Pretty-Printing within the ASF+SDF Meta-Environment: a Generic Approach

Mark van den Brand, Merijn de Jonge

Software Engineering (SEN)

SEN-R9904 March 1999

Report SEN-R9904
ISSN 1386-369X

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

Pretty-Printing within the ASF+SDF Meta-Environment: a Generic Approach

Mark van den Brand
CWI
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands
Mark.van.den.Brand@cwi.nl

Merijn de Jonge
Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
mdejonge@wins.uva.nl

ABSTRACT

The automatic generation of formatters for (programming) languages within the ASF+SDF Meta-Environment is a research topic that is concerned with the construction of language specific formatters (or pretty-printers) given a language definition in SDF.

In this paper, we give an overview of pretty-printers that have been developed within this project and observe that these pretty-printers are either *language dependent* or *non-customizable*. Language independence and customizability are inevitable properties of pretty-printers however, when faced with the problem of formatting many different, evolving languages. Therefore, we introduce in this paper a generic framework for pretty-printing and describe an instantiation of the framework that forms a language independent and customizable pretty-printer.

1991 Computing Reviews Classification System: D.2.1, D.2.3, D.2.6, D.2.7, D.2.m, D.3.2, I.7.2.

Keywords and Phrases: Design, Documentation, Languages, Document Preparation, Program Generators.

Note: Work carried out under project SEN-1.4, ASF+SDF

1 Introduction

The automatic generation of formatters (or pretty-printers) for programming languages within the ASF+SDF Meta-Environment (Klint, 1993) has been a research topic for a long time. This research is concerned with the construction of language specific formatters given a language definition in the syntax definition formalism SDF (Heering *et al.*, 1989). Development of powerful formatting tools is essential for the industrial acceptance of ASF+SDF (Deursen *et al.*, 1996). The renovation factories described in van den Brand *et al.* (1997c), for instance, depend not only on flexible parsing technology and fast transformations but also on powerful pretty-print tools.

The pretty-printers that were designed for and integrated with the Meta-Environment, have one disadvantage in common: they depend on VTP (Borras *et al.*, 1989; Borras, 1989). VTP is the internal tree representation of terms and modules within the Meta-Environment. It is defined as part of the Centaur system, a generic interactive environment generator (Borras *et al.*, 1989), on top of which the ASF+SDF Meta-Environment has been built. This dependency of the pretty-print tools on VTP makes them unusable outside the Centaur system. The development of a new Meta-Environment (van den Brand *et al.*, 1997f, 1997d), not based on Centaur, therefore requires a reimplementation of these tools.

In this paper we emphasize the importance of generic pretty-printers to be *language independent* and *customizable*. The observation that none of the existing pretty-printers do satisfy both properties motivates the development of *new* pretty-printers, rather than reimplementing existing ones. We therefore introduce a generic framework for pretty-printing and discuss a particular instantiation of this framework that forms the pretty-print system for the new Meta-Environment.

The paper is organized as follows: Section 2 describes and classifies the pretty-printers that have been designed before. We discuss the motivation for the development of new pretty-print tools in Section 3. Section 3.2 describes a generic framework for pretty-printing and a particular instantiation of the framework. Conclusions, related work, and directions for future work are described in Section 5. Appendix A contains a complete description of BOX, the markup language that is used in the generic pretty-print framework. An elaborated example of the use of the pretty-print system that we present in this paper is included in Appendix B.

2 Classification of Pretty-Printing Approaches

In this section we will give a brief historical overview of the various pretty-printing approaches that have been used in the ASF+SDF Meta-Environment. A more detailed overview of these pretty-printing facilities can be found in van den Brand (1995). Based on this overview we will give a classification of the approaches with respect to customizability, language dependence, and their dependence on the internal abstract syntax tree representation and implementation of the ASF+SDF Meta-Environment. This classification is used to locate problems in the various approaches and to formulate design properties for a new pretty-print approach that is introduced in this paper.

Approach 1: VTP and PPML. The very first version of the pretty-printer was based on VTP (Borras *et al.*, 1989; Borras, 1989), the internal abstract syntax tree representation of modules and terms, and on the pretty-print engine PPML (Morcos-Chounet and Conchon, 1986). This pretty-printer was implemented in LELISP (Chailloux *et al.*, 1984) and constructs a PPML term for arbitrary VTP trees. For each node in the VTP tree a corresponding PPML term is constructed based on a restricted number of heuristics. The PPML term is interpreted by the PPML formatter engine which generates the formatted textual representation of the VTP tree. The main advantage of this pretty printer was its good performance. The incorrect output that was sometimes generated by the PPML engine and the inflexibility of the pretty-printer (only by writing a LELISP program the output of the pretty-printer could be adapted), motivated the development of a new generation of pretty-printers.

Approach 2: VTP and ToLaTeX. One of the strong features of the ASF+SDF Meta-Environment is the translation of (parts of) ASF+SDF modules to LaTeX. This feature promotes literate programming and the integration of ASF+SDF specifications in LaTeX documents.

The tool `tolatex` (Visser, 1994), traverses a VTP tree and generates LaTeX code. It was developed independent of the pretty-printer based on PPML. This tool only supports the formatting of ASF+SDF modules. Although the tool formats very nicely, it is difficult to adapt because most formatting rules were hard coded in LELISP.

Approach 3: VTP and BOX. Motivated by the restrictions of the first pretty-printer implementation, a preliminary version of the BOX language was designed (Vos, 1990). This work was the starting point of a whole range of tools to improve the pretty-print facilities of the ASF+SDF Meta-Environment. The initial BOX language evolved to the BOX language described in van den Brand and Visser (1996).

BOX is a language independent, intermediate representation used as markup language to describe the intended layout of text. In general, the process of pretty-printing involves the construction of a formatting for an input term over a language L, and the generation of an output term over a language L' which respects this formatting. BOX allows both tasks to be separated in a *front-end* (also called *formatter* in this text), dedicated to the formatting of an input term (i.e., the construction of a BOX term), and a *back-end* which translates BOX to some output format. The advantage of this division is the separation of parts in a pretty-printer that depend on the input language and parts that depend on the output format. A pretty-printer within this setting, is the composition of a front-end and a back-end. The output of a front-end and the input of a back-end are connected by BOX.

BOX first use was to replace PPML in the process of formatting ASF+SDF terms and modules. A front-end, implemented in LELISP, generated a BOX term by traversing the VTP tree. This BOX term was processed by the back-end `box2text` to obtain the corresponding textual representation. The back-end is specified in ASF+SDF and compiled to C using the ASF to C compiler (Kamperman and Walters, 1993). This approach produces correct output but is still inflexible. Nevertheless, this combination of front-end and back-end still forms the default pretty-printer of the ASF+SDF Meta-Environment.

Approach 4: BOX and Formatter Generator. The inability of the user to influence the pretty-printing of terms motivated the development of a formatter generator (van den Brand and Visser, 1996). Given a language definition

Approach	Customizable	Language Dependent	VTP tree Based	Overlapping Code
VTP and PPML	no	no	yes	yes
VTP and `tolatex`	limited	no	yes	yes
VTP and BOX	no	no	yes	no
generated pretty-printer as:				
executable specification	yes	yes	no	no
stand-alone tool	no	yes	no	no

Table 1: Classification of pretty-print approaches.

in SDF, the formatter generator generates a formatter for that language as a set of ASF+SDF conditional rewrite rules. These rewrite rules define mappings from language patterns to BOX expressions. The BOX expressions define how the patterns should be formatted and can be overruled by the user.

The resulting specification can be compiled using the ASF to C compiler to obtain a stand-alone front-end. Together with the back-end `box2text`, it forms a pretty-printer which can replace the default pretty-printer provided by the ASF+SDF Meta-Environment. This approach has been used among others to obtain pretty-printers for SEAL (Koorn, 1994), COBOL (van den Brand *et al.*, 1997e), and PROLOG (Brunekreef, 1996).

Because a generated formatter can be adapted only by modifying rewrite rules, it cannot be customized any further after compilation. As a consequence, a recompilation of the formatter is required whenever its output has to be adapted. A pretty-printer, composed of a generated formatter as front-end, and a back-end, thus can be used in two ways: (i) as executable specification; this pretty-printer is customizable but cannot be used as stand-alone pretty-printer or be used to replace the default pretty-printer; (ii) as stand-alone pretty-printer after compilation to C: the stand-alone pretty-printer thus obtained is non-customizable but can be used to replace the default pretty-printer of the Meta-Environment.

There are several disadvantages related to the approach of the formatter generator. First of all, the generated formatters are language dependent. As a result, a separate formatter is required for each language. Secondly, pretty-printing languages under development is not supported. Instead, each time the language is adapted a new formatter has to be generated, otherwise the formatter would generate incorrect output. Thirdly, the formatter can be used only to format terms. Thus, ASF+SDF specifications cannot be formatted with the generated formatters. Finally, the use of the pretty-printers, either as executable specifications or as stand-alone pretty-printers, is not satisfactory because the executable specifications cannot be used in a stand-alone setting, whereas the compiled pretty-printers are non-customizable.

The generated formatters produce BOX as output which can be processed by back-ends to translate to some output format. Besides `box2text`, a new back-end has been developed that translates BOX expressions to TEX code. The combination of this back-end with generated front-ends provides a facility to develop powerful typesetting tools. In Brunekreef (1995) all PSF specifications are typeset using this combination of techniques.

Classification of Pretty-Printers. The pretty-printers that we have described in this section can be classified according to a number of properties. This classification proved to be helpful in locating problems of existing pretty printers and formulating design properties for new pretty-printers.

We will distinguish four properties: i) Customizable; pretty printers that satisfy this property can be customized by the user to obtain language specific pretty-printers; ii) Language dependent: this property specifies whether or not a pretty-printer is restricted to format programs of a single language only; iii) VTP tree based: specifies whether the pretty-printer depends on a VTP tree traversal; iv) Overlapping code. A pretty-printer forms a mapping from a term over some input language to some output format. Many pretty-print techniques require this complete mapping to be defined separately for each combination of input language and output format. Redefining this mapping results in overlapping code when only the input language (or output format) of several pretty-printers differs. The property 'Overlapping code' indicates whether a significant part of the code of a pretty-printer has been implemented for another pretty-printer as well.

The classification of the pretty-printers described in this section according to these four properties is depicted in Table 1. From this table we see that the pretty-printers are either language dependent or non-customizable. Furthermore, pretty-printers that are based on BOX contain less overlapping code. Finally, we observe that all pretty-printers that have been integrated within the Meta-Environment depend on the Centaur system because they

require a traversal of the VTP tree. This classification of existing pretty-printers is used in the next section to motivate the development of new pretty printer tools for use within a new ASF+SDF Meta-Environment.

3 A Generic Pretty-Print Approach

The pretty-print tools described in the previous section that have been integrated within the Meta-Environment (approaches 1, 2, and 3), all depend on the internal tree representation (VTP) of Centaur. These tools traverse a parse tree represented as VTP tree to pretty-print the concrete syntax. As a consequence they are all bound to the Centaur system. Because Centaur was implemented in LELISP (Chailloux *et al.*, 1984), the pretty-printers were (at least partly) implemented in LELISP as well, in order to have access to VTP trees. This dependence on the Centaur system and their implementation in LELISP not only makes it hard to develop new or adapt (maintain) existing pretty-print tools, it also makes their reuse outside the Centaur system impossible.

Breaking the dependency between the ASF+SDF system and LELISP/Centaur was a strong motivation for the development of a new Meta-Environment (van den Brand *et al.*, 1997f, 1997d). This new Meta-Environment uses AsFix (van den Brand *et al.*, 1997a) as parse tree representation for ASF+SDF modules (see Section 3.1). AsFix is designed to represent structured data and, in contrast with VTP trees, to be exchangeable between a heterogeneous collection of tools.

Tools used within the new Meta-Environment which require access to the abstract representation of ASF+SDF modules have to be designed to operate on AsFix terms. As a consequence, a reimplementation of the pretty-print tools is required to reuse them within this new Meta-Environment.

One could argue that the generated formatters (according to approach 4 in Section 2) do not depend on the implementation of the ASF+SDF Meta-Environment because they do not require direct access to parse trees and thus provide a pretty-print mechanism that is also usable within a new Meta-Environment. Because of the language dependency of the generated formatters however, we do not consider this approach acceptable as *default* pretty-printer in a new Meta-Environment.

For the reimplementation of the pretty-print tools, only the BOX based pretty-print tools as described in the previous section are considered because of the advantage that the separation of a pretty-printer in a front-end and a back-end provides.

Observe that the BOX based tools described in the previous section are either language dependent or non-customizable. A formatter generated by the formatter generator on the one hand, is highly customizable. However, the language dependence of the generated formatters requires separate formatters for each language. The default pretty-printer of the current Meta-Environment on the other hand, is able to format terms over different language definitions. Unfortunately, it does not support customization of the generated output at all.

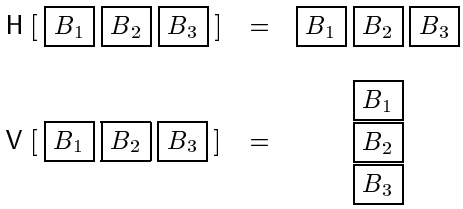
For a pretty-printer to be usable in general, it should be language independent *and* customizable. Both requirements have motivated the development of a new set of pretty-print tools, instead of just reimplementing the existing pretty-print tools. The new tools are again divided in *front-ends* and *back-ends*. A front-end (or formatter) takes a parse tree over an input language L as input and generates a BOX term as output. A back-end takes a BOX term as input and generates as output a term over some output format L'. A pretty-printer is constructed by connecting the output of a front-end to the input of a back-end, using BOX as intermediate representation. Before describing this generic pretty-print framework and an instantiation (which forms the pretty-print system for the new Meta-Environment) in Section 3.2, we will first discuss AsFix and BOX briefly in Section 3.1 and 3.2, respectively. The front-end and back-ends that instantiate the generic pretty-print framework are described in Section 4.1 and 4.2.

3.1 AsFix

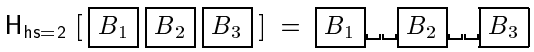
AsFix (ASF+SDF Prefix Notation) (van den Brand *et al.*, 1997a) is a parse tree representation for ASF+SDF terms and modules, designed to be exchangeable between a heterogeneous collection of tools. AsFix contains all information of the original source text, this includes in particular the original source text itself (that is, all keywords, white space, comments, etc. are preserved). AsFix is implemented as instance of the more general ATerm (short for Annotated Terms) representation (van den Brand *et al.*, 1997b). Within the new Meta-Environment AsFix is used to replace VTP as module (and term) representation. AsFix is suitable for pretty-printing because it not only contains the parse tree (used to select pretty-print rules), but also the original source text (required to support comments in the pretty-printed output).

3.2 BOX Language

In the generic pretty-print framework the input language dependent parts and the output format dependent parts are separated in front-ends and back-ends, respectively. Front-ends and back-ends are connected using a language independent intermediate representation, called BOX (see Figure 1). BOX is a mark-up language to describe the intended layout of text. A BOX expression is constructed by composing sub-boxes using BOX operators. These operators specify the relative ordering of boxes. Examples of BOX operators are the H and V operator which format boxes horizontally and vertically, respectively:



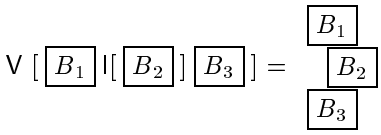
The exact formatting of each BOX operator can be customized using BOX options. For example, to control the horizontal layout between boxes the H operator supports the hs space option.



For a description of BOX in general we refer to van den Brand and Visser (1996) were the BOX language was introduced.

In this section we describe how we deviate from the original BOX language. We slightly adapted the original BOX language described in van den Brand and Visser (1996) because of a few irregularities that would make the development of BOX based tools very hard. A detailed description of the syntax and the functionality of the adapted BOX language can be found in Appendix A.

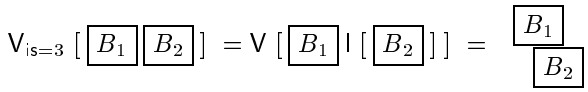
Adaptation of Positional BOX Operators. In general, positional BOX operators (like the H and V operators) specify the relative positioning of their sub-boxes. For example, the H operator specifies that its sub-boxes are separated by horizontal layout. The I and WD operators of the original BOX language deviate from this model. The I operator, which operates on a single box, is used to specify left indentation. It has only effect in a vertical context:



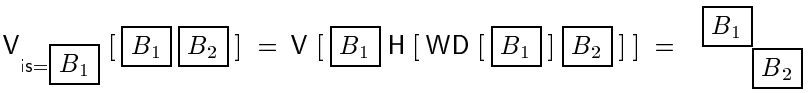
Here we see that the I operator specifies how the boxes B_1 and B_2 are positioned. Although B_1 is not a sub-box, the I operator *does* specify its relative positioning to B_2.

The WD operator from the original BOX language does not specify the relative positioning of its sub-boxes either. The operator translates to an empty box with (horizontal) dimensions equal to its argument.

To make the BOX language more regular, we removed both operators. To provide left indenting similar to the I operator, we introduced a new space option 'is'. The is space option specifies horizontal layout to be placed between sub-boxes. Similar to the I operator, indentation is in effect only in vertical mode.



To be able to specify horizontal spacing equal to the width of some box (like the WD operator), we allow horizontal spacing to be specified as a BOX expression.



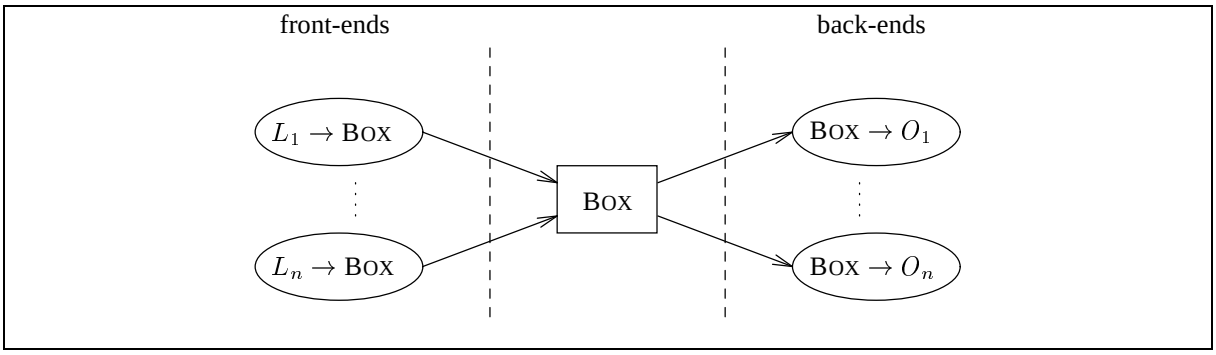


Figure 1: A generic framework for pretty-printing.

Adaptation of Non-Positional BOX Operators. Besides the I and WD positional BOX operators, we removed a few non-positional operators as well.

The rather complicated mechanism to express the formatting of comments using the operators HPAR, VPAR and PAR, has been simplified. We replaced the three operators by the single comment operator 'C'.

Finally, we removed the O operator. The O operator translates to the vertical composition of two of its sub-boxes separated by a horizontal bar. Because this operator is too strongly related to ASF+SDF (it is used solely to format conditional equations of ASF+SDF specifications), it has been replaced by the more general line operator 'L'. The L operator translates to a horizontal bar of width equal to the width of its first sub-box. The bar is constructed from characters of its second sub-box.

4 A Generic Pretty-Print Framework

By using BOX as intermediate representation, we are able to create a generic, extensible pretty-print framework. In Figure 1 this generic framework is depicted. Programs in this figure are denoted by ellipses, the rectangle in the figure denotes data. The framework can be instantiated by *plugging* front-ends and back-ends into the system. Each front-end forms a mapping from terms over an input language L_i to BOX, each back-end translates BOX to some output format O_i. Language independence of the pretty-print system means that the system does not depend on the input language. Language independence is supported within our framework since it allows different front-ends to be plugged in. In Figure 2 a particular instantiation of the framework is depicted. As in Figure 1, we denote data by rectangles and programs by ellipses. The figure shows the generic framework instantiated with one front-end (the program `asfix2box`, see Section 4.1) and three back-ends (the programs `box2asfix`, `box2text`, and `box2latex`, described in Section 4.2). Together, the front-end and back-ends form the pretty-print system of the new Meta-Environment.

The figure displays two combinations of programs that have AsFix as input and as output. The first combination (`asfix2box` and `box2asfix`) shows that after formatting the layout in an AsFix term (containing the parse tree) an AsFix term with the same parse tree is obtained. The fact that parsing the formatted text of a given AsFix term produces an AsFix term again, with the same parse tree, is expressed by the second combination of tools (`asfix2box`, `box2text`, and `parser`).

4.1 From AsFix to BOX

A front-end within the pretty-printer setting takes a term T over some language L as input and generates a BOX term that represents the pretty-printed term T. Because the pretty-print system is to be used within the new Meta-Environment in which terms are represented in AsFix, we initially limited our attention to the development of front-ends based on AsFix only.

Given an SDF definition of a language L, a term T over L can be represented in AsFix and formatted with the formatter `asfix2box`. Using AsFix as term representation makes the formatter language independent because any term represented in AsFix can be formatted by `asfix2box`.

Of course, terms over different languages have to be pretty-printed differently. The front-end `asfix2box` therefore is parameterized such that it can be instantiated with language dependent pretty-print rules. Hence,

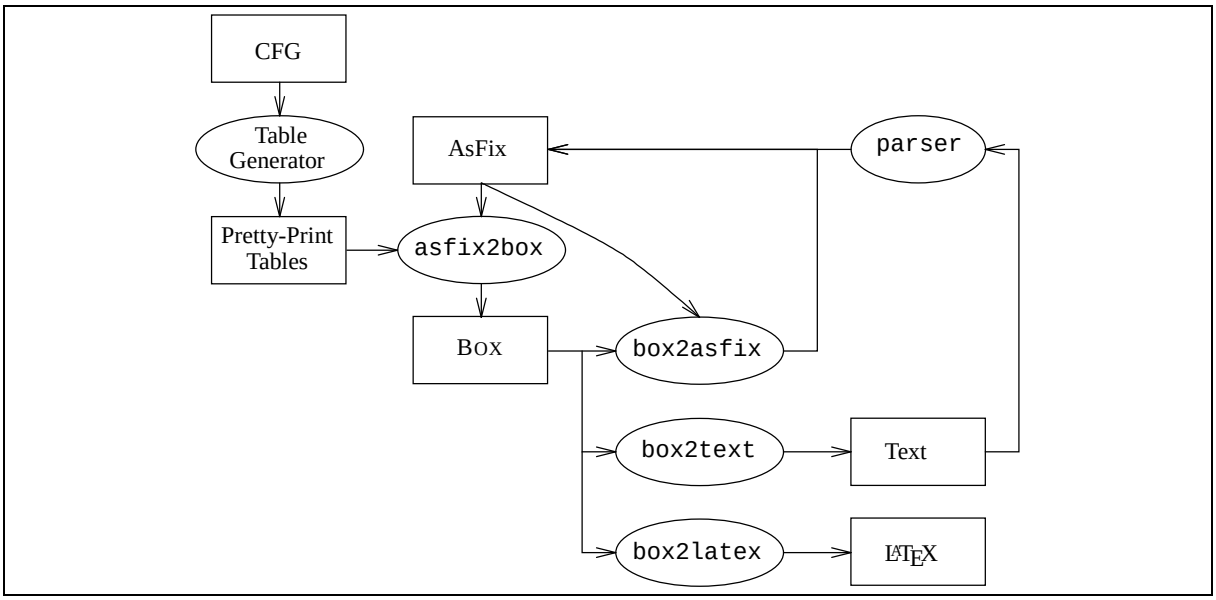


Figure 2: A particular instantiation of the generic pretty-print framework.

`asfix2box` is a *generic* tool because this parameterization separates generic code from language specific pretty-print rules.

The pretty-print rules define how specific language constructs are to be pretty-printed. That is, a pretty-print rule forms a mapping from a context-free language construct to a BOX-term. To be usable in practice, modifying the pretty-print tables by hand should be easy. Pretty-print tables are therefore presented to the user as mappings from SDF productions to BOX expressions. These mappings will be discussed in the next section.

4.1.1 Pretty-Print Tables

The front-end `asfix2box` is a generic, language independent tool. The tool is parameterized with a set of pretty-print tables. Given such a set of tables `asfix2box` constructs a BOX term by applying the pretty-print rules to an input term. The pretty-print tables describe how language constructs should be pretty-printed. The use of pretty-print tables thus separates generic code from language dependent pretty-print rules. The SDF definition of the context-free syntax of pretty-print tables is displayed below[1]:

context-free syntax
Sdf-CfFunction "," BOX → Sdf-Entry
"[" SdfEntry* "]" → SdfTable

A pretty-print table consists of zero or more entries. Each entry is formed by a context-free production in SDF together with a BOX expression, denoting how the language construct should be formatted.

Example 4.1 Consider the language definition of the data type Bool in SDF below:

context-free syntax
"True" → Bool
"False" → Bool
Bool "/\" Bool → Bool
Bool "\/" Bool → Bool

To construct a pretty-print table for this grammar, we specify for each context-free production a BOX expression:

[1] All code examples and program listings in this paper are formatted using the generic pretty-printer described in this paper. First, `asfix2box` is used to obtain a BOX expression describing the intended layout. Then `box2latex` is used to obtain corresponding LaTeX code.

```
[
  "True" → Bool, KW [ "True" ]
  "False" → Bool, KW [ "False" ]
  Bool "/\" Bool → Bool, H [ _1 "/\" _2 ]
  Bool "\/" Bool → Bool, H [ _1 "\/" _2 ]
]
```

Given this table, the constants "True" and "False" will be pretty-printed as keywords. The productions defining the operators "/\" and "\/" are both formatted horizontally. Note the use of numbered place holders ("_1" and "_2") to denote BOX expressions corresponding to non-terminal symbols in the left-hand side of the SDF productions. □

From the example we see that it is rather straightforward to define a pretty-print table given a language definition in SDF. The user just creates a pretty-print entry for each context-free production by specifying the production and the corresponding BOX expression. Productions for which no pretty-print entry exists are formatted horizontally using the H operator.

A more elaborated example of defining pretty-print tables and their use is presented in Appendix B.

4.1.2 Modularization of Pretty-Print Tables

The front-end `asfix2box` accepts a sequence of pretty-print tables as input. A sequence of tables $t_1, t_2, \ldots, t_n$ is ordered in such a way that $t_i > t_j$ whenever $i < j$. From matching entries within different tables only the entry defined in the table with highest order is applied. From matching entries defined in a single table, the first entry is selected. This ordering of pretty-print tables specifies exactly which pretty-print rule will be applied when several matching entries exist.

The ability to pass a sequence of tables to `asfix2box` and the ordering of tables allows for a modular structuring of pretty-print tables. We suggest a modular structuring where the pretty-print tables follow the import structure of the corresponding SDF grammar (i.e., for each constituting module of the grammar a separate pretty-print table is defined). The pretty-print table corresponding to an SDF module can be reused whenever the SDF module itself is reused. The ordering of tables allows the user to customize pretty-print rules by redefining them in pretty-print tables corresponding to importing modules.

4.1.3 Automatic Generation of Pretty-Printers

The generic tool `asfix2box` can be instantiated with a number of pretty-print tables to adapt its default operation. Together, `asfix2box` and the pretty-print tables for a language L form a formatter for the language L. In order to generate a formatter for a language automatically, only a set of pretty-print tables has to be generated.

The generation of pretty-print tables is independent of `asfix2box` and therefore different tools can be used to generate these tables. A typical table generator (see Figure 2) takes a (modular) SDF definition as input and generates for each module a table containing the context-free productions and corresponding BOX expressions. Such a tool can implement the heuristics used in van den Brand and Visser (1996) and in the default pretty-printer of Section 2. The modular structure of the tables generated by such a tool allows for an incremental pretty-print approach. After some production in an SDF module has been added or modified, only the pretty-print table corresponding to that module needs to be regenerated.

Automatic generation and manually fine-tuning of pretty-print tables can interfere. Taking advantage of the ordering of tables, we propose the separation of generated tables and tables that are modified by hand. When the latter is given precedence over the generated table, the user is always able to customize the generated pretty-print rules. Tool support can provide the ability to warn the user for non-matching entries in the non-generated table (i.e., entries for which a corresponding production no longer exists).

4.2 Back-ends

In this section we will briefly discuss the back-ends `box2asfix`, `box2text`, and `box2latex` (see Figure 2). From these back-ends only `box2asfix` is new. The other back-ends have been implemented before, but the adaptation of the BOX language required a reimplementation of these back-ends. Fortunately, the reimplementation

is either trivial (`box2text`) or it forms an overall improvement of the old implementation (in the case of `box2latex`).

From BOX to AsFix. In general a pretty-printer takes a parse tree of a language L as input and generates a string over a language L' as output. Remember from Section 3.1 that AsFix is a parse tree representation that contains layout information. This property of AsFix allows for a pretty-printer that generates a parse tree as output (i.e., it returns the initial parse tree in which layout has been updated). Generation of parse trees with updated layout as output of a pretty-printer is a performance issue basically. A performance improvement is obtained because tools that operate on parse trees can use the output of the pretty-printer directly (no parsing is required to construct the parse tree from the output of the pretty-printer).

The back-end `box2asfix` takes as input an AsFix term and a BOX term describing how the lexical elements in the AsFix term should be formatted. The back-end returns as output the original AsFix term in which the lexical elements are formatted conforming to the BOX term.

From BOX to Text. The back-end `box2text` takes a BOX term as input and generates a string containing the pretty-printed term as output. This translation has been implemented before but cannot be reused unmodified because the BOX language has been adapted (see Section 3.2). Fortunately, the back-end `box2asfix` already implements the translation from BOX to text (the lexical information contained in AsFix terms is represented as text strings), so reimplementing `box2text` is trivial. We just reuse parts of `box2asfix`.

From BOX to LaTeX. The `box2latex` back-end generates LaTeX code according to the layout information contained in BOX terms. A mapping from BOX to TeX has been implemented before, but the modification of the BOX language required a reimplementation. In addition to the tool that has been defined before, `box2latex` supports the mapping from lexical tokens to LaTeX commands, similar to the tool ToLaTeX (Visser, 1994). These mappings give the user fine control over which symbols to use in the generated LaTeX code.

Example 4.2 Mappings of lexical tokens can be defined in a table. Each entry in the table contains a mapping from a BOX string to a LaTeX command. To use the mathematical symbols '$\wedge$' and '$\vee$' for instance, for the specification of Example 4.1, the following table can be passed to `box2latex`:

```
[
  "/\" → "$\wedge$"
  "\/" → "$\vee$"
]
```

`box2latex` will now replace all occurrences of '`/\`' by the LaTeX macro '`$\wedge$`' and '`\/`' by the macro '`$\vee$`'. □

The generated LaTeX code can be processed by `pdflatex` (Thanh, 1998) to produce a PDF document. The cross referencing mechanism of BOX in combination with the hyperref package (Rahtz, 1998) results in interactive documents with hyperlinks.

5 Concluding Remarks

This paper gives an overview of the different pretty-printers that have been developed for the ASF+SDF Meta-Environment. The development of a new Meta-Environment not based on Centaur and LELISP requires a reimplementation of the pretty-printers because the pretty-printers that have been integrated within the ASF+SDF Meta-Environment depend on VTP and LELISP. The observation that existing pretty-printers are either language dependent or non-customizable motivated the development of new pretty-print tools rather than the reimplementation of existing ones.

We introduced a generic framework and described an instantiation of this framework. This instantiation, consisting of a front-end and several back-ends, forms the pretty-printer of the new Meta-Environment. This system has been specified as a number of ASF+SDF specifications. We generated stand-alone executables from the specifications using the new ASF+SDF to C compiler (van den Brand *et al.*, 1999), which translates ASF+SDF specifications to C.

Future Work. The pretty-print system that we developed has several drawbacks that require further investigation. First, we mention the dependency of the generic framework that we introduced upon the BOX language, which is used to connect front-ends to back-ends. This tiny language might turn out to be too weak in practice. We want to investigate whether this language requires extension or whether another intermediate representation proves to be more powerful and useful.

Secondly, the current pretty-print table generator is not fully satisfactory. It creates a pretty-print table for an SDF grammar but it does not take the structure of the productions into account in order to 'guess' suitable BOX expressions. We want to develop a new generator similar to the formatter generator described in Section 2, that uses heuristics for the generation of appropriate BOX expressions.

Finally, we emphasize that the front-end `asfix2box` is not as powerful as the formatter generator of Section 2. The latter supports any pattern to be adapted in the generated formatter, while `asfix2box` only supports adaptation of BOX expressions corresponding to individual productions. This 'locality' of `asfix2box` might be no problem for its use as default pretty-printer in the Meta-Environment. However, in order to have complete control over the pretty-printed output, this limitation of `asfix2box` is too strong. We suggest generating formatters, similar to those generated by the formatter generator, in addition to `asfix2box` which can be plugged into the generic framework and do give more control over the pretty-printed output. Adapting the existing formatter generator to generate formatters which accept terms over the new BOX language would probably be a good approach as well.

Related Work. The notion of boxes for formatting text is not new. BOX like languages include TEX (Knuth, 1984) and PPML (Morcos-Chounet and Conchon, 1986; Borras, 1989). The language BOX is based on PPML and as a consequence, both languages are quite similar. We refer to van den Brand and Visser (1996) for a comprehensive description of formatters and BOX-like languages.

Functional approaches of pretty-printing are described in Hughes (1995) and Swierstra *et al.* (1998). These approaches use a set of combinators to express the desired layout of text. Most of these combinators correspond to BOX operators. For example, the `>|<` combinator is used to place its arguments horizontally (similar to the H operator), the `>-<` combinator places its arguments vertically (like the V operator).

Obtaining interactive documents from BOX terms has been a research topic for several years. In van der Graaf (1997) the back-end `box2html` is developed. Unfortunately, several BOX operators (like the HOV and L operators) cannot be expressed in terms of HTML operators. As a consequence the translation to HTML is currently not satisfactory. Fortunately, we are able to obtain interactive documents anyhow, because the LATEX code that we generate from BOX can be translated to PDF.

Acknowledgments We would like to thank Paul Klint (CWI) and Joost Visser (UvA) for reading earlier drafts of this paper.

A The BOX Language

BOX is a mark-up language to describe the intended layout of text. A preliminary of BOX, based on PPML (Morcos-Chounet and Conchon, 1986) was introduced in Vos (1990). BOX itself was described before in van den Brand and Visser (1996). In this section we describe a new version of the BOX language. Section 3.2 contains the motivation for this new version and a comparison with the version of van den Brand and Visser (1996).

Expressions over the BOX language can be constructed by *composing* boxes using *box-operators*. These operators specify the relative positioning of boxes. BOX supports several of these *positional* operators (described in detail in Section A.1). Examples of positional operators are the H and V operators which format their sub-boxes horizontally and vertically, respectively. The exact formatting of the positional operators can be controlled by means of options. These options allow for instance, the horizontal and vertical layout within the H and V operators to be controlled.

Besides positional operators, BOX also contains *non-positional* operators. These operators are used to control how sub-boxes are displayed. These operators include font operators to specify font parameters (font family, font color, etc.) and operators for cross referencing.

Syntax of the basic positional BOX operators and space options in SDF:

exports

context-free syntax

"hs"	→ SPACE-SYMBOL
"vs"	→ SPACE-SYMBOL
"is"	→ SPACE-SYMBOL
SPACE-SYMBOL "=" INT	→ S-OPTION
SPACE-SYMBOL "=" BOX	→ S-OPTION
S-OPTION*	→ S-OPTIONS

context-free syntax

BOX-STRING	→ BOX
BOX*	→ BOX-LIST
"H" S-OPTIONS "[" BOX-LIST "]"	→ BOX
"V" S-OPTIONS "[" BOX-LIST "]"	→ BOX
"HV" S-OPTIONS "[" BOX-LIST "]"	→ BOX
"HOV" S-OPTIONS "[" BOX-LIST "]"	→ BOX

The syntax of the alignment BOX operator is defined as follows:

context-free syntax

"A" A-OPTIONS S-OPTIONS "[" BOX-LIST "]"	→ BOX
"R" "[" BOX-LIST "]"	→ BOX
"l" S-OPTIONS	→ A-OPTION
"c" S-OPTIONS	→ A-OPTION
"r" S-OPTIONS	→ A-OPTION
"(" {A-OPTION ","}* ")"	→ A-OPTIONS

Observe how space options are combined with column definitions to define *inter-column* spacing. The space options that follow the alignment options define *inter-row* spacing.

Figure 3: Syntax of positional BOX operators in SDF.

A.1 Positional BOX-Operators

The most elementary boxes are strings enclosed in double quotes. Smaller boxes can be composed to form new boxes using the positional BOX operators that specify the relative ordering of the smaller boxes. The syntax in SDF of the positional BOX operators is depicted in Figure 3.

The H and V operators are the basic positional BOX operators. They format their sub-boxes horizontally and vertically, respectively. This behavior is depicted in the diagrams below:

$$\mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,] \;=\; \boxed{B_1}\,\boxed{B_2} \qquad\qquad \mathsf{V}\,[\,\boxed{B_1}\,\boxed{B_2}\,] \;=\; \begin{array}{|c|}\hline B_1 \\ \hline B_2 \\ \hline \end{array}$$

When the line width is taken into account, the H and V operators can be combined to obtain conditional operators. Depending on the amount of space left, boxes are placed horizontally or vertically. BOX contains two conditional operators. The first operator ('HOV'), places all its sub-boxes horizontally when they fit on a single line, otherwise, they are all placed vertically. This can be expressed in terms of H and V operators as:

$$\mathsf{HOV}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \;=\; \begin{array}{l} \mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \\ \quad or \\ \mathsf{V}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \end{array}$$

The HOV operator maximizes the number of lines occupied when its sub-boxes do not fit on a single line.

The second conditional operator minimizes the number of lines occupied. The HV operator calculates a combination of H and V operators as follows:

operator	hs	vs	is	remarks
H	✓			
V		✓	✓	
HOV	✓	✓	✓	Indentation is used only when the formatting occupies multiple lines.
HV	✓	✓	✓	idem.
A	✓	✓		The hs space option defines the *inter-column* spacing, the vs option defines the *inter-row* spacing.
R				

Table 2: Supported space options for each positional BOX operator.

$$\mathsf{HV}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \;=\; \begin{array}{l} \mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \\ \quad or \\ \mathsf{V}\,[\,\mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,]\,\boxed{B_3}\,] \\ \quad or \\ \mathsf{V}\,[\,\boxed{B_1}\,\mathsf{H}\,[\,\boxed{B_2}\,\boxed{B_3}\,]\,] \\ \quad or \\ \mathsf{V}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,] \end{array}$$

The HV operator calculates the layout of the sub-boxes in such a way that the number of lines is minimized and the lines are maximal filled. This operator is typically used when a list of items is to be pretty-printed.

H, V, HOV, and HV are the basic positional BOX operators. Each of these operators can be accompanied with a number of options to control the layout between sub-boxes. The following options are supported: hs for horizontal spacing, vs for vertical spacing, and is for shifting horizontally in a vertical context (indentation). Certain combinations of options and BOX operators do not make sense, e.g., the H operator in combination with the options vs or is. Table 2 summarizes the available space options for each BOX operator.

In order to get a more sophisticated layout such as columns, the BOX alignment operator can be used. It is a combination of the operator A, to indicate that an alignment is constructed, and R operators to represent individual rows. The A operator must be used with a set of alignment options, namely l, c, and r to indicate whether the columns should be left, centered, or right aligned, respectively. The number of alignment options defines the number of columns. For example:

$$\begin{array}{l} \mathsf{A}\ (l,c,r)\ [\ \mathsf{R}\ [\ \boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}\ ] \\ \qquad\qquad \mathsf{R}\ [\ \boxed{B_4}\ \boxed{B_5}\ \boxed{B_6}\ ]\ ] \end{array} \;=\; \begin{array}{lll} \boxed{B_1} & \boxed{B_2} & \boxed{B_3} \\ \boxed{B_4} & \boxed{B_5} & \boxed{B_6} \end{array}$$

To define the layout between columns, each alignment option can contain an hs space option. To define the layout between rows, the A operator supports the vs space option.

font parameter	description
fn	font name
fm	font family
se	font series
sh	font shape
sz	font size
cl	font color

(a)

font operator	used to format
KW	keywords
VAR	variables
NUM	numbers
MATH	mathematical symbols

(b)

Table 3: Table (a) displays the font parameters supported by the F BOX operator, table (b) displays the supported dynamic font operators.

The static font operator F, the dynamic operators, and the font options are defined as:

```
exports
  context-free syntax
    FONT-PARAM "=" BOX-INT          → F-OPTION
    FONT-PARAM "=" FID              → F-OPTION
    F-OPTION*                       → F-OPTIONS
    "F" F-OPTIONS                   → FONT-OPERATOR
    FONT-OPERATOR "[" BOX "]"       → BOX
    FONT-OPERATOR "(" BOX-LIST ")"  → BOX-LIST
    "fn"                            → FONT-PARAM
    "fm"                            → FONT-PARAM
    "se"                            → FONT-PARAM
    "sh"                            → FONT-PARAM
    "sz"                            → FONT-PARAM
    "cl"                            → FONT-PARAM
    "KW"                            → FONT-OPERATOR
    "VAR"                           → FONT-OPERATOR
    "NUM"                           → FONT-OPERATOR
    "MATH"                          → FONT-OPERATOR
```

The cross reference operators are defined as:

```
  context-free syntax
    "LBL" "[" BOX-STRING "," BOX "]" → BOX
    "REF" "[" BOX-STRING "," BOX "]" → BOX
```

The syntax of the comment operator is:

```
  context-free syntax
    "C" "[" BOX-LIST "]" → BOX
```

Finally the line operator looks like:

```
  context-free syntax
    "L" "[" BOX BOX "]" → BOX
```

Figure 4: Syntax of non-positional BOX operators in SDF.

A.2 Non-positional BOX-Operators

There are four different types of non-positional BOX operators: the font operators to change the textual appearance of BOX expressions, the cross reference operators to create links between boxes, the comment operator to indicate that a BOX expression contains comments, and the line operator to draw lines of characters of arbitrary length. The syntax in SDF of these non-positional BOX operators is depicted in Figure 4.

Font Operators. BOX font operators are used to change the textual appearance of the argument box expression. Fonts can be characterized by the parameters font name, font family, font series, font shape, font size, and font color. The most general font operator is F. By means of font parameters the desired font can be controlled. Table 3(a) summarizes the supported font parameters.

By using the F font operator, fonts are defined statically. BOX also support fonts to be defined dynamically using special font operators. These operators are used to format specific language constructs like keywords and variables. The mapping from these operators to fonts is deferred to the back-ends. The user can customize these font mappings at any time after the BOX expression has been constructed. Table 3(b) summarizes the supported dynamic font operators.

Cross Reference Operators. BOX has limited support for cross referencing using the LBL and REF operators. The operator LBL is used to define a label for a box, the operator REF is used to refer to a labeled box. Back-ends can implement these operators to enable cross referencing in the generated output. Currently, only the back-end `box2latex` fully support cross referencing (see Section 4.2). By default, this tool translates the LBL and REF operators to the LaTeX commands '\label' and '\ref', respectively.

```
module Sdf
  exports
    sorts Sdf-Id Iterator CharClass Literal Module Section SyntaxSection CfFunction CfElem Attributes
          FunOpName ListOpName

    context-free syntax
      "module" Sdf-Id Section*                                  → Module
      "imports" Sdf-Id+                                         → Section
      "exports" SyntaxSection+                                  → Section
      "sorts" Sdf-Id+                                           → SyntaxSection
      "context-free" "syntax" CfFunction+                       → SyntaxSection
      "+"                                                       → Iterator
      "*"                                                       → Iterator
      FunOpName CfElem* "→" Sdf-Id Attributes                   → CfFunction
      Literal "(" {CfElem ","}* ")" "→" Sdf-Id Attributes       → CfFunction
      Sdf-Id                                                    → CfElem
      Literal                                                   → CfElem
      Sdf-Id Iterator ListOpName                                → CfElem
      "{" Sdf-Id Literal "}" Iterator ListOpName                → CfElem
                                                                → Attributes
      "{" {Literal ","}+ "}"                                    → Attributes
```

Figure 5: Partial syntax definition of SDF in SDF.

Comment Operator. The comment operator C is used to represent user defined comments in the source text. This operator accepts a sequence of BOX strings as sub-boxes each of which starts a new line of comment. Argument strings may contain newline characters which are interpreted and start a new line. The semantics of the C operator with multiple sub-boxes containing newline characters is depicted below:

$$\mathsf{C}\,[\,\text{“}s_1\backslash n\ \ldots\ \backslash n\,s_n\text{”}\,] \;=\; \mathsf{C}\,[\,\text{“}s_1\text{”}\ \ldots\ \text{“}s_n\text{”}\,]$$

Depending on the back-end these strings are vertically formatted or in a HV style.

Line Operator. The last non-positional operator is L. This operator is used to create a line of characters equal to the width of its first sub-box. This operator can be used for instance, to “draw” the vertical bar in ASF equations. Its use is demonstrated by the following example:

$$\begin{array}{l}\mathsf{V}\,[\,\mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,]\\ \quad \mathsf{L}\,[\,\mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,]\ \texttt{"="}\,]\,]\end{array} \;=\; \underline{\boxed{B_1}\,\boxed{B_2}}$$

B Example

In this section we will demonstrate the pretty-printer that we described in this paper by means of a comprehensive example. We will use part of the language definition of SDF defined in SDF to demonstrate the construction of a pretty-print table and to demonstrate the generated output of the generic pretty-printer when it has been instantiated with the pretty-print rules for SDF.

The Grammar of SDF. Before we construct a pretty-print table for SDF, we first show part of its definition in SDF. The syntax definition of Figure 5 is taken from (Heering *et al.*, 1989) but has been greatly simplified by removing several language constructs that are not of interest for this example. The figure also shows the intended layout of SDF definitions. The BOX expressions that we define in the pretty-print table below, define a formatting of SDF terms equivalent to the formatting of the SDF module in Figure 5.

```
[
  "module" Sdf-Id Section * → Module, V is = 3 [ H [ KW [ "module" ] _1 ] V vs = 1 [ _2 ] ]
  "imports" Sdf-Id + → Section, H [ KW [ "imports" ] HV [ _1 ] ]
  "exports" SyntaxSection + → Section, V is = 3 [ KW [ "exports" ] V vs = 1 [ _1 ] ]
  "sorts" Sdf-Id + → SyntaxSection, H [ KW [ "sorts" ] HV [ _1 ] ]
  "context-free" "syntax" CfFunction + → SyntaxSection,
      V is = 3 [ H [ KW [ "context-free" ] KW [ "syntax" ] ] A ( l , l , l , l ) [ _1 ] ]
  "+" → Iterator, KW [ "+" ]
  "*" → Iterator, KW [ "*" ]
  FunOpName CfElem * "→" Sdf-Id Attributes → CfFunction, R [ H [ _1 _2 ] KW [ "→" ] _3 _4 ]
  Literal "(" {CfElem ","}* ")" "→" Sdf-Id Attributes → CfFunction, R [ H [ _1 "(" _2 ")" ] KW [ "→" ] _3 _4 ]
  Sdf-Id → CfElem, H [ _1 ]
  Literal → CfElem, H [ _1 ]
  Sdf-Id Iterator ListOpName → CfElem, H [ H hs = 0 [ _1 _2 ] _3 ]
  "{" Sdf-Id Literal "}" Iterator ListOpName → CfElem, H [ H hs = 0 [ "{" _1 ] H hs = 0 [ _2 "}" _3 ] _4 ]
  → Attributes, H [ ]
  "{" {Literal ","}+ "}" → Attributes, H hs = 0 [ "{" _1 "}" ]
  {Literal ","}+, H [ _1 "," ]
]
```

Figure 6: A pretty-print table for the syntax definition of Figure 5.

A Pretty-Print Table for SDF. Given the partial language definition of SDF in Figure 5, we can specify a pretty-print table for this language by specifying BOX expressions for each context-free production. This pretty-print table is used to instantiate the tool `asfix2box` to obtain a language specific pretty-printer. Below we will discuss several details of the pretty-print table for SDF. The complete table is displayed in Figure 6.

From Figure 5, we see that an SDF module basically consists of a sequence of sections. This sequence is preceded by the keyword **module** and the name of the module. Similar to the formatting of the syntax definition in Figure 5, these sections will be formatted vertically and left indented. Keywords will be formatted using the KW operator. The keyword **module** and the module name are formatted horizontally. These formatting rules are described by the pretty-print rule below:

```
"module" Sdf-Id Section* → Module,
  V is=3 [ H  [ KW  [ "module"  ] _1 ] V vs = 1 [ _2 ] ]
```

An SDF module can contain zero or more imports sections. Each imports section consists of the keyword **imports** and a number of module names. Formatting the module names horizontally may exceed the line width when the imports section contains many module names. We therefore use the HV operator to allow the formatting of module names to capture multiple lines.

```
"imports" Sdf-Id+ → Section,
  H  [ KW  [ "imports"  ] HV  [ _1 ] ]
```

The context-free functions in Figure 5 are formatted in such a way that the elements at both sides of the arrow are left-aligned. This formatting is achieved using the alignment operator configured with four left-aligned columns. The fourth column (which is not visible from Figure 5) is used to format optional attributes.

```
"context-free" "syntax" CfFunction+ → SyntaxSection,
  V is=3 [ H  [ KW  [ "context-free"  ] KW [ "syntax"  ] ] A (l, l, l, l) [ _1 ] ]
```

sorts Sdf-Id Iterator CharClass Literal Module Section SyntaxSection CfFunction CfElem Attributes FunOpName ListOpName	**sorts** Sdf-Id Iterator CharClass Literal Module Section SyntaxSection CfFunction CfElem Attributes FunOpName ListOpName
(a) Narrow	(b) Wide

Figure 7: Different formatting depending on available width.

This entry specifies that context-free functions are formatted vertically (they are indented from the left by three spaces). The context-free functions together form a tabular with four columns. Text in all four columns is left-aligned.

Each context-free function in an SDF module forms a row in the tabular. Each row consists of four columns: the elements at the left hand side of the arrow, the arrow itself, the target sort of the production and optional attributes. The entries below are used to achieve the desired formatting of individual context-free functions:

```
FunOpName CfElem* "→" Sdf-Id  Attributes  → CfFunction,
  R [ H [ _1 _2 ] KW [ "→"  ] _3 _4 ]

Literal "(" {CfElem ","}* ")" "→" Sdf-Id Attributes  → CfFunction,
  R [ H [ _1 "(" _2 ")" ] KW [ "→"  ] _3 _4 ]
```

The pretty-print entries for the other language constructs are obvious and do not need any further discussion. The complete pretty-print table for the grammar of Figure 5 is displayed in Figure 6.

Using the Pretty-Print Table for SDF. Now that a pretty-print table for the language SDF has been constructed, we can instantiate the generic tool `asfix2box` to obtain a language specific pretty-printer for SDF. This instantiated pretty-printer can be used to generate BOX for terms over SDF according to the rules in the pretty-print table. The generated BOX terms can be translated to different output formats. Figure 5 for example, is obtained by translating BOX to LaTeX.

Depending on the available width for pretty-printing, the final output may differ according to the semantics of the HV and HOV operators. Figure 7 shows how the HV operator formats its sub-boxes differently depending on the available width for pretty-printing. Both figures show the sorts section of the SDF module depicted in Figure 5, pretty-printed with the rules defined in the table of Figure 6. Because of the limited horizontal space available in Figure 7(a), more lines are occupied to format the module names without exceeding the right margin than in Figure 7(b).

References

Borras, P. (1989). *PPML - Reference manual & compiler implementation*. INRIA, Sophia-Antipolis.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Lang, B., and Pascual, V. (1989). CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. Appeared as *SIGPLAN Notices* 24(2).

van den Brand, M. G. J. (1995). Pretty printing in the ASF+SDF Meta-Environment: Past, Present, and future. In M. G. J. v. d. Brand, A. van Deursen, T. Dinesh, J. Kamperman, and E. Visser, editors, *ASF+SDF'95: a workshop on Generating Tools from Algebraic Specifications*, number Technical Report, P9504, pages 155–174. `http://www.wins.uva.nl/research/prog/reports/`.

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

van den Brand, M. G. J., Klint, P., Olivier, P., and Visser, E. (1997a). AsFix. A structured data format for representation of parse trees with an extensive library with generic, language independent functionality. In particular a format for ASF+SDF specifications.

van den Brand, M. G. J., Klint, P., Olivier, P., and Visser, E. (1997b). ATerms: Representing structured data for exchange between heterogeneous tools. Technical report, Programming Research Group, University of Amsterdam.

van den Brand, M. G. J., Sellink, A., and Verhoef, C. (1997c). Generation of components for software renovation factories from context-free grammars. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153. Available at `http://adam.wins.uva.nl/~x/trans/trans.html`.

van den Brand, M. G. J., Kuipers, T., Moonen, L., and Olivier, P. (1997d). 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.

van den Brand, M. G. J., Sellink, A., and Verhoef, C. (1997e). Obtaining a COBOL grammar from legacy code for reengineering purposes. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer verlag.

van den Brand, M. G. J., Heering, J., and Klint, P. (1997f). Renovation of the old ASF+SDF Meta-Environment — current state of affairs. In A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer verlag.

van den Brand, M. G. J., Klint, P., and Olivier, P. (1999). Compilation and memory management for ASF+SDF. To appear in CC'99.

Brunekreef, J. (1995). *On Modular Algebraic Protocol Specification.* Ph.D. thesis, University of Amsterdam.

Brunekreef, J. (1996). A transformation tool for pure Prolog programs: the algebraic specification. Technical Report P9607, University of Amsterdam, Programming Research Group. Available by anonymous ftp at `ftp://ftp.wins.uva.nl/pub/programming-research/reports/1996/P9607.ps.Z`.

Chailloux, J., Devin, M., and Hullor, J.-M. (1984). Le_Lisp, a portable and efficient Lisp system. In *Proceedings ACM symposium on Lisp and Functional Programming*, Austin, Texas.

Deursen, A. v., Heering, J., and Klint, P., editors (1996). *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co.

van der Graaf, M. (1997). A specification of Box to HTML in ASF+SDF. Technical Report P9720, University of Amsterdam, Programming Research Group. Available by anonymous ftp at `ftp://ftp.wins.uva.nl/pub/programming-research/reports/1997/P9720.ps.Z`.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, **24**(11), 43–75. Most recent version available at URL: `http://www.cwi.nl/~gipe/`.

Hughes, J. (1995). The design of a pretty-printing library. In *First International Spring School on Advanced Functional Programming Techniques*, LNCS. Bastad, Sweden.

Kamperman, J. F. T. and Walters, H. (1993). ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica. `ftp://ftp.cwi.nl/pub/gipe/reports/KW93.ps.Z`.

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

Knuth, D. E. (1984). *The TEXbook*, volume A of *Computers & Typesetting*. Addison-Wesley. (Ninth printing, revised, October 1989).

Koorn, J. W. C. (1994). *Generating Uniform User-Interfaces for Interactive Programming Environments.* Ph.D. thesis, University of Amsterdam.

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

Rahtz, S. (1998). Hyperref. available at `http://www.tex.ac.uk/tex-archive/macros/latex2e/contrib/supported/hyperref/`.

Swierstra, D. S., Azero, P., and Saraiva, J. (1998). Designing and implementing combinator languages. In *Third International Summer School on Advanced Functional Programming*. Braga, Portugal.

Thanh, T. H. (1998). A TEX variant which can produce acrobat pdf instead of dvi. available at `ftp://ftp.cstug.cz/pub/tex/local/cstug/thanh/pdftex/`.

Visser, E. (1994). *ASF+SDF to LaTeX, User Manual*. University of Amsterdam, Programming Research Group.

Vos, K. (1990). *Pretty for an easy touch of beauty*. Master's thesis, Programming Research Group, University of Amsterdam.