Recovering Grammar Relationships for the Java Language Specification

Ralf Lämmel and Vadim Zaytsev
Software Languages Team
The University of Koblenz-Landau
Germany

Abstract—**We describe a completed effort to recover the relationships between all the grammars that occur in the different versions of the Java Language Specification (JLS). The relationships are represented as grammar transformations that capture all accidental or intended differences between the JLS grammars. This process is mechanized and it is driven by simple measures of nominal or structural differences between any pair of grammars involved. Our work suggests a form of consistency management for the JLS in particular, and language specifications in general.**

I. Introduction

Many software languages (and programming languages in particular) are described simultaneously by multiple grammars that reside in different software artifacts. For instance, one grammar may reside in a language specification; another grammar may be encoded in a parser specification. Many software languages are also subject to evolution, which means that artifacts with embedded grammars may also occur in different versions. *This diversity of grammars for any single software language represents a fundamental consistency challenge.*

Grammars (and hence grammar-dependent artifacts) may actually disagree on the software language in question in a hard-to-spot manner. Also, the intended, evolution-related differences between two grammars may be obfuscated by other more accidental or superficial differences between the grammars. While such disagreement and obfuscation are certainly not desirable, best practices of grammarware engineering cannot rule them out. This is where the present paper makes a contribution.

In earlier work, we have begun to address the fundamental problem of grammar diversity by initiating a method for *grammar convergence* [1]; this method combines grammar extraction (to obtain raw grammars from artifacts and represent them uniformly), grammar comparison (to determine nominal and structural differences between given grammars), and grammar transformation (to represent the relationships between given grammars by transformations that make the grammars structurally equal).

In the present paper, we report on a completed, major case study for grammar convergence, and we refine the method to provide better scalability and reproducibility.

The case concerns the 3 different versions of the Java Language Specification (JLS; [2], [3], [4]) where each of the 3 versions contains 2 grammars — one grammar is optimized for readability (c.f., *read1–read3* in Fig. 1), and another one is intended to serve as a basis for implementation (c.f., *impl1–impl3* in Fig. 1).

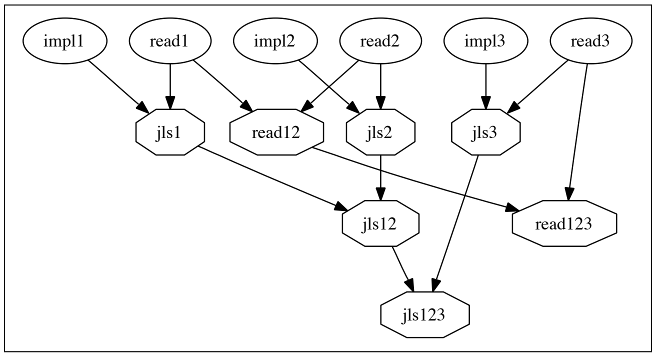

Figure 1. *Binary convergence tree for the JLS grammars. (*The *nodes* in the figure are grammars where the leaves correspond to the original JLS grammars and the other nodes are derived. The *directed edges* denote grammar transformation chains. We use a (cascaded) binary tree here, i.e., each non-leaf node is derived from two grammars.)

Here we note that the JLS is critical to the Java platform — it is a foundation for compilers, code generators, pretty-printers, IDEs, code analysis and transformation tools and other grammarware for the Java language. The JLS is the authoritative specification of Java. One may assume that Sun Microsystems (i.e., the owner of the specification) is interested in an unambiguous, consistent and understandable set of JLS documents.

One would expect that the different grammars per version are essentially equivalent in terms of the generated language. (As a concession to practicality, i.e., implementability in particular, one grammar may be more permissive than the other.) One would also expect that the grammars for the different versions engage in an inclusion ordering (again, in terms of the generated languages) because of the backwards-compatible evolution of the Java language.

Those expected relationships of (liberal) equivalence and inclusion ordering are significantly violated by the JLS grammars, as our case study shows.

It is instructive to wonder why the importance of JLS combined with the assumed scrutiny that went into its preparation still let inconsistencies go unnoticed. We can think of two reasons. First, we note that language equivalence and inclusion is not amenable to any straightforward check; in fact, it is undecidable for context-free grammars. Second, grammar design and evolution is a manual process in practice: grammar engineers design and evolve grammars, as they see fit. They may use simple tools to check the grammar for basic well-formedness or grammar-class compliance. They may also test a parser derived from the grammar. However, such measures cannot guarantee the expected properties for (relaxed) language equivalence and inclusion; oversights happen all too easily (as our case study shows). Our (refined) method of grammar convergence addresses both challenges.

The motivation of our work and its significance is not limited to the mere discovery of bugs in the Java standard or in any other set of grammars for that matter. (In fact, some JLS bugs have been discovered, time and again, by means of informal grammar inspection or other brute-force methods.[1])

The significance of our work is amplified by two arguments. First, we provide a simple and mechanized process that is guaranteed to reveal accidental or intended differences between grammars. Second, we are able to represent the differences in a precise, operational and accessible manner — by means of grammar transformations. In different terms, we are practically able to prove (or disprove) the equivalence of two given grammars. Our method allows for grammar diversity without the drawbacks of disagreement and obfuscation. Different grammars can be effectively related to each other.

Contributions

1) We have recovered nontrivial relationships between grammars of industrial size. (That is, we show that the grammars are equivalent modulo well-defined transformations.) If grammar convergence is compared to related work on recovering, customizing, correcting, completing, restructuring, deriving, and inferring grammars, then it can be said to be original in that *two* or more grammars are given and need to be related to each other — as opposed to any process that starts from a single grammar.
2) We have implemented a mechanized and measurable and reproducible process for grammar convergence. Compared to our initial work [1], the process consists of well-defined phases and progress can be effectively tracked in terms of nominal and structural differences between the grammars at hand.
3) We have worked out a comprehensive operator suite for grammar transformation that substantially extends our previous work on the subject.
4) The JLS case study is publicly available.[2]

Road-map

- §II sketches the corpus of JLS grammars.
- §III describes an approach to grammar transformation.
- §IV describes a (refined) convergence process.
- §V provides a post-mortem for the JLS case.
- §VI discusses related work.
- §VII concludes the paper.

II. Background: the JLS corpus

We recall that each version of the JLS provides a grammar that is optimized for readability (c.f., *read1–read3* in Fig. 1), and another one that is intended to serve as a basis for implementation (c.f., *impl1–impl3* in Fig. 1). We also refer to these grammars as being "more readable" or "more implementable". These notions are not strongly defined, but one can think, for example, of *left factoring* (to help with look ahead) as being used in the more implementable grammars but not in the more readable grammars.

A. Grammar extraction

A JLS document is basically a structured text document with embedded grammar sections. In fact, the more readable grammar is developed throughout the document where the more implementable grammar is given, *en bloc*, in a late section — a de-facto appendix.

The JLS is available electronically in HTML and PDF format. Neither of these formats was designed with convenient access to the grammars in mind. We have opted for the HTML format here.[3] The grammar format slightly varies across the different JLS grammars and versions; we had to collect formatting rules from different documents and sections — in particular from [2, §2.4], [3, §2.4, §18] and [4, §2.4, §18].

[1]There are various accounts that have identified or fixed bugs in the JLS grammars or, in fact, in grammars that were derived from the JLS in some manner. We refer to the work of Richard Bosworth as a particularly operational account; it is a clear list of bugs which is also endorsed by Sun Microsystems: http://www.cmis.brighton.ac.uk/staff/rnb/bosware/javaSyntax/syntaxV2.html. We refer to this list as "known bugs" in our process.

[2]The complete JLS effort (including all the involved sources, transformations, results, and tools) is publicly available through http://slps.sf.net/; see `topics/java/lci` in particular.

[3]In this paper, we show grammar fragments in a pretty-printed format (as opposed to the markup-based source format): nonterminals are in italic type, terminals are enclosed in double quotes, and operators "?", "*" and "+" serve for optionality and lists; top-level choices (alternatives) are represented as a set of productions with the same left-hand side; elisions are shown as "...".

	Grammar class	**Iteration style**
impl1	LALR(1)	left-recursive
read1	none	left-recursive
impl2	unclear	EBNF
read2	none	left-recursive
impl3	"nearly" LL(k)	EBNF
read3	none	left-recursive

Table I
Basic properties of the JLS grammars.

	Productions	Nonterminals	Tops	Bottoms
impl1	282	135	1	7
read1	315	148	1	9
impl2	185	80	6	11
read2	346	151	1	11
impl3	245	114	2	12
read3	435	197	3	14

Table II
Basic metrics of the JLS grammars.

In order to deal with irregularities of the input format, we needed to design and implement a non-classic parser to extract and analyze the grammar segments of the documents and to perform some forms of recovery. There are these categories of irregularities: liberal use of markup tags, misleading indentation, duplicate definitions as well as numerous smaller issues. (About 700 fixes were performed.) The extraction parser is beyond the scope of this paper.

B. Grammar classes and correspondences

JLS1: It is stated [2, §19] that the more implementable grammar has "*been mechanically checked to insure that it is LALR(1)*". The correspondence between *read1* and *impl1* is briefly described by saying [2, §2.3] that *read1* is "*very similar to*" *impl1* "*but more readable*".

JLS2: The second edition of the JLS [3, "Preface to the Second Edition"] "*integrates all the changes made to the Java programming language since [...] the first edition in 1996. The bulk of these changes [...] revolve around the addition of nested type declarations.*" The JLS1/2 grammars themselves are nowhere related explicitly. Upon cursory examination we came to conclude that *read1* and *read2* are strikingly similar (modulo the extensions to be expected), whereas surprisingly, *impl1* and *impl2* appeared as different developments. Also, the LALR(1) claim for *impl1* is not matched by *impl2* for which it is only said [3, §18] to be "*the basis for the reference implementation*".

JLS3: JLS3 extends JLS2 in numerous ways [4, Preface]: "*Generics, annotations, asserts, autoboxing and unboxing, enum types, foreach loops, variable arity methods and static imports have all been added to the language*". Again, the JLS2/3 grammars themselves are nowhere related explicitly, and again, cursory examination suggests that *read2* and *read3* are strikingly similar (modulo the extensions to be expected). This time, *impl2* and *impl3* also bear strong resemblance. No definitive grammar-class claim is made, but an approximation thereof: *impl3* is said [4, §18] to be "*not an LL(1) grammar, though [...] it minimizes the necessary look ahead.*" Hence, *impl3* has definitely departed from LALR(1) (the grammar class of *impl1*).

In addition to grammar class claims for the JLS grammars we have also recorded iteration styles during cursory examination; see Table I. This data already clarifies that we need to bridge the gap between different iteration styles (which is relatively simple) but also different grammar classes (which is more involved) — if we want to recover the relationships between the different grammars by effective transformations.

C. Simple grammar metrics

Table II displays some simple grammar metrics for the various JLS grammars.[4] We have eventually understood that the major differences between the numbers of productions and nonterminals for the two grammars of any given version is mainly implied by the different grammar classes and iteration styles. The decrease of numbers for the step from *impl1* to *impl2* is explainable with the fact that an LALR(1) grammar was replaced by a new development (which does not aim at LALR(1)). Otherwise, the obvious trend is that the numbers of productions and nonterminals go up with the version number.

The *difference in numbers of top-nonterminals is a problem indicator*. There should be only one top-nonterminal: the actual start symbol of the Java grammar. The *difference in numbers of bottom-nonterminals could be reasonable* because a bottom nonterminal may be a lexeme class — those classes are somewhat of a grammar-design issue. However, a review of the nonterminal symbols rapidly reveals that some of them correspond to (undefined) categories of compound syntactic structures.

D. Convergence outline

Let us consider again the convergence tree of Fig. 1. The plan must be to devise transformations such that the two grammars per JLS version are "converged to a common denominator" (see the nodes *jls1–3* in the figure), and all three versions are "converged" (in pairwise fashion) to account for inter-version differences — the extensions to the

[4]A *top nonterminal* is a nonterminal that is defined but never used; a *bottom nonterminal* is a nonterminal that is used but never defined; see [5], [6] for these terms.

Java language in particular (see the nodes *jls12* and *jls123* as well as *read12* and *read123* in the figure).

When deriving *jls1–3*, we give preference to the more implementable grammar as the target of convergence (while some refactoring and correction may still be applied to it). This preference reflects the general rule that an implementation-oriented artifact should be derived from a design-oriented artifact — rather than the other way around. (Incidentally, this direction is also easier to handle by the available transformation operators.)

When relating the different JLS versions, we adopt the redundant approach to relate the common denominators *jls1–3* in one cascade (see the nodes *jls12* and *jls123*), but also the readable grammars *read1–3* in another cascade (see the nodes *read12* and *read123*). The latter cascade is presumably more important because *read1–3* are known to be structurally similar; hence convergence can be expected to be cheap. The additional cascade can also be seen as a sanity and scalability check for the method.

III. Grammar transformation

In this section, we illustrate intended and accidental differences between the JLS grammars and we represent those differences as operational grammar transformations. Simultaneously, we provide an overview of the major operators that are needed for the transformation of concrete syntax definitions in the context of grammar convergence. The operators are independent of the language of study — be it Java or Cobol.

We distinguish semantics-preserving vs. semantics-increasing vs. semantics-decreasing vs. semantics-editing operators [1]. The term semantics refers here to the language generated by the grammar — when considered as a set of strings.

A. Semantics-preserving operators

There are operators to *fold* and *unfold* nonterminal definitions, to *extract* and *inline* specific nonterminals, to *factor* and *distribute* grammar expressions, to "*massage*" grammar expressions (i.e., to rewrite them according to algebraic laws), and to *alter* iteration style (recursion vs. "*"). We also say that all these operators serve *grammar refactoring*.

Several transformation operators serve disciplined "replacement", i.e., they are invoked by the form $o(x, x')$ where o is the operator in question, x is the grammar expression to be located in the input, and x' is the corresponding replacement. For instance, the *factor* operator is applied to an expression and a *factored* variation; the *massage* operator is applied to an expression and an *algebraically equivalent* variation based on a fixed set of laws.

*Example 3.1 (*factor *and* massage *transformations):*

factor*(*
  (Block | *(*`"static"` *Block))* ,
  ((ε | `"static"`*) Block));*
massage*(*
  (ε | `"static"`*)* ,
  `"static"`*?);*

In *read2*, there are distinct alternatives for blocks vs. static blocks. In contrast, in *impl2*, these forms appear in a factored manner. Hence, the *factor* operator is used to factor out the shared reference to *block*. Then, the *massage* operator changes the style of expressing optionality of the keyword "static".

Other grammar transformation operators apply a fixed operation to a specific nonterminal, and hence, they can be invoked by the form $o(n)$ where o is the operator in question, and n is the nonterminal to be affected. For instance, *inlining* a nonterminal can be requested in this manner. Also, the conversion from a recursive definition-based style of iteration to the use of the regular operators "*" and "+" can be requested in this manner. (We call the latter step "de-yaccification" [7], [8].)

Like the previous transformation sample, the next one is taken from a refactoring script that aligns *read2* and *impl2*. The JLS case involves many hundreds of such small refactoring steps; see §V.

*Example 3.2 (*deyaccify *and* inline *transformations):*

deyaccify*(ClassBodyDeclarations);*
inline*(ClassBodyDeclarations);*
massage*(*
  ClassBodyDeclaration$^+$*?* ,
  ClassBodyDeclaration* *);*

In *read2*, recursion-based style of iteration is used. For instance, there is a recursively defined nonterminal *ClassBodyDeclarations* for lists of *ClassBodyDeclaration*. In contrast, in *impl2*, the list form "*" is used. Deyaccification replaces the recursive definition of *ClassBodyDeclarations* by *ClassBodyDeclaration*$^+$. The nonterminal *ClassBodyDeclarations* is no longer needed, and hence inlined. The list of declarations was optional, and hence "+" and "?" can be simplified to "*".

B. Semantics-in/decreasing operators

There are operators to *widen* and *narrow* occurrence constraints (e.g., to change "+" to "*" and vice versa), to *add* and *remove* alternatives (say, productions), and to replace a nonterminal occurrence by one of its productions and vice versa (to which we refer as *downgrading* and *upgrading*). One can also make optional symbols (i.e., those with "?" or "*") to *appear* or *disappear*.

Example 3.3 (Widening an occurrence constraint):

```
widen(
  "static",
  "static"?,
  in ClassBodyDeclaration);
```

This transformation is part of a script that captures the delta between JLS1 and JLS2. The particular widening step enables *instance* initializers in class bodies (where only *static* initializers were admitted before).

The example also demonstrates that transformation operators may carry an extra argument to describe the *scope of replacement*. By default, the scope is universal: all matching expressions in the input grammar would be affected. Selective scopes are nonterminal definitions (specified by a nonterminal — as in the example) or productions (specified by a production label).

Example 3.4 (Adding an alternative):

add(*ConstantModifier: Annotation);*

This transformation is part of a script that captures the delta between JLS2 and JLS3. In JLS2, a constant modifier can be `"public"` or `"static"` or `"final"`. JLS3 offers the additional option *Annotation*.

When we seek relationships between grammars of different versions, then semantics-increasing/-decreasing transformations are clearly to be expected. As a matter of discipline, we prefer to describe the delta by a semantic-increasing transformation to map a version to its successor version (as opposed to the inverse direction). We speak of *grammar extension* in this case.

However, increase (or decrease) may also be needed when two grammars are essentially equivalent — except that one is more permissive than the other. This actually happens in practice: a permissive grammar may be needed as a concession to practicality of say parser implementation. We also speak of *grammar relaxation* in this case. In the JLS case, the different purposes of the grammars (to be more or less readable or implementable resp.) imply the need for relaxation. Similar issues arise with relationships between abstract and concrete syntaxes [1].

Finally, two grammars may differ (with regard to the generated language) in a manner that is purely accidental (read as "incorrect"). We speak of (transformations for) *grammar correction* in this case. Some corrections may be expressed in terms of semantics-increasing/-decreasing operators. (Otherwise we have to use less disciplined operators; see below.)

Example 3.5 (Grammar relaxation):

impl2

Modifier: `"public"` | ...

read2

ClassModifier :	`"public"` | ...
FieldModifier :	`"public"` | ...
InterfaceModifier :	`"public"` | ...
MethodModifier:	`"public"` | ...

In *impl2*, there is only one category of (arbitrary) modifiers. In contrast, in *read2*, there are various precise categories of modifiers for classes, fields, interfaces and methods. Accordingly, the *impl2* grammar is more permissive as far as modifiers are concerned. We omit the neutralizing transformation.

We suggest that a language specification should explicitly call out relaxations so that they are not confused with overlooked inconsistencies (to be modelled as corrections) or evolutionary differences (to be modelled as extensions).

C. Semantics-revising operators

There are operators to *undefine* a nonterminal (i.e., to abandon its definition), to *replace* a grammar expression in an unconstrained manner, to *inject* new components into a production and to *project* away existing components. The operators *inject* and *project* can be invoked by a form such that a *grammar expression with markers* (as in *a <b> c*) is passed as a parameter. These markers highlight the components to be added or removed resp., and thereby state the intention of the operator application more explicitly.

Example 3.6 (Correcting statement syntax in impl2*):*

inject(*Statement:* `"break"` *Identifier*? < `";"` >);

The production for the *break* statement lacks the semicolon which is injected accordingly (left unnoticed in Bosworth's bug list, but obvious when converging with *read2*).

Example 3.7 (Correcting expression syntax):

Incorrect expression syntax in impl2 *and* impl3

```
Expression2: Expression3 [ Expression2Rest ]
Expression2Rest: (InfixOp Expression3)*
Expression2Rest: Expression3 "instanceof" Type
```

Language-revising transformation

```
project(
  Expression2Rest:
    < Expression3 > "instanceof" Type);
```

Corrected expression syntax

```
Expression2: Expression3 [ Expression2Rest ]
Expression2Rest: (InfixOp Expression3)*
Expression2Rest: "instanceof" Type
```

The *impl2* and *impl3* grammars define the Java expression syntax by means of layers, i.e., there are several nonterminals *Expression1*, *Expression2*, ... for the different priorities. We are concerned with one layer here. The second rule for *Expression2Rest* contains an offending occurrence of *Expression3* which needs to be projected away. This issue was revealed by comparing the *impl2* and *impl3* grammars with the *read2* and *read3* grammars (after some prior refactoring).

These two examples pinpoint “grammar bugs”: incorrect syntax. In some cases, incorrect syntax merely arises from representation anomalies of the HTML input used for extraction — as illustrated below.

Example 3.8 (Extraction — post-processing for impl3):

```
replace(
  BlockStatements* ,
  "{" BlockStatements "}" );
```

The source format defines curly brackets to express iteration. However, in the example at hand, they were meant as terminals, and were not recognized due to missing markup. The incorrect list construct is replaced accordingly.

Example 3.9 (Initial correction for impl2):

```
replace(
   Expr,
   Expression);
```

A misnamed nonterminal is found when examining the list of bottom nonterminals before the convergence process starts.

IV. Convergence process

Overall, grammar convergence is an iterative process that alternates two activities [1]: grammar *comparison* (to determine differences) and grammar *transformation* (to resolve differences and thereby move closer towards convergence, i.e., structural equivalence of grammars). There are two elements of choice involved: which difference to pick for treatment at a given point, and what transformations to apply for the resolution of the difference. We refine this process in a substantial manner below, but these elements of choice will remain.

In the sequel, we limit ourselves to convergence for *pairs* of grammars (which is sufficient for the cascaded tree of the JLS case).

A. Grammar differences

We perform (tool-supported) grammar comparison to retrieve nominal or structural grammar differences. We face a *nominal difference* when a nonterminal is defined or referenced in one of the grammars but not in the other. We face a *structural difference* when the definitions of a shared nonterminal differ. For every nonterminal, we actually count the maximum number of unmatched alternatives (of either grammar) as the number of structural differences.

B. Phases of convergence

Grammar convergence is organized in a sequence of phases as follows.

Preparation: This phase involves correcting immediately obvious or a-priori known errors in both grammars. In the JLS case, we incorporated an available bug list at this stage. We also resolved some inaccuracies that were caused by representation anomalies in the HTML input. Further, we added missing definitions. No grammar comparison is involved in this phase.

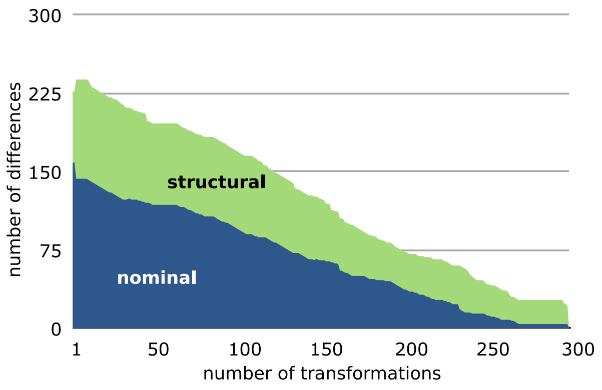

Figure 2. Difference reduction for *read2* towards the convergence target *jls2* in the convergence tree of Fig. 1.

Nominal matching: This phase aligns the syntactic categories of the grammars in terms of their nonterminals. The nominal differences serve as a guidance here to draft renaming as well as extract/inline transformations such that they immediately reduce the number of nominal differences.

This phase relies on two soft assumptions. First, when a nonterminal occurs in both grammars, then it models the same syntactic category (conceptually). Second, the grammar engineer correctly matches different nonterminals from the two grammars. Otherwise, severe structural differences would be encountered eventually as a symptom of error.

Structural matching: This phase aligns the definitions of the nonterminals in a structural sense; this phase dominates the transformation effort. The structural differences serve as a guidance here to draft refactorings such that they immediately reduce the number of differences. The basic idea is that each refactoring operator serves a certain pattern of structural differences, and hence the grammar engineer can systematically exhaust these patterns.

Resolution: This phase consists of three kinds of steps, as discussed in §III: *extension*, *relaxation* and *correction*. In the case of semantics-increasing operators, it is up to the grammar engineer to perform the classification. (Semantics-revising transformations serve correction by definition. Semantics-decreasing transformations must serve correction too because of the directed process that we use in capturing version or permissiveness differences.) Again, each of the possible operators serves a certain pattern of structural differences.

C. Difference reduction

Hence the guiding principle for grammar convergence is to consistently reduce the number of grammar differences

	jls1	**jls12**	**jls123**	**jls2**	**jls3**	**read12**	**read123**	**Total**
Number of lines	682	5116	2847	6772	10715	1639	3082	30853
Number of transformations	67	298	111	395	544	77	135	1627
○ Semantics-preserving	45	239	80	283	381	31	78	1137
○ Semantics-increasing or -decreasing	22	58	31	102	150	39	53	455
○ Semantics-revising	—	1	—	10	13	7	4	35
Preparation phase	1	—	—	15	24	11	14	65
○ Known bugs (Ex. 3.7)	—	—	—	1	11	—	4	16
○ Post-extraction (Ex. 3.8)	—	—	—	7	8	7	5	27
○ Initial correction (Ex. 3.9)	1	—	—	7	5	4	5	22
Resolution phase	21	59	31	97	139	35	43	425
○ Extension (Ex. 3.4)	—	17	26	—	—	31	38	112
○ Relaxation (Ex. 3.5)	18	39	5	75	112	—	2	251
○ Correction (Ex. 3.6)	3	3	—	22	27	4	3	62

Figure 3. *Transformation of the JLS grammars — effort metrics and categorization*

	Productions	Nonterminals	Tops	Bottoms
jls1	278	132	1	7
jls2	178	75	1	7
jls3	236	109	1	7
jls12	178	75	1	7
jls123	236	109	1	7
read12	345	152	1	7
read123	438	201	1	7

Table III
Simple metrics for the derived JLS grammars.

throughout the two matching phases as well as the final resolution phase. Fig. 2 illustrates this principle for one specific JLS grammar and the related convergence. The figure also visualizes that nominal differences tend to be resolved earlier than structural differences.

Our transformation infrastructure is actually aware of the different phases of convergence, and it checks (run-time) the incremental reduction of differences. To this end, we rely on an asymmetric use of convergence with pairs of input grammars where always one grammar serves as a baseline for the other.[5]

V. Post-mortem of the JLS case

Table III shows the same, simple metrics for the derived grammars as we originally presented for the leaves of the convergence tree; c.f., Table II. Top- and bottom-nonterminals are consolidated now. In the case of the "common denominators" *jls1–3*, the numbers of nonterminals and productions reflect that these grammars were derived to be similar to *impl1–3*. Similar correlations hold for the "inter-version" grammars in the rest of the table.

[5]As a concession to a simple design of the operator suite for grammar transformations, we are also allowed to use restructuring steps that slightly increase structural differences as long as we explicitly group them such that the complete "transaction" still achieves reduction.

Fig. 3 measures the extraction effort and the involved grammar transformations. This information was obtained in an automated manner but it relies on some amount of semantic annotation of the transformations for the classifications and phases.

The number of transformations directly refers to the number of *applications of transformation operators*. 33 different operators are used in the JLS case; most of them were introduced in §III. About three quarters of the transformations are semantics-preserving. The remaining quarter is mainly dedicated to semantics-increasing or -decreasing transformations with only 2% of semantics-revising transformations.

In Fig. 3, one can observe that relaxation transformations indeed occur when a more readable and a more implementable grammar are converged. Further, one can observe that the overall transformation effort is particularly high for *jls12* — which signifies the fact (already mentioned above) that *impl1* and *impl2* appear to be different developments. Finally, we have made an effort to incorporate Sun's bug list into the picture (see "Known bugs"). We note that some of the known bugs equally occur in both the more readable and the more implementable grammar, in which case we cannot even discover them by grammar convergence.

VI. Related work

Broadly speaking, grammar convergence and the present case study contribute to grammar(ware) engineering. Within this context, our work is related to *agile parsing* [9] and *grammar recovery or re-engineering of syntax definitions* [5], [6], [7], [10], [11], model-driven parser development [12] as well as *grammar inference* [13], [14], [15], [16], [17]. However, such related work does not involve two central elements of grammar convergence: comparison and simultaneous transformation of two or more grammars. All grammar inference and recovery methods essentially involve code samples, which currently play no role in grammar convergence. It is conceivable to combine methods, e.g., the use of grammar inference techniques to inform a semi-automatic grammar transformation approach.

An interesting blend of recovery and convergence (or consistency checking) is described in [18] where *precedence rules* are recovered from *multiple* grammars and checked for consistency.

There is also related grammar engineering work that incorporates *measures* into some related process (e.g., grammar metrics in the process of grammar recovery) [19], [20], [10]. We have shown that simple measures for grammar differences can drive the process of grammar convergence.

Grammar transformations [9], [8], [21], [22], [23] provide the heavy lifting for grammar convergence. The sketched operator suite (as of §III) builds on top of our previous work on transformation operators [8], [22]. However, we have systematically extended the suite to provide more opportunities for semantics-preserving or reasonably semantics-increasing transformations. For instance, in §III-A, we mentioned *factor* and *distribute* operators that specifically arise from the need to align a less factored (i.e., more readable) grammar with a more factored (i.e., more implementable) grammar.

Grammar comparison can be compared with *schema matching* in ER/relational modeling [24], [25] as well as model and metamodel matching/diffing in model-driven engineering [26], [27], [28] (specifically in the context of model/metamodel evolution). However, our current approach to comparison (as of §IV) is more straightforward than schema matching for two reasons. First, the metamodel of grammars is relatively simple. Second, we only expect nominal differences and structural differences based on matching up alternatives. We will need a more advanced comparison machinery (which could benefit from previous work indeed) once we aim at inference of grammar transformations.

VII. Concluding remarks

We have provided the first published record of recovering and representing the relationships between given grammars of industrial size that serve different audiences (language users and implementers) and that capture different versions of the language. Our results indicate that consistency among the different grammars and versions even for a language as complex as Java is achievable.

The recovery and representation of grammar relationships is based on a systematic and mechanized process that leverages a-priori known grammar bugs, grammar metrics (e.g., for problem indication), grammar comparison for nominal and structural differences, and most notably, grammar transformations. We carefully distinguish transformations for grammar refactoring, extension, correction and relaxation.

While the JLS situation required the recovery of grammar relationships, the ultimate best practice for grammar convergence should require continuous maintenance of relationships. That is, the relationships should be continuously checked and updated whenever necessary along dependent or independent evolution of the involved artifacts.

The approach, as it stands, faces a *productivity problem*. The transformation part of grammar convergence requires substantial effort by the grammar engineer to actually map any given grammar difference into a (short) sequence of applications of operators for grammar transformation. For instance, the JLS transformations required several weeks of work. Such costs may be prohibitive for widespread adoption of grammar convergence.

Notable productivity gains can be expected from advanced tool support. We currently rely on basic batch execution of the transformations. Instead, the transformations could be done interactively and incrementally with good integration for grammar comparison, transformation and error diagnosis.

Other productivity gains are known to be achievable by means of normalization schemes (c.f., de-/yaccification in [7], [8]).

However, ultimately, we need to provide inference of relationships (in fact, transformations). Such inference is a challenging problem because the convergence process involves elements of choice that we need to better understand before we expect reasonable results. For instance, when two syntactic categories are equivalent under fold/unfold modulations, then the grammar engineer is likely to favor one of the two forms — this calls for either an interactive approach or appropriate notions of normal forms or rule-based normalization.

Perhaps the most exciting, remaining problem is to provide a proper formal argument for the "minimality" of the non-semantics-preserving transformations that are involved in a convergence. Currently, we use the pragmatic approach to first align nonterminals, then to align alternatives (by structure) as much as possible, and finally to break out of refactoring and allow ourselves presumably local non-semantics preserving transformations. However, there is no formal guarantee currently for not facing a false positive ("a presumed language difference that is none"). That is, one may accidentally engage in semantics-revising transformations even though the relevant syntactic categories are equivalent, but nonterminal symbols or alternatives are confused by the grammar engineer. Formally, the desired notion of minimality is limited by the undecidability of grammar equivalence, but we are confident that a practical strategy can be devised based on appropriate static analyses of the transformations and the involved grammars.

Acknowledgements The first author is grateful for opportunities to present and discuss some of this work on several occasions: University of Waterloo, IEEE Kitchener-Waterloo, August 6, 2008; Dagstuhl Seminar 08331, 'Perspectives Workshop: Model Engineering of Complex Systems (MECS)', August 12, 2008; École des Mines de Nantes, November 7, 2009; METRIK Workshop, in Berlin, November 21, 2008; the BENEVOL Workshop (invited talk), Eindhoven, December 12, 2008; BX-Grace Meeting near Tokyo, December 15, 2008; DAIMI, University of Aarhus; February 24.

References

[1] R. Lämmel and V. Zaytsev, "An Introduction to Grammar Convergence," in *Integrated Formal Methods, 7th International Conference, IFM 2009, Proceedings*, ser. LNCS, vol. 5423. Springer, 2009, pp. 246–260.

[2] J. Gosling, B. Joy, and G. L. Steele, *The Java Language Specification*. Addison-Wesley, 1996, available at java.sun.com/docs/books/jls.

[3] J. Gosling, B. Joy, G. L. Steele, and G. Bracha, *The Java Language Specification*, 2nd ed. Addison-Wesley, 2000, available at java.sun.com/docs/books/jls.

[4] ——, *The Java Language Specification*, 3rd ed. Addison-Wesley, 2005, available at java.sun.com/docs/books/jls.

[5] R. Lämmel and C. Verhoef, "Semi-automatic Grammar Recovery," *Software—Practice & Experience*, vol. 31, no. 15, pp. 1395–1438, 2001.

[6] M. Sellink and C. Verhoef, "Development, Assessment, and Reengineering of Language Descriptions," in *Proceedings, Conference on Software Maintenance and Reengineering (CSMR'00)*. IEEE, 2000, pp. 151–160.

[7] M. de Jonge and R. Monajemi, "Cost-Effective Maintenance Tools for Proprietary Languages," in *Proceedings, International Conference on Software Maintenance (ICSM'01)*. IEEE, 2001, pp. 240–249.

[8] R. Lämmel, "Grammar Adaptation," in *Proceedings, Formal Methods Europe (FME) 2001*, ser. LNCS, vol. 2021. Springer, 2001, pp. 550–570.

[9] T. Dean, J. Cordy, A. Malton, and K. Schneider, "Agile Parsing in TXL," *Journal of Automated Software Engineering*, vol. 10, no. 4, pp. 311–336, 2003.

[10] E. B. Duffy and B. A. Malloy, "An Automated Approach to Grammar Recovery for a Dialect of the C++ Language," in *Proceedings, 14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 11–20.

[11] O. Nierstrasz, M. Kobel, T. Girba, M. Lanza, and H. Bunke, "Example-Driven Reconstruction of Software Models," in *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. IEEE, 2007, pp. 275–286.

[12] F. Jouault, J. Bézivin, and I. Kurtev, "TCS:: a DSL for the specification of textual concrete syntaxes in model engineering," in *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 2006, pp. 249–254.

[13] M. Črepinšek, M. Mernik, F. Javed, B. R. Bryant, and A. Sprague, "Extracting grammar from programs: evolutionary approach," *SIGPLAN Not.*, vol. 40, no. 4, pp. 39–46, 2005.

[14] A. Dubey, S. K. Aggarwal, and P. Jalote, "A Technique for Extracting Keyword Based Rules from a Set of Programs," in *9th European Conference on Software Maintenance and Reengineering (CSMR 2005), Proceedings*. IEEE, 2005, pp. 217–225.

[15] A. Dubey, P. Jalote, and S. K. Aggarwal, "Inferring Grammar Rules of Programming Language Dialects," in *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Proceedings*, ser. LNCS, vol. 4201. Springer, 2006, pp. 201–213.

[16] M. Di Penta and K. Taneja, "Towards the Automatic Evolution of Reengineering Tools," in *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR '05),*. IEEE, 2005, pp. 241–244.

[17] M. Di Penta, P. Lombardi, K. Taneja, and L. Troiano, "Search-based Inference of Dialect Grammars," *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, vol. 12, no. 1, pp. 51–66, 2008.

[18] E. Bouwers, M. Bravenboer, and E. Visser, "Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking," *ENTCS*, vol. 203, no. 2, pp. 85–101, 2008.

[19] T. Alves and J. Visser, "A case study in grammar engineering," in *Post-proceedings of 1st International Conference on Software Language Engineering (SLE'08)*, ser. LNCS. Springer, 2009, to appear.

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

[21] T. Dean, J. Cordy, A. Malton, and K. Schneider, "Grammar Programming in TXL," in *Proceedings, Source Code Analysis and Manipulation (SCAM'02)*. IEEE, 2002.

[22] R. Lämmel and G. Wachsmuth, "Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment," in *Proceedings, Language Descriptions, Tools and Applications (LDTA'01)*, ser. ENTCS, vol. 44. Elsevier Science, 2001.

[23] D. Wile, "Abstract Syntax From Concrete Syntax," in *Proceedings, International Conference on Software Engineering (ICSE'97)*. ACM Press, 1997, pp. 472–480.

[24] H. H. Do and E. Rahm, "Matching large schemas: Approaches and evaluation," *Information Systems*, vol. 32, no. 6, pp. 857–885, 2007.

[25] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[26] 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)*, ser. LNCS, vol. 5301. Springer, 2008, pp. 326–340.

[27] S. Wenzel and U. Kelter, "Analyzing model evolution," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 831–834.

[28] Z. Xing and E. Stroulia, "Refactoring Detection based on UMLDiff Change-Facts Queries," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 263–274.