses the native Java syntax and the words that are present in the manual.

We hope to have shown that the time for learning a native pattern language when a programmer is native in the language to begin with is close to zero. We believe that this is a very important factor for the transfer of reengineering technology to the IT industry. For, programmers that are fluent in a certain language are much more productive than in a language they are not at all familiar with. See for more information on programming productivity [27].

3.2 Native Patterns Versus Plans

Now we compare native patterns with plans. In the introduction, we already indicated that a plan is a set of components, defining parts of the syntax together with a set of constraints on those components. A good example is the code fragment that from [34] that we used in Section 3.1. First we explain the constraint `CONTROL-FLOW(label, if-stmt)`. This means that there is a control flow path from the label that is used in the COBOL program to the COBOL `IF` statement. In other words: in the COBOL program we first see the label `Procedure-name-1` in our native pattern, with the `IF` statement after the `Procedure-name-1`. Since the native pattern is native COBOL the control flow is already included. Now we explain `CONTROL-DEP(goto-stmt, if-stmt, TRUE)`. In [34] we read: "`CONTROL-DEP(A-stmt, B, C)`: `A` is control dependent (as defined in Section 1) on the `C` (either `TRUE` or `FALSE`) branch of `B`." If we look in Section 1 we find the word control dependent with a footnote: "Intuitively, `A` is control dependent on `B`, if `B` is a control construct having having [sic!] multiple control flow exits (branches). Moreover, `A` must be on a control flow path from some exit of `B` to the end of the program but an alternative path must also exist from some other exit of `B` to the end of the program not passing through `A`. A more formal definition can be found in [17]". So finally, in the footnote we are directed to another paper. The problem with such notations is that some users in the IT industry will not be able to understand what is going on. What is meant is that the `GO TO` is in the `IF` statement. In our native pattern this is just expressed by putting it in the `IF` statement like you would do as a COBOL programmer. This example shows clearly the benefits of our viewpoint towards the notation of patterns. Moreover, our native pattern languages are documented by the manuals that the programmers already have, and since the patterns are native code there can be no question what a pattern actually means. Native patterns are self documenting and self-explanatory. We believe that ad-hoc pattern languages do not contribute to widespread use in IT industry. In the academic arena we hear that the use of REFINE is difficult. In [14] we can read that REFINE tool builders also have to be familiar with Lisp, object orientation, logic, set theory, and compiler construction technology. In [18] we can read that although REFINE is more powerful than GENOA, the latter is easier to learn.

Let us discuss another example. In [56] notation has been developed to describe plans. We recall that a plan contains components describing parts of a program and the constraints describe control flow connections or data flow dependencies in various ways. The underlying algorithm for recognition is often referred to as the constraint satisfaction problem (CSP) approach. In [57] empirical evidence shows that the CSP approach is useful in the sense that the algorithms used are as optimal as possible. Steve Woods communicated to us that the notation is not easy to read or use by uninitiated people. In our opinion, this will hamper the use in IT industry, which has been observed in [57] as well. A combination of the use of native patterns and the more abstract plans could solve this problem. We discuss this issue in a moment.

In [17] plans for Year 2000 recognition are specified using the technology developed in [56, 57]. The notation is transformed into a Lisp representation that is input for 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 input to the recognizer as well. Then the plan recognizing engine will output recognized plan instances. Control flow and data flow information can be assessed via constraints. If they are not present in the plans, this information that is present in the AST is not used.

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. An advantage of plans is that it is not necessary to add what we call context information. We have to express this information using a native pattern. For analysis this is convenient since it enables location of more instances using less plans, but also of false-positives. For native patterns this is crucial information if we also wish to change code automatically. Therefore, we consider both approaches to be complimentary (see below). In Section 5 we treat an example that we found in [17] 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, or by additional constraints. 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. Again this implies that both approaches are complimentary: using, for instance, the plan based approach of [17] 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.

Combining Plans and Native Patterns Rather than having to choose between all the approaches that we discussed for recognizing patterns, we think it is worthwhile to investigate their combination. For instance, it might be possible to write a translator (using native patterns), that turns a native pattern into an abstract plan. It is also possible to write a translation that turns source code into the AST that is needed for the plan recognizer. In fact, in this paper we already combined the two approaches: the plan based approach of [17] was used to make an inventory of the problematic Year 2000 code fragments, and once the problematic code is identified, we used native patterns for transforming them (see Section 5 for details).

3.3 Characterizations of Native Pattern Languages

In [38] we can read that universal algebra has been applied systematically in the theory of automata and formal languages, machine synthesis, the theory and practical development of programming languages, symbolic computations with strings, natural language text, logical formulas, specifying and implementing abstract data types, computations on abstract structures, correctness of programs, etc. It is also possible to use universal algebra to give the definition of a native pattern language for a give programming language. This is not too surprising: in 1830 Peackock already writes "Algebra may be considered, in its most general form, as *the* science which treats of the combinations of arbitrary signs and symbols by means of defined through arbitrary laws [..]" [44].

However, since the reengineering community is not that familiar with universal algebra—as pointed out by a few kind reviewers—we decided to omit the formal definitions of native patterns and native pattern languages in the conference paper. We are of course willing to explain these formal definitions to everyone who wants to know more about that part. From its formal definition it is not too hard to see that a native pattern language can be generated from the grammar of the programming language. In fact, this implies that for each programming language we can pull the handle and generate a language that feels native to a programmer fluent in that language. So what seems to be extremely language specific, is in fact under the surface completely generic. The generic part is universal algebra. A very thorough treatment on this subject can be found in a chapter of the handbook of logics in computer science where Karl Meinke and John Tucker made an effort of explaining universal algebra [38].

We will give an intuitive characterization of native patterns and native pattern languages, in order to give the reader a good impression of the concepts. In Section 3.1 we have seen a few native patterns: a real COBOL program, and a COBOL program using the pseudo-code that is available in the ANSI standard, and a Java pattern that is literally in the language reference manual. We have only one small extension to what is in the COBOL standard: we use a `+` to represent lists of one or more occurrences of a construct and we use a `*` for zero or more occurrences. We note that variables may be numbered at wish as is often done in manuals. So a native pattern language for a given language is all the pseudo-code that is in the language reference manual of this language plus for each variable representing a language construct, like `Statement`, we have variables `Statement`, `Statement+`, `Statement*`, optionally using numbers when we need more than one, like `Statement23+`. The interprestation of `Statement1` is that it matches exactly one arbitrary COBOL statement, `Statement1+` matches one or more COBOL statements, and `Statement1*` zero or more. Note hat the convention of numbers is common in language reference manuals.

Definitions-R-Us This paragraph can be omitted for readers not interested in formal definitions. Our definition of a native pattern language is roughly as follows. A Native pattern language $P(L)$ for a given language L is the collection of terms that can be defined over a many-sorted signature that consists of the context-free functions in distributed fix notation that constitute language L and a set of variables, and two sets of list variables. A native pattern is an element of the above collection. Our definition is a combination of Definition 3.1.1 [38, loc. cit. p. 221], of Definitions 2.1 and 2.2 as given in [20] and of the definitions in Section 2.3 of [25]. In [38, loc. cit. p. 202] an elegant correspondence between many-sorted terms and Backus Naur Forms [2] is made, illustrating the natural connection of universal algebra and formal language definitions. In Section 2.3.1 [38, loc. cit. p. 210] the syntax of while programs is discussed where the connection between distributed fix operations and terms in prefix notation is elegantly illustrated. For more explanations we refer to the technical report version of this paper [52].

Matching and Complexity Many papers on pattern matching focus on other issues rather than the design of the accompanying language. One central issue is the complexity of the algorithms used. The matching algorithm underlying the so-called *constraint satisfaction problem* as used in [17] is know to be NP-hard [24, 54, 58]. We will give some information on complexity issues, as well. In our case, since we omitted a formal definition of patterns, we cannot elaborate on the associated complexity issues in depth. We give some pointers though. First we emphasize that the constraint satisfaction problem is a different problem than the matching problem we deal with here, i.e. solving a different problem. Hence it is not relevant to compare their complexities. Roughly speaking the matching problem we deal with can be stated as follows: given a pattern p and another pattern s, does there exist a substitution σ such that $\sigma(p) = s$? In case the patterns do not contain `+` or `*`, the problem is linear. See [43] for a formal proof. If we use a `+` or a `*` the problem becomes significantly more complex. Solving this general case comes down to solving the so-called *associative matching problem* which is NP-complete, as proved in [3, loc. cit. p. 426]. In fact, the situation is a bit more complex if we compile ASF specifications to C: some special cases are recognized and compiled to efficient implementations. For the complexity of pattern matching as implemented by the (new) ASF2C compiler we refer to [5].

4 Generating Native Pattern Languages

We hope to have illustrated that the native pattern language paradigm is a useful way to design a native pattern language. Of course, such a language should be constructed, as well. This section deals with this issue. In Section 4.1 we discuss the generation process. Next, we briefly discuss support for the programmer in Section 4.2. Finally, we discuss the generated native aspects of transformation and analysis support in Section 4.3.

4.1 The Generation Process

We implemented a development environment for software renovation factories that supports the generation of native pattern languages. We can generate from an arbitrary context-free grammar the native pattern language and generic transformation and analysis support by pressing a button. The generation process for native pattern languages itself is very simple when implemented in the ASF+SDF Meta-Environment and is described in considerable detail in the technical report version of this paper [52]. The generation process of the transformations and analysis support is elaborately discussed in [6].

We recall that we use GLR parser generator technology, so we do not suffer from the problems that are addressed in [49] (see Section 3.1 for details). We describe grammars in a formalism called SDF (Syntax Definition Formalism) [25]. SDF supports compact modular syntax definition by offering a standard interface between lexical and context-free syntax, a standard correspondence between context-free and abstract syntax, powerful disambiguation and list constructs, and an efficient incremental implementation which accepts arbitrary context-free grammars.

In Figure 1 we depicted part of the development environment that we implemented for creating software renovation factories. We will explain the screen dump so that the reader will get an impression of what is generated by us, and how it relates to nativeness of our approach. The code in the window is an SDF definition of a small language called `Cat` that we use to demonstrate the ideas behind factory generation. It stems from [6] where we discuss the generation of transformation and analysis support that use native patterns as input.

The window in Figure 1 is a generic structured editor [32] that understands SDF. The buttons at the left-hand side of the window implement the following functionality:

- `PrettyPrinter` A pretty printer for SDF programs
- `AddQoutes` A tool that takes care of consequent addition of quotes around SDF literals. It is a convenient tool if the grammars become large.
- `AddSorts` A tool that takes care of the declaration of sort names (in SDF we declare sort names). Also a convenient tool when the grammars tend to become large.
- `Tsyn`–`Aeqs` are buttons that generate the syntax and semantics of generic transformation and analysis support. This has been treated elaborately in [6] but see Section 4.3 for their native properties.
- `NatPatLang` the button that generates the native pattern language from a grammar description.

In Figure 2 we depicted the result of pressing the `NatPatLang` button. We scrolled the contents a bit so that we can see the meat of the generation process in more detail. First of all, the button is implemented in SEAL [33] a coordination language for connecting components in the ASF+SDF Meta-Environment. We connected a few components in the `NatPatLang` button. It is the functionality that

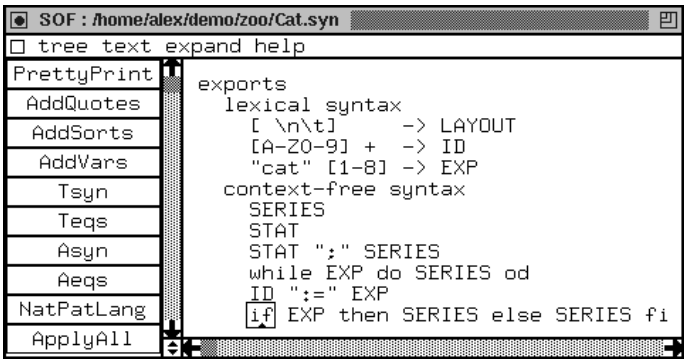

Figure 1: Development environment for renovation factories.

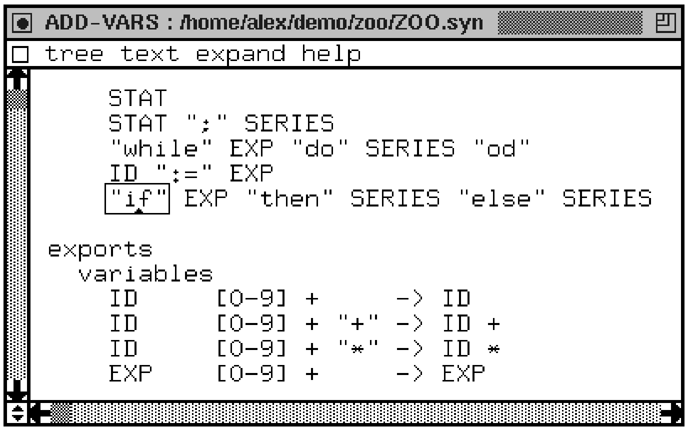

Figure 2: The result of pressing the `NatPatLang` button.

is available in the `AddQuotes`, `AddSorts`, and `AddVars` buttons. So possibly not yet declared sort names are declared, possibly missing quotes are added, and the three types of variables are generated in a special `variables` paragraph. Once we have generated a native pattern language in our development environment for constructing software renovation factories, we can load the native pattern language in a fresh instance of the ASF+SDF Meta-Environment. If we also load the results of the `Tsyn`–`Aeqs` buttons, we have a complete architecture that enables rapid development of software renovation components. For this process we refer to [6].

4.2 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. Since we try to follow the manuals literally, the use of normal available manuals will be very useful already. But what if there is no manual? Or if the manual is known to contain many errors (see [51] for assessment of manuals using tools)? Or if the grammar is updated with undocumented features? In those cases we use the possibility to generate this documentation from the grammar. Moreover, the ASF+SDF Meta-Environment

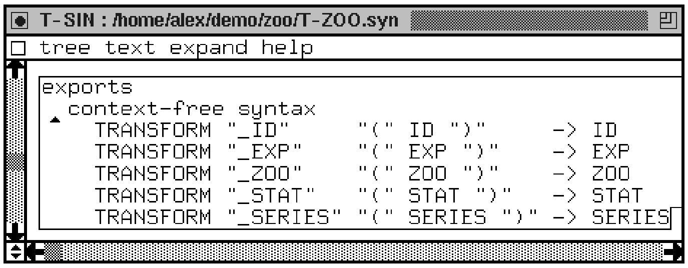

Figure 3: Generation of the generic transformation syntax.

provides a generic structured editor, so always an editor is available that knows the language.

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 [11] we automatically generate a LaTeX document that contains the entire native pattern language in typeset 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 [42] 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 [11]. In [10] it is argued that grammars for reengineering purposes are not always stable, so generated documentation for (also generated) native pattern languages is important, since after modifying a grammar we can generate up-to-date documentation.

The ASF+SDF Meta-Environment supports a so-called generic text and structure editor [32]. 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 [32].

4.3 Generating Native Functional Support

As we have seen, approaches like TXL or REFINE use native facilities in an otherwise fixed language. We generate native syntax to deal with analyses or transformations. In this section we discuss how we deal with syntax for transformations and analyses. The buttons `Tsyn` – `Aeqs` in Figure 1 generate from the grammar syntax and semantics to deal with transformations and analyses. In [6] the semantics and its implications for software renovation factories are elaborately discussed. In this section we will briefly discuss the generated syntax, which is as native as possible.

In Figure 3, we generated (simplified) syntax for generic transformations for the `Cat` language. We obtained this syntax directly from the `Cat` grammar by by pressing the `Tsyn` button of the window displayed in Figure 1. In the semantics we use variables of sort `TRANSFORM`. This means that when we declare `foo` to be of sort `TRANSFORM`, a `Cat` programmer gets for every sort in the grammar the functions `foo_ID`, `foo_EXP`, ..., `foo_SERIES` available. Therefore, we call this native functional support. In Section 5.4, we will see examples of `Leap-year-order_Boolean`, `Leap-year-order_Stat*`, and `Leap-year-corrector_Program`. Also the way of defining the `Leap-year-order` and `Leap-year-corrector` are shown in that section. So also our transformation and analysis syntax is not fixed and uses as much native input as possible. The arguments of these functions are the native patterns. In Section 5.4 some examples are present. More information on generic transformations and analysis support can be found in [6].

5 Using Native Patterns

We discuss the use of our approach for a native pattern language generated for COBOL. The example transformation that we will implement is known to be a notoriously difficult Year 2000 problem and is reported on in [17]. We first discuss the problem in Section 5.1. In Section 5.2 we discuss a native pattern that recognizes the problem. In Section 5.3 we discuss the replacement pattern. In Section 5.4 we discuss the actual transformation. We address how to deal with comments in a transformation in Section 5.5. In Section 5.6 we discuss a way to deal with variations of this Year 2000 problem, by specifying partial semantics for COBOL (using native patterns) . Finally, in Section 5.7 we mention an assembly line that has been implemented to automatically solve this particular Year 2000 problem. Note that we do not claim to solve every instance of the Year 2000 problem. We only show the use of native patterns.

5.1 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 [17]:

```
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 thus far 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 [45] it is claimed that the leap year calculation is frequently missed, since it is relatively complex and the algorithm was not available during programming.

5.2 Recognition Pattern

In [17] 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 native patterns to construct a tool that recognizes the above code and changes it automatically to the corrected code that is also provided in [17].

In this section we start with the input pattern. For comparisons we recall a rule from [17] 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. Normally, we discuss the change strategy with the owners of the code, and we never take the initiative to make changes that they do not like. In this case the code is present in the scientific literature, so we take the freedom to change it a we would like to do it. We want to stress that the changes to follow are not obligatory, they are just a choice. Moreover, this example is to show the use of native patterns. For a real-world set of changes communicated to us by a maintenance team of a customer in the IT industry we refer to [50], also native patterns are used in that paper.

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 5.4 we will treat this preprocessing transformation. In [9] 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, 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 [17] it is checked that the variable that we call `Data-name1` is numeric. In our approach this is not necessary: it is only possible to have a `Data-name` on that location in the native pattern. Then zero or more statements follow. This is expressed with the list variable `Stat1*`. We note that we violate our own rules by abbreviating `Statement-1*` to `Stat1*`. Since COBOL is very verbose, we abbreviated some very long words here, in order to keep explanations smooth. In the code fragment of [17] `Stat1*` 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 should not change the variables `Data-name3` and `Data-name5`. This has been implemented, see Section 5.7. 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.

One might now think that if in the `Stat1*` a `DIVIDE BY 400` is present, the above pattern recognizes a false-positive. This is not true. For, then the `IF` statement making the comparison would also include the test on 400. If it does not contain this test, then indeed the pattern matches and it will duplicate the `DIVIDE BY 400` and repair the `IF` statement. Also then there is no false-positive (only a harmless duplicated `DIVIDE` statement).

5.3 A Replacement Pattern

Now that we have an input pattern, we wish to change the code. In the previous section we demonstrated a pattern that only served the purpose of recognition. When the purpose of recognition is to make changes, this generally asks for another native pattern since changes are often global. In the case of [17] this is also the case. So we revise our input pattern and provide its replacement pattern. This gives us another chance to demonstrate a few native patterns. The reason why the input pattern has to change can be found in the solution that is given in [17]:

```
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 5.7). 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 [17]. We only 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 [17]. 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 400 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 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.

5.4 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 [10] 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 [6] 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 gives access to native functional support (see Section 4.3). 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 language reference manual. 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
```

We discussed the generated syntax in Section 4.3. We recall that `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 5.3. 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`.

5.5 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 difficult 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 a simplified 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 included. 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 native patterns are complete. See [9] for more information on treating comments in transformations.

5.6 Dealing with Variations

Of course, it is possible that programmers use other notation than above to implement their algorithms. On way to deal with that is to mimic those calculations using new native patterns as we have showed above. There is also another way. The ASF+SDF Meta-Environment is capable of describing the semantics of a language as well. So we can add semantics to the transformation above so that it can deal with other implementations of the leap year algorithm as well. We can construct a module `LYS` (Leap Year Semantics) that contains the necessary information. We give an example. Suppose that the programmers not only use `DIVIDE` but also `COMPUTE` statements for the necessary calculations in the leap year algorithm. Then we can add some very specific semantics to the transformation telling the tool that both calculations have the same semantics. We do this in an ASF module that is imported by the transformation. It also uses native patterns:

```
equations
[1] COMPUTE Data-name2 = Data-name1/4
    COMPUTE Data-name3 = Data-name1 - Data-name2*4
    COMPUTE Data-name2 = Data-name2/25
    COMPUTE Data-name5 = Data-name1 - Data-name2*100
=
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
```

So in order to deal with the above COMPUTE algorithm we just explain the tool that the semantics of COBOL is that these two calculations are equal. First, via the LYS semantics the COMPUTE part is transformed into DIVIDE statements and then the transformation makes the correction. Of course, it is necessary to discuss such issues thoroughly with customers since maybe they only want COMPUTE statements. We recall that we do not have the intend to solve the leap year problem in all cases. We just show another use of native patterns that deals with variations.

5.7 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 many the pre and postprocessing tools described in other papers. 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 5.3) are innocent. This list may be seen as hideously ad hoc, procedural, laborious and inefficient. In fact, our list is simple when compared to the list presented in [17, loc. cit. p. 129]: "In addition, simple expression simplification and reordering techniques (e.g., always using less than for comparisons rather than greater than, treating nested IFs and ANDs, simplifying negated conditions by switching the IF and ELSE branches, and so on), allow us to ignore many other variations. And finally, restructuring techniques such as GOTO elimination and expansion of non-recursive procedures allow us to have plans that deal with simple control flow."

Since there is no space left for an elaborate discussion of this assembly line, we refer to other papers where similar assembly lines are discussed. After all, this paper has the intend to explain the use and generation of native pattern languages. We refer to [9, 50, 51] for treatments of assembly lines. We discussed these references briefly in Section 1.

6 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] American National Standards Institute, Inc. *Programming Language – COBOL*, ANSI X3.23–1985 edition, 1985.

[2] J.W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In S. de Picciotto, editor, *Proceedings of the International Conference on Information Processing*, pages 125—131. Unesco, Paris, 1960.

[3] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *LNCS*, pages 417–429. Springer-Verlag, 1985.

[4] 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. Available at: http://adam.wins.uva.nl/~x/sigplan/plan.html.

[5] M.G.J. van den Brand, P. Olivier, J. Heering, and P. Klint. Compiling rewrite systems: The ASF+SDF compiler. Technical report, CWI/University of Amsterdam, 1998. In preparation.

[6] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. Available at http://adam.wins.uva.nl/~x/scp/scp.html. An extended abstract with the same title appeared earlier: [7].

[7] 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 4th Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[8] 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.

[9] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In P. Nesi and F. Lehner, editors, *Proceedings of the 2nd Euromicro Conference on Maintenance and Reengineering*, pages 11–19, 1998. Available at http://adam.wins.uva.nl/~x/cfn/cfn.html.

[10] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[11] 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.

[12] J.R. Cordy and I.H. Carmichael. The TXL programming language syntax and informal semantics version 7. Technical Report 93-355, Queen's University, Canada, 1993.

[13] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[14] A. van Deursen. A comparison of Software Refinery and ASF+SDF. In *Program Analysis for System Renovation*, pages 9–1–9–41. CWI, 1997. Available at: http://www.cwi.nl/~arie/papers/refine.ps.gz.

[15] A. van Deursen and T. Kuipers. Finding classes in legacy code using cluster analysis. In S. Demeyer and H. Gall, editors, *Proceedings of the ESEC/FSE'97 Workshop on Object-Oriented Reengineering*, Report TUV-1841-97-10. Technical University of Vienna, 1997.

[16] A. van Deursen and L. Moonen. Type inference for COBOL systems. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the 5th Working Conference on Reverse Engineering*. IEEE, 1998.

[17] A. van Deursen, S.G. 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.

[18] P.T. Devanbu. GENOA - a language and front-end independent source code analyzer generator. Unpublished revision of [19]. Available at http://seclab.cs.ucdavis.edu/ devanbu/begin.ps.gz.

[19] P.T. Devanbu. GENOA - a language and front-end independent source code analyzer generator. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–319. IEEE, 1992.

[20] W.J. Fokkink and C. Verhoef. A conservative look at term deduction systems with variable binding. *Information and Computation*, 1998. To appear. Available at: http://adam.wins.uva.nl/˜x/conimex.ps.

[21] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[22] S. Haglund and J. Persson. A process for reverse engineering of AXE 10 software. In *Proceedings of the 6th Reengineering Forum*, 1998.

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

[24] J. Hartman. *Automatic Control Understanding for Natural Programs*. PhD thesis, University of Texas at Austin, 1996.

[25] 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. Latest version: ftp://ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z.

[26] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

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

[28] C. Jones. *Assessment and Control of Software Risks*. Prentice-Hall, 1994.

[29] C. Jones. *Patterns of Software System Failure and Success*. International Thomson Computer Press, 1996.

[30] C. Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, 1998.

[31] N. Jones. Year 2000 market overview. Technical report, GartnerGroup, Stamford, CT, USA, 1998.

[32] 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.

[33] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. In H.A. Wijshoff, editor, *Computing Science in the Netherlands (CSN93)*, SION, pages 217–228, 1993.

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

[35] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[36] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, UNIX Programmer's Supplementary Documents, volume 1 (PS1) edition, 1986.

[37] J.R. Levine, T. Mason, and D. Brown. Yacc ambiguities and Conflicts. In *lex & yacc*, pages 217–241. O'Reilly & Associates, Inc., 2nd edition, 1992.

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

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

[40] J.M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–574, September 1984.

[41] J.M. Neighbors. Draco: A method for engineering reusable software systems. In T.J. Biggerstaff and A.J. Perlis, editors, *Software Reusability – Concepts and Models*, volume I, chapter 12, pages 295–319. ACM Press, 1989.

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

[43] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[44] G. Peackock. *A treatise of algebra*. Cambridge, 1830.

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

[46] Reasoning Systems, Palo Alto, California. *DIALECT user's guide*, 1992.

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

[48] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

[49] H. Reubenstein, R. Piazza, and S. Roberts. Separating parsing and analysis in reverse engineering tools. In R.C. Waters and E.J. Chikofsky, editors, *Proceedings of the 1st Working Conference on Reverse Engineering*, pages 117–125, 1993.

[50] M.P.A. Sellink and C. Verhoef. An architecture for automated software maintenance. Technical Report P9807, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/˜x/asm/asm.html.

[51] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. Technical Report P9805, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/˜x/cale/cale.html.

[52] M.P.A. Sellink and C. Verhoef. Native patterns. Technical Report P9804, University of Amsterdam, 1998. Available at http://adam.wins.uva.nl/˜x/pat/pat.html.

[53] M. Tomita. *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.

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

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

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

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

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