Toward an engineering discipline for GRAMMARWARE

Article submitted to TOSEM	August 17, 2003
Reviews received from TOSEM	November 19, 2003
1st revision submitted to TOSEM	July 12, 2004
Reviews received from TOSEM	January 19, 2005
2nd revision submitted to TOSEM	February 22, 2005
2nd revision accepted	April 7, 2005
Files delivered to TOSEM	May 30, 2005

Paul Klint
Centrum voor Wiskunde en Informatica, Amsterdam
Universiteit van Amsterdam
and
Ralf Lämmel
Vrije Universiteit Amsterdam
Centrum voor Wiskunde en Informatica, Amsterdam
and
Chris Verhoef
Vrije Universiteit Amsterdam

Grammarware comprises grammars and all grammar-dependent software. The term grammar is meant here in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, XML schemas as well as some forms of tree and graph grammars. The term grammar-dependent software refers to all software that involves grammar knowledge in an essential manner. Archetypal examples of grammar-dependent software are parsers, program converters, and XML document processors. Despite the pervasive role of grammars in software systems, the engineering aspects of grammarware are insufficiently understood. We lay out an agenda that is meant to promote research on increasing the productivity of grammarware development and on improving the quality of grammarware. To this end, we identify the problems with the current grammarware practises, the barriers that currently hamper research, the promises of an engineering discipline for grammarware, its principles, and the research challenges that have to be addressed.

Categories and Subject Descriptors: D.2.13 [**Software Engineering**]: Reusable Software; D.2.12 [**Software Engineering**]: Interoperability; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems

General Terms: Languages

Additional Key Words and Phrases: grammarware, grammars, grammar-dependent software, automated software engineering, best practises, parsers, software transformation, language processing, generic language technology, model-driven development, meta-modelling, software evolution

Contents

1. INTRODUCTION

1.1 An analogy: linguistics vs. information technology

In linguistics, one must confront and manage a multitude of human languages. The overall attack to deal with such diversity and complexity is to try understanding "the system of principles, conditions, and rules that are elements or properties of all human languages ... *the essence of human language*" [Chomsky 1975]. (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 multitude of programming languages, data representations, protocols, and other entities that are regulated by some sort of grammar. Here, the overall attack must be to understand the principles, conditions, and rules that underly all use cases for grammars. Grammars cannot be reduced to a few formal aspects such as the Chomsky hierarchy and parsing algorithms. We rather need a kind of software engineering that is *grammar-aware* by paying full attention to the engineering aspects of grammars and grammar-dependent software.

1.2 The definition of the term grammarware

We coin the term 'grammarware' to comprise grammars and grammar-dependent software.

—The term *grammar* is used in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, XML schemas as well as some forms of tree and graph grammars. Grammars are used for numerous purposes, e.g., for the definition of concrete or abstract programming language syntax, and for the definition of exchange formats in component-based software applications.

—The term *grammar-dependent software* is meant to refer to all software that involves grammar knowledge in an essential manner. Archetypal examples of grammar-dependent software are parsers, program converters, and XML document processors. All such software either *literally involves* or *encodes grammatical structure*: compare generated vs. hand-crafted parsers.

1.3 A research agenda for grammarware engineering

This paper is a call-to-arms for setting the employment of grammars in software systems on a firm engineering foundation. In fact, this paper is a research agenda that promotes an engineering discipline for grammarware. We use the term "grammarware engineering" to denote this discipline.

Grammarware engineering is focused on the following credo:

> *The development and maintenance of grammarware should be such that the involved grammatical structure is subjected to best practises, tool support and rigorous methods that in turn are based on grammar-aware concepts and techniques for design, customisation, implementation, testing, debugging, versioning and transformation.*

The underlying goal is to improve the quality of grammarware, and to increase the productivity of grammarware development. Grammars permeate (or shape) software systems. Hence, we deserve an engineering discipline for grammarware, and we can expect that grammarware engineering is to the advantage of software development in general.

1.4 Scenarios of grammarware development

Let us consider a few diverse scenarios of software development, in which different sorts of grammar knowledge play an essential role. These scenarios pinpoint some issues and problems regarding the development and maintenance of grammarware:

—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 your's; think of web browsers. Such import functionality requires recovery of the relevant format. Import needs to be robust and adaptive so that all conceivable inputs are parsed and all convertible parts are identified.

—As a developer of database applications, you want to adopt a new screen definition language for an information system. An automated solution requires the ability to parse screen definitions according to the old format, to generate screen definitions according to the new format, and to define a mapping from the old to the new format. Here we presume that screen definitions are not ingrained in program code. Otherwise, additional, perhaps more involved parsing, unparsing, and mapping functionality will be required.

—As an object-oriented developer, you want to improve static typing for XML processing. That is, you want to replace DOM-based XML access by an XML binding. An automated solution requires the ability to locate DOM usage patterns in the code, and to replace them according to the XML binding semantics. We face grammar knowledge of at least two kinds: the syntax of the programming language in which XML access is encoded, and the schema for the accessed XML data.

—As a tool provider for software re-/reverse engineering, you are maintaining a Java code smell detector and a metrics analyser. You have started this effort in 1996 for Java 1.0, while you are currently working on an upgrade for Java 1.5. To support more sophisticated smells and metrics, you add intelligence that recognises and handles various APIs and middleware platforms used in Java applications, e.g., Swing, WebSphere and JBoss. This intelligence boils down to diverse grammar knowledge.

—As a developer of an in-house application generator, you face a redesign of the domain-specific language (DSL) that is used to provide input to the generator. You fail to provide backward compatibility, but you are requested to offer a conversion tool for existing DSL programs. Furthermore, you are required to handle the problem of generator output that was manually customised by the programmers. Hence, you might need to locate and reuse customisation code as it is ingrained in the generated code.

—As a developer of an international standard or vendor-specific reference for a programming language, you would like to guarantee that the language reference contains the complete and correct grammar of the described language and that the shown sample programs are in accordance with the described syntax (modulo elisions). One challenge is here that you need a readable syntax description in the standard or reference as well as an executable syntax definition for validation.

—As an online service provider, you want to meet your clients' request 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, while you want to preserve the conceptual protocol as is. You end up with re-engineering your application such that the alternation of the protocol technology will be easier in the future.

1.5 Typical engineering aspects of grammarware

The aforementioned scenarios involve various engineering aspects regarding grammars:

—What is a “good grammar” in the first place — in terms of style or metrics?
—How to recover the relevant grammars in case they are not readily available?
—How to choose among options for implementing grammar-dependent functionality?
—How to systematically transform grammatical structure when faced with evolution?
—How to maintain links between implemented variations on the same grammar?
—How to test grammar-dependent functionality in a grammar-aware manner?
—How to verify grammar-related properties of grammar-dependent functionality?

(And so on.) Even though a body of versatile techniques is available, in reality, grammarware is typically treated without adhering to a proper engineering discipline. Grammarware seems to be second-class software. For instance, program refactoring is a well-established practise according to modern object-oriented methodology. By contrast, grammar refactoring is weakly understood and hardly practised.

1.6 A concerted, interdisciplinary research effort

In order to make progress with grammarware engineering, we will need a large scale effort in the software engineering and programming language communities. The present agenda takes an inventory, and it identifies open challenges. The next steps are the following. We need dedicated scientific meetings. PhD students need to pick up the listed challenges. We need to start working on an engineering handbook for grammarware. We also need grammarware-aware curricula at universities.

Grammarware engineering *could* have been a classic field of computer science already for decades. After all, grammars and grammar-dependent software are no recent invention. Grammarware engineering fits well with other fields such as generic language technology, generative programming, software re-/reverse engineering, aspect-oriented software development, program transformation, meta-modelling, and model-driven development. That is, grammarware engineering *employs* these fields and *contributes* to them. In this complex context, the focus of grammarware engineering is clearly defined: the engineering aspects of grammars and grammatical structure in software systems.

1.7 Road-map of the agenda

In Sec. 2, we will compile an *inventory of grammarware*. In Sec. 3, we will analyse the reality of dealing with grammarware, which we will have to summarise as *grammarware hacking*. In Sec. 4, we will uncover the *grammarware dilemma* in an attempt to explain the current, suboptimal situation. This agenda has to cut a Gordian knot in order to prepare the ground for a significant research effort on grammarware engineering. In Sec. 5, we will lay out the *promises* of an engineering discipline for grammarware. In Sec. 6, we will identify essential *principles* of the emerging discipline. Ultimately, in Sec. 7, we will compile a substantial list of *research challenges*, which call for basic and applied research projects. Throughout the paper, we will survey existing contributions to the emerging engineering discipline for grammarware.

A fragment of a BNF grammar that defines the concrete syntax of a simple language.

```
[axiom]   program      ::= declarations statements
[decs]    declarations ::= declaration ";" declarations
[nodec]   declarations ::= ϵ
[dec]     declaration  ::= id ":" type
[concat]  statements   ::= statement ";" statements
[skip]    statements   ::= ϵ
[assign]  statement    ::= id ":=" expression
[var]     expression   ::= id
...
```

A fragment of a DTD for the XML representation of the organisational structure in a company.

```
<!DOCTYPE company [
     <!ELEMENT    company   (dept*)                >
     <!ELEMENT    dept      (name, manager, unit*) >
       <!ATTLIST  dept      dept_num ID #REQUIRED  >
     <!ELEMENT    unit      (employee | dept)      >
     <!ELEMENT    employee  (person, salary)       >
       <!ATTLIST  employee  busunit IDREF #IMPLIED >
     <!ELEMENT    person    (name, address)        > ... ]>
```

Some algebraic data types in Haskell notation for event traces of the execution of C programs.

```
data ExecProg  = ExecProg [Either ExecStmt EvalExpr]
data ExecStmt  = ExecStmt [Either ExecStmt EvalExpr]
data EvalExpr  = EvalCall FuncCall
               | EvalAssign Assign
               | EvalOthers [EvalExpr]
data FuncCall  = Call [EvalExpr] [ExecStmt]
data Assign    = Assign [EvalExpr] Desti
data Desti     = ···
```

Fig. 1. Grammar samples: *The syntax definition at the top is perhaps the most obvious example of a grammar. The XML DTD in the middle defines the abstract representation of a company's organisational structure. It makes use of specific XML features such as attributes and references. The signature at the bottom defines the structure of event traces for the execution of C programs. Here, we are specifically interested in tracing assignments and function calls.*

2. AN INVENTORY OF GRAMMARWARE

We use the term grammar as an alias for *structural descriptions* in software systems, i.e.:

Grammar	=	**structural description in software systems**
	=	**description of structures used in software systems**

Some representative examples of grammars are shown in Fig. 1. Whenever a software component involves grammatical structure, then we attest a *grammar dependency*. (We will also say that the component *commits to grammatical structure*.) In this section, we will first demarcate our use of the term "grammar", i.e., "structural description", and we will then compile an inventory of grammar formalisms, grammar notations, grammar use cases, grammar-based formalisms and notations, and forms of grammar dependencies.

2.1 Structural descriptions

When we say that grammars are structural descriptions, we make a number of informal assumptions as to what it means to be a structural description. Firstly, we assume that a grammar (potentially) deals with *several interrelated categories* as opposed to a single category; cf. the nonterminals in a context-free grammar. Secondly, we assume that there are constructs for the *formation of compound structure*. Thirdly, we assume that there are constructs for the *choice among different alternatives*; cf. multiple productions for a nonterminal in a context-free grammar, or the "|" operator in the BNF formalism.

These assumptions are intentionally lax, as to avoid exclusion of grammar forms that we did not think of or that do not yet exist. However, we can further demarcate the term grammar by excluding some artifacts and by identifying borderline cases:

—A parser specification is *not* a grammar, but it is an *enriched* grammar.

—A type declaration for polymorphic lists is a *trivial* (parameterised) grammar.

—An attribute grammar [Knuth 1968] is not a grammar in our restricted sense, but it definitely *comprises* a grammar, i.e., the context-free grammar whose derivation trees are attributed eventually. It is worth noting that the attribute grammar might comprise yet another grammar — the one for the structures that are synthesised.

—What is the relationship between the terms "grammar" and "model" (such as software models in UML)? One direction: a model is not necessarily a grammar because models can describe aspects other than structure. In particular, a software model is *not* a grammar because grammars are models of structures, whereas software models are models of software. However, the class-diagrammatic part of a software model *could* be viewed as a grammar — if the classes, without all behavioural details, lend themselves to a meaningful description of structures. A good example is a source-code *model*. The other direction: a grammar is certainly a model, namely a model of structures, but it is, at best, an incomplete software model because a grammar, by itself, does not model a software application.

—What is the relationship between the terms "grammar" and "meta-model" (in the sense of meta-modelling and model-driven development [metamodel.com 2005; Mellor et al. 2003])? There are varying definitions for the latter term. We adopt the view that a meta-model is a model of models such as a model of software models. That is, meta-models describe language constructs for modelling. One direction: we reckon that a meta-model *includes* a grammar, i.e., the structural description of a modelling language (as opposed to semantic constraints on models, if any). The other direction: *some* grammars are meta-models, namely those that describe language constructs for modelling (in particular, software modelling).

—A relational schema (in the sense of relational databases) is a borderline case. In general, we do not expect grammarware engineering to subsume relational modelling. Technically, the relational model comprises details, such as foreign key constraints, that go arguably beyond plain "formation of structure". Furthermore, the (basic) relational model lacks expressiveness for *general* alternatives; it only allows for NULL vs. NOT NULL values, which correspond to the regular operator "?" in EBNF terminology.

2.2 Grammar formalisms

We presume that the following *formalisms* provide the foundation for grammars:

—Context-free grammars.
—Algebraic signatures.
—Regular tree and graph grammars.

Clearly, these formalisms differ regarding expressiveness and convenience. Context-free grammars happen to enable the definition of concrete syntax of programming languages. Algebraic signatures are suitable for (per-definition) unambiguous abstract syntaxes. Graph grammars and the underlying schemas cater for graph structures.

There exist all kinds of partial, sometimes ad-hoc mappings to relate one formalism to the other. For instance, one can *convert* a context-free grammar into a signature by discarding terminals, by inventing a function symbol per production, and finally by recasting productions as types of function symbols. (Actually, there exists a somewhat forgotten algebraic interpretation of context-free grammars, which precisely formalises this direction.) The inverse direction can also be served by assuming a fixed syntax for function symbols such as prefix notation with parentheses and commas.

A grammar can be amenable to different *interpretations*. Since we want to emphasise that a grammar is a structural description, some interpretations are more meaningful than others. Let us consider some options for context-free grammars. First we note that it is of minor relevance whether we consider an acceptance-based vs. a generation-based semantics. For our purposes, a useful semantics of a context-free grammar is the set of all valid derivation trees [Aho and Ullman 73]. By contrast, the de-facto standard semantics of a context-free grammar is its generated language [Aho and Ullman 73] — a set of strings without attached structure. We contend that this semantics does not emphasise a grammar's role to serve as a structural description.

2.3 Grammar notations

Actual structural descriptions are normally given in some *grammar notation*, for example:

—Backus-Naur Form (BNF [Backus 1960]), Extended BNF (EBNF [ISO 1996]).
—The Syntax Definition Formalisms (SDF [Heering et al. 1989; Visser 1997]).
—The Abstract Syntax Description Language (ASDL [Wang et al. 1997]).
—Abstract Syntax Notation One (ASN.1 [Dubuisson 2000]).
—Syntax diagrams [Herriot 1976; McClure 1989; Braz 1990].
—Algebraic data types as in functional languages.
—Class dictionaries [Lieberherr 1988].
—UML class diagrams without behaviour [Gogolla and Kollmann 2000].
—XML schema definitions (XSD [W3C 2003]).
—Document type definitions (DTD [W3C 2004]).

In fact, there are so many grammar notations that we do not aim at a complete enumeration. It is important to realise that grammar notations do not necessarily reveal their grammar affinity via their official name. For instance, a large part of all grammars in this world are "programmed" in the type language of some programming language, e.g., in the common type system for .NET, or as polymorphic algebraic data types in typed functional programming languages. (We recall the last example in Fig. 1, which employed algebraic data types.)

Some grammar notations directly resemble a specific grammar formalism. For instance, BNF corresponds to context-free grammars. Other grammar notations might be more convenient than the underlying formalism, but not necessarily more expressive — in the *formal* sense of the generated language. For instance, EBNF adds convenience notation for regular operators to BNF. Hence, EBNF allows us to describe structures at a higher-level of abstraction, using a richer set of idioms, when compared to BNF. Yet other grammar notations appeal to a certain programmatic use. For instance, class dictionaries appeal to the object-oriented paradigm; they cater immediately for inheritance hierarchies. Finally, there are also grammar notations that strictly enhance a given formalism or a mix of formalisms. For instance, XSD is often said to have its foundation in tree grammars, but, in fact, it goes beyond simple tree grammars due to its support for references and unstructured data.

As with grammar formalisms, some couples of grammar notations are amenable to unidirectional or even bi-directional conversion. For instance, one can convert an EBNF grammar to a BNF grammar and vice versa. We also call this "yaccification" and "deyaccification" for obvious reasons [Lämmel and Wachsmuth 2001]. The SDF grammar format is richer than pure BNF and EBNF; SDF adds constructs for modularisation and disambiguation. Hence, BNF grammars are easily converted into SDF grammars, but an inverse conversion must be necessarily incomplete.

2.4 Grammar use cases

The grammars in Fig. 1 are *pure grammars*, i.e., plain structural descriptions. Nevertheless, we can infer hints regarding the intended use cases of those grammars. The BNF at the top of the figure comprises details of *concrete syntax* as needed for a language parser (or an unparser). The DTD in the middle favours a *markup-based representation* as needed for XML processing, tool interoperability, or external storage. Also, the provision of references from employees to their departments (cf. ID and IDREF) suggests that the use case asks for "easy" navigation from employees to top-level departments ("business units") — even though this provision is redundant because an employee element is unambiguously nested inside its business unit. The algebraic signature at the bottom of the figure does not involve any concrete syntax or markup, but it addresses nevertheless a specific use case. That is, the description captures the structure of (problem-specific) event traces of program execution. Such event grammars facilitate debugging and assertion checking [Auguston 1995]. Note that the algebraic signature for the event traces differs from the (abstract) syntax definition of the C programming language — even though these two grammatical structures are related in a systematic manner.

For clarity, we use the term *grammar use case* to refer to the purpose of a (possibly enriched) structural description. We distinguish *abstract* vs. *concrete use cases*. An abstract use case covers the overall purpose of a grammar without reference to operational arguments. For instance, the use cases "syntax definition" or "exchange format" are abstract. A concrete use case commits to an actual category of grammar-dependent software, which employs a grammar in a specific, operational manner. For instance, "parsing" or "serialisation" are concrete use cases. Even the most abstract use cases hint at some problem domain. For instance, "syntax definition" hints at programming languages or special-purpose languages, and "exchange format" hints at tool interoperability.

Here are details for representative examples of abstract grammar use cases:

—*Source-code models* are basically syntax definitions, but they are enriched with features such as annotation, scaffolding, and markup [Purtilo and Callahan 1989; Heuring et al. 1989; Koschke and Girard 1998; Sellink and Verhoef 2000b; Mamas and Kontogiannis 2000; Holt et al. 2000; Sim and Koschke 2001; Malton et al. 2001; Cordy et al. 2001; Kort and Lämmel 2003b; Winter 2003]. Also, source-code models tend to be defined such that they are effectively exchange formats at the same time.

—*Intermediate program representations* are akin to syntax definitions except that they are concerned with specific intermediate languages as they are used in compiler middle and back-ends as well as static analysers. Representative examples are the formats PDG and SSA [Ferrante et al. 1987; Cytron et al. 1991]. Compared to plain syntax definitions, these formats cater directly for control-flow and data-flow analyses.

—*Domain-specific exchange formats* cater for interoperation among software components in a given domain. For instance, the ATerm format [van den Brand et al. 2000] addresses the domain of generic language technology, and the GXL format [Holt et al. 2000] addresses the domain of graph-based tools. The former format is a proprietary design, whereas the latter format employs XML through a domain-specific XML schema.

—*Interaction protocols* cater for component communication and stream processing in object-oriented or agent-based systems. The protocols describe the actions to be performed by the collaborators in groups of objects or agents [Odell et al. 2001; Lind 2002]. Such protocols regulate sequences of actions, choices (or branching), and iteration (or recursive interactions). For instance, session types [Vallecillo et al. 2003; Gay et al. 2003] arguably describe interaction protocols in a grammar-like style.

There are just too many concrete grammar use cases to list them all. We would even feel uncomfortable to fully categorise them because this is a research topic on its own. We choose the general problem domain of language processing (including language implementation) to list *some* concrete grammar use case. In fact, we list typical language processors or components thereof. These concrete use cases tend to involve various syntaxes, intermediate representations, source-code models and other sorts of grammars:

—Debuggers [Auguston 1995; Olivier 2000].

—Program specialisers [Jones et al. 1993; Consel et al. 2004].

—Pre-processors [Favre 1996; Spinellis 2003] and post-processors.

—Code generators in back-ends [Emmelmann et al. 1989; Fraser et al. 1992].

—Pretty printers [van den Brand and Visser 1996; de Jonge 2002].

—Documentation generators [Sun Microsystems 2002; Marlow 2002].

In this agenda, all the grammar use cases that we mention are linked to *software engineering* including program development. One could favour an even broader view on grammarware. Indeed, in [Mernik et al. 2004], the authors revamp the classic term "grammar-based system" while including use cases that are not just related to software engineering, but also to artificial intelligence, genetic computing, and other fields in computer science.

2.5 Meta-grammarware

By itself, a grammar is not executable in the immediate sense of a program. It requires commitment to a concrete use case and usually also an enriched grammar before we can view it as an executable specification (or a program). We use the term *meta-grammarware* to refer to any software that supports concrete grammar use cases by some means of meta-programming, generative programming or domain-specific language implementation [Eisenecker and Czarnecki 2000; van Deursen et al. 2000].

The archetypal example of meta-grammarware is a program generator that takes an (enriched) grammar and produces an actual software component such as a parser. In practise, meta-grammarware is often packaged in frameworks for software transformation, program analysis, language processing, and program generation. Examples of such frameworks include the following: ASF+SDF Meta-Environment [Klint 1993; van den Brand et al. 2001], Cocktail [Grosch and Emmelmann 1991], Cornell Synthesizer Generator [Reps and Teitelbaum 1984], DMS [Baxter 1992], Eli [Gray et al. 1992], FermaT [Ward 1999], GENTLE [Schröer 1997], Lrc [Kuiper and Saraiva 1998], Progres [Progres group 2004], Refine [Smith et al. 1985; Abraido-Fandino 1987], RIGAL [Auguston 1990], S/SL [Holt et al. 1982], Stratego [Visser 2001a], Strafunski [Lämmel and Visser 2003], TXL [Cordy et al. 2002].

There are a few use cases of meta-grammarware that allow for the immediate derivation of the desired software component from plain grammatical structure. For instance, the generation of an object-oriented API for matching, building and walking over grammatically structured data [Wallace and Runciman 1999; de Jonge and Visser 2000; Sim 2000; Jong and Olivier 2004; Lämmel and Visser 2003; Moreau et al. 2003] is readily possible for algebraic signatures or suitably restricted context-free grammars.

Most use cases of meta-grammarware require enriched structural descriptions, for instance:

—*Parser specifications* such as those processed by the YACC tool [Johnson 1975] or any other parser generator. These specifications typically contain additional elements such as the parser-to-lexer binding, semantic actions, and pragmas.

—*Test-set specifications* such as those processed by the the DGL tool [Maurer 1990] or any other grammar-based test-data generator. These specifications annotate the basic grammar with control information as to guide test-data generation.

—*Pretty-printing specifications* [van den Brand and Visser 1996; de Jonge 2002]. These specifications attach horizontal and vertical alignment directives to the grammar structure as to guide line breaks and indentation.

—*Serialisable object models*, where meta-data for serialisation is attached to classes and fields in the object model such that serialisation (and de-serialisation) functionality can be generated by a tool or it can be defined in terms of reflection.

Our choice of the term meta-grammarware is inspired by Favre who has coined the term *metaware* [Favre 2003] in the meta-modelling context [metamodel.com 2005]. That is, metaware is application-independent software that helps producing software applications on the basis of suitable meta-models. We emphasise that the term meta-grammarware applies to grammarware rather than software models and meta-modelling.

2.6 Grammar-based formalisms and notations

There are actually a number of more fundamental *grammar-based formalisms* and corresponding notations. These are prominent examples of such grammar-based formalisms:

—Attribute grammars [Knuth 1968; Paakki 1995].
—General tree and graph grammars [Comon et al. 2003; Ehrig et al. 1996].
—Definite clause grammars (DCGs) [Pereira and Warren 1980].
—Advanced grammar formalisms for visual languages [Marriott and Meyer 1998].
—Logic programs (cf. the grammatical view in [Deransart and Maluszyński 1993]).

Corresponding grammar-based notations can be used for the implementation of concrete grammar use cases. For instance, the Progres framework [Progres group 2004] supports graph grammars, while compiler compilers such as Cocktail [Grosch and Emmelmann 1991], Cornell Synthesizer Generator [Reps and Teitelbaum 1984] and Eli [Gray et al. 1992] support attribute grammars.

We note that the distinction fundamental grammar formalisms vs. specification languages for meta-grammarware is not exact. For instance, parser specifications in the sense of YACC are often viewed as an example of attribute grammars. The difference is of an abstract, conceptual kind: grammar-based formalisms provide formal, computational frameworks with different assorted declarative and operational semantics. By contrast, specification languages for concrete grammar use cases were designed back-to-back with the meta-grammarware that supports them.

The aforementioned grammar-based formalisms have in common that the formation of basic grammatical structure is still traceable in the otherwise enriched structural descriptions. In Fig. 2, we provide illustrations. We discuss a few examples of the relationship between basic structural description and complete description:

—An attribute grammar starts from a context-free grammar, while each nonterminal is associated with attributes, and each production is associated with computations and conditions on the attributes of the involved attributes. The basic context-free grammar remains perfectly traceable in the completed attribute grammar.

—Likewise, the attributed multi-set grammar [Golin 1991] in Fig. 2 starts from the productions of a multi-set grammar, while there are geometric attributes and corresponding computations and conditions. The choice of a multi-set grammar (as opposed to a context-free grammar) implies that formation of structure is based on sets rather than sequences.

—The definite clause grammar in Fig. 2 is more entangled in the sense that semantic actions for checking context conditions are injected into the context-free productions. However, the pure productions were easily extracted, if necessary.

—Regular graph grammars are still in accordance with our assumptions for structural descriptions. Most applications of graph grammars [Nagl 1980; Hoffmann 1982; Nagl 1985; Schürr 1990; 1994; 1997] require more general graph grammars. Given a general graph grammar, we can again identify a basic structural description, namely the underlying *graph schema*. Such a schema defines types of nodes and edges.

A definite clause grammar for statically correct programs.

```
program --> declarations([],L), statements(L).

declarations(L0,L2) --> declaration(L0,L1), [";"], declarations(L1,L2).
declarations(L,L)   --> [].

declaration(L,[(I,T)|L])
  --> [id(I)], [":"], type(T), { \+ member((I,_),L) }.

statements(L) --> statement(L), [";"], statements(L).
statements(_) --> [].

statement(L) --> [id(I)], [":="], expression(L,T), { member((I,T),L) }.

expression(L,T) --> [id(I)], { member((I,T),L) }.
...
```

An attributed multi-set grammar for horizontally aligned lists separated by line segments.

$$
\begin{array}{ll}
[a1] & \mathit{List} \rightarrow \mathit{HorLineSeg} \\
& \begin{array}{lcl}
List_0.xmin & := & HorLineSeg.xmin \\
List_0.xmax & := & HorLineSeg.xmax \\
List_0.ymin & := & HorLineSeg.ymin \\
List_0.ymax & := & HorLineSeg.ymax \\
List_0.ycenter & := & HorLineSeg.ycenter
\end{array} \\
[a2] & \mathit{List}_0 \rightarrow \mathit{HorLineSeg}\ \mathit{Element}\ \mathit{List}_1 \\
& \begin{array}{lcl}
Element.ycenter & = & HorLineSeg.ycenter \\
HorLineSeg.ycenter & = & List_1.ycenter \\
HorLineSeg.xmax & = & Element.xmin \\
Element.xmax & = & List_1.xmin \\
List_0.xmin & := & HorLineSeg.xmin \\
List_0.xmax & := & List_1.xmax \\
List_0.ymin & := & min(min(HorLineSeg.ymin, Element.ymin), List_1.ymin) \\
List_0.ymax & := & max(max(HorLineSeg.ymax, Element.ymax), List_1.ymax) \\
List_0.ycenter & := & Element.ycenter
\end{array}
\end{array}
$$

Fig. 2. Illustration of grammar-based formalisms: *The definite clause grammar at the top refines the syntax definition from Fig. 1. Extra semantic actions (cf. {* `...` *}) establish type correctness with regard to a symbol table* `L`. *The attributed multi-set grammar at the bottom defines the visual syntax of horizontally aligned lists: think of* $\_\_x\_y\_\_z\_\_$. *There are constraints on the geometric attributes* $xmax$, $xmin$, *etc. that ensure line segments and list elements to be horizontally aligned along a centre of meaning.*

2.7 Commitment to grammatical structure

It is trivial to observe that parser specifications (and likewise the generated parsers) involve grammar dependencies because each such specification is *based* on a structural description quite obviously. More generally, the use of any grammar-based formalism or meta-grammarware implies grammar dependencies of such a trivial kind. However, software components tend to commit to grammatical structure by merely *mentioning patterns* of grammatical structure giving rise to more scattered grammar dependencies.

The modern, archetypal example is the scenario of a (problem-specific) XML document processor, be it an XSLT program. This program commits to the grammatical structure for the input, as expressed in patterns for *matched input*. Also, the processor is likely to commit to the grammatical structure for the output, as expressed in patterns for *built output*. Notice that the underlying program merely refers to grammatical structure (for input and output), but it cannot be viewed as an enriched structural representation by itself. As an aside, the document processor is "driven" by (the grammatical structure of) the *input* with a subordinated commitment to (the grammatical structure of) the output.

The fact that grammatical structure is entangled in programs is, to some extent, indented and it is inherent to grammar-based programming. Many software components, regardless of the used programming language and programming paradigm, end up committing to grammatical structure. Here are diverse examples:

—In imperative and object-oriented programs, one can use APIs to operate on grammatically structured data, e.g., to match, build and walk over data. This approach is widely used whenever components for language processing or document processing are encoded in mainstream languages. The APIs for data access are often generated by program generators [Grosch 1992; Visser 2001b; Jong and Olivier 2004]. The *use* of the API corresponds to commitment to grammatical structure.

—In functional and logic programs, heterogeneous tree-shaped data is manipulated on a regular basis. Depending on the fact whether we look at a typed or untyped language, the grammatical structure is available explicitly or implicitly (through use in code or documentation). As an aside, there is no need for hand-crafted or generated APIs for data access, when compared to mainstream imperative and OO languages, because functional and logic languages support term matching and building natively.

—Some approaches to term rewriting [van den Brand et al. 1998; Moreau et al. 2003] target language processing. For instance, the ASF+SDF Meta-Environment [Klint 1993; van den Brand et al. 2001] employs a marriage of a syntax definition formalism (SDF [Heering et al. 1989]) for the terms to be processed and an algebraic specification formalism (ASF [Bergstra et al. 1989]) for the actual rewriting rules.

—Grammar knowledge can also be expressed by the mere use of generic combinator libraries for concrete grammar use cases such as parsing, pretty-printing, or generic traversal [Hutton and Meijer 1998; Swierstra 2001; Hughes 1995; Lämmel and Visser 2002]. The required combinators are provided as abstractions in the programming language at hand, e.g., as higher-order functions in the case of functional programming. The encoding of grammatical structure boils down to *applications* of the combinators.

—Reflective and aspect-oriented functionality commits to grammatical structure because the employed metaobject protocols and join point models [Kiczales et al. 1991; Kiczales et al. 1997; Aßmann and Ludwig 1999] are based on grammars. Most notably, these protocols or models are ingeniously related to the abstract syntax of the underlying programming language. A more concrete scenario is debugging based on event grammars [Auguston 1995], where the steps of program execution are abstracted in a grammatical event structure, which is aligned with the abstract syntax of the language.

—Any library (in any language) that offers an API for the construction (or "formation") of functionality presumes that user code commits to the API, which corresponds to commitment to grammatical structure in a broader sense. There are other mechanisms for the systematic construction of functionality or entire software systems, which give rise to similar commitments. Examples include template instantiation, application generation, system composition, and program synthesis [Smith 1990; Eisenecker and Czarnecki 2000; Batory et al. 1994; Jarzabek 1995; Thibault and Consel 1997].

We note that commitment to grammar knowledge in programs does not necessarily imply that *precise patterns* of grammatical structure are to be expected in source code. For instance, industrial compiler front-ends are often hand-crafted. There are even techniques for grammarware development that intentionally depart from a strict grammar-based approach. For instance, the frameworks RIGAL [Auguston 1990] and S/SL [Holt et al. 1982] provide relatively free-wheeling idioms for parsing. An impure style of encoding grammatical structure is also practised in languages like Perl or Python; see [Klusener et al. 2005] for an example.

3. STATE OF THE ART: GRAMMARWARE HACKING

Given the pervasive role of grammars in software systems and development processes, one may expect that there exists a comprehensive set of best practises adding up to an engineering discipline for grammarware. However:

> *In reality, grammarware is treated, to a large extent, in an ad-hoc manner with regard to design, implementation, transformation, recovery, testing, etc.*

We will first contrast a typical case of wide-spread ad-hoc treatment with the potential of an engineering approach. Then, we will substantiate a lack of best practises at a more general level. Afterwards, we will argue that the lack of best practises is not too surprising since even foundations are missing. Also, there are no comprehensive books on the subject, neither do university curricula pay sufficient attention yet.

3.1 Hacking vs. engineering

To give a prototypical example of current ad-hoc approaches, we consider the development of parsers, as needed for software re-/reverse engineering tools. The common approach (shown on the left-hand side in Fig. 3) is to manually encode a grammar in the idiosyncratic input language of a specific parser generator. We encounter just one instance of grammarware tooling in this process: a parser generator. The driving principle is to appeal to the grammar class that is supported by the parser generator — often done by trial and error. The codebase, that must be parsed, is the oracle for this process.

There are a number of techniques that could be put to work in order to convert from hacking to engineering. Some of these techniques are illustrated on the right-hand side in Fig. 3:

- A technology-neutral grammar is *recovered semi-automatically* from available grammar knowledge, e.g., from a language reference that contains “raw” grammatical structure. In this process, the grammar is incrementally improved by transformations that model corrections and provisions of omissions. We can leverage tools for grammar extraction and transformation.
- We assume that the grammar can be executed by a prototype parsing framework. At this stage, the quality of parse trees is irrelevant. Also, we might largely ignore the issue of grammar-class conflicts and grammar ambiguities. We use the grammar as an acceptor only. The codebase drives the incremental improvement of the grammar.
- Parser specifications are *derived semi-automatically* from the recovered grammar using tools that customise grammars for a certain technology. Different parsing technologies

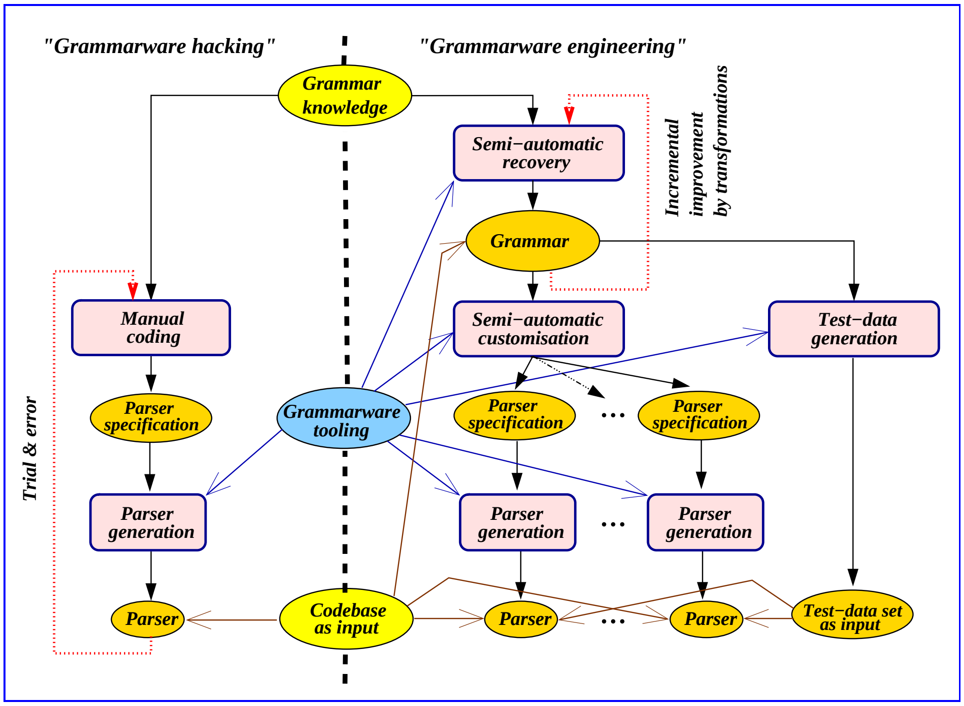

Fig. 3. Parser development: *The left-hand side illustrates common practise. Grammar knowledge, as contained in a language reference, is coded directly as a proprietary parser specification. Options for improvements are shown on the right-hand side. A technology-neutral grammar is recovered from the grammar knowledge, and subsequent customisation can target different parser technologies. The parsers are not just tested against the codebase of interest, but they are also stress-tested. Extra grammarware tooling supports this process.*

can be targeted as opposed to an early commitment to a specific technology. The customisation process is likely to require input from the grammarware engineer.

—There are opportunities for quality assurance by means of testing. We can stress-test the derived parsers using huge generated test-data sets. We can test a reference parser with positive and negative cases (not shown in the figure). We can perform a coverage analysis for the given codebase (not shown in the figure) to see how representative it is.

We have exercised elements of this approach in our team for a string of languages, e.g., for Cobol [Lämmel and Verhoef 2001b], which is widely used in business-critical systems, and for PLEX [Sellink and Verhoef 2000a], which is a proprietary language used at Ericsson.

3.2 Lack of best practises

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 adapting grammars in a traceable and reliable manner — not to mention the even more difficult problem of adapting grammatical structure that is ingrained in grammar-dependent software. This is a major problem because grammatical structure is undoubtedly subject to evolution.

—There is no established approach for maintaining relationships between grammatical structure as it is scattered over different grammar variations and grammar-dependent software components. This situation implies a barrier for evolution of grammarware.

—There is no established approach for delaying commitment to specific technology for the implementation of grammar use cases. Specific technology implies idiosyncratic notations, which make it difficult to alter the chosen technology and to reuse parts of the solution that are conceptually more generic.

The severity of the lack of best practises is best illustrated with yet another example of large scale. There exists a widespread belief that *parser generation* counts as a good grammar-biased example of automated software engineering. This belief is incompatible with the fact that some major compiler vendors do not employ any parser generator. (This claim is based on personal communication. The vendors do not wish to be named here.) One of the reasons that is sometimes cited is the insufficient support for the customisation of generated parsers. Another limitation of parser generators is that they do not provide sufficient programmer support for the grammar's convergence to the properties required by the technology. This leads to laborious hacking: cf. conflict resolution with LALR(1); cf. disambiguation with generalised LR parsing. Parser development is still a black art [van den Brand et al. 1998; Blasband 2001]. So if anyone is saying that grammarware engineering is a reality just because we have (many) parser generators, then this is not just a too restricted understanding of the term grammarware engineering; even the implicit claim about the adoption of parser generators does not hold as such.

3.3 Lack of comprehensive foundations

In fact, there is not just a lack of best practises. Even the fundamentals are missing:

—There is no "discipline of programming" (of the kind [Dijkstra 1976]) for grammars and grammar-dependent software. Likewise, there is no "mathematics of program construction" for grammars and grammar-dependent software. At a pragmatic level, we do not even have design patterns to communicate, and we also lack an effective notion of modular grammarware.

—There is no comprehensive *theory for transforming grammarware*, there are at best some specific kinds of grammar transformations, and some sorts of arguably related program and model transformations. We also lack a dedicated model for version management.

—There is no comprehensive *theory for testing grammarware*; this includes testing grammars themselves as well as testing grammar-dependent software in a grammar-aware manner. We also lack metrics and other quality notions.

—There is no comprehensive *model for debugging grammarware* as there exists for other sorts of programs, e.g., the box/port model for logic programming [Byrd 1980]. Debugging parsers or other grammar-dependent software is a black art.

—There is no unified *framework for relating major grammar forms and notations* in a reasonably operational manner. Theoretical expressiveness results provide little help with the mediation between the grammar forms in actual grammarware development.

3.4 Lack of books on grammarware

It is instructive to notice how little knowledge on grammarware is available in the form of textbooks or engineering handbooks. Even in restricted domains, there are hardly textbooks that cover engineering aspects. For instance, texts on compiler construction, e.g., [Aho and Ullman 73; Aho et al. 1986; Wilhelm and Maurer 1995], go into details of parsing algorithms, but they do not address engineering aspects such as grammar style, grammar metrics, grammar customisation, evolutionary grammar transformations, and grammar testing. There exist a few textbooks that discuss particular frameworks for generic language technology or compiler construction, e.g., [van Deursen et al. 1996; Schröer 1997], without coverage of general engineering aspects. There exist textbooks on problem domains that involve grammar-based programming techniques. For instance, there is a comprehensive textbook on generative programming [Eisenecker and Czarnecki 2000]. There is no such book for grammar-based software transformation. There exist a few textbooks on paradigms for grammar-based programming techniques, e.g., attribute grammars [Alblas and Melichar 1991] and graph transformation [Ehrig et al. 1996]. Again, these books focus on a specific paradigm without noteworthy coverage of the engineering aspects of the involved grammars.

3.5 Lack of coverage in curricula

In the last three decades or so, parsing algorithms and compiler construction formed integral parts of computer science curricula at most universities. The default host for these topics was indeed a compiler class. Some related, theoretical aspects, such as the Chomsky hierarchy, were likely to be covered in a class on foundations of computer science. Engineering aspects of grammarware have never been covered broadly. It is conceivable that a modern compiler class [Griswold 2002] incorporates more software engineering in general, and engineering aspects of grammars (as they occur in compilers) in particular.

A dedicated grammarware class will be more comprehensive in terms of the engineering aspects it can cover. Also, such a class will be a strong host for discussing different problem domains for grammarware *including* compiler construction. Over the last few years, the fields of meta-modelling and model-driven development (MDD) have received ample attention from the research community, and this trend could fully reach curricula soon. A meta-modelling/MDD class can be customised such that it covers technical aspects of grammarware engineering, e.g., the different grammar notations and their relationships, the various grammar use cases and grammar-based testing. Likewise, classes on software re-/reverse engineering, if they became popular, can be made more grammar-aware.

4. THE GRAMMARWARE DILEMMA

We have shown that even though grammarware permeates software systems, its engineering aspects are somewhat neglected. Here is what we call the grammarware dilemma:

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

4.1 Unpopular grammarware research

Part of the answer lies in a *popularity problem* of grammar research. Grammars in the sense of definitions of string languages are well-studied subjects in computer science. Basic research on grammars and parsing was a wave of the 1960s and 1970s. The pervasiveness of grammars in software systems was not yet so obvious at the time. Hence, engineering aspects did not get into the focus. We might see now the beginning of a second wave of grammar research, where a new generation of researchers rediscovers this theme, while being driven by engineering aspects. According to Thomas Kuhn's "The Structure of Scientific Revolutions" [Kuhn 1970], research generally tends to go in such waves, while social issues play an immanent role in this process. When grammar-enthusiastic researchers of the first wave turned into senior researchers, then their junior staff often favoured the exploration of different territory.

4.2 Myths about grammarware

The grammarware dilemma must also be explained in terms of myths about grammarware. These myths are barriers for anyone who wants to do research on grammarware. By naming these myths, we hope to prepare the ground for work on a comprehensive engineering discipline for grammarware.

—*Myth "Grammarware engineering is all about parser development."*

In any language processor, the front-end with its parsing functionality is so overwhelmingly visible that one can easily neglect all the other grammars that occur in a language processor: different abstract syntaxes with variations on annotations, eliminated patterns due to normalisation, preprocessing information, and others. Software components that do not even start from any concrete syntax are easily neglected as grammarware altogether. For instance, a number of mainstream technologies for aspect-oriented programming use XML at the surface for their pointcut languages rather than any concrete syntax. The underlying schema for pointcuts and functionality based on it should still be subjected to grammarware engineering.

—*Myth "Grammarware engineering is all about language processing."*

Incidentally, our reply to the parsing myth invites for such a reduction. However, there are clearly grammar use cases that do not deal with language processing. For instance, the use case "interaction protocol" is not related to language processing according to common sense. Another example: the problem of deriving hierarchical (XML-based) views on relational data in a database, as addressed by various data access APIs in modern programming environments, is about data processing rather than language processing. Nevertheless, the language processing myth is actually a useful approximation of the scope of grammarware engineering, while it is important to adopt a broad view on languages: programming languages, domain-specific languages, configuration languages, modelling languages.

—*Myth "XML is the answer"*

Recall the question: what are the software engineer's methods to design, customise, implement, ... and test grammars; how to handle grammatical structure that is implemented in software components? "XML grammars" (i.e., DTDs, XML schemas, etc.) are in need of an engineering discipline as much as any other grammar notation. Issues of schema evolution, co-evolution of schema-dependent software, and schema-aware testing of schema-based software are all urgent research topics in the "narrow" XML context. Also, XML offers new challenges for grammarware engineering. For instance, the mere mapping between different grammar notations is absolutely non-trivial if an (arbitrary) XML schema is involved on either side. Finally, XML lacks support for some grammar use cases; most notably for concrete syntax definitions.

—*Myth "Meta-modelling is the answer"*

We rehash: Grammarware engineering addresses development and maintenance of grammars and grammar-dependent software. By contrast, meta-modelling focuses on the provision of meta-models, i.e., models of models, in particular: models of software models. According to Sec. 2.1, grammars and meta-models are not in any simple equivalence or subsumption relationship, which implies that meta-modelling and grammarware engineering are complementary. In particular, most grammars tend to be models (of structures) rather than *meta*-models of anything. One might say that "meta-modelling for grammars" can be understood to cover the field of "grammar modelling languages" (BNF, EBNF, ASN.1, etc.), which corresponds, indeed, to a certain part of grammarware engineering. It is hard to see how contemporary meta-modelling would address the technical challenges in grammarware engineering, e.g., transformation and testing of grammar-dependent software, customisation of grammars for use cases, or commitment to common technology options.

—*Myth "Grammarware engineering is a form of model-driven development"*

What is model-driven (software) development (MDD) in the first place? MDD is an emerging field. Our current perception of MDD is inspired by [Mellor et al. 2003; Selic 2003; Favre 2004]: MDD aims at a model-centric approach to software development, where models are systematically transformed into actual software applications. Normally, support for round-trip engineering is also required, i.e., changes to the software can be pushed back into the models. According to Sec. 2.1, grammars and models are not in any simple equivalence or subsumption relationship, but one could still want to argue that grammarware engineering is actually an instance of MDD, i.e., grammar-driven development (GDD) or MDD for grammarware. We do not object to this view, and recent MDD literature indeed recognises grammarware as one typical "technological space" in the broader MDD context [Kurtev et al. 2002; Favre 2004]. In terms of aspirations, the two fields differ as follows:

—MDD aspires to revolutionise software development by favouring models over programs, modelling over programming, model transformations over code revisions.

—Grammarware engineering is grammar-biased and "conservative": it targets grammatical structure in all the grammar use cases that have been existing for decades.

In Fig. 4, we compare the mythical (or perceived) view and the proposed view on grammarware. The mythical view has not triggered an effort on grammarware engineering. The proposed view emphasises the pervasiveness of ingrained grammar dependencies as opposed to merely the grammars that reside within compiler front-ends. The proposed view justifies a major effort on grammarware engineering.

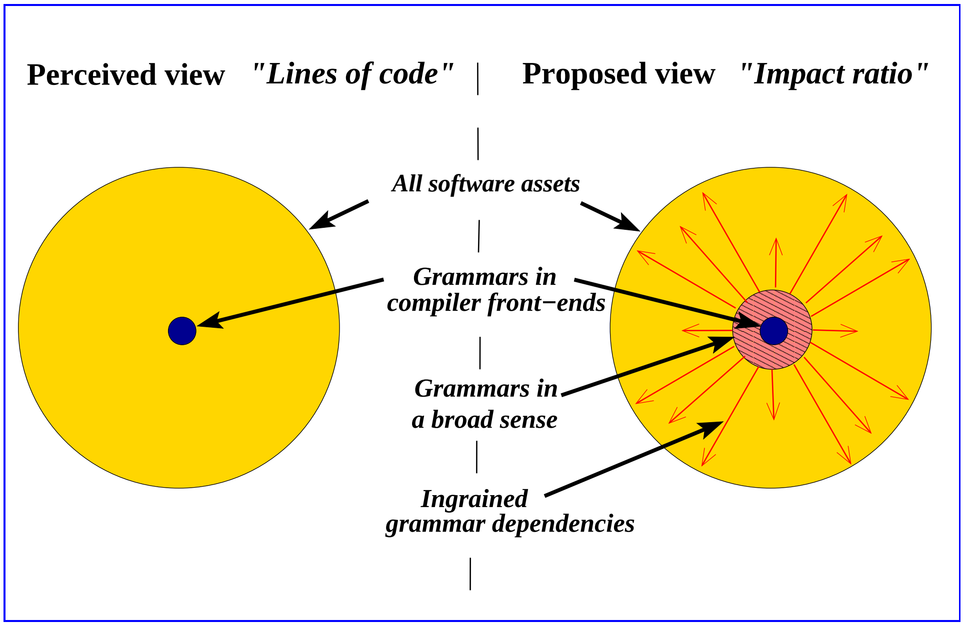

Fig. 4. In need of a paradigm shift: *On the left-hand side, we only care about obvious grammar forms, namely the ratio of "all software" to "grammars in compiler front-ends". On the right-hand side, we admit two important facts: (i) there are many grammars other than those in compiler front-ends; (ii) ingrained grammar dependencies have a deep impact on most software.*

5. PROMISES OF GRAMMARWARE ENGINEERING

At this point, the reader might face the following question:

> *Somehow we managed to deal with all these kinds of grammarware for decades. So what? That is, what are the potential benefits for IT?*

The overall promise of grammarware engineering is that it leads to improved quality of grammarware and to increased productivity of grammarware development. These promises should provide a good incentive since grammars permeate software systems and software development. Of course, it is difficult to justify such general claims at this time. To provide some concrete data, we will report on two showcases (or even success stories). Afterwards, we will identify more detailed promises on the basis of these showcases, but we will also refer to further scattered experiences with engineering aspects of grammarware.

5.1 Showcase: grammar recovery

This showcase is discussed in detail in [Lämmel and Verhoef 2001b; Lämmel 2005]. Using elements of the emerging engineering discipline for grammarware, we were able to rapidly recover a relatively correct and complete syntax definition of VS Cobol II. The starting point for this recovery project was IBM's industrial standard for VS Cobol II [IBM Corporation 1993]. 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 transformation. After that period, we were able to parse all the VS Cobol II code that was available to us (several millions lines). We should note that additional effort will be needed to develop general, mature tools, and to deploy the syntax definitions in different industrial settings. Key to success was a systematic process, automation of grammar transformations, and parser testing based on a prototype technology. This project is part of a series of similar recovery projects [van den Brand et al. 1997; Sellink and Verhoef 2000a; van den Brand et al. 2000]. 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. (Even today, most business-critical code still resides in Cobol portfolios [Arranga et al. 2000].) Industrial Cobol front-ends are always considered intellectual property because the costs for their development and maintenance are considerable and the involved technologies are proprietary.

5.2 Showcase: API-fication

This showcase is discussed in detail in [Jong and Olivier 2004]. Using elements of the emerging engineering discipline for grammarware, members of our team dramatically improved the architecture of the ASF+SDF Meta-Environment [Klint 1993; van den Brand et al. 2001]. This system supports generic language technology on the basis of executable specifications for language-based, interactive tools. The current system is the result of many person years of design, development and evolution. The system is being used in industrial applications dealing with software renovation, domain-specific application generation [van den Brand et al. 1996], and others. The architectural revision of the system concerned the usage of the internal ATerm format [van den Brand et al. 2000] for generic data representation. While infrastructures for generic language functionality normally require such a generic format, a consequence is that programmers are encouraged to encode specific format knowledge of manipulated data in the code. This leads to heavily tangled code. In the case of the C- and Java-based ASF+SDF Meta-Environment, knowledge of several parse-tree formats and other specific formats was scattered all-over the ATerm-based functionality in the system. The architectural revision of the system aimed at an "API-fication". We use this term to denote the process of replacing low-level APIs by higher-level APIs. Here, an API is viewed as a set of C functions, Java methods, and that alike. In the showcase, the low-level API supports processing of plain ATerms, while several high-level APIs support data access for different parse-tree formats and others. The high-level APIs were generated from grammars. The API-fication of the ASF+SDF Meta-Environment led to an explicit representation of specific formats. Also, nearly half of the manually written code was eliminated.

5.3 Promise: increased productivity

The recovery showcase suggests increased productivity as a promise of grammarware engineering because other known figures for the development of quality Cobol grammars are in the range of two or three years [Lämmel and Verhoef 2001b; 2001a]. We analyse the IT value of this speedup in [Lämmel and Verhoef 2001a]. 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 make software re-/reverse engineering scalable in the view of software portfolios in the millions-of-lines-of-code range. Currently, solution providers for legacy modernisation are not able to serve the full spectrum of languages and dialects; as noted by the Gartner Group [Gartner Research 2003]. Apparently, parser development and source-code modelling are very expensive in practise, up to a degree that automated software analysis and modification becomes unaffordable.

Productivity gains are by no means restricted to grammar recovery. Generally, systematic processes and automation in the grammarware life cycle increase productivity.

5.4 Promise: improved evolvability

The API-fication showcase made extra grammatical structure accessible to static typing. This is clearly beneficial for evolution because types make evolutionary adaptations of grammarware more self-checking. In fact, the API-fication effort was triggered by the need to change the parse-tree format, which was found to be too difficult to perform on the original system with its implicit grammar knowledge.

Improved evolvability can also be expected from techniques that *operationalise links* between scattered grammar knowledge. That is, if grammatical structure changes in the context of one use case, then these changes can be propagated to other use cases. An example of an operationalised link is the semi-automatic derivation of a tolerant parser from a more strict grammar [Barnard 1981; Barnard and Holt 1982; Klusener and Lämmel 2003].

5.5 Promise: improved robustness

Static typing of grammarware improves its robustness because it rules out inconsistent grammar patterns in code. That is, the type system of the used specification or programming language is exploited to enforce adherence to a grammar. The API-fication showcase illustrates that generic language technology can require special efforts. The aforementioned *operationalisation of links* between scattered grammar knowledge tackles robustness as well: it makes sure that different components 'talk in the same language', which is clearly important for robust interoperability. Robustness of grammarware will also be improved by effective *reuse*. Unfortunately, we do not yet fully understand how to reuse grammarware. Contemporary grammarware tends to be too monolithic, too technology-dependent, and too application-specific for reuse. Finally, robustness of grammarware will also be improved by *grammar-based testing*. Most notably, differential testing and stress testing can be supported by grammar-based test-data generation using a stochastic approach or even proper coverage criteria. Applications of grammar-based testing are reported in [McKeeman 1998; Sirer and Bershad 1999; Veerman 2005].

5.6 Promise: less patches, more enhancements

The promises of grammarware engineering can be compared with known benefits of modern development methodologies. In [Dekleva 1992], Dekleva addressed the (as it turned out unsubstantiated) assumption that the improved quality of a system's structure and other improvements would reduce maintenance time. This was a shared misconception at that time. Dekleva summarised:

> *"The survey findings do not support the proposition that the application of modern information systems development methodology decreases maintenance*

> *time. However, some benefits are identified. Time spent on emergency error correction, as well as the number of system failures, decreased significantly with the application of modern methodology. Systems developed with modern methodologies seem to facilitate making greater changes in functionality as the system changes.*"

Likewise, we expect that patching work in grammarware maintenance will diminish, failures of grammarware are avoided by construction, so that more time is left for enhancing grammarware, while enhancements do not harm robustness of the grammarware. In fact this is the main motive for aiming at an engineering discipline for grammarware.

6. PRINCIPLES OF GRAMMARWARE ENGINEERING

We contend that an engineering discipline for grammarware is to be based on the principles that follow. None of the principles should be surprising since they are all adopted from contemporary common sense in software engineering. The point is that contemporary grammarware development does *not* adhere to these principles, despite their advisability. However, there exist several supportive samples of using these principles. We will provide corresponding references in due course.

6.1 Principle: start from base-line grammars

When designing grammarware, too early commitment to a concrete use case, specific technology (meta-grammarware), and other implementational choices shall be avoided. To this end, grammarware development shall depart from pure grammars: more or less plain structural descriptions using a fundamental notation. Within the grammarware life cycle, we use the term *base-line grammar* to denote such grammars. Base-line grammars should be sufficiently structured and annotated to be useful in the potential derivation of concrete syntaxes, object models, and other typical forms of use-case specific grammars. If necessary, base-line grammars can be complemented by assorted constraints and semantics for the described structures. The constraints and the semantics shall be "universal", i.e., they must not be specific to a use case.

6.2 Principle: customise for grammar use cases

We derive new grammars and enriched structural specifications via customisation from base-line grammars. Here are some existing techniques that exercise this principle:

—In [Kadhim and Waite 1996; Wile 1997], approaches for the operationalisation of the link between concrete and abstract syntax definition are described. That is, concrete syntax definitions are customised into abstract syntax definitions.

—In [Aho et al. 1986; Lohmann et al. 2004], advanced transformations for the removal of left-recursion in a context-free grammar are described. This sort of customisation is a preparatory step when we want to commit to basic parsing technology for recursive descent. The cited approaches are advanced in so far that transformation is not limited to context-free grammars but the grammar transformation is also lifted to the level of attribute grammars. Here, we assume that the attribute grammars model parse-tree synthesis. The approaches guarantee that the synthesised parse-trees do *not* change, even though the underlying grammar does change.

—Customisation is expected to be useful for converting pure grammars into parser specifications. Relevant idioms for parser specification exist in abundance. For instance, there are idioms that address disambiguation: extra actions for semantics-directed parsing [Parr and Quong 1994; Breuer and Bowen 1995], decorated tokens [Malloy et al. 2003], filters on parse-tree forests [Klint and Visser 1994; van den Brand et al. 2002]. These idioms tend to be coupled with specific technology. Also, one can not exercise these idioms in an incremental fashion such that a given grammar could be adapted in the context of a specific use case.

—A very limited form of grammar customisation is provided by GDK — the Grammar Deployment Kit [Kort et al. 2002], which generates different parser specifications from a general grammar notation. Some minor details of generation can be controlled via a trivial command-line interface. Otherwise, GDK assumes that grammars are prepared prior to export to the chosen parser technology — by means of grammar transformations.

The present-day approach to customisation is predominantly ad-hoc and manual. A general view on automated grammar customisation could be based on concepts of aspect-oriented programming [Kiczales et al. 1997; Elrad et al. 2001] pending an adoption to grammarware. That is, any customisation step could be viewed as the superimposition of advice onto an existing grammar or grammar-dependent software component. This superimposition would be realised by grammarware transformations using a weaving semantics. Furthermore, concepts of model-driven development [Mellor et al. 2003; Selic 2003], in particular, model transformations [Sendall and Kozaczynski 2003] could provide a useful organisation principle for customisation. That is, the base-line grammar in grammarware engineering can be viewed as the platform-independent model (PIM) in model-driven architecture (MDA [OMG 2004]), and each grammar use case, or each intermediate step can be viewed as a platform-specific model (PSM).

6.3 Principle: separate concerns in grammarware

Separation of concerns in software (including grammarware) is supposed to facilitate reuse and modular reasoning [Dijkstra 1976]. A given piece of grammarware indeed tends to deal with several concerns. One can distinguish grammar concerns (i.e., modularisation of the grammar as such), and grammar-based concerns (i.e., modularisation of functionality on top of the grammar). For instance, in a typical re-/reverse engineering front-end, one can find the following grammar concerns (which are unfortunately not separated in practise):

—Base syntax.

—Comments and layout (indentation).

—Preprocessing syntax.

—Error handling rules.

A re-engineering transformation could exhibit the following grammar-based concerns:

—The primary transformation.

—Preparatory or on-the-fly analyses.

—A helper concern for change logging.

—A helper concern for sanity checking.

Some techniques for the separation of grammar concerns are described in [Purtilo and Callahan 1989; Kadhim and Waite 1996; Malton et al. 2001; Cordy 2003]. Research on modular attribute grammars has resulted in some techniques for the separation of grammar-based concerns [Farrow et al. 1992; Kastens and Waite 1994; Lämmel 1999a; Lämmel and Riedewald 1999; Lämmel 1999b; de Moor et al. 2000]. There are mixed techniques such as origin tracking in term rewriting [van Deursen et al. 1993], and parse trees with 'active' annotations [Kort and Lämmel 2003b]. We contend these techniques need to be further developed and marketed before they are widely adopted.

An effective separation of concerns in grammarware often requires advanced means of modularisation. To give an example, let us consider pretty-printing program text. One concern is to define a comprehensive set of pretty-print rules for all constructs. Another potential concern is the preservation of preexisting formatting information [de Jonge 2002]. The challenge is that these concerns (or features) interact with each other in a complicated, so far insufficiently understood manner.

6.4 Principle: evolve grammarware by transformation

The present-day approach to grammarware evolution is predominantly ad-hoc and manual. We propose that evolution of grammarware is operationalised via automated transformations. Since grammars permeate grammar-dependent software, any grammar change has a strong impact. Hence, the evolution of grammatical structure must be effectively transposed to the level of grammar-dependent software components. That is, any grammar transformation has to be completed by a transformation of all grammar-dependent functionality. Likewise, any grammatically structured data is subject to a data transformation in case the type-providing grammar has been changed. Consequently, we face transformations at three levels:

—Grammar transformations.

—Software transformations for grammar-dependent software.

—Data transformations for grammatically structure data.

Evolution must also handle the issue of grammar variations that reside in different software components. The related grammars either evolve jointly, or the evolution of one grammar (use case) must be hidden from the other grammar (use case) by means of a "grammar bridge", i.e., a grammar-based conversion component.

In Fig. 5, we instantiate the different levels of grammarware evolution for XML:

Grammarware	*XML*
Grammar	XML schema (or DTD)
Grammar-dependent program	XML document processor (e.g., XSLT)
Grammatically structured data	XML data (XML stream / document)

The middle layer in the figure represents an XML-schema transformation. The top and the bottom layers complete the primary schema transformation to be meaningful for dependent document-processing functionality and corresponding XML streams.

The derivation of a data transformation from a schema transformation is relatively well understood in the context of databases; cf. database schema mappings coupled with an instance mapping [Hainaut et al. 1993; Henrad et al. 2002; Gogolla and Lindow 2003].

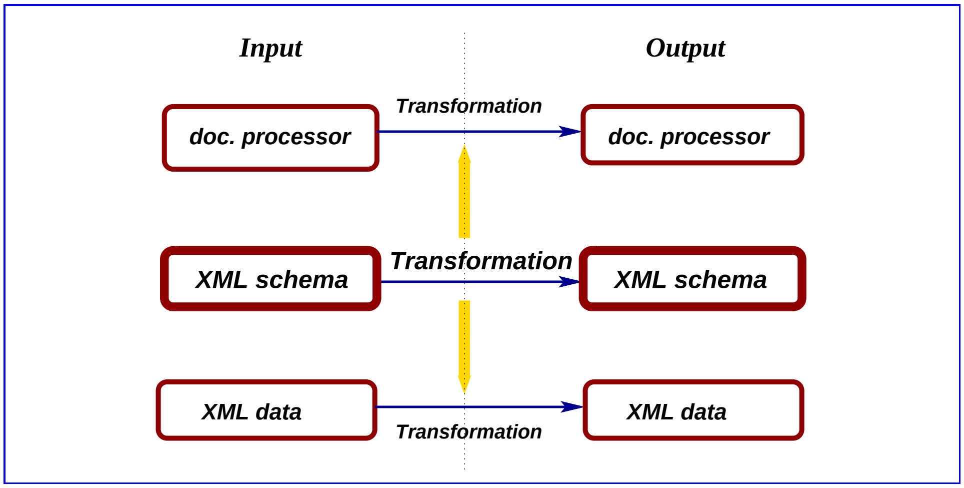

Fig. 5. Multi-level transformations in the XML setting: *The primary transformation is defined at the XML schema level, while the transformations at the document-processing level (e.g., XSLT) and the XML-stream level are supposed to be implied.*

Some similar work has been reported on XML grammars [Lämmel and Lohmann 2001; IBM Research 2002]. More generally, we view pairs of transformations on schema and data as an important instance of the notion of "coupled transformation" [Lämmel 2004a].

The derivation of a program transformation from a schema transformation is weakly understood both in the XML context and the database context. However, object-oriented program refactoring [Griswold and Notkin 1993; Opdyke 1992] instantiates this sort of coupling, where class structures can be refactored and all dependent method implementations are "automatically" updated. Clearly, evolutionary transformations can go beyond mere refactoring. In [Kort and Lämmel 2003a], we consider coupled transformations for types and functions in a functional program, while we even go beyond refactoring. Some forms of model transformations [Sendall and Kozaczynski 2003] (in the sense of the emerging field of model-driven development) might be applicable in the grammarware context.

Evolution comprises refactoring, enhancement, as well as clean-up. In the broader sense, evolution also comprises re-targeting grammarware from one technology to another. Basic grammar transformations for refactoring, enhancement, and clean-up were developed in [Lämmel 2001a]. Evolutionary transformations of software have generally not yet received much attention, except for the refactoring mode of evolution. The situation is not different for grammarware, but some initial ideas are summarised in [Lämmel 1999b; 2004b], where rule-based programs are transformed in a number of ways, including some grammar-biased modifications, some of them going beyond refactoring.

6.5 Principle: reverse-engineer legacy grammarware

We can not assume that suitable base-line grammars are readily available for all legacy grammarware. However, it is fair to assume that there is some encoded grammar knowledge available, from which base-line grammars can be recovered by means of reverse engineering. The grammar knowledge can reside in data, e.g., one can infer an XML schema from given XML documents. The grammar knowledge can also reside in source code or in a semi-structured document, e.g., in a hand-crafted recursive-descent parser or in a semi-formal reference manual. The latter scenario was discussed in detail in Sec. 5.1.

The recovery of base-line grammars is an issue for grammars in a broad sense, not just for syntax definitions of widely used programming languages. It is a common maintenance scenario to recover grammars for DSLs and (data-access) APIs. Typical triggers for such recovery efforts are the following:

—A proprietary language or API must be replaced.

—New grammar-based tools have to be developed.

—The language or API at hand must be documented.

Here are two specific examples that illustrate the link between recovery and enabled forward engineering. In [Sellink and Verhoef 2000a; van den Brand et al. 2000], we describe a project related to the proprietary language PLEX used at Ericsson. The project delivered a recovered PLEX grammar, a documentation of PLEX, and a new parser for PLEX. In [de Jonge and Monajemi 2001], a project is described that relates to the proprietary SDL dialect used at Lucent Technologies. The project delivered a recovered SDL grammar, and a number of SDL tools, e.g., a graph generator for finite state machines.

6.6 Principle: ensure quality of grammarware

We need quality notions or metrics in the first place. We need automated metrics calculation in the second place. We need effective (computable) techniques to assess quality of grammarware and to steer the improvement of quality. This development has to distinguish grammars vs. grammar-dependent software. As far as grammars are concerned, we need to identify grammar metrics, grammar styles, and notions of correctness and completeness. Quality attributes of grammar-dependent software shall be these: correctness in the sense of differential testing, conformance in the sense of conformance testing, performance attributes, complexity metrics, type validation, and others.

Some grammar metrics have been defined and used in [Sellink and Verhoef 2000a] in the context of assessing the code quality and the status of grammars during grammar reverse engineering. Specific notions of relative grammar correctness and completeness were defined in [Lämmel 2001b] with the goal of aligning a grammar to a proprietary (i.e., black box) reference parser.

Techniques for quality assessment and improvement for grammar-dependent software might explicitly involve the grammatical structure at hand, in which case we call these techniques *grammar-based*. For instance, grammar-based testing of grammar-dependent software would be based on test-data sets that cover the underlying grammar [Purdom 1972; Lämmel and Harm 2001]. Grammar-based testing can be partially automated by grammar-based test-data generation; see [Burgess 1994; McKeeman 1998] for compiler testing, and [Maurer 1990; Sirer and Bershad 1999] for other settings. Clearly, validation of a grammar-dependent software component is not necessarily grammar-based. For instance, validation by means of manually developed conformance suites [NIST 2003; Malloy et al. 2002] might focus on I/O behaviour rather than grammatical structure.

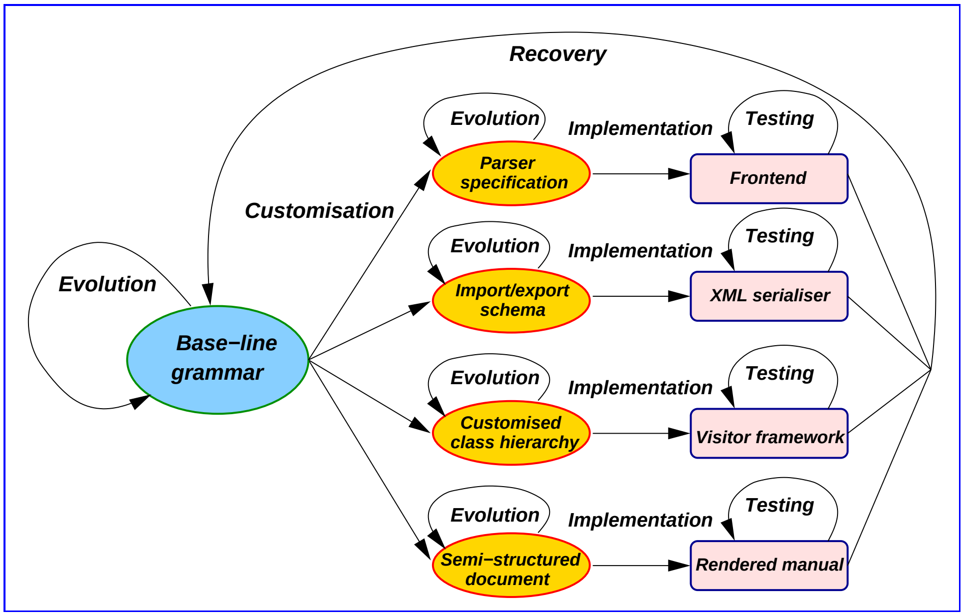

Fig. 6. The grammarware life cycle: *Base-line grammars do not commit to a technology or a use case. The stack in the middle lists some grammar use cases, which are derived by customisation. Both base-line grammars and grammar use cases can be subject to different sorts of evolution, while evolution of base-line grammars should be preferred over evolution of grammar use cases, whenever possible. Grammar use cases can be implemented by meta-grammarware or by grammar-based programming techniques. The grammar life cycle is enabled by grammar recovery, which recovers base-line grammars from implementations or others, if necessary.*

6.7 The grammarware life cycle

The discussed principles can be integrated in a grammarware life cycle; see Fig. 6. By having a proper grammarware life-cycle we can invigorate the normal software life-cycle. Most notably, the distinction of base-line grammars vs. grammar use cases allows us to apply evolutionary transformations to the former such that the adaptations of the latter are mostly implied. That is, grammar use cases are supposed to co-evolve with base-line grammars. There are clearly evolution scenarios that are inherently technology- and use-case-specific, in which case evolutionary transformations must be carried out on grammar use cases.

To align the grammarware life cycle with the normal software life cycle, we will briefly go through Fig. 6. We will focus on *forward engineering* — knowing that we will neglect some trips through the figure. There are the following phases:

—Provision of base-line grammars.

—Customisation to derive grammar use cases.

—Implementation to obtain actual grammar-dependent software.

—(Potentially grammar-based) testing of the grammar-dependent software.

Here is one scenario for forward engineering from Fig. 6: going from a base-line grammar to an object-oriented *visitor framework* through a *customised class hierarchy*. The derivation of the use case requires a class dictionary. (Hence, either the base-line grammar must be a class dictionary, or it must be amenable to a mapping that delivers a class dictionary.) For the sake of an interesting (and realistic) customisation requirement, we assume that the final object structures are supposed to carry extra links for use/def relations. To this end, the customisation has to enhance the class hierarchy accordingly, when compared to the base-line grammar. The enhanced class hierarchy can now be "'implemented" by generating a visitor framework for traversing object structures, as it is pursued in [Visser 2001b] and elsewhere. Ultimately, we obtained a component of grammar-dependent software: a compiled and packaged visitor framework.

6.8 Automated grammar transformations

Several principles of grammarware engineering can be supported through transformations, which are to be automated for reasons of traceability and scalability. We will now focus on grammar transformations, assuming that they can also steer the provision of grammar-aware transformations of grammar-dependent software. Grammarware engineering employs grammar transformations in the sense of a *meta-programming technique*. A grammarware engineer "codes" grammar transformations to express intents of evolution, customisation, and recovery. (This view differs from compiler construction [Aho et al. 1986], where grammar transformations are executed by parser generators and other tools in a black-box fashion.) Grammar transformations can be recorded in scripts. One can envisage interactive tool support for grammar transformation.

Let us consider some examples. We will illustrate recovery transformations for a syntax definition of Cobol. The reported examples were encountered in the aforementioned recovery project [Lämmel and Verhoef 2001b] for a Cobol grammar. According to the industrial standard for VS Cobol II [IBM Corporation 1993], an `ADD` statement can be of the following form (in EBNF notation):

```
add-statement =
  "ADD" (identifier|literal)+ "TO" (identifier "ROUNDED"?)+
  ("ON"? "SIZE" "ERROR" imperative-statement)?
  ("NOT" "ON"? "SIZE" "ERROR" imperative-statement)?
  "END-ADD"?
// two other forms of ADD statements omitted
```

This production is actually incomplete in terms of the intended syntax.
We quote an informal rule from IBM's VS Cobol II reference [IBM Corporation 1993]:

> 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 transformation replaces the occurrences of the nonterminal `imperative-statement` by the EBNF phrase `imperative-statement+`, as suggested by the informal rule. We call this a generalisation because the resulting grammar is more general than the original one — in the formal sense of the generated language. Here is the result:

```
add-statement =
```

```
"ADD" (identifier|literal)+ "TO" (identifier "ROUNDED"?)+
("ON"? "SIZE" "ERROR" imperative-statement+)?
("NOT" "ON"? "SIZE" "ERROR" imperative-statement+)?
"END-ADD"?
```

We will also illustrate transformations for grammar refactoring. The `ON-SIZE-ERROR` and `NOT-ON-SIZE-ERROR` phrases occur in other forms of `ADD`-statements and many other Cobol statements again and again. So we single out these phrases by extraction, which will lead to a more concise grammar. We apply the following transformations:

```
extract "ON"? "SIZE" "ERROR" imperative-statement+
     as on-size-error-phrase
extract "NOT" on-size-error-phrase
     as not-on-size-error-phrase
```

That is, we extract some parts of the productions for `ADD`-statements (and others) such that they constitute new nonterminals `on-size-error` and `not-on-size-error`. Consequently, the modified production looks as follows:

```
add-statement =
  "ADD" ( identifier | literal )+ "TO" ( identifier "ROUNDED"? )+
  on-size-error? not-on-size-error?
  "END-ADD"?
```

Generally, one can classify grammar transformations in terms of usage scenarios (and the assorted preservation properties). We have seen examples of generalisation and extraction. Here is a more profound list of scenarios:

—*Refactoring*: a grammar is improved to become more concise, more readable, better amenable to subsequent changes. Refactoring can be used during evolution, customisation, and recovery. Extraction (see above) is a form of refactoring.

—*Style conversion*: a grammar of a certain normal form ("style") is derived. For instance, regular operators can be eliminated in an EBNF to arrive at a pure BNF. (Style conversions preserve the generated language, just as refactoring does. Style conversion is a global, systematic operation, while refactoring is normally a more specific, programmer-initiated operation.)

—*Generalisation*: productions are added or regular expressions are generalised in the sense of extending the generated language. Generalisation is particularly meaningful during grammar evolution and grammar recovery.

—*Restriction*: the opposite of generalisation.

—*Insertion*: rules are enhanced by inserting extra sub-phrases. For instance, a base-line grammar could be customised as a parse tree format such that inserted sub-phrases cater for position information or comments and layout.

—*Deletion*: the opposite of insertion.

—*Amalgamation*: two ore more rules are merged into a single rule. (This sort of transformation can be viewed as a generalising transformation followed by the elimination of doubles in the rule set.) Amalgamation caters for simplified, problem-specific grammars. A good example of amalgamation can be found in the work on agile parsing [Dean et al. 2002; 2003].

—*Separation*: the opposite of amalgamation.

—*Transformations supporting grammar properties*, e.g., conflict resolution for LALR(1), or disambiguation for generalised LR parsing. Eventually, many of these transformations cannot be described on pure grammars alone, but they rather involve commitment to a richer grammar notation or even to a specific technology (at least, as of today).

A number of systems for language processing have been meanwhile used to support certain forms of automated grammar transformations (in the sense of grammarware engineering); we know of uses of ASF+SDF Meta-Environment, LDL, Popart, Strafunski, Stratego, TXL — as discussed in [Lämmel and Wachsmuth 2001; Lämmel and Verhoef 2001b; Wile 1997; Lämmel and Visser 2003; de Jonge et al. 2001; Dean et al. 2002].

7. A LIST OF RESEARCH CHALLENGES

We have encountered various techniques throughout the agenda, which are indeed very versatile, and which substantiate that we are facing the emergence of an engineering discipline for grammarware. We contend that a proper research effort is needed to study foundations in a systematic manner, and to deliver best practises with a high degree of automation and generality. The required effort should not be underestimated. To give an example, so far, there is no reasonably universal operator suite for grammar transformations despite all reported efforts. Presumably, the toughest challenge is to provide faithful coverage for the many different usage scenarios for these transformations, and to be meaningful to most if not all grammar notations and grammar-based programming setups. This large scale makes us think of a public research agenda as opposed to a short-term project.

The following list entails research issues on foundations, methodology, best practises, tool support and empirical matters. Each item is self-contained, and could serve as a skeleton of a PhD project (except the last one: *miscellaneous*).

7.1 An interoperational web of grammar forms

We have enumerated many different grammar notations. In practise, there exist all kinds of more or less ad-hoc mappings between these notations. For instance, regular operators can be transformed away such that pure BNF notation is sufficient. Also, context-free grammars can be refactored such that the productions correspond immediately to abstract and concrete classes in an object-oriented inheritance hierarchy. Ultimately, we need a comprehensive grammar web, where the side conditions and implications of mapping one notation to the other are described in an operational and pragmatic manner — with reference to details of grammar use cases. Some relevant results can be found in [Koskimies 1991; van der Meulen 1994; de Jonge and Visser 2000; Kort et al. 2002; McLaughlin 2002; Jong and Olivier 2004; Hinze et al. 2004; Herranz and Nogueira 2005]. There exist various theoretical expressiveness results about different grammar forms. These results are relevant and should be exploited, but they must not be confused with practically meaningful mappings between the grammar notations.

7.2 A collection of grammarware properties

What is the complexity of a grammar? What is the grammar-related complexity of grammar-dependent functionality? What are effective notions of grammar equivalence and friends?

What is the distance between two grammars? What are preservation properties, as they can be used to discipline grammar transformations? What is a grammar slice? What is a grammar module? What is the grammar contract that is relied upon in grammar-dependent functionality? What are typical analyses to be performed on grammars? And so on. We presume that the development of a comprehensive framework for grammarware properties can be based on existing work for grammar-flow analysis [Mönck and Wilhelm 1991; Jeuring and Swierstra 1994].

7.3 A framework for grammar transformations

What are suitable primitives? What are the composition principles? What are pre- and post-conditions? How to infer transformations from given grammars? What classes of transformations do exist? How do transformations apply across grammar notation? How to reuse such pure grammar transformations in the context of customisation for grammar use cases? How to support data and grammar integration by grammar transformations? And so on. One should aim at an operator suite that covers the various transformation scenarios including refactoring, disambiguation, normalisation, enhancement and clean-up. The final deliverable can be a domain-specific language for grammar transformation, which is simple to use, and which comes with a dedicated theory for formal reasoning about grammar transformations. Ideally, the transformation language should lend itself to interactive tool support for transformation. Relevant results can be found in [Wile 1997; Pepper 1999; Bernstein and Rahm 2001; Lämmel and Verhoef 2001b; Lämmel 2001a; Lämmel and Wachsmuth 2001; Dean et al. 2002; Erwig 2003].

7.4 Co-evolution of grammar-dependent software

We recall the archetypal example from Sec. 6.4: the co-evolution of an XSLT program in reply to a change of the underlying XML schema. Another example is the co-evolution of a customisation concern for parser tweaking or parse-tree construction in reply to a change of the underlying syntax. There exists related worked on the subject of the joint transformation of grammars and dependent declarative (rule-based) programs [Lämmel 1999b; Lämmel and Riedewald 1999; Lämmel 1999a; Lohmann and Riedewald 2003; Kort and Lämmel 2003a; Lohmann et al. 2004; Lämmel 2004b]. We adopt the term co-evolution from [D'Hondt et al. 2000; Wuyts 2001; Favre 2003], where it was specifically used in the context of joint adaptation of object-oriented designs and implementations. We propose that co-evolution of grammar-dependent software should be approached in a language-parametric manner — as far as the programming language for grammar-dependent functionality is concerned. This sort of genericity is described, to some extent, in [Lämmel 2002; Heering and Lämmel 2004].

7.5 Comprehensive grammarware testing

What are grammar-based coverage criteria? What are means to characterise problem-specific test cases? What techniques are needed to analyse coverage and to generate test data? There exist few coverage criteria for grammars: Purdom's rule coverage [Purdom 1972] for context-free grammars, and refinements thereof [Lämmel and Harm 2001; Lämmel 2001b]. Test-data generation necessitates a string of techniques:

—to deal with the standard oracle problem,

—to minimise test cases that act as symptoms,

—to enforce non-structural constraints,

—to accomplish negative test cases, and

—to achieve scalability for automated testing.

Specific results regarding some of these issues can be found in [Purdom 1972; Celentano et al. 1980; Kastens 1980; Maurer 1990; Burgess 1994; McKeeman 1998; Sirer and Bershad 1999; Harm and Lämmel 2000].

7.6 Parsing technology revisited

Even basic parsing regimes are still subject to ongoing research and defence. What is the ultimate regime? Is it generalised LR-parsing with powerful forms of disambiguation [Klint and Visser 1994; van den Brand et al. 2002]; is it top-down parsing but with idioms for semantics direction [Parr and Quong 1994; Breuer and Bowen 1995]; it is simple LALR(1) parsing with token decoration [Malloy et al. 2003]; is it plain recursive descent parsing with provisions for limiting backtracking [Breuer and Bowen 1995; Kort et al. 2002]? Perhaps, there is no ultimate regime. So then, when to use what regime? How to migrate from one regime to the other? Analysing the engineering aspects of different parsing technologies, and allowing programmers to detach themselves, to some extent, from specific technology is the perfect showcase for grammarware engineering. This showcase really requires best practises and corresponding tool support. Engineering aspects of parser development are largely neglected in the literature, but we refer to [Crawford 1982] for a small but good example, where some engineering guidelines for the construction of LALR grammars are provided.

7.7 Grammar-aware API migration

Consider the following archetypal example. Given is an object-oriented program that access XML data through the simple (generic) Document Object Model (DOM [W3C 2003]). Let us assume that the accessed data is required to always validate against some given XML schema. In that case, static typing of the program could be improved by making use of an XML data binding technology (such as JAXB [Sun Microsystems 2001] in the case of the Java platform). That is, XML access will be based on classes that are generated from the XML schema. The challenge is that API migration is weakly understood in terms of the required code transformations. More generally, the question is: what grammar-based methods can be provided for the support of API migration (potentially also including APIs other than obvious data-access APIs)?

7.8 Modular grammarware development

What advanced means of modular composition can improve reuse of grammars, grammar slices, other grammar fragments, and grammar-dependent functionality? What are generic aspects for grammar-dependent functionality, and what are the means to instantiate them? Modular or even aspect-oriented programming [Kiczales et al. 1997; Elrad et al. 2001] should be fully instantiated for grammarware. Relevant results can be found in [Farrow et al. 1992; van Deursen et al. 1993; Kastens and Waite 1994; Lämmel 1999b; 1999a; de Moor et al. 2000; Malton et al. 2001; Swierstra 2001; Winter 2003; Cordy 2003; Kort and

Lämmel 2003b]. An archetypal scenario is parser development. Achieving an effective modularisation of all the concerns in the following list — on top of mainstream parsing technologies — would be a major step forward in the parsing arena:

—Concrete syntax.

—Abstract syntax.

—Lexical syntax.

—Pre-processing syntax.

—Parse-error recovery.

—Parse-tree construction.

—Semantics-directed parsing.

—Computations for attributed parse-trees.

—Annotation of parse trees with position information.

7.9 Grammarware debugging

It is common practise to debug grammarware just in the same way as any other software — i.e., without actual grammar-awareness. This is not necessarily appropriate. For instance, consider grammar-based programming using visitor techniques in object-oriented programming. Stepping through code for tree walking, one is likely to inspect code that is not related to the problem-specific parts of the traversal. Grammar-aware breakpoints with assorted use-case-specific debug information are needed. There exists related work on visualising the inner workings of compilers [Schmitz 1992], and on debugging models for generic language technology [Olivier 2000]. In addition to debugging grammar-dependent software, there is also a need for debugging grammars, by themselves. For instance, consider the desirable property of a grammar to be unambiguous. While the property is generally undecidable, one can perhaps use static analyses, such as LR(k) conflict analysis for smaller ks, as to obtain indications of sources of ambiguity.

7.10 Adaptive grammarware

In some grammarware development projects, the use of entirely precise grammars is not necessarily the preferred option — from an engineering point of view. Less precise grammars, and more adaptive grammarware might be preferable or even mandatory. For instance, a precise grammar might simply not exist for the use case at hand — as in the case of processing interactive input with transient syntax errors. Even in case a precise grammar is obtainable *in principle*, precision might still be too expensive. Also, over-precision can pose a barrier for evolution of grammarware and for unanticipated variations on grammatical structure. Examples of adaptive techniques are known in parsing [Barnard 1981; Barnard and Holt 1982; Koppler 1997; Moonen 2001; Klusener and Lämmel 2003; Synytskyy et al. 2003]. Clearly, adaptiveness triggers additional concerns such as correctness, as we discuss for parsing in [Klusener and Lämmel 2003]. There is a need for a general methodology for adaptive grammarware.

7.11 Grammar inference put to work

Grammar recovery is an essential phase in the grammarware life cycle. One option for recovery is to extract available traces of grammatical structure, and to issue transformations that lead to a useful grammar [Lämmel and Verhoef 2001b]. An alternative form of grammar recovery can be based on grammar inference. While there is a considerable body of theoretical results on grammar inference of context-free grammars (and other grammars) from data [Mäkinen 1992; Koshiba et al. 2000], there is little experience with applying grammar inference to non-trivial software engineering problems. In particular, known efforts to infer grammars for use in programming-language parsers are quite limited in scale; see, e.g., [Mernik et al. 2003; Javed et al. 2004; Dubey et al. 2005]. For instance, in [Mernik et al. 2003], the syntax of a small domain-specific language is inferred using an evolutionary approach, namely genetic programming. We have not yet seen work that clearly motivates grammar inference from an engineering point of view. How to make sure that the grammar will be meaningful to the grammarware engineer? How to make inference predictable such that similar results are obtained for slightly different inputs? How to take into account informal knowledge about the grammar? How to test the grammar as inference proceeds?

7.12 Reconciliation for meta-grammarware

Consider the following archetypal example, which deals with the evolution of a domain-specific language (DSL [van Deursen et al. 2000]). We assume that the DSL is implemented by the generation of low-level code from high-level DSL code. We assume that the developer can readily customise the generated code, whenever necessary. The evolution of the DSL or alterations of the generator tool make it likely that code has to be regenerated, which poses the following challenge. The newly generated code has to be reconciled with previously customised code. Considering (software) models rather than grammars (or grammarware), such reconciliation issues relate to *round-trip engineering* in model-driven development [Mellor et al. 2003; Selic 2003]. In that case, a platform-independent model (PIM) is transformed into a platform-specific model (PSM) and eventually into code. Any customisation of PSM (or code) would need to be pushed back to the PIM.

7.13 Grammarware life cycling

Processes for typical life-cycle scenarios of recovery, evolution, and customisation need to be defined in detail. This development shall differentiate various grammar notations and grammar use cases. For instance, there will be variations of processes that are specific to document processors vs. language processors. The defined processes are supposed to highlight the potential for automated transformation, quality assessment, and choice points for technology options. This development will eventually add up to a collection of methods, best practises and comprehensive processes that can form the core of an engineering handbook for grammarware.

7.14 Comprehensive grammarware tooling

The future grammarware engineer shall be provided with an environment for *Computer-Aided Grammarware Engineering* (CAGE) — akin to the classic term CASE (Computer-Aided Software Engineering). A CAGE environment should cover interactive and batchmode 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. CAGE tooling needs to be made available in integrated development environments such as Eclipse or Visual Studio. Given the recent surge of model-driven development (MDD), one might add CAGE tooling to MDD environments. For instance, tool support for technology-specific customisation of grammars (as in the parsing context) could be provided as transformation cartridges in the sense of model-driven architecture [OMG 2004].

7.15 Miscellaneous

What are measurable losses caused by grammarware hacking? What are success stories, and what are key factors for success? What is the mid- and long-term perspective for the distribution of different kinds of grammarware? What do organisations know about their grammarware assets? How to enable the creation of such knowledge [Klint and Verhoef 2002]? What are further insights in the grammarware dilemma, and how does this compare to other dilemmas in software engineering? What lessons can be learnt from unsuccessful adoption of grammarware technology? (As a reviewer phrased it: "`lex` and `yacc` are the only tools the world out there has understood; the rest was ignored. Why?".)

8. SUMMARY

We argued that current software engineering practises are insufficiently aware of grammars, which is manifested by an ad-hoc and manual treatment of both — grammars as such and grammatical structure as it occurs in software components. We compiled an agenda that is meant to stimulate research on the engineering aspects of grammarware. We identified promises and principles of the engineering discipline for grammarware.

The promises are increased productivity of grammarware development, improved evolvability and improved robustness of grammarware. The principles are akin to state-of-the-art software engineering. For instance, the principle "implement by customisation" corresponds to a grammarware-tailored instance of model-driven development [Mellor et al. 2003; OMG 2004]; the principle "separate concerns" requires advanced means of modularisation, just as in aspect-oriented programming [Kiczales et al. 1997; Elrad et al. 2001]; the principles "evolve by transformation" and "ensure quality" is well in line with agile methodologies as they are becoming common in today's software engineering.

We called for a major research effort, which is justified by the pervasiveness of grammars 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 grammarware engineering.

Acknowledgement. The three anonymous referees of the TOSEM journal have provided impressive input for the two necessary revisions. We are grateful for collaboration with Steven Klusener and Jan Kort. We are also grateful for discussions with Gerco Ballintijn, Jean Bézivin, Mark van den Brand, Jim Cordy, Jean-Marie Favre, Jan Heering, Hayco de Jong, Wolfgang Lohmann, Erik Meijer, Pablo Nogueira, Bernard Pinon, Günter Riedewald, Wolfram Schulte, Ernst-Jan Verhoeven, Joost Visser, Guido Wachsmuth, and Andreas Winter. We acknowledge opportunities to debug this agenda during related colloquia at the 2nd Workshop Software-Reengineering in Bad Honnef (2000), at the 54th IFIP WG2.1 meeting in Blackheath – London (2000), at the Software Improvement Group in Amsterdam (2000), at the Free University of Amsterdam (2001), at the University of Namur (2001), at the AIRCUBE 2003 workshop hosted by INRIA Lorraine & Loria, at the University of Braga (2003), at the Free University of Brussels (2003), at the University of Kaiserslautern (2003), at the University of Karlsruhe (2003), at the University of Rostock (2003), at the Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development (2004), and at the Microsoft Academic Days in Stockholm (2004).

REFERENCES

ABRAIDO-FANDINO, L. 1987. An overview of Refine 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering, Madrid, Spain*.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers. Principles, Techniques and Tools*. Addison-Wesley.

AHO, A. AND ULLMAN, J. 1972–73. *The theory of parsing, translation, and compiling*. Prentice-Hall, Englewood Cliffs (NJ). Vol. I. Parsing. Vol II. Compiling.

ALBLAS, H. AND MELICHAR, B., Eds. 1991. LNCS, vol. 545. Springer-Verlag.

ARRANGA, E., ARCHBELL, I., BRADLEY, J., COKER, P., LANGER, R., TOWNSEND, C., AND WEATHLEY, M. 2000. In Cobol's Defense. *IEEE Software 17,* 2, 70–72, 75.

ASSMANN, U. AND LUDWIG, A. 1999. Aspect weaving by graph rewriting. In *Proceedings of Generative Component-based Software Engineering (GCSE)*, U. Eisenecker and K. Czarnecki, Eds. LNCS, vol. 1799. Springer-Verlag, 24–36.

AUGUSTON, M. 1990. Programming language RIGAL as a compiler writing tool. *SIGPLAN Notices 25,* 12, 61–69.

AUGUSTON, M. 1995. Program behavior model based on event grammar and its application for debugging automation. In *AADEBUG, 2nd Int'l Workshop on Automated and Algorithmic Debugging*, M. Ducassé, Ed. IRISA-CNRS, 277–291.

BACKUS, J. 1960. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, S. de Picciotto, Ed. Unesco, Paris, 125—131.

BARNARD, D. 1981. Hierarchic Syntax Error Repair. Ph.D. thesis, University of Toronto, Department of Computer Science. 253 pages.

BARNARD, D. AND HOLT, R. 1982. Hierarchic Syntax Error Repair for LR Grammars. *International Journal of Computer and Information Sciences 11,* 4, 231–258.

BATORY, D., SINGHAL, V., THOMAS, J., DASARI, S., GERACI, B., AND SIRKIN, M. 1994. The GenVoca Model of Software Systems Generators. *IEEE Software*, 89–94.

BAXTER, I. D. 1992. Design maintenance systems. *Communications of the ACM 35,* 4 (Apr.), 73–89.

BERGSTRA, J., HEERING, J., AND KLINT, P. 1989. The algebraic specification formalism ASF. In *Algebraic Specification*, J. Bergstra, J. Heering, and P. Klint, Eds. ACM Press Frontier Series. The ACM Press in cooperation with Addison-Wesley, 1–66.

BERNSTEIN, P. A. AND RAHM, E. 2001. On matching schemas automatically. Tech. Rep. MSR-TR-2001-17, Microsoft Research (MSR). Feb.

BLASBAND, D. 2001. Parsing in a hostile world. In *Proceedings, Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society Press, 291–300.

VAN DEN BRAND, M., VAN DEURSEN, A., HEERING, J., JONG, H. D., DE JONGE, M., KUIPERS, T., KLINT, P., MOONEN, L., OLIVIER, P., SCHEERDER, J., VINJU, J., VISSER, E., AND VISSER, J. 2001. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings, Compiler Construction (CC'01)*, R. Wilhelm, Ed. LNCS, vol. 2027. Springer-Verlag, 365–370.

VAN DEN BRAND, M., VAN DEURSEN, A., KLINT, P., KLUSENER, A., AND VAN DER MEULEN, E. 1996. Industrial applications of ASF+SDF. In *41*. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 11.

VAN DEN BRAND, M., JONG, H. D., KLINT, P., AND OLIVIER, P. 2000. Efficient annotated terms. *Software – Practice and Experience 30,* 3 (Mar.), 259–291.

VAN DEN BRAND, M., KLINT, P., AND VERHOEF, V. 1998. Term rewriting for sale. In *Proceedings, Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, C. Kirchner and H. Kirchner, Eds. ENTCS, vol. 15. Elsevier, 139–162. `http://www.elsevier.nl/locate/entcs/volume15.html`.

VAN DEN BRAND, M., SCHEERDER, J., VINJU, J., AND VISSER, E. 2002. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Proceedings, Compiler Construction (CC'02)*, N. Horspool, Ed. LNCS, vol. 2304. Springer-Verlag, 143–158.

VAN DEN BRAND, M., SELLINK, M., AND VERHOEF, C. 1997. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In *Proceedings, 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, M. Sellink, Ed. Electronic Workshops in Computing. Springer-Verlag.

VAN DEN BRAND, M., SELLINK, M., AND VERHOEF, C. 1998. Current parsing techniques in software renovation considered harmful. In *Proceedings, International Workshop on Program Comprehension (IWPC'98)*, S. Tilley and G. Visaggio, Eds. 108–117.

VAN DEN BRAND, M., SELLINK, M., AND VERHOEF, C. 2000. Generation of Components for Software Renovation Factories from Context-free Grammars. *Science of Computer Programming 36,* 2–3, 209–266.

VAN DEN BRAND, M. AND VISSER, E. 1996. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology 5,* 1 (Jan.), 1–41.

BRAZ, L. M. 1990. Visual Syntax Diagrams for Programming Language Statements. In *ACM Eighth International Conference on Systems Documentation*. Visual Issues in Technical Communication. 23–27.

BREUER, P. AND BOWEN, J. 1995. A PREttier Compiler-Compiler: Generating Higher-order Parsers in C. *Software — Practice and Experience 25,* 11 (Nov.), 1263—1297.

BURGESS, C. 1994. The Automated Generation of Test Cases for Compilers. *Software Testing, Verification and Reliability 4,* 2 (jun), 81–99.

BYRD, L. 1980. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, S. Tärnlund, Ed. Debrecen, Hungary, 127–138.

CELENTANO, A., CRESPI-REGHIZZI, S., VIGNA, P. D., GHEZZI, C., GRANATA, G., AND SAVORETTI, F. 1980. Compiler testing using a sentence generator. *Software – Practice and Experience 10,* 11 (Nov.), 897–918.

CHOMSKY, N. 1975. *Reflections on Language*. Pantheon. 269 pages.

COMON, H., DAUCHET, M., GILLERON, R., JACQUEMARD, F., LUGIEZ, D., TISON, S., AND TOMMASI, M. 2003. Tree Automata Techniques and Applications. Book under construction; `http://www.grappa.univ-lille3.fr/tata/`.

CONSEL, C., LAWALL, J. L., AND MEUR, A.-F. L. 2004. A tour of tempo: a program specializer for the C language. *Science of Computer Programming 52,* 1–3, 341–370.

CORDY, J. 2003. Generalized Selective XML Markup of Source Code Using Agile Parsing. In *Proceedings, International Workshop on Program Comprehension (IWPC'03)*. IEEE Computer Society Press, 144–153.

CORDY, J., DEAN, T., MALTON, A., AND SCHNEIDER, K. 2002. Source Transformation in Software Engineering Using the TXL Transformation System. *Journal of Information and Software Technology 44/13,* 827–837. Special issue on source code analysis and manipulation.

CORDY, J., SCHNEIDER, K., DEAN, T., AND MALTON, A. 2001. HSML: Design Directed Source Code Hot Spots. In *Proceedings of International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society Press.

CRAWFORD, J. 1982. Engineering a production code generator. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. ACM Press, 205–215.

CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, F. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

DEAN, T., CORDY, J., MALTON, A., AND SCHNEIDER, K. 2002. Grammar Programming in TXL. In *Proceedings, Source Code Analysis and Manipulation (SCAM'02)*. IEEE Computer Society Press.

DEAN, T., CORDY, J., MALTON, A., AND SCHNEIDER, K. 2003. Agile Parsing in TXL. *Journal of Automated Software Engineering 10,* 4 (Oct.), 311–336.

DEKLEVA, S. M. 1992. The influence of the information systems development approach on maintenance. *MIS Quarterly 16,* 3 (Sept.), 355–372.

DERANSART, P. AND MALUSZYŃSKI, J. 1993. *A Grammatical View of Logic Programming*. The MIT Press.

VAN DEURSEN, A., HEERING, J., AND KLINT, P. 1996. *Language Prototyping*. AMAST Series in Computing, vol. 5. World Scientific.

VAN DEURSEN, A., KLINT, P., AND TIP, F. 1993. Origin Tracking. *Journal of Symbolic Computation 15,* 523–545.

VAN DEURSEN, A., KLINT, P., AND VISSER, J. 2000. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notice 35,* 6 (June), 26–36.

D'HONDT, T., DE VOLDER, K., MENS, K., AND WUYTS, R. 2000. Co-evolution of Object-Oriented Software Design and Implementation. In *Proceedings of the International Symposium on Software Architectures and Component Technology 2000*.

DIJKSTRA, E. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N. J., Chapter 14.

DUBEY, A., AGGARWAL, S., AND JALOTE, P. 2005. A Technique for Extracting Keyword Based Rules from a Set of Programs. In *Proceedings, Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE Computer Society Press, 217–225.

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

EHRIG, H., KREOWSKI, H., MONTANARI, U., AND ROZENBERG, G. 1996. *Handbook of Graph Grammars and Computing by Graph Transformation, Volumes 1–3*. World Scientific.

EISENECKER, U. AND CZARNECKI, K. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect-oriented programming: Introduction. *CACM 44,* 10 (Oct.), 29–32. An introduction to the CACM special issue on AOP.

EMMELMANN, H., SCHRÖER, F.-W., AND LANDWEHR, R. 1989. BEG - A generator for efficient back ends. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (SIGPLAN '89)*, B. Knobe, Ed. ACM Press, Portland, OR, USA, 227–237.

ERWIG, M. 2003. Towards the Automatic Derivation of XML Transformations. In *Workshop on XML Schema and Data Management (XSDM)*. Workshop affiliated with ER2003.

FARROW, R., MARLOWE, T., AND YELLIN, D. 1992. Composable Attribute Grammars. In *Proceedings of 19th ACM Symposium on Principles of Programming Languages (Albuquerque, NM)*. 223–234.

FAVRE, J. 1996. Preprocessors from an abstract point of view. In *Proceedings, International Conference on Software Maintenance*. IEEE Computer Society Press, Washington, 329–339.

FAVRE, J.-M. 2003. Meta-models and Models Co-Evolution in the 3D Software Space. In *Proceedings of International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA 2003)*.

FAVRE, J.-M. 2004. Towards a Basic Theory to Model Model Driven Engineering. In *Workshop on Software Model Engineering, WISME 2004, joint event with UML2004 Lisboa, Portugal, October 11, 2004*. Paper available from `http://www.metamodel.com/wisme-2004/`.

FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems 9,* 3 (July), 319–349.

FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. 1992. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems 1,* 3 (Sept.), 213–226.

GARTNER RESEARCH. 2003. Legacy Modernization Provides Applications For Tomorrow. Research Note, Markets M-19-3671, D. Vecchio.

GAY, S., VASCONCELOS, V., AND RAVARA, A. 2003. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow. Mar.

GOGOLLA, M. AND KOLLMANN, R. 2000. Re-Documentation of Java with UML Class Diagrams. In *Proceedings, 7th Reengineering Forum, Reengineering Week 2000 Zürich*, E. Chikofsky, Ed. Reengineering Forum, Burlington, Massachusetts, 41–48.

GOGOLLA, M. AND LINDOW, A. 2003. Transforming Data Models with UML. In *Knowledge Transformation for the Semantic Web*, B. Omelayenko and M. Klein, Eds. IOS Press, Amsterdam, 18–33.

GOLIN, E. 1991. A method for the specification and parsing of visual languages. Ph.D. thesis, Brown University.

GRAY, R., HEURING, V., LEVI, S., SLOANE, A., AND WAITE, W. 1992. Eli: A complete, flexible compiler construction system. *Communications of the ACM 35,* 2 (Feb.), 121–130.

GRISWOLD, W. 2002. Teaching Software Engineering in a Compiler Project Course. *Journal on Educational Resources in Computing 2,* 4, 1–18.

GRISWOLD, W. G. AND NOTKIN, D. 1993. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol. 2,* 3, 228–269.

GROSCH, J. 1992. Transformation of Attributed Trees Using Pattern Matching. In *Compiler Construction, 4th International Conference on Compiler Construction*, U. Kastens and P. Pfahler, Eds. LNCS, vol. 641. Springer-Verlag, Paderborn, Germany, 1–15.

GROSCH, J. AND EMMELMANN, H. 1991. A Tool Box for Compiler Construction. In *Proceedings, Compiler Compilers, Third International Workshop on Compiler Construction, 22–26 October, 1990*, D. Hammer, Ed. LNCS, vol. 477. Springer, Schwerin, Germany, 106–116.

HAINAUT, J.-L., TONNEAU, C., JORIS, M., AND CHANDELON, M. 1993. Schema Transformation Techniques for Database Reverse Engineering. In *Proceedings, 12th Int. Conf. on ER Approach*. E/R Institute, Arlington-Dallas.

HARM, J. AND LÄMMEL, R. 2000. Two-dimensional Approximation Coverage. *Informatica 24,* 3, 355–369.

HEERING, J., HENDRIKS, P., KLINT, P., AND REKERS, J. 1989. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices 24,* 11, 43–75.

HEERING, J. AND LÄMMEL, R. 2004. Generic Software Transformations (Extended Abstract). In *Presented at Software Transformation Systems Workshop, Satellite Event at OOPSLA'04*. Paper available from the authors' web sites.

HENRAD, J., HICK, J.-M., THIRAN, P., AND HAINAUT, J.-L. 2002. Strategies for Data Reengineering. In *Proceedings, Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society Press, 211–220.

HERRANZ, A. AND NOGUEIRA, P. 2005. More Than Parsing. V Jornadas Sobre Programación y Lenguajes, Conferencia Española de Informática (CEDI'05), Granada, Spain, Thomson-Paraninfo. To appear.

HERRIOT, R. G. 1976. Structured syntax diagrams. *Computer Languages 2,* 1-2, 9–19.

HEURING, V., KASTENS, U., PLUMMER, R., AND WAITE, W. 1989. COMAR: a data format for integration of CFG tools. *The Computer Journal 32,* 5 (Oct.), 445–452.

HINZE, R., JEURING, J., AND LÖH, A. 2004. Type-indexed data types. *Science of Computer Programming 51,* 1-2, 117–151.

HOFFMANN, B. 1982. Modelling Compiler Generation by Graph Grammars. In *Graph Grammars and their Application to Computer Science*, H. Ehrig, M. Nagl, and G. Rozenberg, Eds. LNCS, vol. 153. Springer-Verlag, 159–171.

HOLT, R., WINTER, A., AND SCHÜRR, A. 2000. GXL: Toward a Standard Exchange Format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. IEEE Computer Society Press, 162–171.

HOLT, R. C., CORDY, J. R., AND WORTMAN, D. B. 1982. An Introduction to S/SL: Syntax/Semantic Language. *ACM Trans. Program. Lang. Syst. 4,* 2, 149–178.

HUGHES, J. 1995. The Design of a Pretty-printing Library. In *Summer school proceedings; Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds. LNCS, vol. 925. Springer Verlag, 53–96.

HUTTON, G. AND MEIJER, E. 1998. Functional pearl: Monadic parsing in Haskell. *Journal of Functional Programming 8,* 4 (July), 437–444.

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

IBM RESEARCH. 2002. XML Transformation: Matching & Reconciliation. available at `http://www.research.ibm.com/hyperspace/mr/`.

ISO. 1996. ISO/IEC 14977:1996(E), Information technology — Syntactic metalanguage — Extended BNF. International Organization for Standardization.

JARZABEK, S. 1995. From Reuse Library Experiences to Application Generation Architectures. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability*, M. Samadzadeh and M. Zand, Eds. 114–122.

JAVED, F., BRYANT, B., CREPINEK, M., MERNIK, M., AND SPRAGUE, A. 2004. Context-free grammar induction using genetic programming. In *ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference*. ACM Press, 404–405.

JEURING, J. AND SWIERSTRA, D. 1994. Bottom-up grammar analysis—A functional formulation. In *Proceedings, European Symposium on Programming Languages and Systems (ESOP'94)*, D. Sannella, Ed. LNCS, vol. 788. Springer-Verlag, 317–332.

JOHNSON, S. 1975. YACC - Yet Another Compiler-Compiler. Tech. Rep. Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey.

JONES, N., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

JONG, H. AND OLIVIER, P. A. 2004. Generation of abstract programming interfaces from syntax definitions. *J. Log. Algebr. Program. 59*, 1-2, 35–61.

DE JONGE, M. 2002. Pretty-printing for software reengineering. In *Proceedings, International Conference on Software Maintenance (ICSM'02)*. IEEE Computer Society Press, 550–559.

DE JONGE, M. AND MONAJEMI, R. 2001. Cost-effective maintenance tools for proprietary languages. In *Proceedings, International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society Press, 240–249.

DE JONGE, M., VISSER, E., AND VISSER, J. 2001. XT: a bundle of program transformation tools. In *Proceedings, Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, M. van den Brand and D. Parigot, Eds. ENTCS, vol. 44. Elsevier Science Publishers.

DE JONGE, M. AND VISSER, J. 2000. Grammars as Contracts. In *Proceedings, Generative and Component-based Software Engineering (GCSE'00)*. LNCS, vol. 2177. Springer-Verlag, Erfurt, Germany, 85–99.

KADHIM, B. AND WAITE, W. 1996. Maptool—supporting modular syntax development. In *Proceedings, Compiler Construction (CC'96)*, T. Gyimothy, Ed. LNCS, vol. 1060. Springer, 268–280.

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

KASTENS, U. AND WAITE, W. 1994. Modularity and reusability in attribute grammars. *Acta Informatica 31*, 601–627.

KICZALES, G., DES RIVIERES, J., AND BOBROW, D. 1991. *The Art of the Metaobject Protocol*. MIT Press.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings, European Conference on Object-Oriented Programming (ECOOP'97)*, M. Aksit and S. Matsuoka, Eds. LNCS, vol. 1241. Springer-Verlag, 220–242.

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

KLINT, P. AND VERHOEF, C. 2002. Enabling the creation of knowledge about software assets. *Data & Knowledge Engineering 41*, 2–3 (June), 141–158.

KLINT, P. AND VISSER, E. 1994. Using filters for the disambiguation of contextfree grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*. 1–20. Tech. Rep. 126-1994 Dipartimento di Scienze dell'Informazione, Universita di Milano.

KLUSENER, A. AND LÄMMEL, R. 2003. Deriving tolerant grammars from a base-line grammar. In *Proceedings, International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society Press, 179–189.

KLUSENER, S., LÄMMEL, R., AND VERHOEF, C. 2005. Architectural Modifications to Deployed Software. *Science of Computer Programming 54*, 143–211.

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

KOPPLER, R. 1997. A systematic approach to fuzzy parsing. *Software Practice and Experience 27*, 6, 637–649.

KORT, J. AND LÄMMEL, R. 2003a. A Framework for Datatype Transformation. In *Proceedings Language, Descriptions, Tools, and Applications (LDTA'03)*, B. Bryant and J. Saraiva, Eds. ENTCS, vol. 82. Elsevier. 20 pages.

KORT, J. AND LÄMMEL, R. 2003b. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proceedings, Source Code Analysis and Manipulation (SCAM'03)*. IEEE Computer Society Press, Amsterdam, 161–172.

KORT, J., LÄMMEL, R., AND VERHOEF, C. 2002. The Grammar Deployment Kit. In *Proceedings, Language Descriptions, Tools, and Applications (LDTA'02)*, M. van den Brand and R. Lämmel, Eds. ENTCS, vol. 65. Elsevier Science. 7 pages.

KOSCHKE, R. AND GIRARD, J.-F. 1998. An intermediate representation for reverse engineering analyses. In *Proceedings, Working Conference on Reverse Engineering (WCRE'98)*. IEEE Computer Society Press, 241–250.

KOSHIBA, T., MÄKINEN, E., AND TAKADA, Y. 2000. Inferring pure context-free languages from positive data. *Acta Cybernetica 14*, 469–477.

KOSKIMIES, K. 1991. Object Orientation in Attribute Grammars. See Alblas and Melichar [1991], 297–329.

KUHN, T. 1970. *The Structure of Scientific Revolutions, 2nd edition*. University of Chicago Press, Chicago, Illinois. 1st edition appeared in 1962.

KUIPER, M. AND SARAIVA, J. 1998. Lrc — A generator for Incremental Language-Oriented Tools. In *Compiler Construction CC'98*, K. Koskimies, Ed. LNCS, vol. 1383. Springer-Verlag, 298–301. Tool demonstration.

KURTEV, I., BÉZIVIN, J., AND AKSIT, M. 2002. Technological Spaces: An Initial Appraisal. In *Proceedings, Confederated International Conferences CoopIS, DOA, and ODBASE 2002, Industrial track, Irvine, CA, USA*. `http://www.sciences.univ-nantes.fr/lina/atl/www/papers/PositionPaperKurtev.pdf`.

LÄMMEL, R. 1999a. Declarative Aspect-Oriented Programming. In *Proceedings, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), San Antonio (Texas), BRICS Notes Series NS-99-1*, O. Danvy, Ed. 131–146.

LÄMMEL, R. 1999b. Functional meta-programs towards reusability in the declarative paradigm. Ph.D. thesis, Universität Rostock, Fachbereich Informatik. Published by Shaker Verlag in 1998, ISBN 3-8265-6042-6.

LÄMMEL, R. 2001a. Grammar Adaptation. In *Proceedings, Formal Methods Europe (FME) 2001*, J. Oliveira and P. Zave, Eds. LNCS, vol. 2021. Springer-Verlag, 550–570.

LÄMMEL, R. 2001b. Grammar Testing. In *Proceedings, Fundamental Approaches to Software Engineering (FASE'01)*, H. Hussmann, Ed. LNCS, vol. 2029. Springer-Verlag, 201–216.

LÄMMEL, R. 2002. Towards Generic Refactoring. In *Proceedings of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*. ACM Press, Pittsburgh, USA. 14 pages.

LÄMMEL, R. 2004a. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*.

LÄMMEL, R. 2004b. Evolution of Rule-Based Programs. *Journal of Logic and Algebraic Programming 60–61C*, 141–193. Special Issue on Structural Operational Semantics.

LÄMMEL, R. 2005. The Amsterdam toolkit for language archaeology. In *Post-proceedings of the 2nd International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering (ATEM 2004)*. ENTCS. Elsevier Science.

LÄMMEL, R. AND HARM, J. 2001. Test case characterisation by regular path expressions. In *Proceedings, Formal Approaches to Testing of Software (FATES'01)*, E. Brinksma and J. Tretmans, Eds. Notes Series NS-01-4. BRICS, 109–124.

LÄMMEL, R. AND LOHMANN, W. 2001. Format Evolution. In *Proceedings, Re-Technologies for Information Systems (RETIS'01)*, J. Kouloumdjian, H. Mayr, and A. Erkollar, Eds. books@ocg.at, vol. 155. OCG, 113–134.

LÄMMEL, R. AND RIEDEWALD, G. 1999. Reconstruction of paradigm shifts. In *Second Workshop on Attribute Grammars and their Applications (WAGA 1999)*. 37–56. INRIA, ISBN 2-7261-1138-6.

LÄMMEL, R. AND VERHOEF, C. 2001a. Cracking the 500-Language Problem. *IEEE Software*, 78–88.

LÄMMEL, R. AND VERHOEF, C. 2001b. Semi-automatic Grammar Recovery. *Software—Practice & Experience 31*, 15 (December), 1395–1438.

LÄMMEL, R. AND VISSER, J. 2002. Typed Combinators for Generic Traversal. In *Proceedings, Practical Aspects of Declarative Programming (PADL'02)*, S. Krishnamurthi and C. Ramakrishnan, Eds. LNCS, vol. 2257. Springer-Verlag, 137–154.

LÄMMEL, R. AND VISSER, J. 2003. A Strafunski Application Letter. In *Proceedings of Practical Aspects of Declarative Programming (PADL'03)*, V. Dahl and P. Wadler, Eds. LNCS, vol. 2562. Springer-Verlag, 357–375.

LÄMMEL, R. AND WACHSMUTH, G. 2001. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In *Proceedings, Language Descriptions, Tools and Applications (LDTA'01)*, M. van den Brand and D. Parigot, Eds. ENTCS, vol. 44. Elsevier Science.

LIEBERHERR, K. J. 1988. Object-Oriented Programming with Class Dictionaries. *Lisp and Symbolic Computation 1,* 2 (Sept.), 185–212.

LIND, J. 2002. Specifying Agent Interaction Protocols with Standard UML. In *Proceedings, Agent-Oriented Software Engineering (AOSE'01)*, M. Wooldridge and G. Weiß and P. Ciancarini, Ed. LNCS, vol. 2222. Springer-Verlag, 136–145.

LOHMANN, W. AND RIEDEWALD, G. 2003. Towards automatical migration of transformation rules after grammar extension. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*. IEEE Computer Society Press, 30–39.

LOHMANN, W., RIEDEWALD, G., AND STOY, M. 2004. Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars. In *Proceedings of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*. ENTCS, vol. 110. Elsevier Science, 133–148.

MÄKINEN, E. 1992. On the structural grammatical inference problem for context-free grammars. *Information Processing Letters 42,* 1, 1–5.

MALLOY, B., POWER, J., AND WALDRON, J. 2002. 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*. 75–82. In cooperation with ACM Press.

MALLOY, B. A., GIBBS, T. H., AND POWER, J. F. 2003. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software—Practice & Experience 33,* 1, 1395–1438.

MALTON, A., SCHNEIDER, K., CORDY, J., DEAN, T., COUSINEAU, D., AND REYNOLDS, J. 2001. Processing software source text in automated design recovery and transformation. In *Proceedings, International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society Press.

MAMAS, E. AND KONTOGIANNIS, K. 2000. Towards portable source code representations using XML. In *Proceedings, Working Conference on Reverse Engineering (WCRE'00)*. IEEE Computer Society Press, 172–182.

MARLOW, S. 2002. Haddock: A haskell documentation tool. `http://haskell.cs.yale.edu/haddock/`.

MARRIOTT, K. AND MEYER, B. 1998. *Visual Language Theory*. Springer-Verlag.

MAURER, P. 1990. Generating test data with enhanced context-free grammars. *IEEE Software 7,* 4, 50–56.

MCCLURE, M. 1989. TEX Macros for COBOL syntax diagrams. *TUGboat 10,* 4 (Dec.), 743–750.

MCKEEMAN, W. 1998. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation 10,* 1, 100–107.

MCLAUGHLIN, B. 2002. *Java and XML data binding*. Nutshell handbook. O'Reilly & Associates, Inc.

MELLOR, S. J., CLARK, A. N., AND FUTAGAMI, T. 2003. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 14–18. Special Issue on Model-Driven Development.

MERNIK, M., CREPINSEK, M., GERLIC, G., ZUMER, V., BRYANT, B., , AND SPRAGUE, A. 2003. Learning context-free grammars using an evolutionary approach. Tech. rep. Technical report, University of Maribor and The University of Alabama at Birmingham.

MERNIK, M., GERLIC, G., ZUMER, V., AND BRYANT, B. 2003. Can a parser be generated from examples? In *SAC 2003: Proceedings of the 2003 ACM Symposium on Applied Computing*. ACM Press, 1063–1067.

MERNIK, M., ČREPINŠEK, M., KOSAR, T., REBERNAK, D., AND ŽUMER, V. 2004. Grammar-based systems: Definition and examples. *Informatica 28,* 3 (Nov.), 245–255.

METAMODEL.COM. 2003–2005. Web portal for meta-modelling. `http://www.metamodel.com/`.

VAN DER MEULEN, E. 1994. Incremental rewriting. Ph.D. thesis, University of Amsterdam.

MÖNCK, U. AND WILHELM, R. 1991. Grammar Flow Analysis. See Alblas and Melichar [1991], 151–186.

MOONEN, L. 2001. Generating Robust Parsers using Island Grammars. In *Proceedings, Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society Press, 13–22.

DE MOOR, O., PEYTON-JONES, S., AND VAN WYK, E. 2000. Aspect-Oriented Compilers. In *Generative and Component-Based Software Engineering: First International Symposium, GCSE'99, Erfurt, Germany, September 1999. Revised Papers*, K. Czarnecki and U. Eisenecker, Eds. LNCS, vol. 1799. Springer-Verlag, 121–134.

MOREAU, P.-E., RINGEISSEN, C., AND VITTEK, M. 2003. A pattern matching compiler for multiple target languages. In *12th Conference on Compiler Construction*. LNCS, vol. 2622. Springer-Verlag, 61–76.

NAGL, M. 1980. Graph Rewriting and Automatic, Machine-Independent Program Optimization. In *Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science*, H. Noltemeier, Ed. LNCS, vol. 100. Springer-Verlag, 55–69.

NAGL, M. 1985. Graph technology applied to a software project. In *The Book of L*, G. Rozenberg and A. Salomaa, Eds. Springer-Verlag, 303–322.

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

ODELL, J., VAN DYKE PARUNAK, H., AND BAUER, B. 2001. Representing Agent Interaction Protocols in UML. In *Proceedings, Agent-Oriented Software Engineering (AOSE'00)*, P. Ciancarini and M. Wooldridge, Eds. Springer-Verlag, 121–140.

OLIVIER, P. 2000. A Framework for Debugging Heterogeneous Applications. Ph.D. thesis, Universiteit van Amsterdam. 182 pages.

OMG. 2001–2004. Model Driven Architecture. web portal `http://www.omg.org/mda/`.

OPDYKE, W. 1992. Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign.

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

PARR, T. AND QUONG, R. 1994. Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k). In *Proceedings, Compiler Construction (CC'94)*, P. Fritzon, Ed. LNCS, vol. 786. Springer-Verlag, 263–277.

PEPPER, P. 1999. LR Parsing = Grammar Transformation + LL Parsing. Tech. Rep. CS-99-05, TU Berlin. Apr.

PEREIRA, F. C. N. AND WARREN, D. H. D. 1980. Definite clause grammars for language analysis. In *Readings in Natural Language Processing*, K. S.-J. B. J. Grosz and B. L. Webber, Eds. Morgan Kaufmann, Los Altos, 101–124.

PROGRES GROUP. 2004. Progres — An integrated environment and very high level language for PROgrammed Graph REwriting Systems, web portal. `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/`, a similar site is under construction at the University of Darmstadt.

PURDOM, P. 1972. A sentence generator for testing parsers. *BIT 12*, 3, 366–375.

PURTILO, J. AND CALLAHAN, J. 1989. Parse tree annotations. *Communications of the ACM 32*, 12, 1467–1477.

REPS, T. AND TEITELBAUM, T. 1984. The synthesizer generator. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, 42–48.

SCHMITZ, L. 1992. The Visual Compiler-Compiler SIC (abstract). In *Addendum to the Proceedings of the Conference on Object-Oriented Programming, Systems, Lanuages, and Applications*. 236.

SCHRÖER, F. 1997. *The GENTLE Compiler Construction System*. R. Oldenbourg. 142 pages.

SCHÜRR, A. 1990. Introduction to PROGRESS, an attribute graph grammar based specification language. In *Proceedings of the fifteenth international workshop on Graph-theoretic concepts in computer science*. Springer-Verlag New York, Inc., 151–165.

SCHÜRR, A. 1994. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop*, E. W. Mayr, G. Schmidt, and G. Tinhofer, Eds. LNCS, vol. 903. Springer-Verlag, Herrsching, Germany, 151–163.

SCHÜRR, A. 1997. Developing Graphical (Software Engineering) Tools with PROGRES. In *Proceedings of the 19th International Conference on Software Engineering*. ACM Press, 618–619.

SELIC, B. 2003. The Pragmatics of Model-Driven Development. *IEEE Software*, 19–25. Special Issue on Model-Driven Development.

SELLINK, M. AND VERHOEF, C. 2000a. Development, Assessment, and Reengineering of Language Descriptions. In *Proceedings, Conference on Software Maintenance and Reengineering (CSMR'00)*. IEEE Computer Society Press, 151–160.

SELLINK, M. AND VERHOEF, C. 2000b. Scaffolding for Software Renovation. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR'00)*, J. Ebert and C. Verhoef, Eds. IEEE Computer Society Press, 161–172.

SENDALL, S. AND KOZACZYNSKI, W. 2003. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 42–51. Special Issue on Model-Driven Development.

SIM, S. 2000. Next generation data interchange: Tool-to-tool application program interfaces. In *Proceedings, Working Conference on Reverse Engineering (WCRE'00)*. IEEE Computer Society Press, 278–283.

SIM, S. AND KOSCHKE, R. 2001. WoSEF: Workshop on Standard Exchange Format. *ACM SIGSOFT Software Engineering Notes 26*, 44–49.

SIRER, E. AND BERSHAD, B. 1999. Using Production Grammars in Software Testing. In *Proceedings, Domain-Specific Languages (DSL'99)*, USENIX, Ed. USENIX, 1–13.

SMITH, D., KOTIK, G., AND WESTFOLD, S. 1985. Research on Knowlegde-Based Software Environments at Kestrel Institute. *IEEE Transactions on Software Engineering SE-11,* 11, 1278–1295.

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

SPINELLIS, D. 2003. Global Analysis and Transformations in Preprocessed Languages. *IEEE Transactions on Software Engineering 29,* 11, 1019–1030.

SUN MICROSYSTEMS. 2001. The Java architecture for XML binding (JAXB). `http://java.sun.com/xml/jaxb`.

SUN MICROSYSTEMS. 2002. Java 2 Platform, Standard Edition (J2SE). Javadoc Tool Home Page; `http://java.sun.com/j2se/javadoc/`.

SWIERSTRA, S. 2001. Parser Combinators, from Toys to Tools. In *Proceedings, 2000 ACM SIGPLAN Haskell Workshop*, G. Hutton, Ed. ENTCS, vol. 41. Elsevier Science.

SYNYTSKYY, N., CORDY, J., AND DEAN, T. 2003. Robust Multilingual Parsing Using Island Grammars. In *Proceedings CASCON'03, 13th IBM Centres for Advanced Studies Conference, Toronto*. 149–161.

THIBAULT, S. AND CONSEL, C. 1997. A Framework for Application Generator Design. *ACM SIGSOFT Software Engineering Notes 22,* 3 (May), 131–135.

VALLECILLO, A., VASCONCELOS, V., AND RAVARA, A. 2003. Typing the behavior of objects and components using session types. *ENTCS 68,* 3. Presented at FOCLASA'02.

VEERMAN, N. 2005. Towards lightweight checks for mass maintenance transformations. *Science of Computer Programming*. To appear.

VISSER, E. 1997. Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam.

VISSER, E. 2001a. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)*, A. Middeldorp, Ed. Lecture Notes in Computer Science, vol. 2051. Springer-Verlag, 357–361.

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

W3C. 1997–2003. Document Object Model (DOM). `http://www.w3.org/DOM/`.

W3C. 2000–2003. XML Schema. `http://www.w3.org/XML/Schema`.

W3C. 2004. Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation. `http://www.w3.org/TR/2004/REC-xml-20040204/`.

WALLACE, M. AND RUNCIMAN, C. 1999. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices 34,* 9 (Sept.), 148–159. Proceedings, International Conference on Functional Programming (ICFP'99).

WANG, D., APPEL, A., KORN, J., AND SERRA, C. 1997. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*. USENIX Association, Berkeley, 213–228.

WARD, M. 1999. Assembler to C migration using the FermaT transformation system. In *Proceedings, IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 67–76.

WILE, D. 1997. Abstract syntax from concrete syntax. In *Proceedings, International Conference on Software Engineering (ICSE'97)*. ACM Press, 472–480.

WILHELM, R. AND MAURER, D. 1995. *Compiler Design*. Addison Wesley.

WINTER, A. 2003. Referenzschemata im Reverse Engineering. *Softwaretechnik Trends der GI 23,* 2. 2 pages.

WUYTS, R. 2001. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. Ph.D. thesis, Vrije Universiteit Brussel.