Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

SEN

Software ENgineering

A language independent framework for context-sensitive formatting

M.G.J. van den Brand, A.T. Kooiker, J.J. Vinju, N.P. Veerman

REPORT SEN-R0601 JANUARY 2006

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

A language independent framework for context-sensitive formatting

ABSTRACT
Automated formatting is an important technique for the software maintainer. It is either applied separately to improve the readability of source code, or as part of a source code transformation tool chain. In this paper we report on the application of generic tools for constructing formatters. In an industrial setting automated formatters need to be tailored to the requirements of the customer. The (legacy) programming language or dialect and the corporate formatting conventions are specific and non-negotiable. Can generic formatting tools deal with such unexpected requirements? Driven by an industrial case of nearly 80 thousand lines of Cobol code, several limitations in existing formatting technology have been addressed. We improved its flexibility by replacing a generative phase by a generic tool, and we added a little expressiveness to the formatting back end. Most importantly, we employed a multi-stage formatting framework that can cope with any kind of formatting convention using more computational power.

1998 ACM Computing Classification System: D2.3;D1.2
Keywords and Phrases: context-sensitive formatting; pretty-printing; case study; Cobol; Box

A Language Independent Framework for Context-sensitive Formatting

M.G.J. van den Brand[*], A.T. Kooiker, J.J. Vinju
Centrum voor Wiskunde en Informatica
P.O. Box 94079, NL-1090 GB, Amsterdam, The Netherlands
{Mark.van.den.Brand, Taeke.Kooiker, Jurgen.Vinju}@cwi.nl

N.P. Veerman
Department of Computer Science, Vrije Universiteit Amsterdam
De Boelelaan 1081a, NL-1081 HV, Amsterdam, The Netherlands
nveerman@cs.vu.nl

Abstract

Automated formatting is an important technique for the software maintainer. It is either applied separately to improve the readability of source code, or as part of a source code transformation tool chain. In this paper we report on the application of generic tools for constructing formatters.

In an industrial setting automated formatters need to be tailored to the requirements of the customer. The (legacy) programming language or dialect and the corporate formatting conventions are specific and non-negotiable. Can generic formatting tools deal with such unexpected requirements?

Driven by an industrial case of nearly 80 thousand lines of Cobol code, several limitations in existing formatting technology have been addressed. We improved its flexibility by replacing a generative phase by a generic tool, and we added a little expressiveness to the formatting back end. Most importantly, we employed a multi-stage formatting framework that can cope with any kind of formatting convention using more computational power.

1 Introduction

The layout and style of program source code are crucial for the time required to understand and thus modify it [16]. In fact, these aspects are too integral to the coding aspect to be changed effectively later [15, p399]. However, a system may be initiated with a proper coding standard, years of evolution causes the code to deviate from it. For instance, some programmers format code to a preferred style by hand before making changes. The result is a mixture of different programming styles, which increases the time for maintenance.

A solution to these problems is to format source code with automatic tools. Automatic formatting improves the productivity of maintenance programmers, but it hardly implements the company specific conventions. There exist some language-specific formatting tools that can be used off-the-shelf [9, 22], but these tools suffer from a profusion of command line arguments required to deal with the highly variable formatting conventions that users of the tool will have. Therefore configuring these tools to deal with company specific requirements can be very difficult or even impossible. When a language-specific formatting tool is not available for a certain legacy or domain-specific language, generic formatting (e.g. tools [7, 8, 19, 25]) can be applied. Such tools allow a formatting convention for any language to be defined and can then generate the formatting tool from this definition. They can even provide a more flexible alternative to existing language-specific formatters. However, one of the limitations of existing generic formatters is that these can not use context information or allow arbitrary complex computation, which can be required when formatting according to industrial coding standards.

The goal of this research is to be able to rapidly obtain a formatter for any language that implements arbitrary formatting requirements.

1.1 Contributions

In Section 3 we describe in detail the formatter that meets the requirements needed in our Cobol casestudy (See Section 1.2). First we will evaluate limitations of existing formatting technology, and propose our improvements. We claim that the resulting framework can be expected to han-

[*] As of January 1st, 2006: Software Construction, Technische Universiteit Eindhoven, Den Dolech 2, NL-5612 AZ, Eindhoven, The Netherlands, m.g.j.v.d.brand@tue.nl

Table 1. Required layout standard in the Cobol case study.

Description	Column
Start of divisions, sections, declarations, paragraphs	01
Start of PIC and REDEFINES clauses	41
Start of VALUE, COMP and OCCURS clauses	51
Start of statements	09
Second part of statements (e.g. MOVE Id TO . . .)	25
Third part of Statements (e.g. MOVE Id TO . . .	49
Fourth part of Statements (e.g. MOVE . . . TO Id)	53

Description	Indentation
A nested data declaration should indent with respect to its associated group variable	4
A level 88 data declaration should indent with respect to its preceding variable	4
In a data declaration, a level number and a variable name are separated	2
Indentation of nested statements	4

```
Column   10        20        30        40        50        60        70
12345678901234567890123456789012345678901234567890123456789012345678901
DATA DIVISION.
WORKING-STORAGE SECTION.

01  WD_DATE                                       VALUE ZERO.
    88  WD_NO_DATE                                VALUE ZERO.
    03  WD_DD                           PIC 9(02).
    03  WD_MM                           PIC 9(02).
    03  WD_JJ                           PIC 9(02).

PROCEDURE DIVISION.
INITIALIZE_DATE SECTION.
INIT_00.
        IF              WD_NO_DATE
            ACCEPT      WD_DATE                 FROM DATE
        END-IF
        MOVE            WD_DD                   TO  WR_DD
        MOVE            WD_MM                   TO  WR_MM
        MOVE            WD_JJ                   TO  WR_JJ.
```

Figure 1. Example program with standardized formatting according to Table 1.

dle all kinds of unexpected formatting conventions in any programming language. Specific contributions of this paper are:

- Context-sensitive formatting;
- A formatting framework that needs no default formatter to be generated;
- Demonstrating the use of generic formatting technology on an industrial case.

1.2 An industrial case with context-sensitive requirements

In the context of a software renovation research project, we were asked by a company to enforce a corporate formatting standard to the source code of a medium sized Cobol system. The system had about 78 thousand lines of code spread over 91 Cobol programs.

We were given a document [20] with conventions for formatting a number of Cobol language constructs, which is summarized in Table 1. For several constructs, the assigned starting columns and indentation depths are displayed. The requested standard is illustrated by a code example in Figure 1, containing column numbers, data declarations and a number of statements. The example shows a record structure for storing a date. Level `03` sub-records are used to store the day, month and year, and a level `88` condition entry is declared to check if there is a date. Then in the `PROCEDURE DIVISION`, if no date is present in the record, the date is retrieved from the system clock. In addition to this, the date is stored in another record.

Many formatting conventions are geared to clarify the logical structure of the source code. For example, the statements inside an `IF` are indented to clearly indicate that their execution is subject to the conditional. However, this formatting convention also contains some more unusual rules. Figure 1 shows how the `FROM` part of the `ACCEPT` statement is aligned with the `TO` parts of `MOVE` statements that are *outside* the conditional. This is an example of alignment that crosscuts the logical structure of the program. In a more traditional formatting convention the `FROM` part of the `ACCEPT` statement would be indented relative to the beginning of the entire statement. Instead the convention dictates that we must indent the first `ACCEPT` part and put the `FROM` part at an absolute column, regardless of the current indentation level.

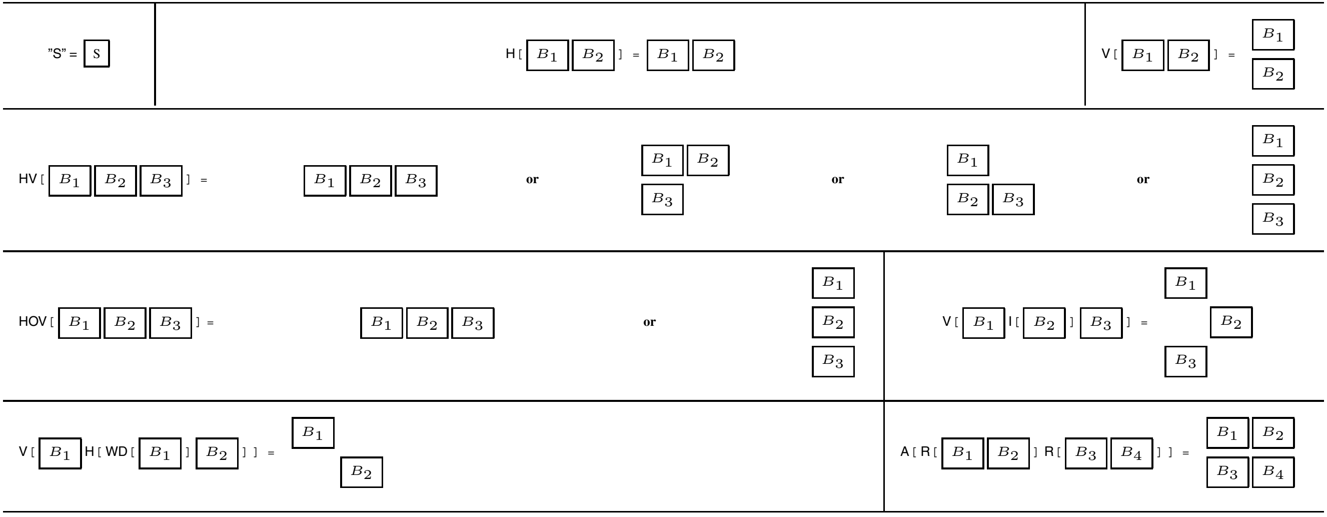
Another point of interest is the `88` declaration. This declaration needs context information to be formatted properly according to our formatting conventions. The trailing `03` fields are not indented with respect to the level `88` field. In fact, when we read the fine print of the standard we found out that trailing fields after an `88` declaration are not to be indented. It is a formatting exception that assigns a particular meaning to the number `88`. The more general scheme is that declarations with a higher number should be indented more. In this example `01` is not indented, while `03` is indented four positions. The Cobol level number thus dictates the indentation depth. The `88` declaration introduces an exception to this rule, since subsequent declarations are not indented. To format the `88` declaration, the context needs to be carefully inspected.

1.3 Generic formatters

Assuming there is no off-the-shelf Cobol formatter with enough command line arguments to serve our purposes, we will have to develop a tailor-made formatter. Generic formatters exist that should automate the boilerplate part of formatting, leaving only some high-level configuration to be expressed by us.

Generic formatters assume the existence of a parser that produces a parse tree of the input program encoded in a

Table 2. Box formatting operators


"S" = S
H [B1 B2] = B1 B2
V [B1 B2] = B1 B2
HV [B1 B2 B3] = B1 B2 B3 or B1 B2 B3 or B1 B2 B3 or B1 B2 B3
HOV [B1 B2 B3] = B1 B2 B3 or B1 B2 B3
V [B1 I [B2] B3] = B1 B2 B3
V [B1 H [WD [B1] B2]] = B1 B2
A [R [B1 B2] R [B3 B4]] = B1 B2 B3 B4

well-known format. These can pretty-print the parse tree to text or other output formats using *default formatting rules*. These also accept high-level *user-defined formatting rules*, in order to adapt the formatter to specific requirements. The expressiveness of these user-defined rules defines the amount of variability that is allowed for defining a formatter.

Many formatting systems use some variant of the Box language as an intermediate format [1, 2, 7, 8, 25]. The Box language, introduced in [6] for defining user interfaces, has operators that define exactly the two-dimensional relative positions of source code elements such as keywords and identifiers. From a Box structure, a reusable Box back-end generates formatted text, or any other visual representation of the source code (e.g. in HTML).

The remainder of this section will give a brief explanation of each Box operator. Table 2 shows an example of each Box operator being used. Box operators can be combined recursively. String literals, denoted as S in Table 2, are atomic Box expressions. H boxes place their operands horizontally, while V boxes place them vertically with respect to each other. The two hybrid operators, HV and HOV place their operands either horizontally or vertically. The first maximizes the number of horizontally placed operands, while the second puts all operands vertically if they do not fit next to each other on a single line. The I and WD operators are used to define static and dynamic indentation respectively. Finally, an A box declares an alignment environment, in which R boxes are aligned in columns. The absolute placement of boxes on a two-dimensional page follows from their relative positioning, their width, and the available space. Each Box operator has various parameters to fine-tune it. For example, H hs=2 will separate each operand of the H box by two spaces. Note that Box operators never loose operands, or change the order of appearance of their operands on a page read left-to-right, and top-to-bottom.

1.4 Related work

The development of generic formatters [18, 21] has been tackled in various ways. Many of these approaches are in one way or the other influenced by the early work by Oppen [19] on pretty-printing and Coutaz [6] on user interfaces. Oppen's work can be characterized as the lexical approach. A stream of lexical tokens interwoven with escape characters to direct the pretty-printing is the input for Oppen's pretty-printer. TXL [4] uses an approach which strongly resembles the original approach of Oppen (see Chapter 5 of [5]). TXL grammar rules can also be used to define formatting conventions. As such, a grammar must be adapted to each set of formatting requirements. The Box layout abstraction offers a general way of expressing two dimensional formatting and decouples the formatting of back-ends from syntax trees. The pretty-print engines of PPML [1, 17] pioneered the use the Box language for formatting source code. Box was extended in [27] and [25] to be able to cope with the details of larger programming languages, such as Pascal and Cobol.

One approach to obtain Box based formatters is to generate pretty-print rewrite rules that map syntax constructs to Box operators. This approach is described in [25]. The starting point of that work is the generation of a collection of *default* pretty-print rules. Default rules are generated using heuristics that analyze typical structures of programming languages. If the behavior of a specific rule does not satisfy the required layout, a new rule can be added by the user that supersedes the behavior of the default rules. The drawback of this approach is that this technique is volatile to

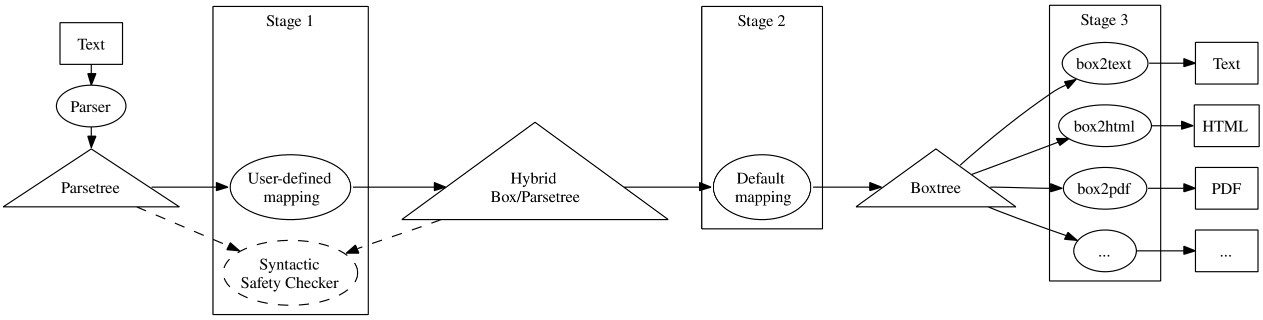


Figure 2. A multistage formatting pipeline that allows any formatting convention, and still automates the bulk of formatting.

modifications in the underlying grammar. If the grammar is adapted the large set of default rules has to be regenerated. An advantage of this approach is that context information can be taken into consideration during pretty-printing. Although this can only be achieved by overriding almost all generated default rules. The implementation details of the generated rules can not be hidden from the user.

An alternative approach to the generation of a collection of pretty-print rules was presented in [7]. The starting point of this work is a table consisting of names of abstract syntax tree nodes and default pretty-print rules, which is also based on a Box-like language. These rules can be overridden by giving an alternative pretty-print rule for a specific abstract syntax tree node. The advantage of this approach is that it is less sensitive to modifications in the underlying grammar. The table approach has been used to develop pretty-printers for re-engineering purposes [8].

The Box language is also used by the commercial DMS program transformation system [2]. It uses a mapping from production rules to Box constructs, which allows similar expressibility to the table approach described above.

A drawback of the two latter approaches is that they do not take context information into account and thus any unforeseen requirements are not easily dealt with.

2 A description of the framework

Our industrial case illustrates that formatting is a process which heavily depends on specific user requirements, requiring context information in some cases. The application of formatting source codes is bound by strict, but possibly irregular rules given by the owner of the code base. Both the used language, as well as the corporate conventions may be unique.

Existing generic pretty-printing, apart from [25], do not cope well with unexpected formatting conventions that require more elaborate analysis of the source code. Extra information such as nesting depth, specific identifiers, or relative positions between several constructs in a language is often important; this is context-sensitive information.

When language constructs are mapped during pretty-printing, we should allow more elaborate user-defined computation. Nevertheless, *default* pretty-printing is very practical because it automates the major part of creating a formatting tool. Therefore, the formatting process in our framework is split into three stages, see Figure 2.

2.1 Stage 1: user-defined mapping

The input for *Stage 1* consists of a parse tree. The user-defined mapping that is applied to this parse tree consists of transforming particular language constructs to Box constructs. The language constructs that are transformed, depend on the formatting requirements that are not handled as desired by the default mapping in *Stage 2*. Any programming language or tool can be used to map selected language constructs to Box constructs. For our Cobol application we used ASF+SDF [26], because its application domain is in these kind of language transformations. We may use any kind of context information to construct this mapping.

Applying a user-defined mapping on the input parse tree results in a *hybrid* parse tree containing both source language constructs and Box constructs. Figure 3 illustrates such a hybrid tree, where Box language operators can have programming language constructs as children, and vice versa. The borders between the source code language formalism and Box formalism are guarded by encapsulating nodes which are marked by two special node attributes: `from-box` and `to-box`. The outermost pyramid shows a Cobol parse tree that is partially formatted. It has one child that has been transformed to Box constructs. The transition from Cobol to Box is guarded by a `from-box` node.

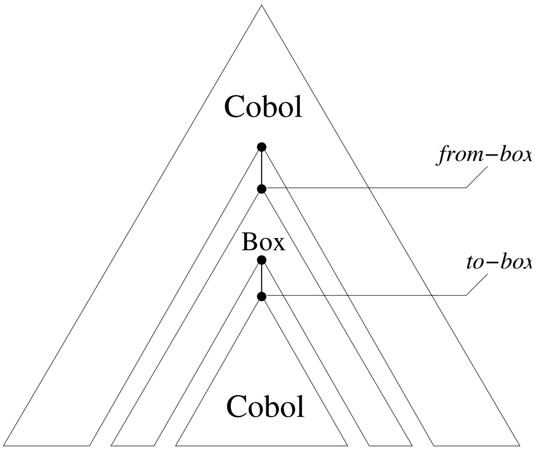


Figure 3. A hybrid tree with guarding `from-box` and `to-box` nodes.

This part of the tree does contain an unformatted Cobol part again. The transition from Box back to Cobol is guarded by a `to-box` node.

2.2 Stage 2: default mapping

In *Stage 2* the default formatting engine applies default pretty-print rules to the hybrid parse tree result of *Stage 1*. All Cobol constructs that are left in the hybrid tree are then mapped to Box operators. The algorithm used by the formatting engine skips over all Box expressions that are between a `from-box` and a `to-box` node, since they have been formatted already (see Figure 3). The resulting tree contains only Box expressions. It is guaranteed that all source language constructs have been transformed into Box constructs.

It is surprising that in general more nodes are formatted in *Stage 2* than in *Stage 1*. Programming languages share typical syntactic idioms that can be formatted in a similar way. The most obvious example is the block structure: a syntax rule that begins and ends with a literal, and has a list of other constructs in the middle. There is an easy opportunity for reuse. *Stage 2* benefits from these similarities by using some smart heuristics. It extracts information from parse trees to identify syntactical idioms, and maps them to Box expressions. We reuse the default mapping that was proposed in [25], but now we implement it on the hybrid tree instead of generating a default implementation that the user needs to adapt. The benefit is twofold: the user can choose the technology he prefers to use for the user-defined formatter and we avoid common maintenance problems with generated code altogether.

2.3 Stage 3: Box back-ends

For *Stage 3* several reusable Box back-ends are available [7, 8, 25] that can be reused to output formatted programs. Using the Box tree from *Stage 2*, the size of every Box will be computed and the output will depend on these sizes.

2.4 Syntax safety

The default formatter in *Stage 2* is constructed in such a manner that it guarantees *syntax safety*. It will not throw away, flip, or invent programs or program parts. With a user-defined formatter, that might use arbitrary computational power we do not have such a guarantee. Therefore, *Stage 1* also contains a tool that correlates the hybrid tree with the original parse tree. If the user-defined formatter terminates, this tool can assert that at least the resulting Box language constructs represent exactly the characters from the original parse tree, and output an error message if not. Note that this tool uses the information in the parse tree with a complete representation of the context-free and lexical grammars.

3 Case study: Cobol layout standardization

In this project, we were driven by an industrial application: the layout of a medium sized DEC Cobol system had to be standardized according to specific conventions (Table 1). We describe what kind of effort was needed to create a formatter that meets all requirements using the above described formatting pipeline (Figure 2). In particular, we describe how we dealt with specific language constructs that had to be standardized. Furthermore, we report on some performance measurements.

3.1 Implementation of the formatting framework

For implementing both the parser and the user-defined formatting we have used the language specification formalism ASF+SDF [26]. This is a formal language that is well-equipped for transformations of source code. Using SDF grammar productions, the syntax of Cobol has been defined. The SDF grammar was derived from the on-line IBM VS Cobol II grammar [12–14] and adapted to be able to parse DEC Cobol-specific constructs; the adaptation was done using the Grammar Deployment Kit [11]. The adapted grammar productions were used to generate a parser for DEC Cobol. The parser outputs a parse tree that can be used in *Stage 1* (see Figure 2).

In the case that the default mapping was different from the formatting convention, ASF rewrite rules have been defined to implement the mapping of Cobol constructs to Box language constructs. These rewrite rules are applied to the parse tree in *Stage 1*.

The Cobol SDF grammar is not discussed here, except for production rules to illustrate a few examples. Instead we focus on the formatter written in ASF. ASF rewrite rules, have a tag, a left-hand side, a right-hand side and optionally some conditions that guard the application of a rewrite rule. If a language construct in a parsed program matches the left-hand side of a rewrite rule, it is replaced by the right-hand side. A rewrite rule may have complex matching patterns. By defining parameterized functions, rewrite rules can also receive context information to guide a transformation. In ASF information flows by passing parameters [23].

As our examples will illustrate, ASF rewrite rules use the concrete syntax of the manipulated language on both sides of a rule and in the conditions. The mapping of Cobol constructs to Box constructs is therefore immediately recognizable as such.

For efficiency reasons, *Stage 2* and *3* are linked together in a new tool called Pandora. Pandora is distributed with the ASF+SDF Meta-Environment [3,10,24] and can be used with and without a user-defined mapping. Having Pandora as a separate tool allows us to maintain and evolve it without influencing user-defined rules too much. Pandora can therefore be implemented in any language. As our industrial application will show, we need most operators of the Box language as presented in Table 2, and one important extension: tab stops. Tab stops support placing of constructs at fixed columns independent of the current indentation level. The crosscutting concern mentioned in Section 1.2, where parts of a language construction have to be placed at fixed columns, has inspired this extension.

3.2 General implementation scheme

The implementation of a pretty-printer involves specifying at least one rewrite rule for each construct that the standardization document describes, unless the default mapping (*Stage 2*) coincides with the standard. We will discuss a number of Cobol constructs with their implementation:

- Data declarations;
- Structured statements (e.g. `IF`);
- Non-structured statements (e.g. `MOVE`).

The general implementation scheme is illustrated by Figure 4. An example construct, with keywords `BEGIN` and `END` is formatted by a single rewrite rule. The syntax is defined by a production rule, and we define some meta variables. We prefix the variables with hash signs in our examples for readability, and use `?` to indicate a variable matches an optional construct. The left-hand side of the rule tagged `format-example-construct` matches all instances of this construct. The conditions of a rule guard its application and may introduce local variables. On the right-hand side of the rule, we replace the construct by a Box expression. In this example we format all parts of the construct vertically using a V box. One of the members of this construct is indented two positions using I is=2.

```
context-free syntax
 "BEGIN" Sort1 Sort2 "END" -> Sort3
context-free syntax
  from-box( Box  )  -> Sort3 {from-box}
  to-box  ( Sort1 ) -> Box   {to-box}
  to-box  ( Sort2 ) -> Box   {to-box}
variables
  "#Sort1" -> Sort1
  "#Sort2" -> Sort2
equations
[format-example-construct]
 ... (optional conditions)
 ===>
 BEGIN #Sort1 #Sort2 END
 =
 from-box(
 V [ "BEGIN"
     to-box(#Sort1)
     I is=2 [to-box(Sort2)]
     "END"
   ])
```

Figure 4. General scheme of mapping a language construct to Box using ASF+SDF.

```
equations
[] IndStatement            =  9
[] IndSecondPartStatement = 25
[] IndDDFirstClause       = 41
[] IndThirdPartStatement  = 49
[] IndDDOtherClauses      = 51
[] IndFourthPartStatement = 53

[] DatanameSpace          =  2
[] IndDataEntry           =  4
[] IndStructured          =  4

[] determine-ind (#Level)
   = ... (not shown)
[] get-group-variable(#Data-desc-entries)
   = ... (not shown)
```

Figure 5. Constant and parameterized functions for column numbers.

The `from-box` and `to-box` constructs (see Section 2.1) mark the borders between the Cobol and Box formalisms and make the hybrid trees type correct. The members of the example construct, captured by the variables `#Sort1` and `#Sort2`, are nested in the Box expression using `to-box` productions. The entire construct is translated to Box, and this Box expression is nested in its surroundings using a `from-box` production. The above scheme is applied to all constructs we format.

We implemented the conventions from Table 1 as a number of constant ASF functions, and some parameterized functions. Figure 5 displays the defined constants, and the left-hand sides of the parameterized functions. determine-ind computes the indentation level of data declarations based on the level number. For example, for 01 it returns 0, and for 03 it returns 4. The function get-group-variable is used to retrieve the level number of previous data declarations. This is a typical example of context-information being used to influence indentation.

In the remainder of this section, we describe some of the rules for a number of Cobol constructs we implemented in our case-study. For each construct, we show the SDF production that defines its syntax, together with the corresponding rewrite rule that defines its formatting.

3.3 Formatting data declarations

In Cobol, there are various types of data declarations. We briefly summarize the syntax and semantics of this construct.

A single data declaration is called a data description entry, and several entries are grouped into a WORKING-STORAGE SECTION. See Figure 6 for the context-free production rules in SDF. An entry consists of a level number (e.g. 01, 03, ...), an optional data-name or FILLER, and zero or more clauses for specifying properties of the data item. In Cobol, a filler is a data item which can not be referred to. Properties of data items are for instance their size, usage, and value. The declaration is terminated by a period. A number of data description entries can be grouped to form a record structure. The grouping is only indicated by the different level numbers, not by the tree structure of the program. An entry with a higher level number is subordinate to an entry with a lower level, i.e., 03 is a sub-record of 01. This is also shown in Figure 1. The picture clause, indicated by the PIC keyword, is used to specify the size of the variable. The value clause, indicated by the VALUE keyword, is used to initialize the value of the variable. A special level number is the level 88 declaration; it can be used as a flag for the associated group variable in conditional statements, such as an IF statement.

In order to formalize the layout standard for data description entries using ASF+SDF, we implemented a number a rewrite rules. To give an idea of what such a rewrite rule contains, we show two of the actual rules for the data description entry from our specification (Figure 6). On the left-hand side of the first rule, we see the syntax defined by the left-hand side of the production rule for Data-desc-entry. On the right-hand side of the rewrite rule, we see the formalization of the layout, expressed with Box operators. The following will help to understand this Box term:

- The first H hs=0 specifies that the parts of a data description should be printed horizontal with no spaces in between, since the spacing is set by tab stops;
- Next, H ts=... defines the tab stop position of the data names or FILLERs. We use the helper function determine-ind that was described earlier. The hs=2 specifies that the #Level and the optional dataname or FILLER are separated by two spaces;
- Then the #Picture-clause is set at a certain tab stop;
- The #Data-desc-entry-clauses are printed at a different tab stop;
- Finally, we close the statement with the atomic box ".".

So the rewrite rule in Figure 6 is used to pretty-print data declarations. In addition to this rule, we implemented several other rules to deal with different flavors of data declarations. For instance, a data description entry without a picture clause requires an additional rule.

Level 88 data declarations are slightly more complicated. Its formatting convention requires context sensitive information. The second rule in Figure 6 deals with 88 declarations. The rule matches a larger context, namely a list of data description entries in a working storage section, instead of a single declaration. This provides the context needed to determine the group variable of the current record. We retrieve this information by applying the function get-group-level-number to the declarations that precede an 88 declaration. Using the retrieved level number, the indentation of the record is determined, and by adding IndDataEntry we calculate the required indentation for the level 88 variable.

3.4 Formatting structured statements

In Cobol there are a number of structured statements. A structured statement is a statement which can contain other statements. Some examples of such statements in Cobol are: IF statement (conditional), EVALUATE statement (switch), PERFORM statement (loop). There are some more structured statements in Cobol, but we do not discuss them here. We focus on the IF statement.

In Figure 7, two SDF production rules show the syntax of the Cobol IF statement, and two ASF rewrite rules show the formatting conventions for IF. In the first rule, the IF statement is divided into three parts, which are printed in vertical mode:

- The first part consists of the IF keyword, the #Condition and the optional THEN. These are printed in horizontal mode. In addition, the #Condition is printed at a certain tab stop using ts=...;
- The second part is the #Statement-list, which is formatted with an indentation using an I box;

```
context-free syntax
 "WORKING-STORAGE" "SECTION" "." Data-desc-entry*        -> Working-storage-section
 Level (Data-name|"FILLER")? Data-desc-entry-clauses "." -> Data-desc-entry

equations
[format-data-desc-entry-using-level-number]
 #Level != 88
 ===>
 #Level #Data-nameOrFILLER? #Picture-clause #Data-desc-entry-clauses,
 =
 from-box(
 H hs=0 [
   H ts=determine-ind(#Level) hs=2 [to-box(#Level) to-box(#Data-nameOrFILLER?)]
   H ts=IndDDFirstClause  [to-box(#Picture-clause)]
   H ts=IndDDOtherClauses [to-box(#Data-desc-entry-clauses)]
   "."
 ])

equations
[format-data-desc-entry-level-88-using-context-information]
 #Level     := get-group-level-number(#Data-desc-entries1),
 #Ind-level := determine-ind(#Level) + IndDataEntry
 ===>
 WORKING-STORAGE SECTION.
   #Data-desc-entries1
   88 #Data-nameOrFILLER? #Data-desc-entry-clauses .
   #Data-desc-entries2
 =
 WORKING-STORAGE SECTION.
   #Data-desc-entries1
   from-box(
   H hs=0 [
     H ts=#Ind-level hs=2  ["88" to-box(#Data-nameOrFILLER?)]
     H ts=IndDDFirstClause [to-box(#Data-desc-entry-clauses)]
     "."
   ])
   #Data-desc-entries2
```

Figure 6. Two SDF production rules for Cobol data declarations, and two conditional ASF rewrite rules that map them to Box.

- The last part is the optional `#Else-phrase`, which is formatted below the `#Statement-list` because of the outermost V.

The second rule from Figure 7 specifies that the closing keyword `END-IF` should be printed in vertical mode using a V box.

3.5 Formatting non-structured statements

In Cobol, a non-structured statement consists of a single statement. For instance, the `ADD`, `COMPUTE` and `DIVIDE` statements without `SIZE ERROR` parts for error handling are non-structured. All such statements are formatted on a single line, using an H box. When possible they are divided into three parts. Each part has its defined tab stop according to the standardization document.

A statement which always belongs to the category of non-structured statements is the `MOVE` statement, since it has no error handling parts. An example is shown below:

```
MOVE A TO B
```

According to the syntax, which is shown in 8, an identifier or literal can be moved to one or more receiving identifiers. In Figure 8, the ASF rewrite rule is shown for formalizing layout of the `MOVE` statement. The rule defines the required layout convention: each part of the statement is printed in a different column using three applications of the H operator with a certain tab stop value.

3.6 Summary

In this case study, we implemented formatting rules to enforce the requested layout standardization on several constructs from the Cobol grammar. We implemented rules for 10 different statements, and a number of rules for the data declarations. In addition, rules were implemented to format divisions, sections and paragraphs. In total, about 50 ASF rewrite rules were implemented to cover the constructs that appeared in the corporate convention (*Stage 1*), the rest of the constructs were formatted according to the default rules (*Stage 2*).

Several rewrite rules, such as for the `MOVE` statements are simple one-to-one mappings. They do not use any con-

```
context-free syntax
 "IF" Condition "THEN"? Statement-list Else-phrase? -> If-statement-non-closed
 If-statement-non-closed "END-IF"                   -> If-statement

equations
[if-statement-non-closed]
 IF #Condition #THEN? #Statement-list #Else-phrase?
 =
 from-box(
 V [ H [ "IF" H ts=IndSecondPartStatement [to-box(#Condition)] to-box(#THEN?)]
     I is=IndNestedStatement [to-box(#Statement-list)]
     to-box(#Else-phrase?)
 ])


[if-statement]
 #If-statement-non-closed END-IF
 =
 from-box(
 V [ to-box(#If-statement-non-closed)
     "END-IF"
 ])
```

Figure 7. Two SDF production rules for the Cobol `IF` statement, and two ASF rewrite rules to formalize its layout standard.

```
context-free syntax
 "MOVE" ( Identifier | Literal ) "TO" Identifier-list -> Move-statement

equations
[move-statement]
 MOVE #IdentifierOrLiteral TO #Identifier-list
 =
 from-box(
 H [ "MOVE"
     H ts=IndSecondPartStatement [to-box(#IdentifierOrLiteral)]
     H ts=IndThirdPartStatement  ["TO"]
     H ts=IndFourthPartStatement [to-box(#Identifier-list)]
 ])
```

Figure 8. An SDF production rule for the Cobol `MOVE` statement, and the ASF rewrite rule to formalize its layout standard.

text information, but they do use the tab stop feature of the Box language, to be able to define alignment that crosscuts the logical structure of a program. Note that the absolute tab stop feature is not available in most implementations of the Box language.

On the other hand, with the Cobol data declarations we have shown how to use context information and how to assign specific semantics to selected level numbers. We used freedom of applying arbitrary computational power to analyze a program and influence its formatting.

3.7 Performance

We measured the performance of application of formatting 78 thousand lines of Cobol code. Parsing 78 KLOC was done in 420 seconds. *Stage 1* using compiled rewrite rules, took only 22 seconds, while Pandora took 74 seconds to perform *Stage 2* and *3*, of which 32 seconds are spent by the *default* mapping. The above measurements show that formatting 78 thousand lines of code using this framework is feasible.

4 Conclusions

We have taken a given set of formatting requirements for a Cobol system as spelled out in a standardization document, and applied generic formatting technology to implement them. It appeared that corporate conventions can dictate alignment that crosscuts the logical structure of a program, and can even dictate indentation that needs to be computed from context information.

We have developed and implemented a formatting framework that allows arbitrary computational power for mapping language constructs to the Box language. The enabling feature is a hybrid format that merges Box expressions with parse trees. Much of the boilerplate part of formatting can still be automated by a default mapping to Box. Absolute tab stops, an important feature which is not found in many Box back-ends, are used extensively in our case study.

5 Acknowledgments

This research was done at the Centrum voor Wiskunde en Informatica and the Vrije Universiteit. The case study was carried out at the Vrije Universiteit in cooperation with PinkRoccade The Netherlands. We would like to thank Steven Klusener and Paul Klint for their contributions. We also like to thank Rob Economopoulos and Chris Verhoef for proofreading this paper.

The research was supported by the Dutch Ministry of Economic Affairs via contract SENTER-TSIT3018 *CaLCE: Computer-aided Life Cycle Enabling of Software Assets*.

References

[1] I. Attali, C. Courbis, P. Degenne, A. Fau, and D. Parigot. Smarttools: a generator of interactive environments tools. In R. Wilhelm, editor, *Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 355–360. Springer-Verlag, Genova, Italy, April 2001.

[2] I. D. Baxter, C. Pidgeon, and M. Mehlich. Dms: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[3] Centrum voor Wiskunde en Informatica. The ASF+SDF Meta-Environment, 2005. http://www.cwi.nl/projects/MetaEnv/.

[4] J. R. Cordy. Txl — a language for programming language tools and applications. In G. Hedin and E. van Wyk, editors, *4th International Workshop on Language Descriptions, Tools and Applications (LDTA'2004)*, Electronic Notes in Theoretical Computer Science, pages 1–27. Elsevier, 2004.

[5] J. R. Cordy, I. H. Carmichael, and R. Halliday. *TXL — Programming Language*, 2005. Version 10.4.

[6] J. Coutaz. The box, a layout abstraction for user interface toolkits. Technical report, Carnegie Mellon University, 1984. CMU-CS-84-167.

[7] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, June 2000.

[8] M. de Jonge. Pretty-printing for software reengineering. In *Proceedings: International Conference on Software Maintenance (ICSM 2002)*, pages 550–559. IEEE Computer Society Press, Oct. 2002.

[9] GNU. indent, 2005. http://www.gnu.org/software/indent.

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

[11] J. Kort, R. Lämmel, and C. Verhoef. The grammar deployment kit. *Electronic Notes in Theoretical Computer Science*, 65(3), 2002.

[12] R. Lämmel and C. Verhoef. VS Cobol II Grammar Version 1.0.3, 1999. http://www.cs.vu.nl/grammars/vs-cobol-ii/.

[13] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

[14] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[15] S. McConnel. *Code Complete*. Microsoft Press, 1993.

[16] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *ACM*, 26(11):861–867, 1983.

[17] E. Morcos-Chounet and A. Conchon. PPML: a general formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier, 1986.

[18] J. M. Neighbors. *Software Construction using Components*. PhD thesis, University of California, Irvine, Department of Information and Computer Science, 1980. Technical Report UCI-ICS-TR-160.

[19] D. C. Oppen. Prettyprinting. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980.

[20] PinkRoccade. *Formatting conventions (internal document)*, 2004.

[21] D. V. Schorre. Meta ii a syntax-oriented compiler writing language. In *Proceedings of the 1964 19th ACM national conference*, pages 41.301–41.3011, New York, NY, USA, 1964. ACM Press.

[22] Software & Solutions. Jindent, 2005. http://www.jindent.com.

[23] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[24] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[25] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, 1996.

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

[27] K. Vos. PRETTY for an easy touch of beauty. Master's thesis, Universiteit van Amsterdam, 1990.