Centrum voor Wiskunde en Informatica

A Pretty-Printer for Every Occasion

M. de Jonge

Software Engineering (SEN)

SEN-R0115 May 31, 2001

Report SEN-R0115
ISSN 1386-369X

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

A Pretty-Printer for Every Occasion

Merijn de Jonge
email:mdejonge@cwi.nl

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

ABSTRACT

Tool builders dealing with many different languages, and language designers require sophisticated pretty-print techniques to minimize the time needed for constructing and adapting pretty-printers. We combined new and existing pretty-print techniques in a generic pretty-printer that satisfies modern pretty-print requirements. Its features include language independence, customization, and incremental pretty-printer generation.

Furthermore, we emphasize that the recent acceptance of XML as international standard for the representation of structured data demands flexible pretty-print techniques, and we demonstrate that our pretty-printer provides such technology.

1998 ACM Computing Classification System: D.2.3, D.2.7, D.2.13
Keywords and Phrases: pretty-printing, documentation, languages, XML, tool construction, software engineering

1. INTRODUCTION

Pretty-printing is concerned with formatting and presentation of computer languages. These languages include ordinary programming languages and languages defining data structures. XML [6], recently accepted as international standard for the representation of structured data, brings formatting issues (related to the transformation of XML documents to user-readable form) towards a broad community of tool builders.

These tool builders as well as language designers demand advanced pretty-print techniques to minimize the time required for developing new or adapting existing pretty-printers. For both it is essential to maximize language independence of pretty-printers and to be able to add support for new languages easily. Moreover, pretty-printers should minimize code duplication, be customizable, extensible, and easy to integrate.

Most pretty-print technology used in industry today does not meet these requirements. This lack of sophisticated technology makes development and maintenance costs of pretty-printers high. Despite the academic research in this field which has yielded advanced pretty-print techniques, we observe that these techniques have not come available for practical use yet.

In this paper we combine new and existing techniques to form a pretty-print system that satisfies modern pretty-printer requirements. It features language independence, extensibility, customization, pretty-printer generation, and it supports multiple output formats including plain text, HTML, and LaTeX. Furthermore, the pretty-printer can easily be integrated in existing systems and is freely available.

This article is organized as follows. Section 2 describes several aspects of pretty-printing by summarizing earlier work in this field. In Section 3 we describe the design and implementation of the generic pretty-printer GPP. Several case studies are discussed in Section 4. Section 5 explains how our pretty-printer can be used to format XML documents depending on their document type definition (DTD) and how it may function as alternative to the extensible style language (XSL). Contributions and future work are addressed in Section 6.

2. STATE OF THE ART

Traditionally, mostly ad-hoc solutions have been used to cope with the problem of formatting computer languages. Not only were traditional pretty-printers bound to specific languages, they also contained hard-coded

formatting rules which made them non-customizable.

The first general solution to the pretty-print problem was formulated by Oppen [20]. He described a *language independent* pretty-print algorithm operating on a sequence of logically continuous blocks of strings. The division of the input by delimiters (either block delimiters or white space) provides information about were line breaks are allowed.

Oppen also introduced *conditional formatting* to support different formattings when a block cannot fit on a single line. He distinguishes inconsistent breaking, which minimizes the number of newlines that are inserted in a block to make it fit within the page margins, and consistent breaking, which maximizes the number of newlines. Conditional formatting has been adopted in most modern pretty-printers.

In addition to Oppen, many language independent pretty-print *algorithms* are described in the literature. Traditional algorithms which are more or less similar to Oppen's include [23, 18, 24, 19, 30]. A consequence of conditional formatting is an exponential growth of the possible formattings. While the traditional algorithms only consider a small subset of these formattings in order to limit execution time, more advanced formatting algorithms are designed in the community of functional programming [12, 26, 14, 34]. These algorithms heavily depend on lazy evaluation to abstract over execution time. This allows the pretty-printers to select an optimal formatting in a lazy fashion from all possible ones.

Several formatting primitives have been suggested as alternative to the blanks and blocks of Oppen. Modern pretty-printers describe formatting in terms of *boxes* (as introduced by [15] and [18]). PPML[19] defines a formalism based on boxes to define the structuring of displays. It introduces different types of boxes for different formatting. Examples are the h box for horizontal formatting and v for vertical formatting. Based on PPML, [30] introduces the language BOX, mainly to solve some technical problems of PPML. Another similar approach to PPML is described by Boulton [5]. He describes a formalism to annotate a grammar with, among others, abstract syntax and formatting rules. The syntax for specifying formatting is based on PPML.

Oppen [20] observed that the process of pretty-printing can be divided in a language dependent *front-end* for the translation of a program text to some language independent formatting, and a language independent *back-end* which translates the language independent formatting to an output format. All current pretty-printers that we are aware of follow this structure.

The division of a pretty-printer in a front-end and back-end not only makes a back-end language independent, it also makes a front-end output format independent. Despite this fact, by far the most back-ends that are described in the literature concentrate on the translation from a language independent input term to plain text. Articles which address the translation to other output formats include [19, 30, 27].

A nice formatting is a question of style and personal taste [16]. Blaschek and Sametinger [4] emphasize that the ability to *customize* the generated output of a pretty-printer to one's favorite style can improve the readability and maintainability of programs significantly. Customizing existing pretty-printers mostly requires changing the code manually, or modifying the formatting rules as annotations of the grammar (which, as a result, also modifies the grammar). An ordinary user cannot be expected to perform such modifications. A more user-friendly approach of customizable pretty-printing is described by [4]. They introduce user-adaptable pretty-printing using personal profiles which provide individual formatting rules for general language constructs.

A front-end for a language can be constructed by hard-coding the formatting rules manually, or be generated from a grammar annotated with formatting rules. The first approach is most commonly used, for example in [13, 19]. The latter approach, suggested by Oppen (who emphasized the importance of separating pretty-print information from code), is used in [23, 24, 5].

A front-end can also be generated from a grammar without annotated format rules by a pretty-printer *generator* that analyses the structure of a grammar to "guess" a suitable layout. Despite the usefulness of such generators in environments where a large number of evolving languages are used, little work has been carried out on this topic. The only pretty-printer generator that we are aware of is described in [30]. They describe a generator which produces dedicated, language specific front-ends. These front-ends contain formatting rules and the code to perform the formatting. The actual formatting can be customized by adapting or extending the generated code. Their approach yields highly customizable formatters but the formatters are language dependent and customization requires modifying the generated code (and thus requires understanding the generated code).

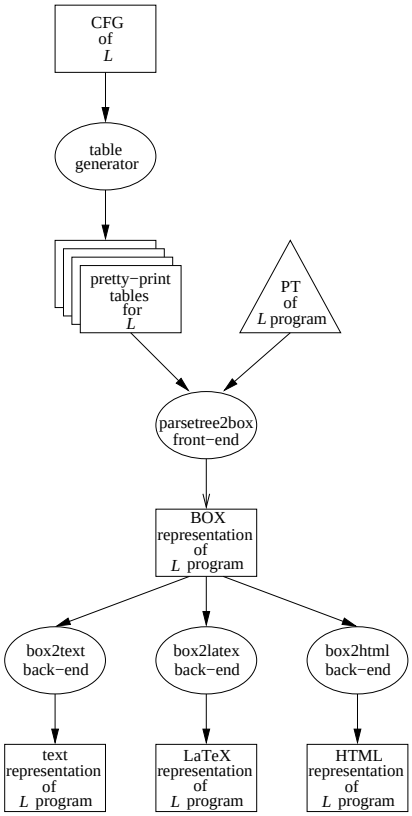


Figure 1: An overview of the generic pretty-printer GPP. It consists of a table generator, a front-end (`parsetree2box`), and three back-ends which produce plain text, HTML and LaTeX, respectively.

3. A PRETTY-PRINTER FOR EVERY OCCASION

Despite all research on the topic of pretty-printing, most pretty-printers that are used in practice are language specific, inflexible, and support only a very restricted number of output formats. Moreover, for many languages not even a pretty-printer exists. Adding support for a new language or a new output format often means implementing a new pretty-printer from scratch. This is not only a time consuming task, but also introduces much code duplication which increases maintenance costs.

On the other hand, more advanced pretty-printers that have been developed as part of research projects are often incomplete (because they only address a limited number of pretty-print aspects), or are tightly coupled to a particular system [19, 30] which make them hard to use in general.

Summarizing, there is a great need for advanced pretty-print techniques in industry which are flexible, customizable, easy to use, and language independent. Despite the research in this field there are currently no such pretty-printers for practical use available.

In the remainder of this section we will describe the architecture and design of the generic pretty-printer GPP which satisfies modern pretty-print requirements. The pretty-printer is language independent and divided in front-ends and back-ends to make future extensions easy to incorporate. A box based intermediate format (called BOX), which supports comment preservation and which is prepared for incremental and conservative pretty-printing [25], is used to define the formatting of languages and to connect front-ends with back-ends. Furthermore, the pretty-printer uses new techniques to support customization of pretty-printers (based on reusable, modular pretty-print tables), and incremental pretty-printer generation. We support multiple output formats including plain text, HTML, and LaTeX. Finally, the pretty-printer can be integrated easily in existing

operator	*options*	*description*
H	hs	Formats its sub-boxes horizontally.
V	vs, is	Formats its sub-boxes vertically.
HV	hs, vs, is	Inconsistent line breaking. Respects line width by formatting its sub-boxes horizontally and vertically.
A	hs, vs	Formats its sub-boxes in a tabular.
ALT		Depending on the available width, formats its first or second sub-box.

Table 1: Positional BOX operators and supported space options (hs defines horizontal layout between boxes, vs defines vertical layout between boxes, and is defines left indentation).

systems, or be used stand-alone and is freely available. Figure 1 gives a general overview of the architecture of GPP.

3.1 An open framework for pretty-printing

We followed the well-known approach of dividing a pretty-printer in a language dependent *front-end* and a language independent *back-end*. This allows for an open pretty-print system which can easily be extended to support new languages and output formats. A front-end for language L expresses the language specific layout of L in terms of a generic formatting language. A back-end producing output format O translates terms over this formatting language to O. A pretty-printer for L producing O as output can now be constructed by connecting the output of the L specific front-end to the input of the back-end for O. This architecture thus isolates language specific code in the front-end and output format dependent code in back-ends. Adding support for a new language only requires developing a new front-end for the language, likewise, to add support for a new output format, only a new back-end has to be developed.

We used the domain specific language BOX [30] to connect the output of front-ends to the input of back-ends (see Section 3.2 for a description of the BOX language). By using BOX to glue front-ends and back-ends, the framework allows any BOX producer to be connected to any BOX consumer. This flexibility allows a whole range of front-ends and back-ends of different complexity to be connected to the pretty-print framework. For example, multiple front-ends for a single language may exist simultaneously, providing different functionality or different quality. One of them might be optimized for speed, performing only basic formatting for instance, while another is designed to produce optimal results at the cost of decreased performance.

3.2 The box markup language

BOX is a language independent markup language designed to describe the intended layout of text. Being a box-based language, it allows a formatting of text to be expressed as a composition of horizontal and vertical boxes. BOX is based on PPML[19] and contains similar operators to describe layout and conditional operators to define formatting depending on the available width. In addition to PPML, BOX supports tables, fonts, and formatting comments. In the remainder of this section we will give a brief overview of BOX (for a more complete description of the language we refer to [27]).

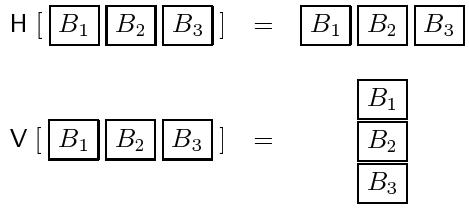
A term over the BOX language consists of a nested composition of boxes. The most elementary boxes are strings, more complex boxes can be constructed by composing boxes using *positional operators* and *non-positional operators*. The first (see Table 1 for a list of available positional operators) specify the relative positioning of boxes. The latter (see Table 2) specify the visual appearance of boxes (by defining color and font parameters), define labels, and format comments.

Examples of positional operators are the H and V operators, which format their sub-boxes horizontally and

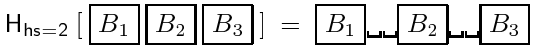
operator	*description*
F	Operator to specify fonts and font attributes.
KW	Font operator to format keywords.
VAR	Font operator to format variables.
NUM	Font operator to format numbers.
MATH	Font operator to format mathematical symbols.
LBL	Operator used to define a label for a box.
REF	Operator to refer to a labeled box.
C	Operator to represent lines of comments.

Table 2: Non-positional BOX operators.

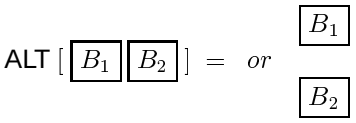
vertically, respectively:



The exact formatting of positional box operators can be controlled using *space options*. For example, to control the amount of horizontal layout between boxes, the H operator supports the hs space option:



BOX as we use it slightly differs from its initial design as described in [30]. We simplified the language (mainly to improve comment handling) and made it more consistent. Furthermore, we introduced a generalization of the conditional HOV operator. This operator, which is available in some form or another in most formatting languages, formats its contents either completely horizontally or completely vertically depending on the available width (consistent line breaking). We introduced as generalization the ALT operator:



This operator chooses among two alternative formattings depending on the available width. It chooses for its first sub-box when sufficient space is available and for its second sub-box otherwise.

3.3 Pretty-print tables

We introduce the notion of interpreted formatting in which a front-end (see Section 3.4) formats its input by interpreting a set of language specific formatting rules. Formatting rules and code are separated by defining the formatting rules in pretty-print tables. Each formatting rule forms a mapping of the form $p_L \longrightarrow b$ (where p_L denotes a production of the grammar of the language L and b denotes the corresponding BOX expression) and specifies how the language construct p_L should be formatted.

Representing formatting rules in tables instead of having a single dedicated pretty-printer that contains all pretty-print rules for a language provides the following advantages. First, tables support a modular design of pretty-printers. As a consequence, a pretty-printer can follow the same modular structure as the corresponding modular grammar and re-use is promoted. Second, pretty-print tables promote incremental pretty-printer

```
"package" Name ";" → PackagedDeclaration — H [KW["package"] H hs=0 [_1 ";"]],
"import" Name ";" → ImportDeclaration — H [KW["import"] H hs=0 [_1 ";"]],
"import" Name "." "*" ";" → ImportDeclaration — H [KW["import"] H hs=0 [_1 "." "*" ";"]]
```

Figure 2: A sample of a pretty-print table. The table contains mappings from grammar productions in SDF (on the left-hand side of '—') to corresponding BOX expressions (on the right-hand side of '—').

generation. When one or more modules of a modular grammar are modified, only the tables corresponding to the modified modules have to be re-generated. Third, tables allow easy personal customization by separating globally defined or generated formatting rules, and customized rules in different tables. Defining an ordering on tables determines which formatting rule should be applied when multiple rules exist for a single language construct. It allows a user to customize the pretty-printer by defining additional rules with higher precedence. Fourth, the separation of formatting rules in tables allows for a generic BOX producer which, when instantiated with language specific pretty-print tables, performs language specific formatting (see Section 3.4).

We use the syntax definition formalism SDF [11] to express language constructs in pretty-print tables. SDF in combination with generalized-LR parser generation [22] offers advanced language technology that handles the full class of context-free grammars. By using this technology in the pretty-printer we also obtain pretty-print support for this class of grammars. In addition to SDF, the general idea of pretty-print tables containing mappings from language constructs to BOX expressions can easily be implemented for other syntax definition formalisms (like BNF) or XML as well.

Figure 2 shows an example of a pretty-print table which defines a format for three language constructs of the programming language Java. The first entry in the table defines a formatting for PackagedDeclaration[1]. This language construct consists of the terminal symbols package and ';', and the non-terminal symbol Name. The formatting rule expresses that these three elements are layout horizontally, that package is formatted as keyword, and that no white space is inserted between the non-terminal Name and the semicolon. Observe the use of the numbered place holder ('_1') to denote the BOX expression corresponding to the formatted non-terminal symbol Name. The remaining formatting entries define similar formattings for the two import declaration constructs of Java.

3.4 A generic box producer

We designed a generic, language independent front-end which applies formatting rules defined in an ordered sequence of pretty-print tables to a parse tree. Separating the language specific formatting rules in tables allows the generic front-end to be re-used unmodified to format any language. Constructing a pretty-printer for a new language only requires language specific formatting rules to be defined in tables.

The front-end operates on a universal format for the representation of parse trees (called AsFix [9]), which preserves layout and comments. Operating on parse trees in general has the advantage that lexical information for disambiguation is available. Therefore we do not have to deal with the insertion of brackets to disambiguate the generated output[2]. Because AsFix is a universal parse tree format, it can represent parse-trees for any language and therefore allows generic parse-tree operations to be defined in language independent tools. As a result, the transformation of a parse tree to BOX can be defined language independently in the single tool `parsetree2box` (see Figure 1). Using AsFix has the additional advantage that all layout is preserved in the tree which simplifies comment handling.

The front-end `parsetree2box` constructs a BOX term for a parse tree of a language by traversing the parse tree in depth first order and simultaneously constructing a BOX term according to the language specific formatting rules in the pretty-print tables. For each node in the tree that corresponds to a production of the language `parsetree2box` searches the tables for the corresponding BOX expression. When a format rule for a production does not exist, `parsetree2box` automatically generates a default rule (this approach makes

[1] Please note that productions in SDF are reversed with respect to formalisms like BNF. On the right-hand side of the arrow is the non-terminal symbol that is produced by the symbols on the left-hand side of the arrow.

[2] We do not consider constructing valid parse trees (i.e., parse trees containing all lexical information for disambiguation) as part of pretty-printing. In case a tree is not constructed by a parser directly, disambiguation (like described in [30] and [21]) might be needed and has to be performed by third party tools.

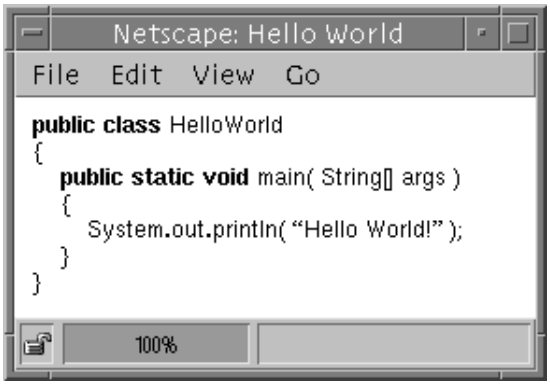


Figure 3: A screen dump showing the result of the HTML code of a Java code fragment as produced by `box2html`.

pretty-print entries optional because simple formattings are constructed dynamically for missing entries). The BOX term thus obtained is then modified to include original comments, and is instantiated with BOX terms representing the formatted non-terminal symbols of the production. Original comments are restored by inserting C boxes (containing the textual representation of comments) in the BOX term, and by positioning these comment boxes using the H and V operators to preserve their original location.

3.5 Pretty-printer generation

Constructing a pretty-printer for a language by hand is a time consuming task. The ability to quickly and easily obtain pretty-printers becomes more and more important when the number of languages and dialects in use increases. For example, development of domain specific languages (DSLs), and language proto-typing requires the use of a large number of pretty-printers and demands enhanced technology for the construction of pretty-printers.

Pretty-printer generation, based on grammars without annotated format rules, is such technology. This technology supports the generation of a pretty-printer for a language by “guessing” a suitable layout based on grammar analysis and formatting heuristics. Obviously, the result of such generated pretty-printers will not satisfy completely in most cases and the ability to adapt generated pretty-printers strongly increases the usefulness of the generator and its generated formatters.

In addition to the pretty-printer generator described in [30], which produces dedicated, language specific front-ends, we introduce an alternative technique for the generation of pretty-printers which benefits from the table based pretty-print approach. Due to the separation of language specific formatting rules and generic code to perform a formatting, there is no need to generate any code. Only pretty-print tables have to be generated and the generic formatting engine `parsetree2box` can be re-used for each language to perform the actual formatting. This approach completely separates data (the pretty-print tables) and code (the generic formatting engine). The user can customize the formatting by overruling generated formatting rules in tables with higher precedence (see Section 3.3).

In our approach, a pretty-printer generator only consists of a table generator. We developed such a table generator which constructs a separate pretty-print table for each module of a modular SDF grammar. The generator currently only uses simple techniques to generate formatting rules for a language. Improving the generation process by using more advanced heuristics and grammar analysis is a current research topic. Another approach to improve the generated pretty-print tables would be to guide the generation process by means of user profiles (similar to [4]).

3.6 Box consumers

A back-end transforms a language independent BOX term to an output format. The advantage of using GPP depends on the number of available output formats. GPP currently supports the output formats plain text, HTML,

```
public class HelloWorld
{
  public static void main( String[] args )
  {
    System.out.println( “Hello World!” );
  }
}
```

Figure 4: The result of formatting a Java code fragment using the back-end `box2latex`.

and LaTeX, which are produced by the back-ends `box2text`, `box2html`, and `box2latex`, respectively. PDF can also be generated but indirectly from generated LaTeX code.

From the three back-ends `box2text` is the most complicated because it has to perform all formatting itself. The translation to HTML and LaTeX is less complicated because the actual formatting is not performed by the back-end but by a WEB browser or LaTeX. The implementation of these back-ends therefore consists of a translation from a BOX term to native HTML or LaTeX code.

The translation to text consists of two phases. During the first phase the BOX term is normalized to contain only horizontal operators, vertical operators, and comments. During the second phase the simplified BOX term is translated to text and the final layout is calculated.

The formatting defined in a BOX term is expressed in HTML as a complex nested sequence of HTML tables. In contrast to BOX, HTML is designed to format a text logically (consisting of a title, a sequence of paragraphs etc.), not as a composition of horizontal and vertical boxes. Only the use of HTML tables (in which individual rows correspond to horizontal boxes and tables to vertical boxes) yielded a correct HTML representation of the formatting defined in a BOX term. Figure 3 shows a screen dump of a pretty-printed Java code fragment produced by `box2html`.

LaTeX code, representing the formatting defined in a BOX term, is obtained by translating the BOX term to corresponding BOX specific LaTeX environments. These environments provide the same formatting primitives as BOX in LaTeX. As an additional feature, `box2latex` allows one to define a translation from BOX strings to native LaTeX code. This feature is used to improve the final output, for instance by introducing mathematical symbols which were not available in the original source text (for example, it allows one to introduce the symbol ‘ϕ’ in the output where the word `phi` was used in the original source text). Figure 4 shows the result of processing a small Java code fragment by `box2latex`.

3.7 Implementation

For the implementation of the individual tools of GPP we combined modern parsing techniques with compiled algebraic specifications. The parsing techniques, based on SGLR (scannerless generalized-LR) parsing [32], allow us to easily define and adapt grammars and automatically generate parsers from them. The basic functionality of the individual tools is implemented as a number of executable specifications in the algebraic specification formalism ASF+SDF [11, 3, 31]. From these specifications we obtained C code by compiling the specification using the ASF+SDF compiler [28]. The generated C code is efficient and gives a promising performance of GPP despite of its interpreted approach based on pretty-print tables, and its implementation as algebraic specification.

The generated parsers and compiled specifications are glued together into a single component using Unix scripts. We use `make` in combination with dynamically generated Makefiles as performance improvement, to prevent doing redundant work.

In order to process files as produced by `box2latex` by `latex`, the style file `boxenv` is required which contains the implementation of the BOX specific environments. For a general usage of this style file and for an in-depth discussion of its implementation we refer to [7].

4. CASE STUDIES

4.1 Formatting real-world languages

We experimented with the pretty-printer and its generator and constructed pretty-printers for some real-world languages. These languages include the programming language Java and the extensible markup language XML. An application of the pretty-printer in industry is its use as formatter for Risla [2], a domain specific language for describing financial products.

For the Java pretty-printer we first constructed a grammar in SDF according to the Java Language Specification [10]. Then we generated pretty-print tables from this grammar. Finally, we customized the pretty-printer manually to meet our requirements. Figure 3 and Figure 4 show the result of formatting a small Java program. Figure 3 is obtained by using `box2html`, for Figure 4 we used `box2latex`.

The XML formatter is another application of GPP for real-world languages. Its development was very similar to the construction of the Java formatter. We first constructed a grammar from XML in SDF according to [6], then we generated and customized pretty-print tables. Thanks to the table based approach, we were able to re-use these tables for the pretty-printer of the language depicted in Figure 5 and 6. Similar to the grammar of this language, which combines the languages XML and BOX (see Section 5), we were able to also construct a corresponding pretty-printer for this language by combining (and re-using) the pretty-printers of XML and BOX.

4.2 Tool construction

The individual components of GPP provide basic language independent pretty-print facilities. These components can easily be used in combination with additional software to construct advanced special-purpose tools. We have combined these generic tools for instance, with language specific features to form two advanced formatting engines for the algebraic specification formalism ASF+SDF [3, 11, 31]. The tool `tolatex` generates a modular LaTeX document from an ASF+SDF specification by formatting each individual module incrementally, and combining them to form a single document with a table of contents and cross references between modules. Similarly, the tool `tohtml` generates hyper-linked HTML documents from a modular specification, featuring visualization of the import structure of the specification and hyper-links between modules.

Other examples of the use of the individual components for tool construction include the integration of GPP in the interactive ASF+SDF Meta-Environment [29], and its integration and distribution as part of XT [8], a distribution of tools for the construction of program transformation systems.

5. FORMATTING XML DOCUMENTS

The extensible markup language XML [6] is a universal format for the abstract representation of structured documents and data. Pretty-print techniques are used to transform XML documents to user-readable form. Formatting XML documents is being standardized in the extensible style language XSL [1]. The combination of XML and XSL separate content (XML) from format (XSL). Since the intended use of XML initially was limited to WEB documents, techniques for pretty-printing XML documents mostly concentrated on the transformation to HTML.

We expect that the need to represent XML documents in other formats than HTML will grow rapidly. Moreover, alternatives to XSL are desirable because the translation from XML to HTML using XSL is considered to be difficult [17]. Although XSL is powerful, its design might prove to be unnecessarily difficult for the common case and thus makes more simple pretty-print techniques sensible.

Our pretty-printer provides such techniques and combined with its ability to produces different output formats makes it suitable for formatting XML documents.

5.1 Using box to format xml documents

The Document Type Definition (DTD) of an XML document defines the structure of a document. The DTD of an XML document can thus be seen as language definition or grammar, and its contents as a term over that language.

A pretty-printer for a language can be constructed by defining mappings from language productions to BOX expressions. Similarly, a pretty-printer for a particular DTD can be constructed by defining mappings from DTD

```
<!ELEMENT person (name, surname, age)> —
    V is=3 ["person" _1 _2 _3],
<!ELEMENT name (#PCDATA)> —
    H ["name: " _1],
<!ELEMENT surname (#PCDATA)> —
    H ["surname: " _1],
<!ELEMENT age (#PCDATA)> —
    H ["age: " _1]
```

Figure 5: A simple XML DTD annotated with BOX formatting rules

constructs to BOX. Once such pretty-print tables have been defined, well-formed XML documents over that DTD can be transformed to all output formats for which a back-end is available.

Example 5.1 In Figure 5 we define a simple DTD which structures personal data (name, surname, and age). The DTD is annotated with BOX formatting rules. These rules formulate that the contents of records should be formatted vertically, left indented, and preceded by the string "person".

Below the textual representation of a typical well-formed document over this DTD is displayed after formatting by `box2text`:

```
person
   name: Johny
   surname: Walker
   age: 5
```

Of course, the formatting can be improved, for instance by using tables to align field names and field values.

Example 5.1 demonstrates that the use of BOX as formatting language in combination with XML, and the use of the available back-ends allows XML documents to be formatted easily.

Currently, we do not support formatting rules to be defined as annotations of a DTD directly (as we did in Figure 5). Instead, we first generate an SDF grammar from a DTD, then we use the SDF grammar to generate a pretty-print table. This indirection allows us to experiment with XML by using existing pretty-print tools, minimizing the need for additional software.

5.2 An alternative style language

The obvious way to transform an XML document to HTML currently is by using XSL stylesheets. An XSL stylesheet specifies how particular documents should be presented in terms of some XML formatting vocabulary. An XSL stylesheet thus describes a structural transformation between the original document and the formatting vocabulary. HTML is used as formatting vocabulary when an XML document has to be transformed into a traditional WEB document.

In spite of its advantage of separating presentation and content, and its expressive power, we agree with [17] that XSL is difficult. First because the language uses the XML syntax which make XSL stylesheets difficult to read. Furthermore, the language is large as a result of the intention to make XSL stylesheets generally applicable. Finally, the combination of a formatting language and a transformation language makes XSL stylesheets complex and difficult to maintain because one has to deal with formatting and transformation issues (by means of tree traversals) simultaneously.

We think that these negative aspects make XSL stylesheets too difficult for many simple transformations. Separation of traversals and presentation, and a less complex language would ease describing simple presentations of XML documents.

With `parsetree2box` simple presentations of XML documents can be defined based on an implicit traversal of the parse-tree. Pretty-print tables are suitable to express a formatting in terms of a formatting vocabulary other than BOX. The combination of an implicit traversal and pretty-print tables as little language to express a transformation to HTML thus forms an alternative to XSL for simple formatting purposes.

<**!ELEMENT** *person* (*name*, *surname*, *age*)> —
“<html>” “<head>” “<title>” _1 _2 “</title>” “</head>” “<body>” _1 _2 “ and ” _3 “</body>”
“</html>”,
<**!ELEMENT** *name* (#PCDATA)> —
“my name is ” _1,
<**!ELEMENT** *surname* (#PCDATA)> —
_1,
<**!ELEMENT** *age* (#PCDATA)> —
“I am ” _1 “years old”

Figure 6: Pretty-print tables used as language to define a simple transformation from XML to HTML.

Example 5.2 Example 5.1 demonstrated how formatting in terms of horizontal and vertical boxes can be defined for a DTD. Formatting a document according to these rules yields an unstructured representation of the document. Figure 6 shows how pretty-print tables can also be used to define a structured representation in terms of HTML.

The mappings in Figure 6 define for each production of the XML DTD the corresponding HTML code. Formatting a well-formed document using `box2text` according to these rules will yield:

```
<html>
  <head>
    <title>
      my name is Johny Walker
    </title>
  </head>
  <body>
    my name is Johny Walker
    and I’m 5 years old
  </body>
</html>
```

This HTML document can then be displayed by the user using an HTML browser.

Example 5.2 demonstrates how simple transformation rules of XML documents can be separated from code that defines traversals. This provides, in combination with the implicit tree traversals of `parsetree2box`, a simple formatting mechanism of XML documents and may serve as alternative to XSL.

For more complex transformations where implicit traversals are too restricted, we are planning to investigate on using languages designed primarily for transformations as alternative to XSL. An example of such a language is Stratego [33], which has more powerful transformation facilities and a better syntax. We expect that both will help to improve readability and maintainability.

6. Concluding Remarks

6.1 Contributions

In this paper we described the design, implementation, and use of the generic language independent pretty-printer GPP. The system can easily be extended in order to add support for more languages or more output formats. It can also easily be adapted to extend pretty-print support for existing languages. The system combines known techniques (like language independent pretty-printing, division of pretty-printers in front-ends and back-ends, and pretty-printer generation) with new techniques to provide advanced pretty-print support. Our contributions are: i) Formulation of formatting rules in pretty-print tables, which allows for a modular pretty-printer design and supports incremental pretty-printer generation. ii) Customization of pretty-printers by means of ordered pretty-print tables. iii) We developed a generic format engine (`parsetree2box`) which operates on a universal parse tree format and interprets language specific format rules contained in pretty-print tables. iv) We designed a table generator which generates pretty-print tables for a language by inspecting

the corresponding grammar. v) We implemented three back-ends which make plain text, HTML, and LaTeX output available for all formatters. vi) The pretty-printer is designed as stand-alone system and can therefore easily be integrated in third-party systems. Moreover, the system is free and can be downloaded from `http://www.cwi.nl/~mdejonge/gpp/`.

Furthermore, we discovered that XML is a relative new application area of pretty-printing. We experimented with XML and we found two useful applications of our pretty-printer. First, the pretty-printer can be used to easily format an XML document depending on its DTD and to translate it to several different output formats. Second, the pretty-printer can be used for simple term transformations as alternative to XSL. For complex transformations we suggest using more advanced transformation systems (like the programming language Stratego for instance) as alternative to XSL.

6.2 Future work

This pretty-print project was initiated as part of the development of a new ASF+SDF Meta-Environment and its integration as default formatter was the intended goal. The integration of the pretty-printer in this interactive programming environment is not finished yet but is planned to be completed soon.

The table generator is the one component of the pretty-print system that still needs additional research. This research includes experimenting with more advanced heuristics and grammar analysis to guess a suitable layout, and experimenting with user profiles to guide the generation process in order to respect user preferred formatting styles.

The recent experiments with XML proved the usefulness of the generic pretty-print approach that we followed. The rapid growing importance of XML and of formatting XML documents makes it an interesting application area for our pretty-printer and a natural extension of our research.

Acknowledgments The author wants to thank Arie van Deursen for all his suggestions and for the many pleasant discussions we had. The author also thanks Mark van den Brand and Paul Klint for commenting on earlier versions of this paper.

References

1. S. Adler, A. Berglund, J. Caruso, S. Deach, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible Stylesheet Language (XSL) version 1.0. Technical Report WD-xsl-20000112, World Wide Web Consortium, 2000.
2. B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.
3. J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.
4. G. Blaschek and J. Sametinger. User-adaptable Prettyprinting. *Software – Practice and Experience*, 19(7):687 – 702, 1989.
5. R. J. Boulton. SYN: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical report, Computer laboratory, University of Cambridge, 1996.
6. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.
7. M. de Jonge. boxenv.sty: A LaTeX style file for formatting BOX expressions. Technical Report SEN-R9911, CWI, 1999. Available from `http://www.cwi.nl/~mdejonge/gpp/`.
8. M. de Jonge, E. Visser, and J. Visser. XT: Transformation Tools. Available from `http://www.cs.uu.nl/~visser/xt/`.
9. M. de Jonge, E. Visser, and J. Visser. Definition of the syntax definition formalism Sdf and its parse tree format AsFix. Technical report, CWI, 2000. In preparation.
10. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Java Series. Sun Microsysems, 1996.
11. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
12. J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques*, number 925 in LNCS, pages 53 – 96. Springer Verlag, 1995.
13. M. O. Jokinen. A Language-independent Prettyprinter. *Software—practice and experience*, 19(9):839–856, 1989.
14. S. P. Jones. A Pretty Printer Library in Haskell, Version 3.0, 1997. Available from `http://www.dcs.gla.ac.uk/~simonpj/pretty.html`.
15. D. E. Knuth. *TEX and METAFONT : new directions in typesetting.* Digital Press and the American Mathematical Society, 1979.

16. G. T. Leavens. Prettyprinting Styles for Various Languages. *SIGPLAN Notices*, 19(2):75–79, 1984.
17. P. M. Marden and E. V. Munson. Today's style sheet standards: The great vision blinded. *IEEE Computer*, 32(11):123–125, 1999.
18. M. Mikelsons. Prettyprinting in an interactive programming environment. *SIGPLAN Notices*, 16(6):108–116, 1981.
19. 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.
20. D. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.
21. N. Ramsey. Unparsing Expressiong With Prefix and Postfix Operators. *Software – Practice and Experience*, 28(12):1327–1356, 1998.
22. J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.
23. G. A. Rose and J. Welsh. Formatted Programming Languages. *Software—practice and experience*, 11:651–669, 1981.
24. L. F. Rubin. Syntax-Directed Pretty Printing – A First Step Towards a Syntax-Directed Editor. *IEEE Transactions on Software Engineering*, SE-9:119–127, 1983.
25. M. Ruckert. Conservative Pretty-Printing. *SIGPLAN Notices*, 23(2):39–44, 1996.
26. D. S. Swierstra, P. Azero, and J. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming*, number 1608 in LNCS, pages 150 – 206. Springer Verlag, Braga, Portugal, 1998.
27. M. G. J. van den Brand and M. de Jonge. Pretty Printing within the ASF+SDF Meta-Environment: a Generic Approach. Technical Report SEN-R9904, CWI, 1999.
28. M. G. J. van den Brand, P. Klint, and P. Olivier. Compilation and Memory Management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC'99)*, volume 1575 of *LNCS*, pages 198–213, 1999.
29. M. G. J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new ASF+SDF Meta-Environment. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer verlag, 1997.
30. 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):1 – 41, 1996.
31. 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.
32. E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.
33. E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30 – 44. Springer-Verlag, 1999.
34. P. Wadler. A Prettier Printer, 1998. Available from `http://cm.bell-labs.com/cm/cs/who/wadler`.