

A Unified Format for Language Documents

Vadim Zaytsev and Ralf Lämmel

Software Languages Team
Universität Koblenz-Landau
Universitätsstraße 1
56072 Koblenz
Germany

Abstract. We have analyzed a substantial number of language documentation artifacts, including language standards, language specifications, language reference manuals, as well as internal documents of standardization bodies. We have reverse-engineered their intended internal structure, and compared the results. The Language Document Format (LDF), was developed to specifically support the documentation domain. We have also integrated LDF into an engineering discipline for language documents including tool support, for example, for rendering language documents, extracting grammars and samples, and migrating existing documents into LDF. The definition of LDF, tool support for LDF, and LDF applications are freely available through SourceForge.

Keywords: language documentation, language document engineering, grammar engineering, software language engineering

1 Introduction

Language documents form an important basis for software language engineering activities because they are primary references for the development of grammar-based tools. These documents are often viewed as static, read-only artifacts. We contend that this view is outdated. Language documents contain formalized elements of knowledge such as grammars and code examples. These elements should be checked and made available for the development of grammarware. Also, language documents may contain other formal statements, e.g., assertions about backward compatibility or the applicability of parsing technology. Again, such assertions should be validated in an automated fashion. Furthermore, the maintenance of language documents should be supported by designated tools for the benefit of improved consistency and traceability. In an earlier publication, a note for ISO [KZ05], we have explained why a language standardization body needs grammar engineering (or document engineering).

This paper presents a data model (say, metamodel or grammar) for developing language documents. Upon analyzing and reverse-engineering a wide range of language documents, which included international ISO-approved standards and vendor-specific 4GL manuals, we have designed a general format for language documents, the Language Document Format (LDF), which supports the

documentation of languages in a domain-specific manner. We have integrated LDF with a formalism for syntax definition that we designed and successfully utilized in previous work [LZ09a,LZ09b,Zay10a].

We have integrated LDF also with existing tools and methods for grammar engineering from our previous work; see the grammar life cycle at the top of [Figure 1](#) for an illustration. Furthermore, we have added LDF-specific tools, and begun working towards a discipline of *language document engineering*. There is support for creating, rendering, testing, and transforming language documents; see the document life cycle at the bottom of [Figure 1](#) for an illustration. Given the new format LDF, it is particularly important that there are document extractors so that one can construct consistent LDF documents from existing language documents.

In this paper, we will be mainly interested in LDF as the format for language documents, and the survey that supports the synthesis of LDF documents. The broader discussion of language document engineering including aspects of rich tool support is only sketched here, and deserves substantial future work efforts.

LDF can be seen as an application of literate programming [Knu84] ideology to the domain of language documentation: we aim to have one artifact that is both readable and executable. By “readable” we mean its readability, understandability and information retrievability qualities. By “executable” we assume a proper environment such as a compiler compiler (for parser definitions) or a web browser (for hyperlinked grammars). LDF provides us with a data model narrowly tailored to the domain; it allows us to focus on one baseline artifact which is meant for both understanding and formal specification. Other artifacts such as grammars, test sets, web pages, language manuals and change documents are considered secondary in that they are to be generated or programmed. The full implementation of this approach relies on a transformation language for language documents that we will briefly discuss.

Summary of contributions

- We have analyzed a substantial number of language documentation artifacts, including language standards, specifications and manuals of languages such as BNF dialects, C, C++, C#, Cobol dialects, Fortran, 4GLs, Haskell, Jovial, Python, SDF, XML, and other data modeling languages. Company-specific internal documents and software engineering books that document a software language (e.g., [GHJV95] with the well-known design patterns), were also researched. The objective of the analysis was to identify domain concepts and structuring principles of language documentation.
- We have designed the Language Document Format (LDF) to specifically support the documentation domain, and to make available language documents to language document engineering.

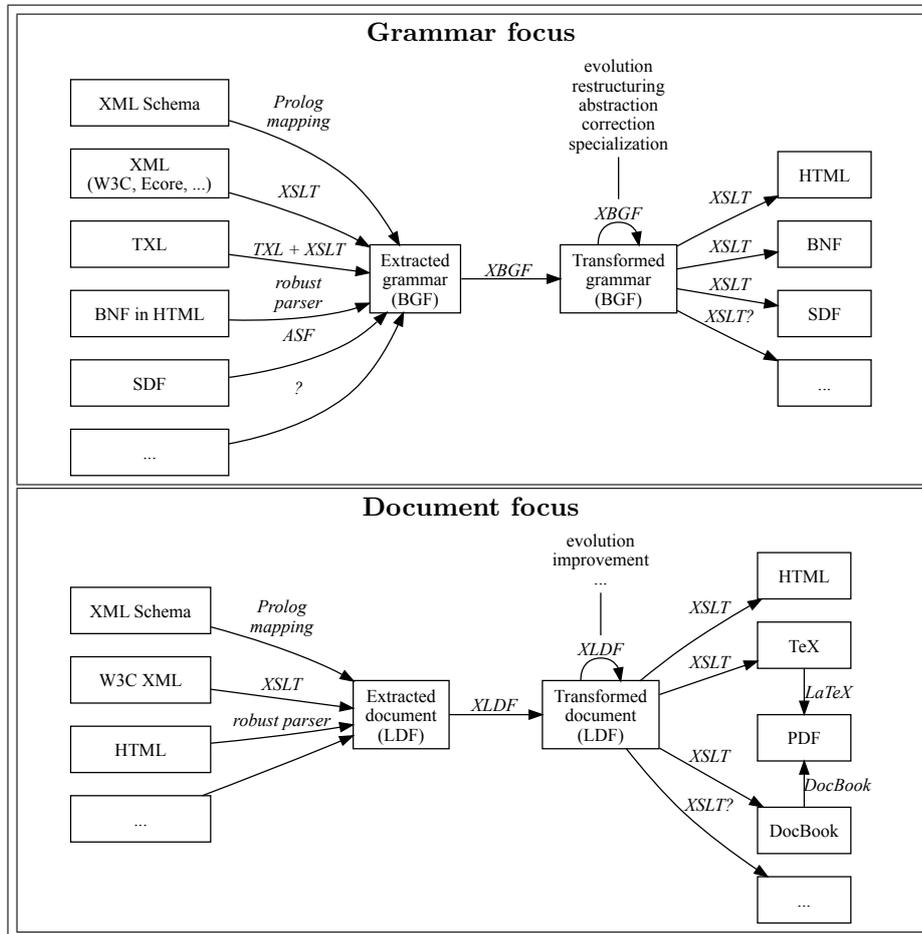


Fig. 1. Megamodels related to language document engineering. At the top, we see the life cycle of grammar extraction, recovery, and deployment. Grammars are extracted from existing software artifacts on the left, and represented in the unified format BGF. Grammars may then be subject to transformation using the XBGF transformation language. Parsers, browsable grammars, and other “executable” artifacts are delivered on the right. Such grammar engineering feeds into language document engineering. At the bottom, we see the life cycle of language document extraction, language (document) evolution, generation of end-user documents, extraction of grammars and test suites. Non-LDF documents can be converted to LDF through the extraction shown on the left. Document transformation may be needed for very different reasons, e.g., structure recovery or language evolution; see the reference to XLDF, which is the transformation language for LDF.

Validation

We have applied LDF to a number of language documentation problems, but a detailed discussion of such problems is not feasible in this paper for space reasons. For instance, we have applied language document engineering systematically to the documentation of XBGF—the transformation language for BGF grammars which is used extensively in our work on grammar convergence; the outcome of this case study is available online [Zay09]. In the current paper, we briefly consider mapping W3C XML to LDF, specifically W3C’s XPath standard; this case study is available online, too [W3C]. In the former case, we rely on a document extractor that processes XSD schemata in a specific manner. In the latter case, the extractor maps W3C’s XML Spec Schema to LDF.

More generally, the SourceForge project “Software Processing Language Suite” (SLPS) hosts the abovementioned two case studies, the LDF definition, tool support for LDF, other LDF applications, and all other grammars and tools mentioned in this paper and our referenced, previous work. For instance, we refer to the SLPS Zoo, slps.sf.net/zoo; it contains a collection of grammars that we extracted from diverse language documents. The next step would be to properly LDF-enable all these documents.

Road-map

The rest of the paper is organized as follows. §2 discusses the state of the art in language documentation as far as it affects our focus on a format for language documents and its role in language document engineering. §3 identifies the concepts of language documentation as they are to be supported by a unified format for language documents, and as they can be inferred, to some extent, from existing language documents. §4 describes the Language Document Format (LDF) in terms of the definitional grammar for LDF. It also provides a small scenario for language document transformation. §5 discusses related work (beyond the state of the art section). §6 concludes the paper.

2 State of the art in language documentation

As a means of motivation for our research on a unified format for language documents, let us study the state of the art in this area. The bottom line of this discussion is that real-world language documents are engineered at a relatively low level of support for the *language* documentation domain.

2.1 Background on language standardization

In practice, all mainstream languages are somehow standardized; the standard of a mainstream language would need to be considered the primary language document. For instance, the typical standard for a programming language entails grammar knowledge and substantial textual parts for the benefit of understanding the language.

Let us provide some background on language standardization. In particular, we list standardization bodies, and we discuss some of the characteristics of language standards. Standardization bodies that produce, maintain and distribute language standards, are, among others:

- American National Standards Institute (ANSI, since 1918), ansi.org
- European Computer Manufacturers Association (ECMA, since 1961), ecma-international.org
- Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA, since 1884), standards.ieee.org
- International Electrotechnical Commission (IEC, since 1906), iec.ch
- International Organization for Standardization (ISO, since 1947), open-std.org
- International Telecommunication Union (ITU, since 1865), itu.int
- Internet Engineering Task Force (IETF, since 1986), ietf.org
- Object Management Group (OMG, since 1989), omg.org
- Organization for the Advancement of Structured Information Standards (OASIS, since 1993), oasis-open.org
- Website Standards Association (WSA, since 2006), websitestandards.org
- World Wide Web Consortium (W3C, since 1994), w3.org

A language specification (programming language standard) is a complex document that may consist of hundreds of pages: the latest COBOL standard, ISO/IEC 1989:2002 [ISO02], has more than 800 pages; the latest C [ISO05] and C# [Sta06] standards contain over 500 pages each, C++ draft is already well over 1100 pages [ISO07]. It has not always been like that. For example, the Algol 60 standard [BBG⁺63] is not much longer than 30 pages, and yet, it claimed to contain a complete definition of the language. However, programming languages evolve, their specifications grow in size. Also, complicated structure of modern language documents reflects the complicated structure of modern programming languages and the associated ecosystems.

2.2 The language documentation challenge

Writing and maintaining such a document and keeping it consistent is as complex as writing and maintaining a large software system—these processes have a lot in common.

Defining a programming language in a standardized specification is often considered as a process that is executed just once. The dynamic and evolving nature of programming languages is frequently underestimated and overlooked [Fav05]. Not only software itself, but programming languages that are used to make it, evolve over time. This process usually comes naturally in the sense that the first version of a language does not have all the features desired by its creator. Also, new requirements may be discovered for a language, and hence, the language needs to be extended or revised. However, it is desirable for that process to be guided and controlled for the sake of the quality of resulting specifications.

There are tools like parsers and compilers whose development is based on a language specification. Inconsistencies in the language documents may lead to non-conformant language tools; such inconsistencies certainly challenge the effective use of the language documents. Languages need to evolve, and hence, it should be easy enough to evolve language documents. However, with the current practice of language standardization, evolution of language documents may be too ad-hoc, error-prone and labor-intensive; see, for example, our previous study on the language documentation for the Java Language Specification [LZ09b].

Overall, it is difficult to support language evolution for programming languages or software languages that are widely used. We contend that a systematic approach to language documents is an important contribution to a reliable and scalable approach to language evolution in practice.

2.3 Language documentation approaches

In practice, language documents are created and maintained with various *technologies*, e.g., L^AT_EX [ISO08], HTML [BBC⁺07], Framemaker [ISO02], home-grown DSLs based on the language being defined in the document [Bru05], XML Schema [Zay09], DITA, DocBook. The creation and maintenance of language document is also regulated by *practices* of design committees and standardization bodies or simply language document editors. The practices are often constrained by the technologies (or vice versa). We make an attempt to organize technologies and practices. To this end, we identify language documentation approaches.

The text- and presentation-oriented approach considers a language specification as a text document subject to text editing. The editor manually adds text to the document, manages section structure, moves around paragraphs and other units of text, performs layout and formatting operations. Typically, the text is meant to be immediately ready for presentation—perhaps even based on WYSIWYG.

The course of action for an editor of a language document is often described in a separate “change document” that is created before the actual change takes place or directly after it. The change document comprises a list of intended modifications. Once the editing process reaches a certain milestone, a new “revision” is delivered and stored in the repository. Once all the modifications approved by the language design committee are brought upon the main document, a new “version” is delivered and officially distributed within the terms of its license. This approach tends to utilize programs like Adobe Framemaker (ISO/IEC JTC1/SC22/WG4¹), Microsoft Word (Microsoft version of C# [Mic03]), etc. It is also possible to use HTML (early W3C [Rag97]) in such a way that the main document is edited manually and the changes are discussed and/or documented elsewhere.

¹ ISO/IEC JTC1/SC22/WG4 — COBOL Standardization Working Group, <http://www.cobolstandard.info/wg4/wg4.html>.

This approach involves significant low-level text editing. The links between the change documents and the main document revisions are often not verified. (Versioning and change tracking facilities can be too constraining.) Any structured content that is a part of a language document must be formatted in a way dictated by the medium: e.g., the formulæ can only use the symbols available in the font. It is also common to have several differently organized layers in the infrastructure: e.g., the main document is edited by one person following the instructions in the change document, but the change documents circulate in the form of co-authored Word documents.

The structure-oriented approach operates on documentation domain concepts such as “sections” or “divisions”. The approach may leverage existing editing software to support maintenance activities at the central repository of structured data. The approach also leverages backend tools that produce PDF, \LaTeX , and other types of deliverables. An example of such a documentation support system is DocBook [WM99]. It is a mature, well-documented, actively used technology. Microsoft is known to use DocBook to generate help files for Windows applications.

The separation between the content and its presentation can be sufficient in DocBook and similar systems. However, their orientation on books does not anticipate documents that have several intertwined hierarchies. For example, a grammar production that is a part of the corresponding section, is also a part of the complete grammar in the appendix, and should appear there automatically (as opposed to being manually cloned). In principle, one could leverage transformations (such as XSLT for DocBook) for the representation of the evolution of a (language) document. We are not aware of related work of this kind.

The topic-oriented approach operate in terms of “topics” that should be covered in order for the documentation to be complete. The DocBook counterpart in this group of approaches is Darwin Information Typing Architecture (DITA) [OAS07] which was designed specifically for authoring, producing and delivering technical information. IBM uses DITA for their hardware documentation. PDF, HTML, Windows help files and other output formats are possible. DITA is a relatively modern technology (2004 versus 1991 for DocBook), its support is growing, but is not as mature as for DocBook. A more lightweight approach is wiki technology that allows for topics to be left uncovered, showing explicitly which parts of the documentation are intended to be written in the future.

Language documentation is not naturally organized in topics and tasks, and thus is not anticipated by DITA. In principle, it is possible to use DITA to represent our proposed model (LDF). In order to do that, necessary element types—like grammar productions, code examples, notes concerning version differences, optional feature descriptions, possible implementation remarks, language engineering explanations—would need to be defined. Designated backends will also be required. There is no apparent benefit of using DITA, when compared to the XML/XSD-based approach that we chose for LDF’s description.

The XML Spec Schema, available from <http://www.w3.org/2002/xmlspec>, combines elements of structure and topic orientation in a manner that brings us closer to the domain of *language* documentation. The XML Spec Schema is a DTD that is used for some W3C recommendations. It is based on the literate programming tag set SWEB and the text encoding tag set TEI Lite. The Spec Schema covers some elements of the language documentation domain such as tagging facilities for grammar fragments; it does not capture the rich classification of sections in language documents.

3 Concepts of language documentation

As a preparatory step towards introducing LDF, we identify the concepts of the language documentation domain. We set up a control group to this end, and we also illustrate several concepts specifically for one member of the control group: the XPath W3C Recommendation.

3.1 Control group for the domain model

As we have indicated in the introduction, we have consulted a large set of language documents to eventually synthesize a unified format. For reasons of scalability, we have selected a smaller set of documents which we use here to present the results of our reverse-engineering efforts and to prepare the synthesis of a unified format for language documents. The reference set of documents has been chosen for its diversity. Table 1 shows some basic metadata about the language documents for the reference set. We describe the reference set in more detail as follows:

Property	IAL [Bac60]	Jovial [MIL84]	Patterns [GHJV95]	Smalltalk [Sha97]	Informix [IBM03]	C# [Sta06]	MOF [MOF06]	XPath [BBC ⁺ 07]
Body	ACM	DoD	—	ANSI	IBM	ECMA, ISO	OMG	W3C
Company	IBM	—	Pearson	—	IBM	Microsoft	—	—
Year	1960	1984	1995	1997	2003	2006	2006	2007
Pages	21	158	395	304	1344	548	88	111
Notation	BNF	BNF	UML	BNF	RT	BNF	UML	EBNF

Table 1. Some basic metadata of the standards chosen for the survey.

- **IAL** stands for International Algebraic Language that later became known as Algol-58 [Bac60]. It is historically the first programming language document, and as such it is the first time that the notation for specifying grammar productions was explicitly defined. The majority of all other standards produced over the following decades re-used this notation and extended it.
- **JOVIAL**, or J73 [MIL84] is a Military Standard of 1984, which “has been reviewed and determined to be valid” in 1994. It is approved for use by the Department of the Air Force and is available for use by all other Departments and Agencies of

the Department of Defense of USA. The version that was examined in this survey is a result of a second upgrade of the original language. It is less than 200 pages and very strictly composed: basically every section has a syntax, semantics and constraints subsections, with rare notes or examples. A traditional BNF is used for syntax, plain English for semantics.

- *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95] is a well-known book by Erich Gamma et al., which defines 23 well-known design patterns. Since design patterns can be considered a special language, their definition can be considered a language document—and Table 2 only proves that, letting the 400 pages long book’s structure fit in the general data model perfectly.
- **ANSI Smalltalk** [Sha97] is an NCITS J20 draft of 1997, 300+ pages long, it describes both the language (ANSI Smalltalk is derived from Smalltalk-80) and the Standard Class Library.
- **Informix** [IBM03] is an **IBM** manual for a proprietary fourth generation language. It exemplifies industrial standards, which are extensively strictly structured, contain minimum extra sections and have impressive volume. Informix specification utilizes “railroad track” syntax diagrams, which can be mapped more or less directly to EBNF.
- **C#** specification [Mic03,Sta06] is both an **ISO** and an **ECMA** standard, yet it was developed entirely within **Microsoft** and only approved by standardization bodies. The ECMA version used for this survey is 550 pages long and very loosely structured, explaining a lot of issues in running text and using arbitrary subsectioning.
- **MOF Core Specification** [MOF06] is a 90-pages long document describing Meta Object Facility. It uses UML and presents the information in a significantly different way, being oriented on diagrams, properties, operations and constraints. However, the overall information structuring turns out to be similar to conventional (E)BNF-based standards.
- The structure of **XPath W3C Recommendation** [BBC⁺07] is rather volatile, following the tradition of all other W3C recommendations. Each section contains one or several EBNF formulæ, the definition for a domain concept modeled by it and a body of text organized arbitrarily in lists and subsections.

3.2 Identification of concepts

The core domain concepts of LDF are: **synopsis**, **description** (an extended textual definition), **syntax** (associated grammar productions), **constraints** (restricting the use of the construct), **references** (to other language constructs), **relationship** (with other language constructs), **semantics**, **rationale**, **example**, **update** (from the previous language version), **default** (for absent parts). Four additional concepts can occur multiple times: **value** (associated named piece of metadata), **list** (itemized data), **section** (volatile textual content), **subtopic** (structured section).

Table 2 compares the documents from the reference sets in terms of the domain concepts. The cells in the table are filled with names of the sections,

subsections or otherwise identifiable paragraphs in the corresponding documents, unless noted otherwise. The coverage graph shows fully covered parts of LDF in black (represented by section names in table cells), partially covered in gray (“~” in a table cell means that the information is given but lacks any specific markup) and not covered in white (“—” in a cell means that this kind of information is absent from the language document). Gray concepts are interesting in so far that we face instances of implicit structure which can only be recovered with human intervention or advanced information retrieval techniques in the extraction tool.

3.3 Example: the XPath language document

The discovery of a language document’s structure and underlying domain concepts is a genuine process which we would like to sketch here for one example. We have chosen XPath 1.0 for this purpose—mainly because of its modest size.

The XML Path Language 1.0 specification [CD99] is one of the small standards, it contains only 32 pages in the printed version. We perform a cursory examination of it, trying to locate the domain concepts identified in the previous section:

Synopsis — is not automatically retrievable. We note that in some sections it is possible to use the first sentence as a synopsis (e.g., “Every axis has a principal node type.”), but we can only defer it to the transformation phase.

Description — if no specific structure can be recovered, we will treat all section content as a description.

Syntax — when we use the XML version of the specification as a source, all grammar production are easily identifiable by the `<scrap>` tag. A specific parser needed to be developed in XSLT to deal with the mix of plain text (e.g., for EBNF metasympols) and XML tags (e.g., for nonterminal symbols).

Constraints — some of the Notes are mentioning constraints (e.g., “The number function should not be used...”), but they are not automatically distinguishable from other Notes.

References — since all nonterminal names are always annotated with hyperlinks to the corresponding sections, no explicit references are required.

Relationship — there are mentions of relationships, some of which are even inter-documentary (e.g., the `mod` operator is being compared to the `%` operator in ECMAScript and the IEEE 754 remainder operation, but it is impossible to derive them naturally during recovery).

Semantics — is defined in plain English in running text.

Rationale — almost all Notes can be classified as providing rationales. We decide to map them all to rationales at the extraction step and sort the exceptions later with more advanced recovery techniques or programmed transformations.

Example — as typical for a W3C document, examples sections are inlined, but preceded by the sentences like “for example,” or “here are some examples”.

Update — XPath 1.0 is the first specification of its kind, which means that it contains no updates.

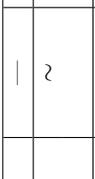
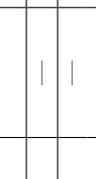
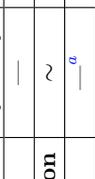
Domain concept	IAL [Bac60]	Jovial [MIL84]	Design Patterns [GHJV95]	Smalltalk [Sha97]	Informix [IBM03]	C# [Sta06]	MOF [MOF06]	XPath [BBC+07]
synopsis	—	~	intent	synopsis definition	~	~	~	—
description	~	—	motivation	~	usage	~	—	~
syntax	— ^a	syntax	structure	~	~	~	—	[NN] ^b
constraints	—	constraints	applicability	errors	restrictions	~	constraints	~
references	—	—	related patterns	—	references	~	—	~
relationship	—	—	consequences	return value, refinement	related	return type	—	~
semantics	—	semantics	collaborations	—	important	~	semantics	~
rationale	~	notes	implementation	rationale	GLS, ES ^c	note	rationale	note
example	examples	examples	sample code, known uses	—	~	example	—	~
update	—	—	—	—	—	— ^d	changes	—
default	—	—	—	—	note	default values	—	—
value	—	—	also known as	conforms to	—	—	—	—
list	~	—	—	messages, parameters	<i>terminals</i>	—	properties	~
section	~	—	—	—	~	~	—	~
subtopic	—	types	participants	—	fields	parameters, methods	operations	functions
Coverage of LDF								

Table 2. Mapping language definitions to domain concepts for language documentation

^a The absence of **syntax** elements means that grammar productions only occur within the designated part of a standard.

^b All productions in XPath standard are numbered and marked as [1], [2], etc.

^c GLS — Global Language Support, ES — an IBM Informix database type.

^d For every version of C#, there is a separate document that summarizes the changes brought to the language.

Default, Value — not found in this standard.

List — found inside the `<ulist>` and `<slist>` tags in the XML version of the document.

Section — Data Model section contains simple subsections.

Subtopic — every function description (the `<proto>` tag) can be treated as a subtopic. They are never long, but still can contain structured information such as lists and examples.

The global structure of the XPath specification is mapped to LDF in a straightforward fashion: for example, specific sections within the `<header>` such as Abstract and Status form a front matter part; `<body>` subsections populate the core part; `<back>` subsections become the back matter part. The mapping is mainly terminological: i.e., Status becomes “scope”, “Introduction” becomes “foreword”, etc.

4 A unified format for language documents

We will now describe the Language Document Format (LDF)—a unified format for language documents (say, language documentation). Given the motivation of LDF in previous sections, we will focus here on the actual language description for LDF. LDF’s description and related infrastructure are available online through the SLPS SourceForge project.² This section presents and discusses a full grammar for (current) LDF. The grammar notation we use here is a pretty-printed EBNF dialect called BGF [LZ09b,Zay10a], for BNF-like Grammar Format, which should be intuitively comprehensible. For brevity’s sake, some more routine (obvious) format elements are skipped in the discussion.

4.1 Language document partitioning

Consider the following productions concerning the **document** top sort and top level sections. For example, a **document** always contains one **document metainfo**, and one or more **parts**:

```
document:
    document-metainfo part+
document-metainfo:
    ((body number::string) | author::string+) topic::string status
    (version::string | edition::string) previous* date::time-stamp
body:
    ansi::ε | ecma::ε | ieee::ε | iso::ε | iso/iet::ε | itu::ε | iec::ε
    | ietf::ε | oasis::ε | omg::ε | wsa::ε | w3c::ε
status:
    unknown::ε | draft::ε | candidate::ε | proposed::ε | approved::ε
    | revised::ε | obsolete::ε | withdrawn::ε | collection::ε
    | trial::ε | errata::ε | report::ε
```

² LDF’s primary description leverages XSD: [shared/xsd/ldf.xsd](#)

```

previous:
  title::string (version::string | edition::string) uri::any-uri
part:
  part-metainfo section+
part-metainfo:
  part-role title::string? author::string* id::id?
part-role:
  front-matter:: $\epsilon$  | core-part:: $\epsilon$  | back-matter:: $\epsilon$  | annex:: $\epsilon$ 
section:
  placeholder | simple-section | lexical-section | structured-section
  | list-section
placeholder:
  index:: $\epsilon$  | full-grammar:: $\epsilon$  | list-of-tables:: $\epsilon$  | list-of-authors:: $\epsilon$ 
  | list-of-contents:: $\epsilon$  | list-of-references:: $\epsilon$ 

```

Most of the structural facets and elements should be self-explanatory. Let us highlight here the mandatory division of each language **document** into **parts**. In this manner, we encourage more structure than a simple list of top-level chapters. Existing documents vary greatly in the order of sections and their presentation. For instance, a “conformance” section is usually found in the **front matter** between the title page (**document-metainfo**) and the core chapters, but in the XPath 1.0 standard [CD99], it is the last core chapter. However, another typical front matter section, namely “normative references”, is found in XPath as one of the appendices. LDF’s parts encourage some grouping among the many sections.

4.2 Simple sections

“Simple” sections do not have any intricate internal structure and are usually found in **front matter**, **back matter**, **annex** or as an extra subsection in the usual structure.

```

simple-section:
  section-metainfo section-content::textual-content
section-metainfo:
  section-role title::string? author::string* id::id?
section-role:
  abstract:: $\epsilon$  | conformance:: $\epsilon$  | compliance:: $\epsilon$  | compatibility:: $\epsilon$ 
  | document-structure:: $\epsilon$  | notation:: $\epsilon$  | normative-references:: $\epsilon$ 
  | design-goals:: $\epsilon$  | scope:: $\epsilon$  | whatsnew:: $\epsilon$  | foreword:: $\epsilon$ 

```

As shown above, the **metainfo** of any simple section contains its role, a possible specific **title** (if absent, assumed to be equal to the role), a possible list of authors (if absent, assumed to be equal to the list of the document authors), and a possible **id** that is used to refer to this section from elsewhere. If the id is missing, one can still use an XPath expression over the document structure to access the section at hand. However, explicit ids are potentially preferred because of their greater robustness with regard to document evolution.

The list of **section roles** was synthesized from the reverse-engineered language documents. The roles should be intuitively understandable, but the con-

create wording may vary: a particular **foreword** can be called “introduction” the same way that an **obsolete** standard can be called “rescinded”.

4.3 Lexical sections

Our analysis showed that the sections about lexical details are significantly differently structured than other core sections or front/back matter simple sections.

```
lexical-section:
    lexical-metainfo lexical-section-content::textual-content
lexical-metainfo:
    lexical-section-role title::string? author::string* id::id?
lexical-section-role:
    line-continuations::ε | whitespace::ε | tokens::ε
    | preprocessor::ε | literals::ε | lexical-issue::ε
```

The list of **lexical section roles** was derived similarly to the list of **section roles**. The last role, **lexical issue**, is reserved for all other special cases, which all together occur more rarely than any of the other predefined roles.

4.4 List sections

Lists can occur anywhere in the **document** naturally, but we are concerned here with a specific kind of lists, referred to as **list section**. Such lists are commonly found in the **front matter**.

```
list-section:
    list-section-metainfo list-section-content::(term+)
list-section-metainfo:
    list-section-role title::string? author::string* id::id?
list-section-role:
    definitions::ε | abbreviations::ε | language-overview::ε
    | normative-references::ε
term:
    name::string definition::textual-content
```

Normative references list, if strictly and properly structured, forms a list section, but some standards *discuss* references instead of presenting them in an itemized list. This is the reason for normative references to appear as a possible role for a **simple section**.

4.5 Structured sections

A **structured section** is any section of a document that describes a separate language construct in a structured way. The following productions resemble the domain concepts of [Table 2](#).

```
structured-section:
    structured-section-metainfo structured-section-content
```

```

structured-section-metainfo:
  title::string author::string+ id::id?
structured-section-content:
  structured-section-element+
structured-section-element:
  subtopic::structured-section | references::list | placeholder
  | value::(key::string data::string) | (element-role simple-section)
element-role:
  normative-role | informative-role | specific-section::ε
normative-role:
  synopsis::ε | description::ε | syntax::ε | constraints::ε
  | relationship::ε | semantics::ε | default::ε
informative:
  rationale::ε | example::ε | update::ε

```

One can notice that the **metainfo** of a **structured section** is different from any other **metainfo** we have seen so far: there is no **role** involved since every section is dedicated to a language construct, and their set of such constructs varies from language to language, and varies greatly from paradigm to paradigm. The mandatory **title** is assumed to identify the language construct.

The **elements** of a **structured section** do have roles, and many of them have already been seen in [Table 2](#)—LDF as a DSL contains those domain concepts explicitly. The difference between a **specific section** and a **subtopic** becomes apparent here since they come up at different levels: the former is a possible role of a **simple section**; the latter is a **structured section** placed inside another **structured section**.

Theoretically, it is possible to have a **normative** (i.e., not just **informative**) section with examples—for instance, when we have a standard test set integrated in the language standard. Yet we have never seen this in practice.

4.6 Detailed content

Language documents, especially modern standards, have structured content even at the textual level of a section: hyperlinks, other references, tables, figures, formulae, lists, inline code fragments are among the most commonly used formatting elements.

```

textual-content:
  text-element+
text-element:
  empty::ε | code::string | text::mixed-type | figure | table | list
  | formula | sample::(string source::string) | production
figure:
  figure-metainfo figure-source+
figure-metainfo:
  short-caption::string? caption::string id::id?
figure-type:
  PDF::ε | PostScript::ε | SVG::ε | PNG::ε | GIF::ε | JPEG::ε
figure-source:

```

```

    type::figure-type (local-file::string | uri::any-uri)
table:
  header::table-row+ row::table-row+
table-row:
  table-cell::textual-content+
list:
  item::mixed-type+

```

For formulæ we reuse MathML, which definition is omitted here. For productions we reuse BGF [LZ09b,Zay10a]—the notation we use in this section.

We allow multiple **figure sources** so that the rendering tools for LDF can pick the source that is most convenient for the desired output format. For instance, a bitmap (**PNG**, **GIF**, **JPEG**) picture can be easily inserted into a web page, but a **PDF** file cannot be used in this manner. However, PDF may be preferred when LDF is rendered with pdfL^AT_EX.

4.7 Transformation of LDF documents

In the introduction, we mentioned the pivotal role of transformations for enabling the life cycle of language documents. In this section, we want to briefly illustrate such document transformations on top of LDF.

Let us set up a challenge for document transformation. Consider the two standards of XPath: versions 1.0 [CD99] and 2.0 [BBC⁺07]. They are vastly different documents, the one being three times the size of the other; with different author teams, and generally following different structure. Thus, there is no correspondence (neither explicitly defined nor easily conceived) between the two versions, except for the backwards compatibility section in the latter, which statements cannot be validated explicitly. However, using language document engineering—including document transformations—we should be able to represent the delta between the two versions through a script of appropriate transformation steps.

We are working on a transformation language for LDF, i.e., XLDF, which should be ultimately sufficient in addressing conveniently the above challenge. We refer to [Zay10b] for a more extensive discussion of the XLDF effort, and we sketch XLDF in the sequel. Our current XLDF design and implementation has been useful already for simpler problems. For [Zay09], we extracted a complete XBGF manual from the corresponding XML Schema, improved it with a few XLDF transformation steps and delivered a browsable version at the end. Such steps were needed because the assumed profile of XML Schema does not cover all LDF functionality.

XLDF is to LDF what XBGF [Zay09] is to BGF [LZ09a]. That is, in the same sense as grammars can be adapted programmatically with XBGF, language documents would be adapted with XLDF. Apart from `xldf:transform-grammar` operator that lifts grammar transformations, XLDF also contains operators for introducing and moving content. Consider the following illustration where a number of operators are applied in a transformation sequence.

```
xldf:add-section(structured-section:(title:"For Expressions",
```

```

        id:"id-for-expressions"),
        ...));
xldf:move-section(id:"section-Function-Calls",
                 inside:"id-primary-expressions");
xldf:rename-id(from:"section-Function-Calls",
              to:"id-function-calls");

```

One could even think of meta-level transformations that affect the grammar notation used in LDF. For instance, XPath 1.0's grammar notation uses single quotes, while XPath 2.0's grammar notation uses double quotes:

```

xldf:change-grammar-notation(start-terminal,"");
xldf:change-grammar-notation(end-terminal,"");

```

Executing such XLDF commands would have to involve transforming the transformations that pretty-print BGF productions. Higher-order transformations are one of the few challenges of the future work on XLDF.

5 Additional related work

A discussion about general documentation approaches was already included in §2. Below we will discuss related work more broadly.

We have carried out a previously published case study for Cobol [Läm05] where the grammar of Cobol is extracted from the Cobol standard; it is then refactored, made consistent and finally put back into the standard without detailed parsing of the standard's structure. This is a limited case of document engineering where only grammar parts are affected, but it goes beyond grammar extraction due to the persistent link between the grammar and the manual.

In [Wai02,Wei02], respected experts in the field of technical documentation advocate the engineering approach to documentation, as opposed to the artistic one—without though covering the kind of domain support or life cycle that is enabled by LDF.

Original verification techniques on language documentation are presented in [SWJF09]. Checks include formulae like “for all reading paths, a term X must be defined before it is used”. These ideas are complementary to ours.

The use of highly interactive eBooks for technical documentation is proposed in [DMW05]. In our domain, we use “browsable grammars” to enable interaction with language documentation.

Extraction for documentation is not necessarily restricted to text; extraction in [TL08] operates on graphic-rich documents. We could think of visual languages, UML-like models and “railroad tracks” kind of syntax diagrams.

One may also use Natural Language Generation (NLG) in deriving readable documents. For instance, in [RML98], the text is automatically generated with NLG when creating a final PDF out of the domain knowledge stored in a well-structured way. On a related account, several OMG technologies such as Knowledge Discovery Metamodel [KDM09] and Semantics of Business Vocabulary and Business Rules [SBV08] try to capture ontological concepts of the software domain and a means to make formal statements about them.

In [HR07], an industrial (Hewlett Packard) case study on documentation is presented. It involves user guides, manpages, context-sensitive help and white papers. The approach caters for a primary artifact from which a heterogeneous set of deliverables is generated with XMLware. To this end, disparate pieces of related information are positioned into final documents. A conceptually similar relationship between different documentation artifacts considered in [BM06], where a view-based approach to software documentation is proposed.

6 Concluding remarks

We have described the Language Document Format (LDF)—a unified format for language documents (say, language documentation). The unique characteristics of LDF are that i) it is derived by abstracting over a substantial and diverse body of actual language documents, and ii) it is integrated well with our previous research and infrastructure for grammar extraction, grammar recovery, grammar convergence, and grammar transformation.

Language document engineering with LDF brings us a step closer to the technical and methodological feasibility of life-cycle-enabled language documents so that state of the art documents could be migrated to a more structured setup of language documentation that is amenable to i) continuous validation, ii) systematic reuse of all embedded formal parts (grammars, examples) in other grammar engineering activities, and iii) transformational support for evolution.

There are these major areas for future work on the subject. First, we will further improve our infrastructure for engineering language documents so that we serve a number of input and output formats with sufficient quality, for example, in terms of “recall” for extraction or “roundtripping” for re-exporting to legacy formats. Second, our current approach to supporting evolution of language documents is not fully developed. Language design work and possibly tool support is needed for the transformation language XLDF. Third, a proper case study is required where an important language document (say, Cobol’s or Java’s standard) is converted into LDF, and the various benefits of our approach (language document engineering) are properly illustrated, with language evolution as one of the most important point.

References on language documentation

- Bac60. J. W. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In S. de Picciotto, editor, *Proceedings of the International Conference on Information Processing*, pages 125–131, Unesco, Paris, 1960.
- BBC⁺07. A. Berglund, S. Boag, D. Chamberlin, M. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. *W3C Recommendation*, 23 January 2007. www.w3.org/TR/2007/REC-xpath20-20070123.
- BBG⁺63. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised Report on the Algorithmic Language

- ALGOL 60. *Numerische Mathematik*, 4:420–453, Springer-Verlag, Berlin, Heidelberg, New York, 1963. International Federation for Information Processing 1962. Edited by Peter Naur.
- Bru05. R. Brukardt. ISO/IEC JTC1/SC22/WG9 Document N465—Report on “Grammar Engineering”, 2005.
- CD99. J. Clark and S. DeRose. XML Path Language (XPath) 1.0. *W3C Recommendation*, 16 November 1999. www.w3.org/TR/1999/REC-xpath-19991116.
- GHJV95. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- IBM03. IBM. *Informix 4GL Reference Manual*, 7.32 edition, March 2003.
- ISO02. ISO/IEC 1989:2002. *Information Technology—Programming Languages—COBOL*, 2002.
- ISO05. ISO/IEC 9899:TC2. *Information Technology—Programming Languages—C, Committee Draft WG14/N1124*, 2005.
- ISO07. ISO/IEC 14882. *Information Technology—Programming Languages—C++, Committee Draft WG21/N2315*, 2007. Available at <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2315.pdf>. Accessed in September 2010.
- ISO08. ISO/IEC N2723=08-0233. *Working Draft, Standard for Programming Language C++*, 2008.
- KDM09. Object Management Group. *Knowledge Discovery Metamodel (KDM)*, 1.1 edition, January 2009. Available at <http://www.omg.org/spec/KDM/1.1/>.
- Mic03. Microsoft .NET Framework Developer Center. *ECMA and ISO/IEC C# and Common Language Infrastructure Standards*, 2003. Available at msdn.microsoft.com/netframework/ecma and mirror sites.
- MIL84. MIL-STD-1589C. *Military Standard Jovial (J73)*, July 1984.
- MOF06. Object Management Group. *Meta-Object Facility (MOFTM) Core Specification*, 2.0 edition, January 2006. Available at <http://omg.org/spec/MOF/2.0>.
- OAS07. OASIS. DITA Version 1.1 Language Specification Approved as an OASIS Standard. *Committee Specification 01*, 31 May 2007. docs.oasis-open.org/dita/v1.1/CS01/langspec.
- Rag97. D. Raggett. HTML 3.2 Reference Specification. *W3C Recommendation*, 14 January 1997. www.w3.org/TR/REC-html32.
- SBV08. Object Management Group. *Semantics of Business Vocabulary and Rules (SBVR)*, 1.0 edition, January 2008. Available at omg.org/spec/SBVR/1.0.
- Sha97. Y.-P. Shan et al. *NCITS J20 DRAFT of ANSI Smalltalk Standard, Revision 1.9*, December 1997. Available at wiki.squeak.org/squeak/uploads/172/standard_v1_9_indexed.pdf. Accessed in June 2007.
- Sta06. Standard ECMA-334. *C# Language Specification*, 4th edition, June 2006. Available at <http://www.ecma-international.org/publications/standards/Ecma-334.htm>.
- W3C. Software Language Processing Suite: Mapping Spec Schema to LDF. <http://slps.sf.net/w3c>.
- WM99. N. Walsh and L. Meullner. *DocBook: The Definitive Guide*. O’Reilly, 1999.
- Zay09. V. Zaytsev. *XBGF Manual: BGF Transformation Operator Suite v.1.0*, August 2009. Available at <http://slps.sf.net/xbgf>.

Other references

- BM06. J. Bayer and D. Muthig. A View-Based Approach for Improving Software Documentation Practices. In *Proceedings of the 13th Annual IEEE Inter-*

- national Symposium and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 269–278, Washington, DC, USA, 2006. IEEE Computer Society.
- DMW05. G. Davison, S. Murphy, and R. Wong. The use of eBooks and interactive multimedia as alternative forms of technical documentation. In *Proceedings of the 23rd annual international conference on Design of communication (SIGDOC)*, pages 108–115, New York, NY, USA, 2005. ACM.
- Fav05. J.-M. Favre. Languages Evolve Too! Changing the Software Time Scale. In IEEE, editor, *8th International Workshop on Principles of Software Evolution, IWPSE*, 2005.
- HR07. K. Haramundanis and L. Rowland. Experience Paper: a Content Reuse Documentation Design Experience. In *Proceedings of the 25th annual ACM international conference on Design of communication (SIGDOC)*, pages 229–233, New York, NY, USA, 2007. ACM.
- Knu84. D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.
- KZ05. S. Klusener and V. Zaytsev. ISO/IEC JTC1/SC22 Document N3977—Language Standardization Needs Grammarware. Available at <http://www.open-std.org/jtc1/sc22/open/n3977.pdf>, 2005.
- Läm05. R. Lämmel. The Amsterdam Toolkit for Language Archaeology. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 137(3):43–55, 8 September 2005. Proceedings of the Second International Workshop on Metamodels, Schemas and Grammars for Reverse Engineering (ATEM’04).
- LZ09a. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proceedings of 7th International Conference on Integrated Formal Methods (iFM’09)*, volume 5423 of *LNCS*, pages 246–260. Springer, 2009.
- LZ09b. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 178–186. IEEE, September 2009.
- RML98. E. Reiter, C. Mellish, and J. Levine. Automatic Generation of Technical Documentation. *Readings in Intelligent User Interfaces*, pages 141–156, 1998.
- SWJF09. C. Schönberg, F. Weigl, M. Jakšić, and B. Freitag. Logic-based Verification of Technical Documentation. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 251–252, New York, NY, USA, 2009. ACM.
- TL08. K. Tombre and B. Lamiroy. Pattern Recognition Methods for Querying and Browsing Technical Documentation. In *Proceedings of the 13th Iberoamerican congress on Pattern Recognition*, pages 504–518, Berlin, Heidelberg, 2008. Springer-Verlag.
- Wai02. B. Waite. Consequences of the Engineering Approach to Technical Writing. *ACM Journal of Computer Documentation (JCD)*, 26(1):22–26, 2002.
- Wei02. E. H. Weiss. Egoless Writing: Improving Quality by Replacing Artistic Impulse with Engineering Discipline. *ACM Journal of Computer Documentation (JCD)*, 26(1):3–10, 2002.
- Zay10a. V. Zaytsev. Language Convergence Infrastructure. In *Post-proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE’09)*, July 2010. In print.
- Zay10b. V. Zaytsev. *Recovery, Convergence and Documentation of Languages*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, October 2010.