An Introduction to Grammar Convergence

Ralf Lämmel and Vadim Zaytsev

Software Languages Team, The University of Koblenz-Landau, Germany

Abstract. Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in all kinds of software artifacts, e.g., object models, XML schemas, parser descriptions, or language documents. The central idea is to extract grammars from diverse software artifacts, and to transform the grammars until they become syntactically identical. The present paper introduces and illustrates the basics of grammar convergence.

1 Introduction

Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in all kinds of software artifacts. In fact, it is an integrated method that works purposely across different programming and specification languages as well as different approaches to software development. Here are few use cases for grammar convergence:

– Given are Java classes for a specific domain, say financial exchange. There is also an independently designed XML schema that is meant to standardize that domain. One needs to establish the agreement between the object model and the schema.
– Given is a compiler for a programming language, say gcc for C++. There is also a reverse/re- engineering tool for the same language based on a different parsing infrastructure. One needs to establish that both tools agree on the language at hand.
– Given is an XML-data binding technology. One needs to test the (customizable) mapping from XML schemas to object models. The oracle for testing relies on establishing an agreement between XML schemas and object models.
– Given are 3 versions of the Java language specification, with 2 grammars per version. One needs to align grammars per version and express the evolution from version to version. (We have done such a case study; see the authors' website.)

The central idea of grammar convergence is to extract grammars from diverse software artifacts, and to transform the grammars until they become syntactically identical. In more detail, the method entails the following core ingredients:

1. A unified *grammar format* that effectively supports abstraction from specialities or idiosyncrasies of the grammars as they occur in software artifacts in practice.
2. A *grammar extractor* for each kind of artifact – e.g., a Java extractor maps Java classes to the unified grammar format.
3. A *grammar comparer* that determines and reports grammar differences in the sense of deviations from syntactical equality (if any).

4. A framework for automated *grammar transformation* that can be used to refactor, or to otherwise more liberally edit grammars until they become syntactically identical.

The method also entails the following optional ingredients:

5. Grammar convergence may be extended to the 'instance level' so that instances (such as parse trees or XML documents) are also extracted, compared and transformed.
6. The transformations of grammar convergence may be semi-automatically derived ('inferred') from grammar differences.

The present paper only covers the core ingredients 1.-4.

Contributions

- Grammar convergence helps in relating grammar knowledge that is readily ingrained in diverse forms of software artifacts; it complements the use of generative (or model-driven) approaches, when they are not used, have not been used, or cannot (yet) be used.
- Grammar convergence delivers conceptually simple grammar transformations to software artifacts of kinds that would normally require more complicated transformations, e.g., XML schemas, and object models. This is possible because of the abstraction done during extraction.
- An implementation of grammar convergence is publicly available.[1]

Roadmap §2 describes the basics of grammar convergence and introduces the running example of the paper. §3 outlines BGF — the BNF-like Grammar Format, i.e., the unified grammar format that we use in our implementation of grammar convergence. §4 describes and illustrates the concept of grammar extraction. §5 sketches our suite of programmable transformations for grammar convergence. §6 discusses related work. §7 concludes the paper.

2 Basics and running example

We use a trivial programming language FL ('Factorial Language'; available from the quoted repository) as a running example. That is, we converge grammars for FL that were obtained from different FL language processors such as interpreters and optimizers. Here is an illustrative program in the FL language; it defines two functions: one for multiplication; another for the factorial function; the latter in terms of the former:

```
mult n m = if (n == 0) then 0 else (m + (mult (n - 1) m))
fac n = if (n == 0) then 1 else (mult n (fac (n - 1)))
```

[1] https://sourceforge.net/projects/slps/

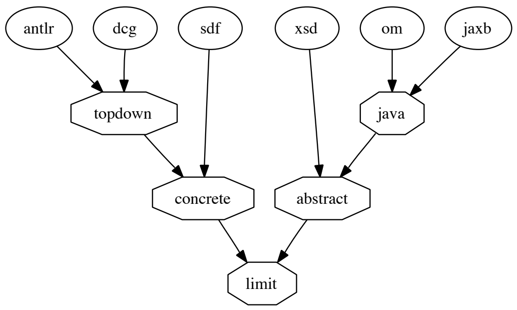

Fig. 1. The overall convergence graph for the 'Factorial Language'.

```
program(Fs) −−> +(function,Fs).
 function (N,Ns,E)) −−> name(N), +(name,Ns), @("="), expr(E), +(newline).

expr(E)                        −−> lassoc(ops,atom,binary,E).
expr(apply(N,Es))              −−> name(N), +(atom,Es).
expr(ifThenElse(E1,E2,E3))     −−> reserved("if"), expr(E1), ...

atom( literal (I))     −−> int(I).
atom(argument(N))      −−> name(N).
atom(E)                −−> @("("), expr(E), @(")").

ops(equal)   −−> @("==").
ops(plus)    −−> @("+").
ops(minus)   −−> @("−").
```

Fig. 2. Definite Clause Grammar for FL. (The clauses construct a term representation; see the arguments of the various predicates. The DCG leverages higher-order predicates for EBNF-like expressiveness and left-associative tree construction (c.f., '+' and 'lassoc'). The priorities on expression forms are expressed by means of a layered definition; c.f., 'expr' vs. 'atom'.)

```
Function+                            → Program
Name Name+ "=" Expr Newline+         → Function
Expr Ops Expr                        → Expr     { left , prefer , cons(binary)}
Name Expr+                           → Expr     {avoid,cons(apply)}
"if" Expr "then" Expr "else" Expr    → Expr     {cons(ifThenElse)}
"(" Expr ")"                         → Expr     {bracket}
Name                                 → Expr     {cons(argument)}
 Int                                 → Expr     {cons( literal )}
"−"                                  → Ops      {cons(minus)}
"+"                                  → Ops      {cons(plus)}
"=="                                 → Ops      {cons(equal)}
```

Fig. 3. SDF grammar for FL. (Only (context-free) SDF productions are shown. Notice that the 'defining expression' of a production appears on the left side of the arrow, and the 'defined nonterminal' on the right side. Productions can be annotated in certain ways between the braces, e.g., with constructor names (c.f., cons), or directions for disambiguation (c.f., prefer, avoid).)

2.1 Sources of convergence

Fig. 1 shows a convergence tree for some FL components. The *leafs* of the tree (see at the top) denote different *sources*. We use the term source to mean 'software artifact containing grammar knowledge'. Here is short description of the sources for FL:

antlr This is a parser description in the input language of ANTLR[2]. Semantic actions (in Java) are intertwined with EBNF-like productions.
dcg This is a logic program written in the style of definite clause grammars; c.f. Fig. 2.
sdf This is a concrete syntax definition in the notation of SDF/SGLR[3] with scannerless generalized LR parsing as parsing model; c.f. Fig. 3.
xsd This is an XML schema[4] for the abstract syntax of FL.
om This is a hand-crafted object model (Java classes) for the abstract syntax of FL.
jaxb This object model was generated by JAXB[5] from an XML schema for FL.

2.2 Targets of convergence

Consider again Fig. 1. The *inner nodes and the root* denote a number of *targets* for FL. We use the term target to mean 'derived grammars that establish the correspondence between some sources'. Here is short description of the targets for FL:

topdown The sources *antlr* and *dcg* both leverage top-down parsing. Their correspondence can be established by a few simple refactoring steps.
concrete This target converges all concrete syntax definitions. A noteworthy difference is that *sdf* uses one expression nonterminal, whereas *topdown* uses two 'layers'.
java The sources *om* and *jaxb* are both object models whose correspondence can be established by simple refactoring steps.
abstract The target *java* and the XML schema are to be converged to an abstract syntax definition. The corresponding refactorings need to neutralize the style differences implied by the data models: OO vs. XML.
limit The targets *concrete* and *abstract* are converged (to an even more abstract syntax). For instance, terminals are removed from *concrete*.

3 BGF — BNF-like Grammar Format

3.1 Design rationale

In principle, we could try to leverage an existing syntax definition formalism (e.g., SDF/SGLR (see an earlier footnote) or a meta-modeling facility (e.g., EMF[6]). In contrast, we have derived BGF such that it covers the grammar-like expressiveness that we encountered in different kinds of software artifacts. Also, BGF allows us to avoid any sort of bias towards a particular parsing model or other details of operational semantics. For convenience, we can still represent BGF in other notations (using a generative approach as in [13]).

2 `http://antlr.org`
3 `http://www.program-transformation.org/Sdf/SGLR`
4 `http://www.w3.org/XML/Schema`
5 `http://jaxb.dev.java.net/`
6 `http://www.eclipse.org/modeling/emf/`

3.2 BGF concepts

We start with the most trivial aspects:

– Terminals and nonterminals.
– Regular expression-like composition and grouping:
  • Sequential composition (infix ‘,’ – also called ‘sequence’).
  • Alternative composition (infix ‘;’ – also called ‘choice’).
  • Epsilon (’true’) and the ‘empty language’.
  • Iteration and optionality (‘*’, ‘+’, ‘?’).
– A production is a pair of ‘defined nonterminal’ and ‘defining expression’.
– A grammar consists of a set of start symbols and a set of productions.

At this point, we have reached ‘representation capability’ for textbook-style BNF and EBNF (when restricted to context-free syntax). Only few more concepts are needed to represent essential extras of XML schemas, object models, and algebraic signatures:

Production labels Extraction may populate these labels from names of OO subclasses, derived XML schema types, or algebraic term constructors. As a bonus, labels are convenient in addressing productions in programmable grammar transformations.

Expression selectors While a flat record-like grammar structure with top-level selectors is sufficient to represent typical object models, more liberal selectors are needed to represent arbitrarily nested element declarations of XML Schema.

Simple types Types such as string and int are added to cover the simple types used in algebraic data types, object models, and XML schemas. Even syntax definitions indirectly involve simple types through the attributes associated with lexemes.

Universal type This type is a fallback for extraction whenever no precise grammar structure can be determined, e.g., when mapping the OO base type ‘object’, wildcards of XML Schema, or dynamics in functional programming.

Namespaces Various kinds of sources are organized in namespaces, c.f., Java’s packages for object models, Haskell’s hierarchical module system, or XML Schema’s foundation on XML namespaces. Such organization can be preserved by extraction.

3.3 Self-representation

In our implementation, BGF is primarily defined by an XML Schema, which naturally, is too voluminous to be shown here. While the XML-based representation of BGF grammar may be convenient for data exchange, several of our components use a Prolog-based term notation. (For instance, the transformation component is implemented in Prolog.) Fig. 4 lists the BGF of BGF in the Prolog-based term notation.[7] The interesting status of the shown grammar is that it has been computed from the BGF that was extracted from the primary XML schema for BGF. That is, we have applied grammar convergence to align the XML schema with the expected Prolog-based term notation for BGF.

[7] The notation uses predominantly prefix terms with the exception of special list notation and infix functors ‘,’ and ‘;’ for sequences and choices. Note that optionality is represented via lists — using the empty list [] for the case of absence, and the singleton list otherwise. Certain symbols need to be escaped by quotes or parentheses, as one can see in the figure.

```
g( [g,p,x,v,l,n,s,t], [
  p([g], g, (*(n(n)), *(n(p)))),          -- grammar = start symbols + productions
  p([p], p, (?(n(l)), n(n), n(x))),       -- production = label + LHS + RHS
  p([true], x, true),                     -- epsilon
  p([fail], x, true),                     -- empty language
  p([v], x, n(v)),                        -- values of simple types
  p([a], x, true),                        -- all (universal type)
  p([t], x, n(t)),                        -- terminals
  p([n], x, n(n)),                        -- nonterminals
  p([s], x, (n(s), n(x))),                -- selector expressions
  p([(',')], x, (n(x), n(x))),            -- sequence
  p([(;)], x, (n(x), n(x))),              -- choice
  p([?], x, n(x)),                        -- optionality
  p([+], x, n(x)),                        -- 1 or more repetitions
  p([*], x, n(x)),                        -- 0,1 or more repetitions
  p([int], v, true),                   -- integer values
  p([string], v, true),                -- strings
  p([], l, v(string)),              -- labels are strings
  p([], n, v(string)),              -- nonterminal symbols are strings
  p([], s, v(string)),              -- selectors are strings
  p([], t, v(string))               -- terminal symbols are strings
])
```

Fig. 4. BGF of BGF w/o namespaces.

4 Grammar extraction

4.1 Abstraction by extraction

The limited expressiveness of BGF, when compared to any possible source format, implies that some of the details of the source format are not conveyed into the extracted grammar; we call this effect 'abstraction by extraction'. Such abstraction simplifies proofs of grammar correspondences at the cost of potentially missing certain kinds of grammar differences. Here are examples of details that are abstracted away in this manner; they are grouped by kinds of grammarware:

Parser descriptions
- Semantic actions
- Lexical syntax descriptions
- Precedence declarations

Object models
- Constructors, static methods, initializers
- Specific types of collection classes
- Distinction of classes vs. interfaces and fields vs. methods

Algebraic data types
- Distinction of nominal types vs. types aliases
- Higher-order and quantified types (represented universally)

XML schemas

- Distinction of elements, attributes, complex types, and groups
- Simple type constraints

4.2 Grammar extractors

An extractor is simply a software component that processes a software artifact and produces a (BGF) grammar. In the typical case, extraction boils down to a straightforward mapping defined by a single pass over the input. Extractors are preferably implemented within the computational framework of the source artifact at hand, or in its affinity, e.g.:

- ANTLR: ANTLR
- DCG: Prolog
- Java: java.lang.reflect or com.sun.source.tree
- SDF: ASF+SDF Meta-Environment[8] or Stratego/XT[9]

On the output side, an extractor leverages the XML format for BGF.

4.3 Extraction samples

Fig. 5 contrasts the extraction results for several FL sources. The differences between the grammars can be summarized as follows:

- Only the ANTLR&DCG&SDF extracts contain terminals.
- The ANTLR&DCG extracts contain expressions layers expr and atom.
- The SDF&XSD extracts contain a single expression layer.
- Only the XSD extract contains selectors.
- The ANTLR&XSD extracts leverage choices.
- The DCG&SDF extracts leverage nonterminals with multiple productions.
- There is also some variation on using production labels.
- More trivially, the grammars disagree on names, upper and lower case.

5 Programmable grammar transformations

In the more preferable case, two different grammars can be refactored to become syntactically identical. We use the term *refactoring* in the established sense of semantics-preserving transformations. In the less preferable case, non-semantics-preserving transformations are due, in which case weaker properties should limit the impact.

[8] `http://www.meta-environment.org/`
[9] `http://strategoxt.org/`

BGF extracted from the SDF for FL as of Fig. 3

```
p([binary], 'Expr', (n('Expr'), n('Ops'), n('Expr'))),
p([apply], 'Expr', (n('Name'), +n('Expr'))),
p([ifThenElse], 'Expr', (t(if), n('Expr'), t(then), n('Expr'), t(else), n('Expr'))),
p([], 'Expr', (t('('), n('Expr'), t(')'))),
p([argument], 'Expr', n('Name')),
p([ literal ], 'Expr', n('Int')),
p([minus], 'Ops', t(−)),
p([plus], 'Ops', t(+)),
p([equal], 'Ops', t(==))
```

BGF extracted from the DCG for FL as of Fig. 2

```
p([binary], expr, (n(atom), ∗((n(ops), n(atom))))),
p([apply], expr, (n(name), +n(atom))),
p([ifThenElse], expr, (t(if), n(expr), t(then), n(expr), t(else), n(expr))),
p([ literal ], atom, n(int)),
p([argument], atom, n(name)),
p([], atom, (t('('), n(expr), t(')'))),
p([equal], ops, t(==)),
p([plus], ops, t(+)),
p([minus], ops, t(−))
```

BGF extracted from an ANTLR frontend for FL

```
p([], expr, (n(binary);n(apply);n(ifThenElse))),
p([], binary, (n(atom), ∗((n(ops), n(atom))))),
p([], apply, (n('ID'), +n(atom))),
p([], ifThenElse, (t(if), n(expr), t(then), n(expr), t(else), n(expr))),
p([], atom, (n('ID');n('INT');t('('), n(expr), t(')'))),
p([], ops, (t(==);t(+);t(−)))
```

BGF extracted from an XML schema for FL

```
p([], 'Function', (s(name, v(string)), +s(arg, v(string)), s(rhs, n('Expr')))),
p([], 'Expr', (n('Literal');n('Argument');n('Binary');n('IfThenElse');n('Apply'))),
p([], 'Literal', s(info, v(int))),
p([], 'Argument', s(name, v(string))),
p([], 'Binary', (s(ops, n('Ops')), s(left, n('Expr')), s(right, n('Expr')))),
p([], 'Ops', (s('Equal', true);s('Plus', true);s('Minus', true))),
p([], 'IfThenElse', (s(ifExpr, n('Expr')), s(thenExpr, n('Expr')), s(elseExpr, n('Expr')))),
p([], 'Apply', (s(name, v(string)), +s(arg, n('Expr'))))
```

Fig. 5. Some extraction results for FL. (Only expression syntax is shown.)

5.1 Transformation properties

We may refer to the semantics of a grammar as the language (set of strings) generated by the grammar, as it is common for formal languages — for context-free grammars, in particular. With the string-oriented semantics in mind, few transformations are semantics-preserving. Examples include renaming of nonterminals, and fold/unfold manipulations. To give an example where different semantics are needed consider the scenario of aligning a concrete and an abstract syntax.

When necessary, we may apply the algebraic interpretation of a grammar, where grammar productions constitute an algebraic signature subject to a term-algebraic model. In this case, the terminal occurrences in any given production do no longer carry semantic meaning; they are part of the function symbol. (Hence, abstract and concrete syntaxes can be aligned now.) Some transformations that were effortlessly semantics-preserving w.r.t. the string-oriented semantics, require designated bijective mappings w.r.t. the term-oriented semantics, e.g., fold/unfold manipulations, but generally, the term-oriented semantics admits a larger class of semantics-preserving transformations than the string-oriented one.

For brevity, we omit the discussion of another alternative: graph-oriented semantics.

Transformations that are not semantics-preserving may still be 'reasonable' if they model data refinement [8,25].[10] A simple way to think of data refinement in our context is that a transformation increases or decreases the number of 'representational options', e.g., by making a certain syntactic structure optional or mandatory. Here we assume the term-oriented semantics with its term-algebraically defined domains.

Some grammar differences may require more arbitrary replacements. In this case, one would want to be sure that *a)* indeed no more preserving transformation is possible, and *b)* the scope of replacement is as small as possible. To this end, we have developed an effective strategy, which however is beyond the scope of the present paper.

5.2 Grammar refactoring

Let us demonstrate a number of refactoring operators. In our running example, there are two sources that are very close to each other: *antlr* and *dcg*; c.f., Fig. 5. Both sources serve top-down parsing. The remaining differences are neutralized by the following refactorings to be applied to the ANTLR grammar; we show the applications of the transformation operators combined with an explanatory comment:

```
renameN('NEWLINE', newline)                    % use lower case
renameN('INT', int)                            % use lower case
renameN('ID', name)                            % rename ID to name
verticalN(expr)                                % many expr productions
unchain(p([], expr, n(apply)))                 % inline apply production
unchain(p([], expr, n(binary)))                % inline binary production
unchain(p([], expr, n(ifThenElse)))            % inline ifThenElse
verticalN(atom)                                % many atom productions
deanonymize(p([literal], atom, n(int)))        % add label for literals
deanonymize(p([argument], atom, n(name)))      % add label for arg refs
verticalN(ops)                                 % many ops productions
deanonymize(p([equal], ops, t(==)))            % label == production with equal
deanonymize(p([plus], ops, t(+)))              % label + production with plus
deanonymize(p([minus], ops, t(−)))             % label − production with minus
```

Fig. 6 briefly describes a small suite of refactoring operators. All operators except ***permute*** are semantics-preserving w.r.t. string-oriented semantics. Without exception, the operators are semantics-preserving w.r.t. term-oriented semantics.

5.3 Grammar editing

We use the term *grammar editing* for transformations that go beyond refactoring. Let us consider an example. The *antlr* and *dcg* sources of FL use two expression layers

[10] We say that a data type (domain) A can be refined to a data type (domain) B, denoted by the inequality $A \leq B$, if there is an injective, total function $to : A \rightarrow B$ (the representation function), and a surjective, possibly partial function $from : B \rightarrow A$ (the abstraction function) such that $from.to = id_A$, where id_A is the identity function on A.

renameN(N_1, N_2) renames all occurrences of the nonterminal N_1 to N_2, provided N_2 does not occur in G. There are also operators ***renameL*** and ***renameS*** for renaming labels and selectors. In ***renameS***(OL, S_1, S_2), OL is an optional label; if present, S_1 is renamed only in the scope of the identified production, or globally otherwise.

permute(P) replaces a production say P' by P, where P and P' must agree on their defined nonterminal and (optional) label while their defining expressions must be permutations of each other (with regard to sequential composition). Here is an example:
- A production: **p**([binary], expr, (**n**(expr), **n**(ops), **n**(expr)))
- A permutation: **p**([binary], expr, (**n**(ops), **n**(expr), **n**(expr)))

verticalN(N) converts the choice-based definition of N to multiple productions. Each alternative of the choice becomes another production. An outermost selector, if present, is reused as a production label (but must not yet be in use in G). The variation ***verticalP***(P) limits the conversion to a production P. There is the opposite operator ***horizontal***.

unchain(P) replaces a chain production P and the production P' that defines the nonterminal of its defining expression by a production that inlines P' in P. (There is also the opposite operator ***chain***.) Here is an example:
- The chain production: **p** ([], expr , **n**(literal))
- The referenced definition: **p** ([], literal , **n**(**int**))
- The result of unchaining: **p**([literal], expr , **n**(**int**))

deanonymize(P) replaces an unlabeled production say P' by its labeled variant P. There is also the opposite operator ***anonymize***.

lassoc(P) replaces list-based recursion by binary recursion. (The 'l' in ***lassoc*** is for *left* association hinting at the expected effect at the instance level. There is also an operator ***rassoc*** hence.) Here, P describes binary recursion. Their must be a corresponding production in G that uses list-based recursion. Here is an example:
- Binary recursion: **p**([binary], expr, (**n**(expr), **n**(ops), **n**(expr)))
- List-based recursion: **p**([binary], expr, (**n**(expr), $*$((**n**(ops), **n**(expr)))))

Fig. 6. Operators for grammar refactoring. (G refers to the input grammar.)

project(P) replaces a production say P' by P, where P and P' must agree on their defined nonterminal and (optional) label, and the defining expression of P must be a sub-sequence of the one of P' (with regard to sequential composition).

stripTs removes all terminals.

stripSs removes all selectors.

skip(P) removes a reflexive chain production P.

unite(N_1, N_2) recursively merges the definitions of N_1 and N_2 into one by replacing all defining and using occurrences of N_1 by N_2.

define(Ps) adds the productions Ps as a definition, assuming that all productions agree on a defined nonterminal that is used but not yet defined in G. We take the view that an undefined nonterminal is implicitly defined to be equal to the universal type. Hence, the ***define*** operator essentially 'narrows' a definition in a semantic sense. There is also the opposite operator ***undefine*** for discarding the explicit definition of a nonterminal.

Fig. 7. Operators for grammar editing. (G refers to the input grammar.)

(expr and atom), whereas the *sdf* source only uses one expression layer (and deals with priorities by extra annotations). The following transformation uses an editing operator ***unite*** to merge the two layers (i.e., nonterminals) in one:

unite (atom, expr)

Consider another example. The grammars in Fig. 5 differ with regard to the grammatical details regarding FL's literals and function or argument names. The *xsd* source uses precise (simple) types int and string, whereas the other grammars leave the corresponding nonterminals undefined (because the extraction only returned immediate context-free structure in those cases). The following transformations resolve the undefined nonterminals in accordance to the *xsd* source:

```
define ([p ([], name, v( string ))])     % names are strings
define ([p ([], int , v( int ))])        % ints ( literals ) are ints
```

Consider a final example. The convergence of concrete and abstract syntax definitions requires a transformation that removes all details that are specific to concrete syntax definitions. That is, we project away the reference to newline, strip off all terminals, remove the bracketing production, and permute the ingredients of binary expressions to resemble prefix instead of infix notation. Thus:

```
project (p ([], function , (v( string ), +v( string ), t (=), n(expr ))))
stripTs
skip (p ([], expr, n(expr )))
permute(p([binary ], expr, ’,’([n(ops ), n(expr ), n(expr )])))
```

Fig. 7 briefly describes a small suite of editing operators. In fact, the editing operators ***stripTs*** and ***stripSs*** are semantics-preserving w.r.t. the term-oriented semantics because terminals and selectors are irrelevant for interpreting a grammar as a signature. All but one of the remaining operators model data refinement in one direction or the other, i.e., from input (I) to output (O), or vice versa: ***skip***: $O \leq I$, ***unite***: $I \leq O$, ***define***: $O \leq I$, ***undefine***: $I \leq O$. The operator ***project*** does not model data refinement; rather it models 'data disposal'. Its I-to-O mapping for ***project*** is total, surjective, non-injective; its O-to-I mapping is not generally defined.

6 Related work

Interoperability The consistent use of structural and nominal types (to be compared here with grammar knowledge) is a goal shared with programming-language type systems, exchange formats, and interface definition languages (IDLs). IDLs are specifically used in distributed programming. Exchange formats are widely used for any sort of data- and communication-intensive programming. A domain with classic grammar-like exchange formats including bridges between different formats is reverse engineering [9, 14]. In the broad context of interoperability, *grammar convergence provides added value in the situation where diverse, related grammar-like knowledge is ingrained in different software artifacts*. The use of extraction and transformation compensates for the lack of consistent use of a common type system, IDL, or exchange format, and it allows for flexible correspondence relationships.

Testing grammarware The I/O behavior of grammarware (e.g., the acceptor behavior of a frontend) can be tested by 'sampling' — subject to test-data generation and test suites [16, 19, 23, 31]. Such approaches are specifically useful for differential testing of grammarware. *Grammar convergence is complementary in that it provides a static verification of the correspondence between different software artifacts based on access to the internal structure of the artifacts.* It can also be applied to specify the 'distance' between grammars.

Generators and synchronizers If two artifacts are meant to use the same grammar (type, etc.) modulo its realization in the software artifact, then, arguably one grammar (or software artifact) should be generated from the other. One scenario of that kind is XML-object mapping where object models are derived from XML schemas or vice versa [18]. Another scenario is the provision of text-to-model and model-to-text capabilities in model-driven engineering, where, for example, a parser description may be generated from a sufficiently rich ('annotated') metamodel [11].

One may go beyond generation, and even require bidirectional synchronization between scattered grammar knowledge, akin to bidirectional model/model or model/code synchronization in model-driven engineering [32]. As should be clear from the list of use cases in the introduction, *grammar convergence is applicable even when generators or bidirectional synchronizers are not, have not been, or cannot (yet) be used for whatever technical or other reason*. In particular, existing components do not need to be adapted, in any way, when applying the method of grammar convergence.

As an illustration, let us consider two concrete scenarios. First, consider the problem of different versions of a highly idiosyncratic parser description [27]. Bidirectional synchronization is not available in this context, but grammar convergence applies, and establishes the correspondence between the grammar versions. Second, consider the derivation of a technology-specific parser description from a technology-neutral baseline grammar. Only simpler cases of this process can be automated [13, 11].

Grammar recovery Our work is heavily influenced by the idea of grammar recovery [2, 6, 10, 17, 20, 30], especially those forms that begin with the extraction of grammar knowledge from an artifact like a standard (containing syntax) or an implementation (based on an idiosyncratic parsing technology). Just like grammar convergence, grammar recovery involves (manual or automated) grammar transformations, which we discuss below. While grammar recovery has focused on (mostly concrete) syntax definitions, grammar convergence applies to a very broad interpretation of grammars (XML schemas, object models, etc.). Grammar recovery is a reverse-engineering method that relies on conservative parser testing to derive a quality grammar from the source. In contrast, *grammar convergence is a verification method that establishes and maintains grammatical correspondences between software artifacts*.

Grammar transformation (Automated) grammar transformation has seen a surge of interest over the last decade, but the concept is much older because parsing technologies tend to require some internal transformations, c.f., the classic example of left-recursion removal [1,22,24]. There are several modern use cases for grammar transformation that support automated software engineering and grammar-based programming in one way or another: grammar recovery (see above), derivation of an abstract from a concrete syntax [33], problem-specific customization of grammars [4], and mediation between different grammar classes [28]. Ultimately, we speak of grammar programming or programmable grammar transformations [3].

Grammar convergence relies on an advanced operator suite for grammar transformation the design of which is driven by the unified grammar format, and the kinds of grammar differences that we have encountered. The design of the operator suite has not yet fully stabilized; we are still pursuing research on principled properties of grammar transformations — at all levels: single operators, operator suites, and sequences of operator applications. This work is based on earlier research by the first author [15,21].

Grammar convergence Finally, we mention grammar engineering techniques that can be seen as specific forms of grammar convergence. In [2], the compatibility of (different implementations of) precedence rules in grammars is checked. Our (current) grammar convergence approach does not address any parsing techniques specifically, but in return, it is more generic (with regard to the notion of grammar), and programmable (with regard to deltas between grammars). In [12], the correspondence between a concrete and abstract syntax definition is addressed: the specifications for both syntaxes may be incomplete, as long as they complement each other consistently. Grammar convergence provides a general tool for 'programming' such relationships, and verifying them. In [27], the problem of proliferation of grammar-based artifacts (in fact, parser descriptions with semantic actions) due to grammar evolution or always new grammar use cases is addressed. Based on ideas of version control, a parser description remains associated with its 'prototype', so that revisions of the prototype can be signaled to derivatives. Grammar convergence also covers this scenario, except that it cannot detect modifications that are gone after grammar extraction.

7 Concluding remarks

If unit testing is the simple, pragmatic, and effective method to generally validate the I/O behavior of software modules, then grammar convergence is the simple, pragmatic, and effective method to keep scattered grammar knowledge 'in sync'. The method can also be used to capture and henceforth verify the differences between scattered grammar knowledge — both intended differences (due to evolution or implementational choices) and accidental differences (that cannot be resolved immediately). In one case study (see the authors' website), we have applied the method to the various grammars in the Java language specification; we have accurately captured the evolution from version to version, and we have spotted a substantial number of inconsistencies.

We currently work on the application of grammar convergence at the instance level (c.f., XML trees, derivation trees, parse trees, etc.) so that one can compare and converge 'data' from different software artifacts. Our implementation already supports some operators at the instance level so that instances of one grammar can be converted to instances of another grammar.

Our current implementation of grammar convergence does not infer transformation candidates in any way; the software engineer must use the output of grammar comparison intelligently. This is an obvious target for future work, and we expect useful input from other areas of software engineering: schema matching and data integration in the field of data modeling and databases [29]; comparison of UML models or metamodels in the context of model-driven engineering [34,7]; the computation of refactorings from different OO program versions [26,5].

Acknowledgement The first author is grateful for opportunities to present and discuss some of this work at University of Waterloo, IEEE Kitchener-Waterloo, on August 6, 2008, at Dagstuhl Seminar 08331, 'Perspectives Workshop: Model Engineering of Complex Systems (MECS)', on August 12, 2008, and at invited talks at the METRIK Workshop, in Berlin, on November 21, 2008, and at the BENEVOL Workshop, in Eindhoven, on December 12, 2008.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. E. Bouwers, M. Bravenboer, and E. Visser. Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. *ENTCS*, 203(2):85–101, 2008.
3. T. Dean, J. Cordy, A. Malton, and K. Schneider. Grammar Programming in TXL. In *Proceedings of Source Code Analysis and Manipulation (SCAM'02)*. IEEE, 2002.
4. T. Dean, J. Cordy, A. Malton, and K. Schneider. Agile Parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, 2003.
5. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceeding of 20th European Conference (ECOOP 2006)*, volume 4067 of *LNCS*, pages 404–428. Springer, 2006.
6. E. B. Duffy and B. A. Malloy. An Automated Approach to Grammar Recovery for a Dialect of the C++ Language. In *Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 11–20. IEEE, 2007.
7. J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *Proceedings of Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *LNCS*, pages 326–340. Springer, 2008.
8. C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.
9. D. Jin, J. Cordy, and T. Dean. Where's the Schema? A Taxonomy of Patterns for Software Exchange. In *Proceedings of International Workshop on Program Comprehension (IWPC'02)*, pages 65–74. IEEE, 2002.
10. M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings of International Conference on Software Maintenance (ICSM'01)*, pages 240–249. IEEE, 2001.
11. F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of Generative programming and component engineering (GPCE 2006)*, pages 249–254. ACM, 2006.
12. B. Kadhim and W. Waite. Maptool—supporting modular syntax development. In T. Gyimothy, editor, *Proceedings of Compiler Construction (CC'96)*, volume 1060 of *LNCS*, pages 268–280. Springer, 1996.
13. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. *ENTCS*, 65(3):7 pages, 2002. Proceedings of Language Descriptions, Tools, and Applications (LDTA'02).
14. N. A. Kraft, B. A. Malloy, and J. F. Power. An infrastructure to support interoperability in reverse engineering. *Information & Software Technology*, 49(3):292–307, 2007.
15. R. Lämmel. Grammar Adaptation. In J. Oliveira and P. Zave, editors, *Proceedings of Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.
16. R. Lämmel. Grammar Testing. In *Proceedings of Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of *LNCS*, pages 201–216. Springer, 2001.

17. R. Lämmel. The Amsterdam toolkit for language archaeology. *ENTCS*, 137(3):43–55, 2005. Post-proceedings of the 2nd International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering (ATEM 2004).
18. R. Lämmel and E. Meijer. Revealing the X/O Impedance Mismatch. In *Datatype-Generic Programming, International Spring School, SSDGP 2006, Nottingham, UK, Revised Lectures*, volume 4719 of *LNCS*, pages 285–368. Springer, 2006.
19. R. Lämmel and W. Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Proceedings of 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems, TestCom 2006*, volume 3964 of *LNCS*, pages 19–38. Springer, 2006.
20. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, 2001.
21. R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. *ENTCS*, 44(2), 2001. Proceedings of Language Descriptions, Tools and Applications (LDTA'01).
22. W. Lohmann, G. Riedewald, and M. Stoy. Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars. *ENTCS*, 110:133–148, 2004. Proceedings of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004).
23. B. Malloy, J. Power, and J. Waldron. Applying software engineering techniques to parser design: the development of a C# parser. In *Proceedings of Conference of the South African institute of computer scientists and information technologists*, pages 75–82, 2002. In cooperation with ACM Press.
24. R. C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 249–255. Morgan Kaufmann Publishers Inc., 2000.
25. C. Morgan. *Programming from Specifications*. Prentice Hall International, 1990.
26. M. O'Keeffe and M. O. Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution*, 20(5):345–364, 2008.
27. T. Parr. The Reuse of Grammars with Embedded Semantic Actions. In *Proceedings of the 16th IEEE Conference on Program Comprehension (ICPC 2008)*, pages 5–10. IEEE, 2008.
28. P. Pepper. LR Parsing = Grammar Transformation + LL Parsing. Technical Report CS-99-05, TU Berlin, 1999.
29. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
30. M. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 151–160. IEEE, 2000.
31. E. Sirer and B. Bershad. Using Production Grammars in Software Testing. In USENIX, editor, *Proceedings of Domain-Specific Languages (DSL 1999)*, pages 1–13. USENIX, 1999.
32. P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *Proceedings of Model Driven Engineering Languages and Systems (MoDELS 2007)*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.
33. D. Wile. Abstract syntax from concrete syntax. In *Proceedings of International Conference on Software Engineering (ICSE'97)*, pages 472–480. ACM Press, 1997.
34. Z. Xing and E. Stroulia. Refactoring Detection based on UMLDiff Change-Facts Queries. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 263–274, Washington, DC, USA, 2006. IEEE.