Towards an engineering discipline for GRAMMARWARE

Draft as of August 17, 2003

Paul Klint [1,2] Ralf Lämmel [3,1] Chris Verhoef [3]

[1] CWI — Centrum voor Wiskunde en Informatica, Amsterdam
[2] UvA — Universiteit van Amsterdam
[3] VUA — Vrije Universiteit Amsterdam

Abstract

Grammarware comprises grammars and all grammar-dependent software, i.e., software artifacts that directly involve grammar knowledge. The term grammar is meant here in the widest sense to include XML schemas, syntax definitions, interface descriptions, APIs, and interaction protocols. The most obvious examples of grammar-dependent software are document processors, parsers, import/export functionality, and generative programming tools. Even though grammarware is so omnipresent, it is somewhat neglected —from an engineering point of view. We lay out an agenda that is meant to promote research on improving the quality of grammarware and on increasing the productivity of grammarware development. To this end, we identify the problems with current foundations and practices, the promises of an engineering discipline for grammarware, its ingredients, and research challenges along the way.

Keywords: software engineering, grammarware, grammars, schemas, XML, APIs, program transformation, automated software engineering, evolution, co-evolution, software life cycle, generative programming, aspect-oriented programming

Contents

1 In need of well-engineered grammarware

Linguistics vs. information technology In linguistics, one has to face a tremendous multitude of human languages. The overall attack to deal with such diversity and complexity is to try to understand "the system of principles, conditions, and rules that are elements or properties of all human languages . . . *the essence of human language*" [21]. (This is Chomsky's controversial definition of the "universal grammar".) Such research cannot be separated from sociology, and other human sciences. Similarly, in information technology, we are faced with a tremendous multitude of technical notations, APIs, interaction protocols, and programming languages. Here, the overall attack must be, again, to understand the principles, conditions, and rules that underly all these 'grammars' and that enable their efficient use, reuse and evolution as well as their seamless interoperation. Grammars cannot be reduced to a few formal aspects such as the Chomsky hierarchy and parsing algorithms. It is in the interest of a more complete software engineering discipline that we make it grammar-aware by paying full attention to the *engineering* aspects of grammars and grammar-dependent software.

Engineering of grammarware We coin the term 'grammarware' to comprise not just grammars in the widest sense but also grammar-dependent software, i.e., all software artifacts that directly involve grammar knowledge. To see what we mean by 'engineering of grammarware', we will consider a few scenarios:

- As a developer of database applications, you want to make a transition to a new screen definition language, which has to be adopted gradually for all existing and future information systems in your company.
- As a developer of Commercial Off-The-Shelf software, you want to import user profiles in order to promote the user's transition from an old to a new version, or from a competing product to yours; think of web browsers.
- As an object-oriented application developer, you want to revise your application to take advantage of a new inhouse API, or to use the latest release of a standard API such as the SWING toolkit for user interfaces as part of the Java platform.
- As a manager for software configuration, you want to adapt your make infrastructure to work for a new version or a different dialect of the tools make, automake, or others. Also, you want to migrate from one shell dialect to another.
- As a developer of an inhouse domain-specific language (DSL), you give up on trying to provide an upward-compatible redesign of the DSL, but you want to provide at least a tool to convert existing DSL programs.
- As an online service provider, you want to meet your clients' requirements to serve new XML-based protocols for system use. For example, you want to replace an ad-hoc, CGI-based protocol by instant messaging via Jabber/XMPP.

By "engineering of grammarware" we mean that such scenarios should normally be realised in a way that the involved grammars are systematically recovered, designed, adapted, tested, customised, implemented, and so forth. While there is certainly a body of versatile techniques available, grammarware is typically not treated as an engineering artifact.

A research agenda The present paper serves as an agenda for the stimulation of a research effort on an engineering discipline for grammarware. This emerging discipline is expected to improve the quality of grammarware and to increase the productivity of grammarware development. The agenda substantiates that we need to join efforts in the software engineering and programming language communities to make progress with this goal as opposed to small-scale, short-term research activities. The agenda also attempts an identification of a basis for such a joint effort. As an outlook, initial scientific meetings, regular scientific events, and special issues in journals are needed to progress from here.

The contribution of the agenda Because grammarware is so omnipresent, an effort on an engineering discipline for grammarware will be to the advantage of software in general. Furthermore, the proposed discipline will invigorate automated software engineering. This is because software development tools such as CASE tools, compilers, refactoring browsers and others are prime forms of grammarware. In essence, the required research has to provide foundations as well as best practices for engineering of grammarware. This research faces major challenges. Most notably, a grammar tends to reside in several artifacts that all deal with different grammar uses in different grammar notations. Also, grammar structure tends to be all-dominant and other aspects end up being entangled in grammarware. This makes it rather difficult to keep track of all grammar dependencies in grammarware, and to master evolution of grammars as well as co-evolution of grammarware. The present agenda integrates the body of knowledge that addresses such crucial aspects of the proposed discipline.

Structure of the agenda In Sec. 2, we recall the *omnipresence* of grammarware in software systems. This is followed by a rude awakening in Sec. 3, where we substantiate that the reality of dealing with grammarware must be called *hacking*. In Sec. 4, we analyse the *dilemma* underlying the current situation. Cutting this Gordian knot prepares the ground for a significant research effort on engineering of grammarware. In Sec. 5, we lay out the *promises* of an engineering discipline for grammarware. In Sec. 6, we list the so-far identified *ingredients* of the emerging discipline. Special attention is paid to the role of *automated transformations* in Section 7. This is followed by a related work discussion in Sec. 8, which substantiates that we are indeed concerned with an *emerging discipline* rather than a hypothetical one. Ultimately, in Sec. 9, we compile a substantial list of *research challenges*.

Acknowledgement We are grateful for collaboration with Steven Klusener and Jan Kort. We also grateful for discussions with Gerco Ballintijn, Mark van den Brand, Jim Cordy, Jan Heering, Hayco de Jong, Wolfgang Lohmann, Erik Meijer, Bernard Pinon, Günter Riedewald, Wolfram Schulte, Ernst-Jan Verhoeven, and Andreas Winter. Finally, we are grateful for opportunities to debug this agenda during related colloquia at the 2nd Workshop Software-Reengineering in Bad Honnef, at the 54th IFIP WG2.1 meeting in Blackheath – London, at the AIRCUBE 2003 workshop hosted by INRIA Lorraine & Loria, at Free University of Amsterdam, Free University of Brussels, University of Kaiserslautern, University of Karlsruhe, University of Rostock, and at the Software Improvement Group in Amsterdam.

2 Omnipresence of grammarware

Grammar forms and notations Grammarware comprises grammars in the widest sense. Here is a categorisation of grammars with an indication of all the grammar notations in use:

- *Definitions of interchange and storage formats* for application data — in the form of XML schemas, or declarations of serialisable data in a programming language, or domain-specific formalisms such as ASN.1 [31] (used primarily in the telecommunication context).
- *Interface descriptions* for abstract datatypes, components in component-based or distributed applications — in the form of CORBA's IDLs, or interfaces in the sense of the facade design pattern [38], or APIs offered by class libraries.
- *Specifications of interaction protocols* — normally in a form with specification elements for structural concerns (i.e., sequences, alternatives, iteration, etc.); interaction protocols describe how groups of objects or agents cooperate to complete a task [90, 79], e.g., there are two forms of interaction diagrams in UML, namely sequence and collaboration diagrams.
- *Concrete and abstract syntax definitions* of programming languages and domain-specific languages [26], and meta-models — in Backus-Naur Form, or in the abstract syntax description language ASDL [112] (used for interoperable compiler technology), or in graphical notations for BNF (e.g., the syntax diagrams in [46]), or in the form of algebraic signatures, or object-oriented class dictionaries subject to the design patterns visitor, composite and interpreter [38].
- *Definitions of intermediate and internal representations* such as annotated parse trees [67], document models [30], intermediate representations in compilers [35, 23], system logging in the sense of the command design pattern [38] — using some of the grammar notations discussed for the other categories.

Grammar-dependent software In addition to plain grammars, the term grammarware also comprises grammar-dependent software, i.e., software artifacts that directly involve grammar knowledge. Again, the term software artifact is meant here in the widest sense including source code, documentation, and models or specifications at the levels of design and analysis. It is fair to say that grammarware is everywhere. To use a metaphor: grammars are the hardware of software, say grammars are the backbone of (grammar-dependent) software. Here are examples of grammar-dependent software, which are clearly shaped by grammars:

- Application generators [108] or tools for generative programming [32], aspect-oriented weavers [59, 3], tools for program synthesis [105], tools for automated software engineering, CASE tools (e.g., Rational Rose).
- Distributed or component-based applications where grammars occur in the sense of required vs. supported interfaces as well as formats for input vs. output; most business-critical systems are of that kind.

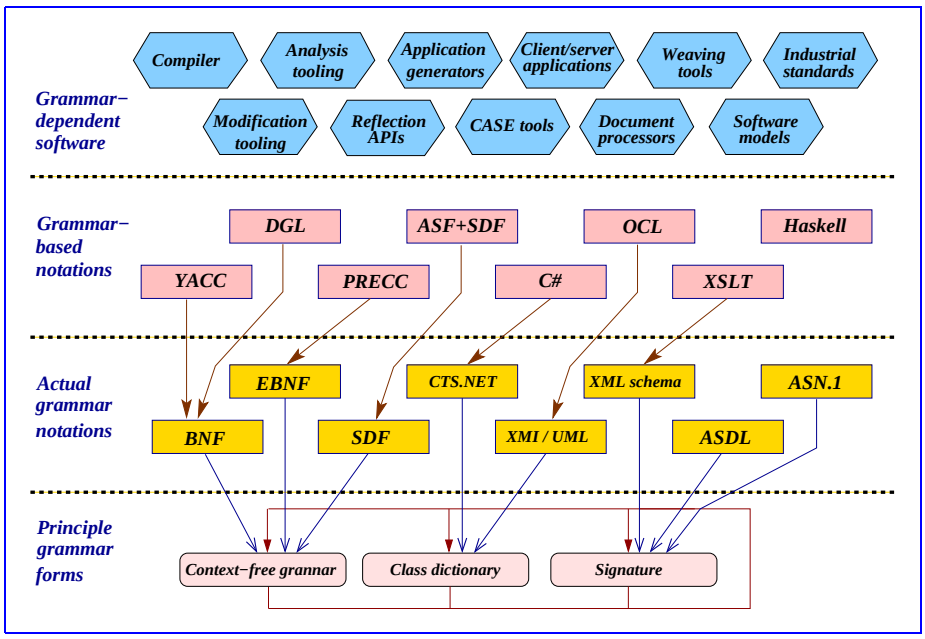

Figure 1: So much grammarware, so little engineering.

- Functionality in language implementations, e.g., compilers, animators, documentation generators [106, 84], profilers, debuggers [91]: a typical compiler involves several grammars and several components that use grammars, e.g., components for optimisation, program specialisation [51], preprocessing, parsing, code generation [33, 37].
- Functionality for automated software re-documentation [27], analysis and modification, e.g., source-code model extractors [69], transformation functionality [16, 76], pretty printers [17, 53], pre- and post-processors.
- Reference manuals, style guides, and industrial standards where grammars are used for the documentation and the definition of an API, or a programming language (e.g., [46]), or an interaction protocol.

Our criterion for saying that a software artifact directly involves grammar knowledge is that grammar structure is encoded in the artifact, e.g., some productions defined by a context-free grammar, or some patterns defined by a signature or a schemas. For example, an XSLT program, via its templates, makes assumptions about element type declarations that would need to be contained in the XML schema of a suitable input document. An even stronger form of dependency holds if artifacts are *systematically derived* from a grammar. For example, the definition of functionality for the serialisation of object structures follows the underlying class structure very closely.

Grammarware at a glance In Fig. 1, we visualise the omnipresence of grammarware. The bottom layer lists *principle grammar forms*. The round-trip of the arrows indicates that context-free grammars, class dictionaries and algebraic signatures can be translated into each other — in a formal sense. Nevertheless, these forms make original contributions. Context-free grammars are immediately prepared to deal with concrete syntax as needed for the definition of programming languages. Signatures are more concerned with abstract syntax and other abstract representations. Class dictionaries appeal to the object-oriented paradigm; they model inheritance and aggregation relationships. The next layer lists *actual grammar notations*; the arrows connect these notations with the principle form. Context-free grammars are normally specified using some notation for (extended) Backus Naur Form (BNF) [4, 36]. Further context-free notations typically add some specification constructs. For example, the SDF syntax definition formalism [41, 109] adds constructs for modularisation and disambiguation. We list two notations for class dictionaries: the common type system (CTS) for .NET's object structures and the XMI meta-data interchange format for UML models. For signatures, we list ASDL [112] and ASN.1 [31] as formalisms for abstract syntax definition. We also list XML schemas, which serve for the representation of semi-structured data. The layer above actual grammar notations deals with *grammar-based notations*. These notations add specification or programming constructs to a basic grammar notation. The semantics of such grammar-based notations can involve domain-specific elements. For example, YACC [50] and PRECC [18] serve for parsing with actions for parse-tree construction and others, whereas DGL [85] is not for parsing but for test set generation. Many programming languages, e.g., C# and Haskell, are grammar-based notations when the types of the language are viewed as grammars. Several specification languages are grammar-based notations, e.g., ASF+SDF, which is a marriage of the SDF syntax definition formalism and the ASF algebraic specification formalism [7]. The top layer in Fig. 1 lists typical categories of grammar-dependent software.

Multi-dimensional grammarware It is crucial to notice that software artifacts can depend on several grammars at the same time in quite different ways. In an object-oriented program, for example, there is a predominant class structure (i.e., a grammar). In addition, the program's methods are likely to employ some APIs (i.e., grammars). Also, there could be an interaction protocol (i.e., a grammar) assumed for the use of the program's services. Furthermore, the program could be subject to wrapping according to the adaptor pattern [38]. That is, the program's services are made available to clients that assume a specific interface or message protocol (i.e., a grammar).

Tangling of grammarware Grammar knowledge that is involved in a software artifact is not necessarily available in a pure form. For example, the typical industrial compiler does not employ a parser generator, but its frontend is hand-crafted where one can at best claim that a grammar is encoded in the frontend. More generally, grammars tend to be entangled in functionality of grammar-dependent software in a more or less traceable manner. There is normally no simple way to replace entangled grammars; so they are indeed all-dominant. An XSLT program, for example, refers to the underlying XML schema in all of its templates on the basis of the XPath parts.

Grammars as structural concerns We want to draw a line here regarding the question what we count as grammars. Regarding the purpose of grammars, we view grammars as means to deal with primarily *structural* concerns in software development rather than inherently behavioural ones. However, this borderline is somewhat fuzzy because grammars are sometimes used as structural means to describe behaviour — with UML's interaction diagrams being a good example. Regarding the expressiveness of grammars, Fig. 1 suggests that we indeed restrict ourselves to context-free grammar-like formalisms. For example, we refrain from considering arbitrary *attribute* grammars [64, 93] as grammars in our sense. (In fact, attribute grammars belong to the layer of grammar-based notations in Fig. 1. A separation of context-free structure and attribute computations resembles the distinction of a predominant class structure vs. the method implementations in an object-oriented program.) However, it is generally acknowledged that context-free grammar-like formalisms lack convenience and expressiveness to deal with advanced structural concerns such as the definition of unambiguous concrete syntax or context conditions. Hence, we have to take more expressive notations into account when this is considered essential for structural concerns.

3 State of the art: grammarware hacking

At this point, the reader might face the dilemma that we describe later more closely: "This omnipresence is obvious. Somehow we managed to deal with all these kinds of grammarware for decades. So what?" Here is an important observation:

> Given the omnipresence of grammarware, one may expect that grammarware is treated as an engineering artifact — subject to reasonable common or best practices. In reality, grammarware is predominantly treated in an ad-hoc manner.

To give a concrete example, we consider the development of parsers for software re- and reverse engineering; see Fig. 2. The common approach is to manually encode a grammar in the idiosyncratic input language of a specific parser generator. The only grammarware tool that occurs in this process is a parser generator. By contrast, an engineering approach would be based on the following steps:

- A neutral grammar specification is *recovered semi-automatically* from the grammar knowledge, e.g., from a semi-formal language reference. Extraction and transformation tools are involved in this step. Thereby, the link between language reference and the ultimate parser is preserved.
- Parser specifications are *derived semi-automatically* from the grammar specification using a generative programming approach. Different parsing technologies can be targeted as opposed to an eternal commitment to a specific technology.

This approach appears to be feasible because we and others have exercised parts of it for a string of languages [101, 48, 75, 54, 80, 68], e.g., Cobol, which is widely used in business-critical systems, and the proprietary language PLEX used at Ericsson.

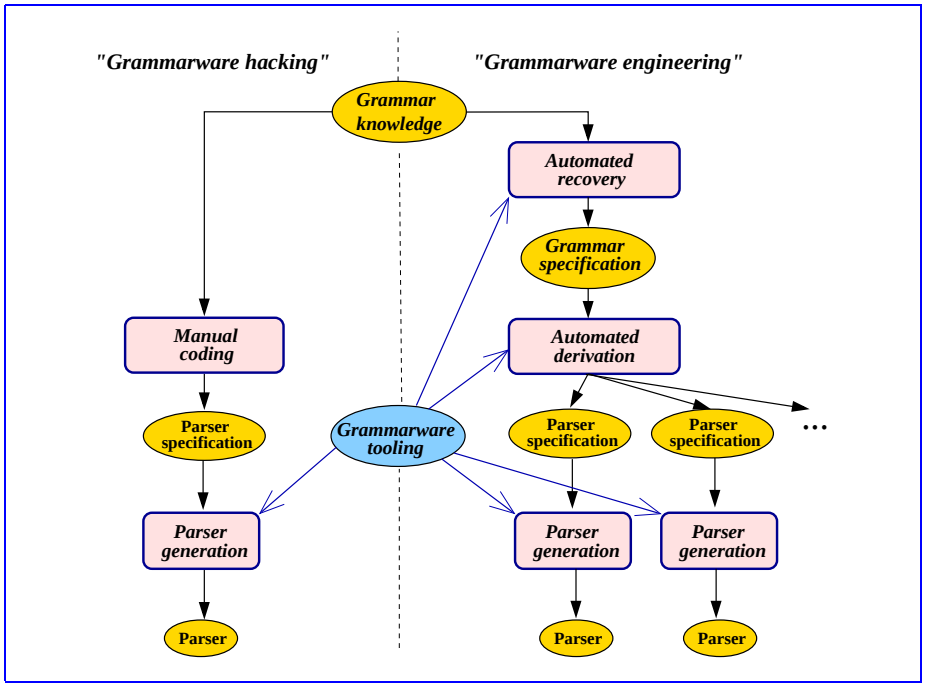

Figure 2: Parser development in the context of software re- and reverse engineering.

Lack of best practices Our claim about grammarware hacking can be substantiated with a number of general observations that concern the treatment of grammars in software development:

- There is no established approach for performing grammar evolution in a traceable manner — not to mention the even more difficult problem of co-evolution of grammar-dependent software. This is a major problem because grammar structure is undoubtedly subject to evolution, e.g., an API of a long-living class library, or the interfaces in a distributed application for which enhancements are continuously necessary.

- There is no established approach for maintaining consistency between the incarnations of conceptually the same grammar. It is, in fact, quite common to be faced with various incarnations. For example, a grammar tends to reside in various artifacts, e.g., in a CASE tool *and* in the documentation of its meta-model. Also, in a given artifact, several incarnations of the same grammar may reside, e.g., a concrete syntax *and* an abstract syntax in a compiler frontend.

- Grammars are immediately implemented using specific technology, which implies the use of idiosyncratic notations. An obvious example is parser specification in YACC [50]. Using idiosyncratic notations in the first place makes it difficult to alter the chosen technology or application context, e.g., to derive a pretty printer rather than a parser. (A more subtle example is the use of attributes in XML, which are often said not to be a part of the data but to provide information for the software that wants to manipulate the data. This distinction between content and other information is rather fragile and it hampers evolution.)

Lack of comprehensive foundations In fact, there is not just a lack of best or common practices. Even designated fundamental notions are missing. For example, there is no comprehensive theory of testing grammarware; neither is there one for transforming grammarware. Some further topics that are largely unexplored for grammarware are version management, quality assessment, design patterns, and debugging. Such foundations are needed for the sound provision of best practices that treat grammarware as an engineering artifact.

4 The grammarware dilemma

We have shown that even though grammarware is so omnipresent, it is somewhat neglected — from an engineering point of view. Is the lingering software crisis maybe to some extent a grammarware crisis? Here is what we call the grammarware dilemma:

> Improving on grammarware hacking sounds like such a good idea!
> Why did it not happen?

We think that the answer to this question lies in a number of grammar myths. These myths are like barriers for anyone who wants to do research on grammarware. So by addressing these myths, we hope to prepare the ground for working on an engineering discipline for grammarware.

Myth "Grammars are dead" While grammars are neglected engineering artifacts, grammars in the sense of definitions of formal languages are well-studied subjects in computer science. For the last three decades, the Chomsky hierarchy and parsing algorithms have formed integral parts of university curricula. This intensive exposition to grammars makes many of us maybe believe that grammars are a 'buried subject'. To refresh the researcher's liaison with grammars, it really needs a re-commencement, and a focus on engineering concerns, which were largely dismissed in the past. The current XML hype seems to be helpful in this respect.

Myth "Engineering of grammarware = parser generation" One might feel tempted to think that the widely established use of parser generators testifies a reasonable degree of engineering for at least syntax definitions or parsers. However, reducing an engineering discipline for grammarware like this is as narrow as reducing software engineering to coding. Nevertheless, even parsing by itself has to be reconsidered from

an engineering point of view. Many compiler developers refuse using parser generators for various reasons. For example, the means to customise the generated parsers are often considered insufficient. In any case — with and without parser generators, industrial parser development requires considerable tweaking, and hence remains a black art [15, 8].

Myth "XML is the answer" Recall the question: what are the software engineer's methods to deal with all forms of grammarware? XML is in need of an engineering discipline as much as any other grammar notation and application domain. For example, issues of schema evolution, co-evolution, and testing of schema-based software are all urgent research topics even in the 'narrow' XML context. The proper engineering of formats contributes to the interoperability promise of the XML age. Besides, it is important to notice that XML is merely an interchange format as opposed to a universal grammar notation. So while XML specifically addresses the issue of semi-structured data, it does not address the issue of concrete syntax in the sense of programming languages. However, the latter issue is obviously of importance for our society's software assets, which are normally encoded in some programming language.

Myth "Grammarware is all about programming languages" The significance of the engineering discipline for the 'narrow' programming language context is certainly considerable, and it is not so much about compiler development, but more urgently about software maintenance [74]. However, nowadays' software assets depend more and more on grammars other than programming language syntaxes, namely APIs, exchange formats, interaction protocols, and domain-specific languages. Just think of the many APIs that nowadays come and go. Those APIs that stay for longer, tend to evolve. The introduction, the evolution, and the displacement of such grammars are prime issues to be addressed by the proposed engineering discipline.

In need of a paradigm shift To summarise, we visualise a comparison of the mythical view and the proposed view on grammarware in Fig. 3. Left side: one might feel tempted to just count the lines of code spent on grammars in compiler frontends in relation to all software. This is not likely to trigger an effort on engineering of grammarware. Right side: we emphasise the impact ratio of grammars. That is, they make it into other software by direct or indirect dependencies; see the many arrows departing from the inner circle. We also emphasise that the grammars that reside within compiler frontends only provide a fraction of all the grammars there are. In fact, nowadays, more and more grammar lines of code are in existence that are concerned with structural aspects other than concrete or abstract syntax for programming languages.

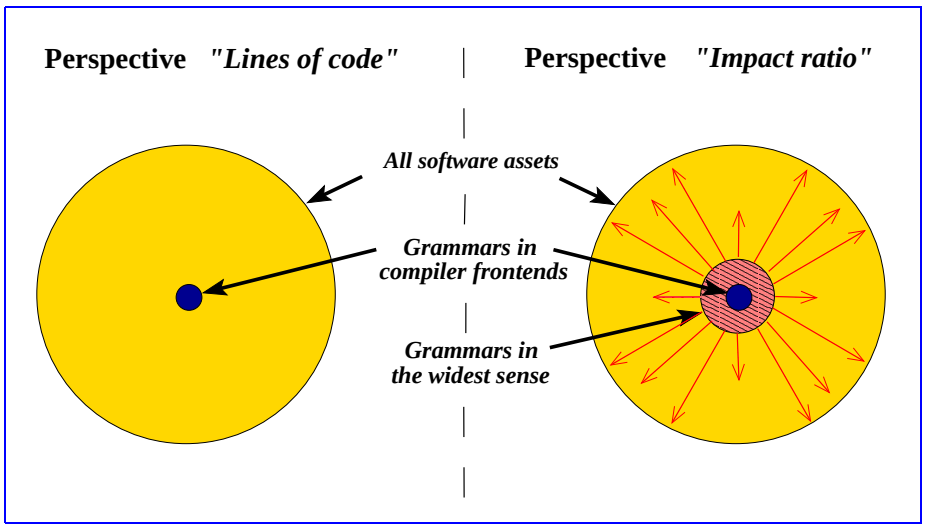

Figure 3: The role of grammars from two different perspectives.

5 Engineering of grammarware —promises

An engineering discipline for grammarware clearly contributes to the software engineering science. The new discipline poses certainly several research challenges, but it is maybe even more important to stress its potential benefits for IT. At a general level, the overall promise of the engineering discipline is to improve the quality of grammarware and to increase the productivity of grammarware development. Recall that grammarware is everywhere. Any business-critical system depends on a larger number of grammars that shape its architecture or that reside in the processes for its development and maintenance. So quality and productivity gains for grammarware will be to the advantage of software systems in general. To substantiate these general remarks, we will first review showcases. We will then go on to identify some more detailed promises on the basis of these showcases and further scattered experiences with viewing grammarware as an engineering artifact.

Showcase: grammar recovery Using elements of the emerging engineering discipline for grammarware, two of the present authors were able to rapidly recover a relatively correct and complete syntax definition of VS Cobol II [75]. The starting point for this recovery project was IBM's industrial standard for VS Cobol II [46]. The syntax diagrams had to be extracted from the semi-formal document, and about 400 transformations were applied to the raw syntax in order to add missing constructs, to fix errors, and to ultimately obtain a grammar that could be used for parsing. The recovery project was completed in just a few weeks, which included the development of simple tools for diagram extraction and grammar transformations. After that period, we were able to parse the several millions lines of VS Cobol II code that were available to us. (Additional effort would be needed to develop general, mature tools, to deploy the syntax

definitions in different industrial settings. We refer to [75] for a detailed discussion of this showcase.) Key to success was a systematic process, automation of grammar transformations, and separation of concerns such as development of a grammar specification vs. an operational parser committing to specific parsing technology. The underlying concepts matured during a series of similar recovery projects [14, 101, 16]. The recovered syntax definition for Cobol is widely used by tool developers and researchers around the world. This was the first freely available, high-quality syntax definition for Cobol in the 40 years of this language. (The costs for the industrial development of a Cobol grammar imply that it is considered intellectual property. Recall that most business-critical code today still resides in Cobol portfolios.)

Showcase: API-fication Generic language technology is a primary research theme in our team. Here our work is centred around the ASF+SDF Meta-Environment [60, 9], which supports executable specification of language-based tools, generation of parsers, interactive language-based tools, and others. The current system is the result of many man years of design, development and evolution. The system is being used in industrial applications dealing, for example, with software renovation and application generation from domain-specific specifications [10]. The development of generic language functionality normally involves a generic format for data representation. In the case of the ASF+SDF Meta-Environment, the ATerm format [11] is employed for this purpose. Such formats encourage programmers to incorporate structural knowledge of the manipulated data into the code, which leads to heavily tangled code. Such tangling is inherent to generic functionality when it is encoded in mainstream languages. In the case of the C- and Java-based ASF+SDF Meta-Environment, structural knowledge of parse-tree formats was scattered all-over the system. In fact, there were several parse-tree formats, and other representation formats for the kind of data structures and notations used in the system. In [52], de Jong and Olivier describe the 'API-fication' of the ASF+SDF Meta-Environment. By API-fication, we mean the process of replacing low-level APIs such as arbitrary ATerms by higher-level APIs for parse trees or other formats. An essential element is here that these APIs are preferably generated from grammars (again in the widest sense). The generated API amounts to a set of Java methods or C functions that provide access functionality according to the grammar at hand. The API-fication of the ASF+SDF Meta-Environment resulted in the elimination of almost half (!) of the manually written code; see [52] for details.

Productivity of grammarware development The showcase for the recovery of a Cobol grammar suggests that an important promise of the emerging engineering discipline is productivity. Indeed, other known figures for the development of a quality Cobol grammar are more in the range of two or three years [75, 74]. But why is such a speedup important for IT? We answer this question in [74]. In essence, the ability to recover grammars for the 500+ languages *in use* enables the rapid production of quality tools for automated software analysis and modification. Such tools are needed to make software maintenance and other parts of software engineering scalable in the view of software portfolios in the millions-of-lines-of-code range. The reality of producing tools for automated software analysis and modification is that parser hacking domi-

nates the budget of corresponding projects or makes them even unaffordable. Major productivity gains are by no means restricted to grammar recovery. For example, testing grammarware with manually designed test suites alone is time-consuming. Some aspects of testing can be both effectively and efficiently dealt with via automated testing, e.g., stress testing or differential testing of different versions or products from different vendors. We refer to [86, 104] for success stories on automated testing reported by others.

Maintainability and evolvability of grammarware The API-fication showcase suggests maintainability as a promise. This is not just because of the code size that was cut in half, but the resulting code is also better prepared for changing formats. This was indeed the triggering motivation for the API-fication of the ASF+SDF Meta-Environment. That is, a new parse-tree format had been designed, but the transition to it was found to be difficult because of the hard-coded structural knowledge of the old formats. By the transition to higher-level APIs some amount of static typing was added, and such static typing makes evolution more self-checking. While static typing has become a routine weapon in the implementation of normal grammarware, the API-fication showcase is more involved because it deals with generic language technology. (For normal grammarware, it is at least in principle clear how to employ grammars such as abstract syntaxes or XML schemas as 'types' in programs.) The maintainability / evolvability promise is further strengthened when we complement static typing by another static guarantee, namely the claim that all scattered incarnations of a given grammar agree to each other. This requires to trace all links between all grammars in all grammarware. Examples of such links include the consistency of an XML schema with respect to an underlying syntax definition [98], or the correctness of a tolerant grammar with respect to a base-line grammar [6, 63].

Robustness of grammarware Static typing and the preservation of links between grammar artifacts already contribute to a robustness promise in a vital manner. Let us also review further approved means of gaining confidence in the proper functioning of software, namely *reuse* and *testing*, and let us see how these means carry over to grammarware:

- Reuse is a generally accepted means to take advantage of prior maturation as opposed to the risks of developing functionality or data structures from scratch. Reuse is hardly exercised for grammars because existing means of reuse are too weak. They basically support reuse 'as is' via simple forms of modular composition. Grammarware tends to be too monolithic, too technology-dependent, too application-specific for reuse. Improved reusability shall be an obvious contribution of an engineering discipline for grammarware.

- Testing is a generally accepted means to validate functionality. Testing is a particularly appealing method in a setting of data processing as opposed to an abstract behavioural view. Empirical results (e.g., [86, 104, 80]) demonstrate that grammar-based testing makes grammarware more robust. The proposed engineering discipline for grammarware shall contribute a comprehensive testing approach including testing during development, automated test-set generation, and testing for intermediate formats or APIs.

6 Engineering of grammarware —ingredients

We contend that an engineering discipline for grammarware is to be based on the following principles:

- *Generality*. We have enumerated the many different forms of grammars, notations for grammars, and application domains for grammarware. A unified engineering discipline for grammarware shall identify commonalities while categorising all common variations.

- *Abstraction*. Grammar notations must not force us to commit ourselves too early to specific technology and implementational choices. That is, we need technology- and application-independent grammar specifications, say *pure* grammars. Such a distinction of design vs. implementation improves reusability of grammars because it encourages the derivation of actual grammarware solutions from pure grammars.

- *Customisation*. The derivation of grammarware from pure grammars involves customisation steps, which express commitment to specific technology and/or a specific grammar use. For example, a grammar can be customised to serve as a parser specification. Then, customisation will opt for specific parsing technology, e.g., in the way how disambiguation is achieved. (So we could opt for decorated tokens [81], extra actions for semantics-directed parsing [95, 18], or filters that are applied to parse forests [62, 13].)

- *Separation of concerns*. A piece of grammarware normally deals with several concerns. Advanced means of modularisation are needed to effectively separate these concerns [28]. To give an example, let us consider the scenario of "rendering", which encompasses document processing for formatting stylesheets as well as pretty-printing program text. Here, one concern is to associate all the possible grammar patterns with rules for rendering them. Another concern could be the preservation of preexisting formatting information wherever available [53]. The challenge is that these concerns actually interact with each other.

- *Evolution*. The evolution of formats, APIs, interaction protocols, and others must be effectively supported by a unified methodology. This is also challenged by the fact that different incarnations of the same grammar could either need to evolve jointly or independently of each other.

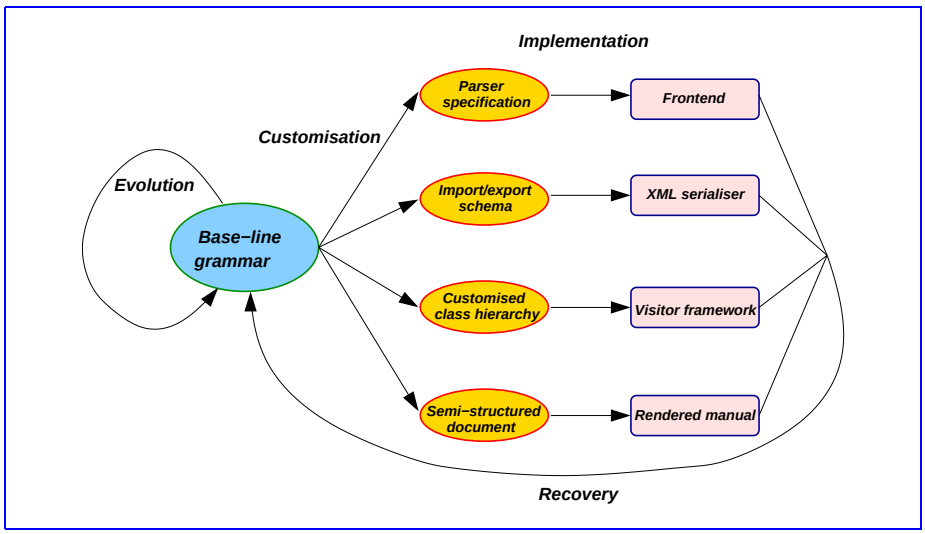

Figure 4: The grammarware life cycle.

- *Assessment*. Claims about the quality of grammars in the sense of metrics, or correctness and completeness must be feasible. In addition, *grammar-based* methods are needed to enable claims about the quality of grammar-dependent software, e.g., grammar-based testing.

- *Automation*. This principle is meant to help with achieving traceability and scalability of the engineering processes for customisation, separation of concerns, evolution, and assessment. For example, customisation and evolution suggest using transformation technology as opposed to manual adaptation. Assessment can be supported by automated metrics calculation and test-set generation.

We assume that establishing these principles for grammarware is more tractable than for arbitrary software. This is because actual grammars are specifications with a relatively simple syntax and semantics. Also, the traces of grammars in grammar-dependent software artifacts can normally be identified in a systematic manner.

Base-line grammars and derived artifacts On the basis of the above principles, we can now provide a *life cycle* for grammarware; see Fig. 4. The life cycle is centred around the notion of a base-line grammar, which is meant to serve as a reference specification for the derivation of grammarware by means of customisation and implementation steps. Evolution should also be performed on the base-line grammar rather than on customised or implemented grammars. Some evolution tasks might however concern the customisation or implementation itself. We briefly discuss one of the life-cycle scenarios from Fig. 4: going from a base-line grammar to a *visitor framework* via a *customised class hierarchy*. Here the base-line grammar would be preferably in the form of a class dictionary, but such a class dictionary could also be obtained from other

grammar forms. Customisation is concerned here with the derivation of an object-oriented class hierarchy for the chosen object-oriented language. Extra customisation effort could be needed to serve a certain application scenario, e.g., the representation of annotations such as use/def relations. The resulting class hierarchy can now be 'implemented', which means for the chosen scenario that a visitor framework for traversing object structures is derived. This can be served by generative programming [32] as pursued in [110].

Recovery of base-line grammars We cannot simply assume that a suitable base-line grammar is always readily available. However, it is fair to assume that there is some grammar knowledge available, which can serve as a starting point for building a base-line grammar. The grammar knowledge might reside in data, e.g., one can attempt to recover an XML schema from given XML documents. The knowledge might also reside in some piece of grammarware, e.g., in a hand-crafted recursive-descent parser or in a reference manual. Hence, the life cycle can be enabled by recovering the base-line grammar first. For an illustration of the recovery phase in the life cycle, we refer back to the recovery showcase from Sec. 5. Recovery is an issue for grammars in the widest sense, not just for syntax definitions of mainstream programming languages. IT companies normally use DSLs, inhouse code generators, import/export functionality, preprocessors, and others, which all involve some grammars. It is a common scenario in software maintenance that such grammars need to be recovered — be it to replace proprietary languages, or to develop new grammar-based tools, or just for documentation purposes. Here are two examples. A kind of recovery project for the proprietary language PLEX used at Ericsson is described in [101, 16]. The project delivered a documentation of PLEX and a new parser for PLEX. In [54], life-cycle enabling was performed for the proprietary SDL dialect used at Lucent Technologies. The recovered grammar was used then to develop a number of SDL tools using generic language technology.

7 The role of automated transformations

The treatment of grammarware as an engineering artifact crucially relies on automated transformations to make all steps in the grammarware life cycle more traceable and scalable than in the present-day approach, which is ad-hoc and manual. Several kinds of transformations can be distinguished:

- *Grammar transformations* to refactor or to revise the described format, syntax, API, and so on in the course of grammar recovery, evolution, and customisation.
- *Program transformations* for grammar-dependent software, which must co-evolve with the underlying grammar structure.
- *Data transformations* to adapt data structures so that they are in compliance with the adapted format (i.e., grammar) assumed by the co-evolved programs.
- *Customising transformations* to derive grammarware that deals with specific grammar uses and specific technology.

The customising transformations involve elements of aspect-oriented programming [59]. For example, the derivation of a proper parser (say, frontend) from a plain grammar can be seen as a weaving process to add several aspects to the basic grammar structure, e.g.:

- Parse-tree construction.
- Parse-tree annotation, e.g., for line and position information.
- Attribute computation, e.g., for types or use/def relations.
- Symbol-table management, maybe also used for semantics-directed parsing.
- Technology-biased tweaking, e.g., for disambiguation.

One can also view some, if not all, customising transformations as ordinary grammar transformations defined on a designated grammar notation. The implementation of grammars according to the grammarware life cycle can often be supported by generative programming [32].

Grammar transformations From the list above it becomes obvious that the transformation of grammars is a key concept. (It should be noted however that grammar transformations are far from constituting a common practice!) To give a typical example, we consider a transformation for the completion of the Cobol grammar as it occurred during the recovery project in [75]. To this end, we quote an informal rule from IBM's VS Cobol II reference [46]:

> A series of imperative statements can be specified
> whenever an imperative statement is allowed.

To implement this rule, we can apply a transformation operator **generalise** as follows:

```
generalise imperative-statement
to imperative-statement+
```

The grammarware engineer writes this grammar transformation down in a transformation script, or (s)he operates in a designated console for transformations, but interactive support is also conceivable. The semantics of the operator **generalise** are such that it replaces the EBNF phrase `imperative-statement` by the phrase `imperative-statement+` as suggested by the informal rule. We call this a generalisation because the new phrase is more general than the original phrase in the formal sense of the generated language.

Categorisation of grammar transformations In Fig. 5, we compile an open-ended list of different kinds of grammar transformations. We refer to [71, 77, 24] for first results on the definition, the implementation and the deployment of some of these categories in the specific context of correcting, completing, or customising definitions of concrete syntax. In all phases of the grammarware life cycle, *grammar refactoring* makes sense. That is, the grammar is restructured without affecting its generated language, which is formally defined for any kind of grammar notation. Here is an example of refactoring a Cobol grammar:

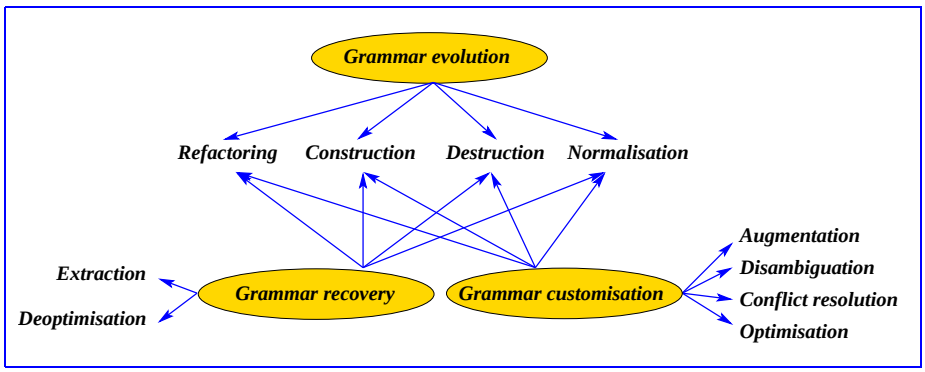

Figure 5: Grammar transformations as a paradigm.

```
extract "ON"? "SIZE" "ERROR" imperative-statement+
as on-size-error
from add-statement-format-i
```

That is, we extract some part of a production for `add-statement-format-i` to constitute a new nonterminal `on-size-error`. Going beyond pure refactoring is frequently necessary. For example, grammar evolution tends to include new productions, exclude obsolete productions, and revise existing productions. This is called *construction* and *destruction* in the figure. Another mode of evolution is *normalisation*, e.g., the systematic elimination of regular operators by using recursive definitions instead, or vice versa. Normalisation preserves the generated language as refactoring does, but it is an exhaustive operation as opposed to a point-wise operation. All kinds of transformations for evolution are immediately useful during grammar recovery and customisation as well. The figure also lists several kinds of transformations that specifically deal with recovery and customisation. These transformations either fix plain grammar structure as in the case of evolution, or they operate on an enriched grammar notation. Disambiguation is a good example for both options. Some amount of disambiguation can be performed by means of plain grammar refactoring. Some issues could remain, which require technology-biased customisation, e.g., adding actions for semantics-directed parsing [95, 18].

Multi-level transformations We switch to the XML setting to briefly illustrate how the primary grammar transformations tend to imply transformations at other levels. In Fig. 6, the middle layer is about XML schema transformation. The top and the bottom layers complete this schema transformation to also be meaningful for dependent document-processing functionality as well as corresponding XML streams. The arrows "↑" and "↓" are meant to indicate that the transformations at the top and the bottom layers are (to some extent) implied by the schema transformation at the middle layer. The arrow "↓" is discussed in some detail in [73]. At IBM Research, related work is underway [47]. The overall problem is similar to database schema evolution coupled with instance mapping as studied by Hainaut and others [40, 42]. The arrow

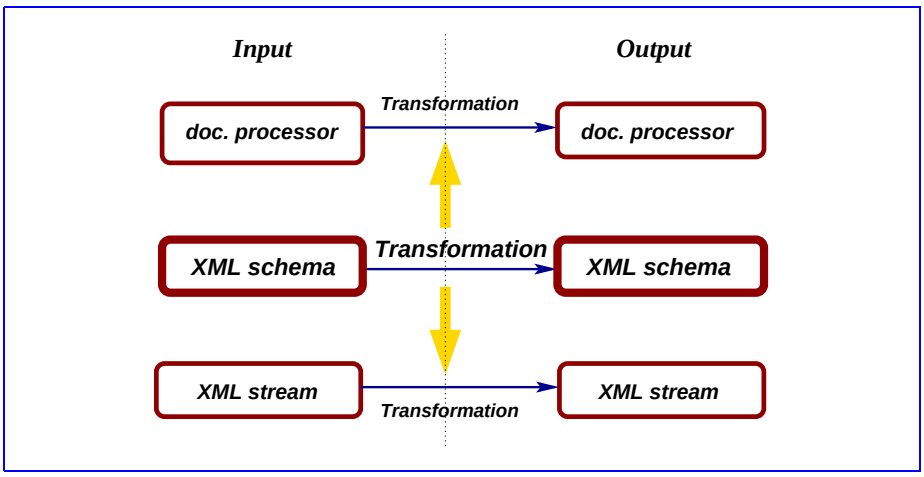

Figure 6: Multi-level transformations in the XML setting with a primary schema transformation and co-evolving document processors as well as migrated XML streams.

"↑" has not been studied in the narrow XML or grammar context. However, the overall problem is similar to Opdyke's refactoring of class hierarchies in object-oriented programming [92]. (Recall class dictionaries are essentially grammars.)

8 An emerging discipline

The essence, the omnipresence, the promises and the ingredients of a comprehensive engineering discipline for grammarware have not been identified before, but relevant ideas and concepts are found in previous work, which is already clear from the many pointers given throughout the paper. The following related work discussion substantiates that we have identified an emerging discipline. All the referenced techniques are versatile, but we contend that a proper research effort is needed to study foundations in a systematic manner, and to deliver well-founded and well-engineered practices.

Grammar transformations They play an established role in implementing parsers, e.g., for the elimination of chain productions; see any text on compilation [2, 1], or Pepper's unification of LL and LR parsing [97]. By contrast, the proposed engineering discipline relies on grammar transformations in the sense of a programming tool. A grammarware engineer programs grammar transformations to perform refactoring, disambiguation, and others. Wile's transformations for deriving an abstract from a concrete syntax [113] and Cordy's et al. transformations to enable 'agile parsing' based on problem-specific grammars [24] are instances of this concept. More generally, the paradigm of grammar transformations can be expected to adopt ideas from refactoring [39, 92, 66], schema evolution [40, 42], and transformational programming [96, 100].

Grammar customisation This concept is related to the notion of generic language technology [60, 12, 55, 9], where one generates software components from language definitions, e.g., interpreters and compilers [78], APIs for processing source code [102, 52], or visitor frameworks for traversal [110]. Customisation poses two challenges:

1. The links between different grammar-like artifacts have to preserved.
2. Support for a separation of concerns that jointly occur in grammarware is needed.

An example for (1.) is the Maptool in the system Eli for language implementation, which addresses the link between concrete and abstract syntax [56]. There are many examples for (2.), especially in the context of technology for program transformation [25, 82, 22, 67]. One can distinguish separation of structural concerns, e.g., concrete syntax, comments, layout, preprocessing statements vs. behavioural concerns, e.g., the primary transformation as opposed to change logging or error handling. We contend that a comprehensive, well-engineered approach to customisation has to combine ideas from generative programming [32], aspect-oriented programming [59], generic programming on datatypes [44], and modular attribute grammars [34, 58, 70].

Grammar assessment Some form of grammar assessment has been previously exercised for the purpose of measuring the quality of grammars in the context of reverse-engineering language syntaxes [101, 72]. A uniform approach for analysing grammars and grammar-dependent software could be based on grammar-flow analysis by Wilhelm and others [87, 49]. The validation of grammar-dependent software can be based on manually developed conformance suites [89], or stochastic test-set generation; see [19, 86] for compiler testing, and [85, 104] for other settings. Test set generation necessitates a whole string of techniques to deal with the standard oracle problem, to minimise test cases that act as symptoms, to enforce semantic constraints as opposed to purely structural generation [20, 57], and to accomplish negative test cases. Validation can also be based on formal coverage criteria, e.g., Purdom's rule coverage for context-free grammars, or vital extensions thereof [99, 72].

Standard formats There has been a recent flurry on standard formats in a re- and reverse engineering context [69, 83, 45, 103, 114]. The overall idea of standard formats has already proved to be useful in the context of compilation and static program analysis for long; think of widely adopted formats such as PDG or SSA [43, 35, 23]. Standard formats complement our notion of a base-line grammar since they are normally less language-specific than base-line grammars. Recent work has focused on formats that support interoperability of tools operating on source code models, e.g., the XML-based format GXL [45] as well as layers defined on top of it. A current topic is improved reuse for language-specific schemas (say, base-line grammars) on the basis of their modular composition from library fragments [114]. Another active research topic is the provision of API generation for standard formats, parse trees, and other intermediate representations [111, 55, 102, 52, 76]. This is an improvement over the more basic approach to use generic data representations such as plain ATerms [11] in generic language technology or the Document Object Model [30] (DOM) for accessing XML content.

Parsing This is an active research field, and the open problems are of relevance for the proposed engineering discipline. There is a body of work that demonstrates how paradigm shifts (regarding parsing technology) attack grammarware hacking [95, 18, 15, 8, 13, 81]. It is widely acknowledged that the common development process for industrial parser development is far from being optimal. An improvement of common practices is showcased in [80], where a C# parser is designed while focusing on the adoption of software engineering techniques. A current topic is *tolerant parsing*, which is meant to address the dialect problem and other problems that are in conflict with insistence on precise grammars [5, 65, 88, 63]. Not even the basic battle about the ultimate parsing regime is decided: is it generalised LR-parsing with powerful forms of disambiguation [62, 13]; is it top-down parsing but then with idioms for semantics direction [95, 18]; it is simple LARL(1) parsing with token decoration [81]; is it plain recursive descent parsing but then with provisions for limiting backtracking [18, 68], or what else? This multitude of options supports our separation of pure grammars vs. technology-biased grammar customisation.

9 Research challenges

The effort for arriving at general *and* well-founded *and* automated engineering practices for grammarware should not be underestimated. To give an example, so far there is no somewhat universal operator suite for grammar transformations despite all efforts in the last few years; recall the related work discussion. Developing a universal suite, which will be meaningful to software engineering as a whole, appears to be challenging. Such indications for the complexity of the needed research effort makes us think in terms of a public research agenda as opposed to a short-term project.

The following list entails research issues on foundations, methodology, tool support and empirical matters. The items are listed in no particular order. Each item is self-contained, and could serve as a skeleton of a PhD project.

1. *A framework for grammar transformations*. This effort culminates in a domain-specific language based on appropriate basic operators. The faced challenges are about orthogonality of the basic operators, full coverage of all transformation scenarios (recall refactoring, construction, etc.), simplicity of use, and a suitable theory for formal reasoning. Initial results can be found in [113, 75, 71, 24].

2. *An approach for co-evolution of grammar-dependent software*. The approach should be largely parametric in the language for which co-evolution is supported. A prototypical example is the evolution of an XSLT program in the view of changes of the underlying XML schema. Another example is the co-evolution of an aspect for parser tweaking or parse-tree construction in the view of changes of the underlying concrete syntax.

3. *A comprehensive theory of grammarware testing*. This includes coverage criteria for both grammars and grammar-dependent software as well as means for test-case characterisation. A non-trivial issue is here to go beyond structural properties. Also, making coverage analysis and test-set generation both versatile and scalable is challenging. Initial results can be found in [85, 19, 86, 104, 72].

4. *A collection of formal grammarware properties*. This includes notions of correctness and completeness for grammars, e.g., relative to a testsuite, or a testing oracle. This further includes metrics and slicing criteria for grammar-dependent software. Ultimately, this includes distance metrics and preservation properties for grammar transformations. The development of a uniform framework for properties of grammars, grammar-dependent software and transformations can be based on grammar-flow analysis [87, 49].

5. *Aspect-oriented grammarware development*. A prototypical example is parser development, which starts from a pure grammar to be elaborated by aspects for pre-processing, error recovery, parse-tree construction, attribute computations, technology-biased disambiguation, annotation of parse trees with position information or comments, and others. The challenge is indeed to separate these aspects as to constitute proper modules, and then to derive the complete artifact by advanced modular composition. Initial results can be found in [34, 58, 70, 82, 107, 22, 67].

6. *A model for debugging grammarware*. This includes static analyses, e.g., an analysis to give indications of sources of ambiguity on the basis of an LR(k) conflict analysis for smaller ks. This also includes debugging of grammar-dependent software while paying attention to the involved grammars. For example, structural patterns such as nests of some construct could be integrated into the debugging model. The generic debugging model in [91] could be useful in this context.

7. *A detailed life cycle for grammarware*. Processes for typical life-cycle scenarios of recovery, evolution, and customisation need to be defined in detail, e.g., processes for the alternation of technology vs. the evolution of grammarware functionality. This development differentiates the kinds of grammarware, e.g., document processors vs. APIs vs. parsers. The defined processes highlight the potential for automation and quality assessment in the various phases.

8. *Tolerant grammarware*. Entirely precise grammars are often not the preferred option from an engineering point of view. Firstly, this is the case when semi-structured data or input with loose structure has to be handled, e.g., when user input is processed real-time in an IDE editing session. Secondly, problem-specific grammars can also be more efficient in terms of effort and costs whenever the problem at hand does not inherently require an entirely precise grammar, e.g., analysis or transformation problems that are only concerned with few language constructs. However, the consideration of tolerant grammarware triggers additional correctness problems as discussed for language parsers in [63]. A general methodology for tolerant grammarware has to be delivered.

9. *Grammar inference*. In several settings, suitable base-line grammars are not readily available, e.g., in software re-engineering. There is previous work on grammar recovery on the basis of grammar knowledge such as a semi-formal language reference [75, 74]. By contrast, little is known about grammar inference from sample data. In fact, inference of XML schemas from XML streams is relatively simple because the markup text mentions element and attribute names. However, other forms of grammars are not easily inferred. The inference of interaction protocols requires non-trivial program analyses and heuristics. The inference of concrete syntax definitions requires learning a language in an AI sense as it has been studied for natural languages [94].

10. *Conception of an IDE plug-in supporting the life cycle of grammarware*. An interactive development environment (IDE) for a given language would be extended to facilitate the development of grammarware. The underlying technology needs to be very flexible and open, e.g., the combination Eclipse (`http://www.eclipse.org/`) and the ASF+SDF Meta-Environment [60, 9] appears to be suitable. The plug-in would integrate tooling for interactive and batch-mode grammar transformations, co-evolution of grammar-dependent programs, test-set generation, coverage visualisation, calculation of grammar metrics, indication of bad smells, customisation of grammars, and others.

11. *CASE meets CAGE*. The automation aspect of the engineering discipline can be termed as CAGE — computer-aided grammarware engineering. This raises the issue of how to integrate CAGE with established CASE — computer-aided software engineering. This would clarify the role of the new engineering discipline for the analysis and the design of complex, long-living software systems as opposed to studying grammarware solely at the code level. For example, one should investigate the necessary refinement of CASE approaches such as Rational's Unified Process and corresponding CASE tools.

12. *Grammar-aware asset management*. In many software projects and organisations the knowledge about software assets is largely missing and a form of knowledge management helps to increase the understanding of the software portfolio [61]. Given the special nature of grammarware, specialised knowledge management techniques may exist for grammarware. Also, the overall approach for understanding and assessing software assets may need adjustment once grammar-awareness is increased.

13. *Reconcilable application generation*. Implementations of domain-specific languages [26] (DSLs) fall completely within the scope of grammarware. An application generator for the implementation of a DSL depends on grammars for the high-level, domain-specific input notation as well as the target language(s). Co-evolution is very meaningful in the DSL context. That is, modifications to the high-level notation imply that the DSL implementation needs to be revised, but also that existing DSL programs have to be upgraded. A particularly hard problem is that generated code could have been customised by the programmer. Hence, a scheme is needed to reconcile the newly generated code with the previously customised code.

14. *Empirical research on grammarware*. What are measurable losses implied by grammarware hacking? What are success stories, and what are the key factors for success? What are further insights in the grammarware dilemma, and how does this compare to other dilemmas in software engineering? What is the mid- and long-term perspective for the distribution of different kinds of grammarware? Will it be useful to broaden the scope of the engineering discipline for grammarware, e.g., by providing a uniform paradigm for grammarware, databases, the semantic web [29], and model-driven architecture?

10 Concluding remarks

We have argued that current software engineering is insufficiently aware of grammars, which is manifested by an ad-hoc treatment of grammarware. We layed out an agenda that is meant to contribute to a trend towards research on grammarware from an engineering point of view. We justified the envisaged research effort on the basis of the omnipresence of grammarware in software systems and development processes. We provided a substantial list of challenges, which can be viewed as skeletons for PhD projects. Such challenges need to be addressed in order to make progress with the emerging discipline for engineering of grammarware.

Research is needed that focuses on modularity, robustness, and evolvability of grammarware and also on the automation of grammarware development. To this end, we team up with current trends in software engineering. Firstly, grammarware is a form of aspect-oriented software, where grammars can be seen as the all-dominant concern. Secondly, generative programming is the essential concept to customise grammars. Thirdly, automated transformations are employed to manage evolution of grammars and co-evolution of grammar-dependent software. Finally, there are several provisions to measure quality. Putting all this together, the ultimate outcome of an engineering discipline for grammarware is a more automated, more predictable, and more agile information technology.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] A. Aho and J. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Englewood Cliffs (NJ), 1972–73. Vol. I. Parsing. Vol II. Compiling.

[3] U. Aßmann and A. Ludwig. Aspect weaving by graph rewriting. In U. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, volume 1799 of *LNCS*, pages 24–36. Springer-Verlag, Oct. 1999.

[4] J. 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.

[5] D. Barnard. *Hierarchic Syntax Error Repair*. PhD thesis, University of Toronto, Department of Computer Science, Mar. 1981. 253 pages.

[6] D. Barnard and R. Holt. Hierarchic Syntax Error Repair for LR Grammars. *International Journal of Computer and Information Sciences*, 11(4):231–258, 1982.

[7] J. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specifi cation*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[8] D. Blasband. Parsing in a hostile world. In *Proc. Working Conference on Reverse Engineering (WCRE'01)*, pages 291–300. IEEE Press, Oct. 2001.

[9] M. v. d. Brand, A. v. Deursen, J. Heering, H. d. Jong, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Proc. Compiler Construction (CC'01)*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[10] M. v. d. Brand, A. v. Deursen, P. Klint, A. S. Klusener, and E. v. d. Meulen. Industrial applications of ASF+SDF. In *41*, page 11. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, June 30 1996.

[11] M. v. d. Brand, H. d. Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software – Practice and Experience*, 30(3):259–291, Mar. 2000.

[12] M. v. d. Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM Sigplan Notices*, 32(2):54–61, 1997.

[13] M. v. d. Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In N. Horspool, editor, *Proc. Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.

[14] M. v. d. Brand, M. Sellink, and C. Verhoef. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In M. Sellink, editor, *Proc. 2nd International Workshop on the Theory and Practice of Algebraic Specifi cations*, electronic Workshops in Computing. Springer-Verlag, 1997.

[15] M. v. d. Brand, M. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proc. International Workshop on Program Comprehension (IWPC'98)*, pages 108–117, 1998.

[16] M. v. d. Brand, M. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[17] M. v. d. Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, Jan. 1996.

[18] P. Breuer and J. Bowen. A PREttier Compiler-Compiler: Generating Higher-order Parsers in C. *Software — Practice and Experience*, 25(11):1263—1297, Nov. 1995.

[19] C. Burgess. The Automated Generation of Test Cases for Compilers. *Software Testing, Verifi cation and Reliability*, 4(2):81–99, jun 1994.

[20] A. Celentano, S. Crespi-Reghizzi, P. D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software – Practice and Experience*, 10(11):897–918, Nov. 1980.

[21] N. Chomsky. *Reflections on Language*. Pantheon, 1975. 269 pages.

[22] J. Cordy. Generalized Selective XML Markup of Source Code Using Agile Parsing. In *Proc. International Workshop on Program Comprehension (IWPC'03)*, pages 144–153. IEEE Press, May 2003.

[23] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[24] T. Dean, J. Cordy, A. Malton, and K. Schneider. Grammar Programming in TXL. In *Proc. Source Code Analysis and Manipulation (SCAM'02)*. IEEE Press, Oct. 2002.

[25] A. v. Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[26] A. v. Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notice*, 35(6):26–36, June 2000.

[27] A. v. Deursen and T. Kuipers. Building Documentation Generators. In *Proc. International Conference on Software Maintenance (ICSM'99)*, pages 40–49, 1999.

[28] E. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[29] Y. Ding, D. Fensel, M. Klein, and B. Omelayenko. The semantic web: yet another hip? *Data & Knowledge Engineering*, 41(2–3):205–227, June 2002.

[30] W. A. domain. Document Object Model (DOM), 1997–2003. `http://www.w3.org/DOM/`.

[31] O. Dubuisson. *ASN.1 – Communication between heterogeneous systems*. Morgan Kaufmann Publishers, Oct. 2000. 588 p.; Translated from French by Philippe Fouquart.

[32] U. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[33] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG - A generator for efficient back ends. In B. Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (SIGPLAN '89)*, pages 227–237, Portland, OR, USA, June 1989. ACM Press.

[34] R. Farrow, T. Marlowe, and D. Yellin. Composable Attribute Grammars. In *Proc. of 19th ACM Symposium on Principles of Programming Languages (Albuquerque, NM)*, pages 223–234, Jan. 1992.

[35] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[36] I. O. for Standardization. ISO/IEC 14977:1996(E), Information technology —Syntactic metalanguage —Extended BNF, 1996.

[37] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[39] W. Griswold and D. Notkin. Program restructuring as an aid to software maintenance. Technical report, Seattle, WA, USA, August 1990.

[40] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Schema Transformation Techniques for Database Reverse Engineering. In *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, 1993. E/R Institute.

[41] J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[42] J. Henrad, J.-M. Hick, P. Thiran, and J.-L. Hainaut. Strategies for Data Reengineering. In *Proc. Working Conference on Reverse Engineering (WCRE'02)*, pages 211–220. IEEE Press, Nov. 2002.

[43] V. P. Heuring, U. Kastens, R. G. Plummer, and W. M. Waite. COMAR: a data format for integration of CFG tools. *The Computer Journal*, 32(5):445–452, Oct. 1989.

[44] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In E. Boiten and B. Möller, editors, *Proc. Mathematics of Program Construction (MPC'02)*, 2002.

[45] R. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. In *Proc. Working Conference on Reverse Engineering (WCRE'00)*, Nov. 2000.

[46] IBM Corporation. *VS COBOL II Application Programming Language Reference*, 4. Publication number GC26-4047-07 edition, 1993.

[47] IBM Research. XML Transformation: Matching & Reconciliation, 2002. available at `http://www.research.ibm.com/hyperspace/mr/`.

[48] R. Jain. Extracting Grammar from Programs. Master's thesis, Indian Institute of Technology, Kanpur, 2000.

[49] J. Jeuring and D. Swierstra. Bottom-up grammar analysis—A functional formulation. In D. Sannella, editor, *Proc. European Symposium on Programming Languages and Systems (ESOP'94*, volume 788 of *LNCS*, pages 317–332. Springer-Verlag, 1994.

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

[51] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[52] H. d. Jong and P. Olivier. Generation of abstract programming interfaces from syntax definitions. Technical Report SEN-R0212, St. Centrum voor Wiskunde en Informatica (CWI), Aug. 2002. To appear in Journal of Logic and Algebraic Programming.

[53] M. d. Jonge. Pretty-printing for software reengineering. In *Proc. International Conference on Software Maintenance (ICSM'02)*, pages 550–559. IEEE Computer Society Press, Oct. 2002.

[54] M. d. Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proc. International Conference on Software Maintenance (ICSM'01)*, pages 240–249. IEEE Press, Nov. 2001.

[55] M. d. Jonge and J. Visser. Grammars as Contracts. In *Proc. Generative and Component-based Software Engineeringi (GCSE'00)*, volume 2177 of *LNCS*, pages 85–99, Erfurt, Germany, Oct. 2000. Springer-Verlag.

[56] B. Kadhim and W. Waite. Maptool—supporting modular syntax development. In T. Gyimothy, editor, *Proc. Compiler Construction (CC'96)*, volume 1060 of *LNCS*, pages 268–280. Springer, Apr. 1996.

[57] U. Kastens. Studie zur Erzeugung von Testprogrammen für Übersetzer. Bericht 12/80, Institut für Informatik II, University Karlsruhe, 1980.

[58] U. Kastens and W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica 31*, pages 601–627, 1994.

[59] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 9–13 June 1997.

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

[61] P. Klint and C. Verhoef. Enabling the creation of knowledge about software assets. *Data & Knowledge Engineering*, 41(2–3):141–158, June 2002.

[62] P. Klint and E. Visser. Using filters for the disambiguation of contextfree grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, pages 1–20, 1994. Tech. Rep. 126-1994 Dipartimento di Scienze dell'Informazione, Universita di Milano.

[63] S. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar. In *Proc. International Conference on Software Maintenance (ICSM'03)*. IEEE Press, 2003. 10 pages, to appear.

[64] D. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2:127–145, 1968. Corrections in 5:95-96, 1971.

[65] R. Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27(6):637–649, 1997.

[66] J. Kort and R. Lämmel. A Framework for Datatype Transformation. In B. Bryant and J. Saraiva, editors, *Proc. Language, Descriptions, Tools, and Applications (LDTA'03)*, volume 82 of *ENTCS*. Elsevier, Apr. 2003. 20 pages.

[67] J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proc. Source Code Analysis and Manipulation (SCAM'03)*, Amsterdam, Sept. 2003. IEEE Press. To appear.

[68] J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M. v. d. Brand and R. Lämmel, editors, *Proc. Language Descriptions, Tools, and Applications (LDTA'02)*, volume 65 of *ENTCS*. Elsevier Science, Apr. 2002. 7 pages.

[69] R. Koschke and J.-F. Girard. An intermediate representation for reverse engineering analyses. In *Proc. Working Conference on Reverse Engineering (WCRE'98)*, pages 241–250. IEEE Press, Oct. 1998.

[70] R. Lämmel. Declarative Aspect-Oriented Programming. In O. Danvy, editor, *Proc. of 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, Jan. 1999.

[71] R. Lämmel. Grammar Adaptation. In J. Oliveira and P. Zave, editors, *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[72] R. Lämmel. Grammar Testing. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[73] R. Lämmel and W. Lohmann. Format Evolution. In J. Kouloumdjian, H. Mayr, and A. Erkollar, editors, *Proc. Re-Technologies for Information Systems (RETIS'01)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

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

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

[76] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, Jan. 2003.

[77] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M. v. d. Brand and D. Parigot, editors, *Proc. Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001.

[78] P. Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, 1989.

[79] J. Lind. Specifying Agent Interaction Protocols with Standard UML. In M. Wooldridge and G. Weiß and P. Ciancarini, editor, *Proc. Agent-Oriented Software Engineering (AOSE'01)*, volume 2222 of *LNCS*, pages 136–145. Springer-Verlag, 2002.

[80] B. Malloy, J. Power, and J. Waldron. Applying software engineering techniques to parser design: the development of a C# parser. In *Proceedings of the 2002 conference of the South African institute of computer scientists and information technologists*, pages 75–82, 2002. In cooperation with ACM Press.

[81] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software—Practice & Experience*, 33(1):1395–1438, 2003.

[82] A. Malton, K. Schneider, J. Cordy, T. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *Proc. International Workshop on Program Comprehension (IWPC'01)*. IEEE Press, May 2001.

[83] E. Mamas and K. Kontogiannis. Towards portable source code representations using XML. In *Proc. Working Conference on Reverse Engineering (WCRE'00)*, pages 172–182. IEEE Press, Nov. 2000.

[84] S. Marlow. Haddock: A haskell documentation tool, 2002. `http://haskell.cs.yale.edu/haddock/`.

[85] P. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–56, 1990.

[86] W. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[87] U. Mönck and R. Wilhelm. Grammar Flow Analysis. In H. Alblas and B. Melichar, editors, *Proc. International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 151–186. Springer-Verlag, June 1991.

[88] L. Moonen. Generating Robust Parsers using Island Grammars. In *Proc. Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE Press, Oct. 2001.

[89] NIST, Information Technology Laboratory, Software Diagnostics and Conformance Testing Division, Standards and Conformance Testing Group. Conformance test suite software, 2003. `http://www.itl.nist.gov/div897/ctg/software.htm`.

[90] J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing Agent Interaction Protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Proc. Agent-Oriented Software Engineering (AOSE'00)*, pages 121–140. Springer-Verlag, 2001.

[91] P. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, Universiteit van Amsterdam, Dec. 2000. 182 pages.

[92] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[93] J. Paakki. Attribute Grammar Paradigms —A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[94] R. Parekh and V. Honavar. Grammar Inference, Automata Induction, and Language Acquisition. In R. Dale, H. Moisl, and H. Somers, editors, *A Handbook of Natural Language Processing: Techniques and Applications for the Processing of Language as Text*. Marcel Dekker Press, 2000.

[95] T. Parr and R. Quong. Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k). In P. Fritzon, editor, *Proc. Compiler Construction (CC'94)*, volume 786 of *LNCS*, pages 263–277. Springer-Verlag, Apr. 1994.

[96] H. Partsch. *Specifi cation and Transformation of Programs*. Springer-Verlag, 1990.

[97] P. Pepper. LR Parsing = Grammar Transformation + LL Parsing. Technical Report CS-99-05, TU Berlin, Apr. 1999.

[98] J. Power and B. Malloy. Program annotation in XML: a parse-tree based approach. In *Proc. Working Conference on Reverse Engineering (WCRE'02)*, pages 190–198. IEEE Press, Nov. 2002.

[99] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.

[100] Roever, W.-P. de and K. Engelhardt. *Data Refi nement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[101] M. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In *Proc. Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 151–160. IEEE Press, March 2000.

[102] S. Sim. Next generation data interchange: Tool-to-tool application program interfaces. In *Proc. Working Conference on Reverse Engineering (WCRE'00)*, pages 278–283. IEEE Press, Nov. 2000.

[103] S. Sim and R. Koschke. WoSEF: Workshop on Standard Exchange Format. *ACM SIGSOFT Software Engineering Notes*, 26:44–49, Jan. 2001.

[104] E. Sirer and B. Bershad. Using Production Grammars in Software Testing. In USENIX, editor, *Proc. Domain-Specifi c Languages (DSL'99)*, pages 1–13. USENIX, 1999.

[105] D. R. Smith. KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[106] Sun. Java 2 Platform, Standard Edition (J2SE). Javadoc tool home page, 2002. `http://java.sun.com/j2se/javadoc/`.

[107] S. Swierstra. Parser Combinators, from Toys to Tools. In G. Hutton, editor, *Proc. 2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *ENTCS*. Elsevier Science, Aug. 2001.

[108] S. Thibault and C. Consel. A Framework for Application Generator Design. *ACM SIGSOFT Software Engineering Notes*, 22(3):131–135, May 1997.

[109] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sept. 1997.

[110] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, Nov. 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

[111] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices*, 34(9):148–159, Sept. 1999. Proc. of ICFP'99.

[112] D. Wang, A. Appel, J. Korn, and C. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 213–228, Berkeley, Oct. 15–17 1997. USENIX Association.

[113] D. Wile. Abstract syntax from concrete syntax. In *Proc. International Conference on Software Engineering (ICSE'97)*, pages 472–480. ACM Press, 1997.

[114] A. Winter. Referenzschemata im Reverse Engineering. *Softwaretechnik Trends der GI*, 23(2), 2003. 2 pages.