Noname manuscript No.
(will be inserted by the editor)

Recovering Grammar Relationships for the Java Language Specification

Ralf Lämmel · Vadim Zaytsev

Received: date / Accepted: date

Abstract Grammar convergence is a method that helps discovering relationships between different grammars of the same language or different language versions. The key element of the method is the operational, transformation-based representation of those relationships. Given input grammars for convergence, they are transformed until they are structurally equal. The transformations are composed from primitive operators; properties of these operators and the composed chains provide quantitative and qualitative insight into the relationships between the grammars at hand.

We describe a refined method for grammar convergence, and we use it in a major study, where we 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 transformation chains that capture all accidental or intended differences between the JLS grammars. This method is mechanized and driven by nominal and structural differences between pairs of grammars that are subject to asymmetric, binary convergence steps.

We present the underlying operator suite for grammar transformation in detail, and we illustrate the suite with many examples of transformations on the JLS grammars. We also describe the extraction effort, which was needed to make the JLS grammars amenable to automated processing. We include substantial metadata about the convergence process for the JLS so that the effort becomes reproducible and transparent.

Keywords grammar convergence · grammar transformation · grammar recovery · grammar extraction · language documentation

R. Lämmel
Software Languages Team
The University of Koblenz-Landau
Germany
E-mail: laemmel@uni-koblenz.de

V. Zaytsev
Software Languages Team
The University of Koblenz-Landau
Germany
E-mail: zaytsev@uni-koblenz.de

1 Introduction

Overall, this paper is concerned with **the problem of grammar consistency checking**. Many software languages (and programming languages, in particular) are described simultaneously by multiple grammars that are found in different software artifacts. For instance, one grammar may reside in a language specification; another grammar may be encoded in a parser specification; yet another grammar may be present in an XML schema for tool-independent data exchange. Ideally, one would want to reliably establish and continuously maintain that all co-existing (potentially embedded) grammars describe the same intended language. Without such guarantee, grammar inconsistencies may go unnoticed, and grammar-based software artifacts may get brittle. Some existing ad-hoc or brute-force methods partially address this problem, but ultimately grammar consistency checking is an open software engineering problem without a satisfying best practice. A good example is the Java Language Specification (JLS; Gosling et al, 1996, 2000, 2005), which is the target of the present paper. The JLS is a critical specification in the software industry, yet it contains substantial inconsistencies.

Let us sketch **the obstacles for grammar consistency checking.** Consider the problem of establishing or maintaining that some given BNFs (i.e., grammars) describe the same language. An automated solution is constrained by the formal undecidability of grammar equivalence. Such a formal limit is certainly part of the problem that there is no best practice for grammar consistency checking. Obviously, the problem becomes even more challenging once we consider the practical situation of grammars of many different forms: BNFs, parser descriptions, XML schemata, software models, etc. Such variation implies impedance mismatches. As a result, it may be hard to mentally or automatically map one grammar to the other. The present paper describes a method that addresses those obstacles effectively.

In essence, **grammar consistency checking deals with grammar differences** in a systematic manner. Grammars for the same language may be different for various, practically viable reasons. For instance, grammars may be tailored for a certain purpose or quality such as “readability”. (In the present paper, we deal with “more readable” vs. “more implementable” grammars for the Java language.) Some grammars may have been designed independently of one another, and hence they are likely to be vastly different in the sense of structural equality of the grammar specifications. Other grammars may have been affected heavily by compromises required by implementation technologies (e.g., parsing techniques), or data models (e.g., XML Schema as opposed to BNF). To summarize, in practice, there are many intended, accidental, idiosyncratic, superficial, and substantial differences between co-existing grammars of a language.

In fact, we need a generalized form of grammar consistency checking that also account for **versatile grammar relationships due to software and language evolution**. Both, software languages as such (e.g., in the form of language documentation) and grammar-based software artifacts (e.g., compilers, source code analysis tools, IDEs) are subject to possibly independent evolution. The grammars of different versions are not even intended to describe the same language, but one would still want to understand their relative correspondence in terms of a delta between these versions. As a result, there are even more grammars to be checked for consistency. Also, we are no longer restricted to plain grammar equivalence, but language extensions, restrictions, or revisions would need to be captured and checked.

Another related challenge of language evolution is migration of data (programs, words, etc.) across versions. We do not discuss this challenge in the present paper, even though the underlying method may be potentially useful in such a context.

In Lämmel and Zaytsev (2009), we have begun to address the fundamental problem of grammar diversity by initiating a method for **grammar convergence**. 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). Grammar convergence is another method of grammar engineering—as such, it is a companion of grammar recovery, adaptation, and inference. The specific property of convergence is that it genuinely takes several grammars as input—as opposed to any process that starts from a single grammar.

In the present paper, we describe **the JLS study**—a major study for grammar convergence for the Java language. We also deliver **a refined method for grammar convergence with improved scalability and reproducibility**.[1] The study in this paper concerns the 3 different versions of the Java Language Specification (JLS; Gosling et al, 1996, 2000, 2005). Each of the 3 JLS versions contains 2 grammars: one grammar is said to be optimized for readability, and the other one is intended as a basis for implementation.

Let us briefly discuss the JLS situation. 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 generate languages that engage in an inclusion ordering 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 study shows.*

The JLS is critical to the Java platform—it is a foundation for compilers, code generators, pretty-printers, IDEs, source code analysis and manipulation tools, and other grammarware for the Java language. The JLS is the authoritative specification of Java. Hence, there is a strong incentive for an unambiguous, consistent and understandable set of JLS documents. Still, our study discovers substantial inconsistencies with the help of grammar convergence.

Our work is in no way restricted to the JLS. We notice **a broader impact on language standardization and engineering**. Based on the major JLS study of the present paper, previous work on grammar recovery, and general trends in software language engineering, we contend that grammar convergence improves the state of the art in creation, maintenance and evolution of language documentation. Ideally, we would hope for standardization bodies and language documenters to incorporate grammar convergence into their methodology. For instance, it would be clearly desirable for Oracle to abandon manual grammar editing in the next version of Java and the JLS. We refer to Klusener and Zaytsev (2005) for a proposal, in fact, an ISO document, that hints at the application of grammar engineering techniques such as grammar convergence in the context of creating, maintaining, or evolving language documents. Realistically, though, it will be difficult to replace current ad-hoc techniques of dealing with multiple grammars in language documents and otherwise. We will discuss some of the limitations of the current grammar convergence method in the conclusion. There is the particular issue of adoption: it would take Oracle, ISO, and other such organizations substantial effort to incorporate additional methods and tools into their processes, and to adjust existing documents. Overall, grammar convergence is still an emerging method.

[1] An earlier and abbreviated account on this work has been published in the Proceedings of Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, pp. 178–186.

Contributions

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.) The significance of our work is amplified by two arguments. First, we provide a simple and mechanized process for discovering 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.

Here is an itemized summary of the contributions of this work:

1. We have recovered nontrivial relationships between grammars of industrial size. (That is, we show that the grammars are equivalent modulo well-defined transformations.)
2. We have designed a mechanized, measurable and reproducible process for grammar convergence. Compared to the initial work on grammar convergence (Lämmel and Zaytsev, 2009), the process consists of well-defined phases and its progress can be effectively tracked in terms of the numbers of nominal and structural differences between the grammars at hand.
3. We have worked out a comprehensive operator suite for grammar transformation driven by the scale of the present JLS study. The suite substantially improves on prior art.
4. The complete JLS effort (including all the involved sources, transformations, results, and tools) is publicly available through SourceForge.[2]

Roadmap

§2 gives an overview on grammar convergence method, it prepare the application of the method to the JLS, and it describes phases of a refined process of convergence that we extracted from the reported JLS study. §3 describes the extraction phase of grammar convergence for the JLS. §4 describes an operator suite for grammar transformation, and applies it to the JLS. §4.3 provides a postmortem for the reported JLS study. §5 discusses related work. §6 concludes the paper.

2 Grammar convergence

The central idea of grammar convergence (Lämmel and Zaytsev, 2009) is to extract grammars from diverse software artifacts, and to discover and represent the relationships between the grammars by chains of transformation steps that make the grammars structurally equal. In this section, we will describe the method in detail and prepare its application to the JLS. The method relies on the following core ingredients:

– A unified *grammar format* that effectively supports abstraction from specialities or idiosyncrasies of the grammars as they occur in software artifacts in practice.
– A *grammar extractor* for each kind of artifact. (In the present JLS study, we had to extract grammars from the JLS documents, which are available in HTML and PDF.)

[2] `http://slps.sf.net/`; see `topics/java/lci` in particular.

read2 (Gosling et al, 2000, §8.1)

ClassDeclaration:
    ClassModifiers? `"class"` *Identifier Super*? *Interfaces*? *ClassBody*

read3 (Gosling et al, 2005, §8.1, §8.9)

ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration
NormalClassDeclaration:
    ClassModifiers? `"class"` *Identifier TypeParameters*? *Super*? *Interfaces*? *ClassBody*
EnumDeclaration:
    ClassModifiers? `"enum"` *Identifier Interfaces*? *EnumBody*

Fig. 1 Two similar grammar excerpts from different versions of the JLS. The second excerpt involves two more nonterminals than the first excerpt: *NormalClassDeclaration*, which looks similar to the nonterminal from the first grammar, and *EnumDeclaration*, which is completely new. Hence, we speak of two nominal differences (two nonterminals in *read3* that do not match *read2*), and of two structural differences (two unmatched branches in *ClassDeclaration*).

– A *grammar comparator* that determines and reports grammar differences in the sense of deviations from structural equality.
– A framework for automated *grammar transformation* that can be used to refactor, or to otherwise more liberally edit grammars until they become structurally equal.

2.1 Grammar comparison

The grammar comparator is used to discover grammar differences, and thereby, to help with drafting transformations in a stepwise manner. We distinguish nominal vs. 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 two given grammars. Some of the nominal differences will be eliminated by a simple renaming, while others will disappear gradually when dealing with structural differences that involve folding/unfolding.

In order to give better results, we assume that grammar comparison operates on a slightly normalized grammar format. The assumed, straightforward normalization rules are presented in Appendix A.

Let us consider a simple example, without paying attention yet to the specific grammar notation and transformation operators. For instance, consider the two grammar excerpts from the "more readable" grammars of JLS2 and JLS3 (*read2* and *read3* from now on) in Figure 1. Conceptually, the grammars are different in the following manner. The *read3* grammar covers additional syntax for enumeration declarations; it also uses an auxiliary nonterminal *NormalClassDeclaration* for the class-declaration syntax that is declared directly by *ClassDeclaration* in the *read2* grammar. The comparator reports four differences that are directly related to these observations:

- Nominal differences:
  - read2: nonterminal *NormalClassDeclaration* missing.
  - read2: nonterminal *EnumDeclaration* missing.
- Structural differences:
  - Nonterminal *ClassDeclaration*: no matching alternatives
    (counts as 2 because the definitions have a maximum of 2 alternatives).

Arguably, these differences should help the grammar engineer who will typically try to find definitions for missing nonterminals by extracting their inlined counterparts. The counterpart for *NormalClassDeclaration* is relatively obvious because of the combination of a nonterminal that is entirely missing in one grammar while it occurs in a structural different and unmatched alternative in the other grammar.

2.2 **Grammar transformation**

Since the goal of grammar convergence is to relate all sources to each other, the relationships between grammars will be represented as grammar transformations. We say that grammars g_1 and g_2 are f-equal, if $f(g_1) = g_2$ (where "=" refers to structural equality on grammars, and f denotes the meaning of a grammar transformation). When f is a refactoring (i.e., a semantics-preserving transformation), then f-equality coincides with grammar equivalence. If f is a semantics-increasing (-decreasing) transformation, then we have shown an inclusion ordering for the languages generated by the two grammars.

We use the terms "semantics-preserving", "-increasing" and "-decreasing" in the formal sense of the language generated by a grammar. Clearly, the composition of (sufficiently expressive) increasing and decreasing operators allows us to relate arbitrary grammars, in principle. Hence, more restrictions are needed for accumulating reasonable grammar relationships, as we will discuss below. We also mention that there is a rare need for operators that are neither semantics-increasing nor -decreasing. In this case, we speak of a semantics-revising operator. Consider, for example, an unconstrained **replace** operator for expressions in grammar productions that may be needed if we face conflicting definitions of a nonterminal in two given grammars.

The baseline scenario for grammar transformation in the context is convergence is as follows. Given are two grammars: g_1 and g_2. The goal is to find f such that g_1 and g_2 are f-equal. In this case, one has to gradually aggregate f by addressing the various differences reported by the comparator. In our current implementation of grammar comparison, we do not make any effort to propose any transformation operators to the user, but this is clearly desirable and possible.

In JLS, given the differences reported by the comparator and presented in the previous section, the grammar engineer authors an transformation to add an extra chain production for *NormalClassDeclaration*. This transformation and a few subsequent ones as well as all intermediate results are listed in Figure 2.

The idea is now that such compare/transformation steps are repeated. Hence, we compare the intermediate result, as obtained above, with the grammar *read3*. It is clear that the nominal difference for *NormalClassDeclaration* has been eliminated. The comparator reports the three following differences:

Using a grammar transformation operator to introduce a chain production

```
chain(
 ClassDeclaration:
        NormalClassDeclaration );
```

read2’ (Intermediate grammar fragment)

```
ClassDeclaration:
        NormalClassDeclaration
NormalClassDeclaration:
        ClassModifiers? "class" Identifier Super? Interfaces? ClassBody
```

Using grammar transformation operators to add enumeration declarations

```
introduce( ... );
addV(
 ClassDeclaration:
        EnumDeclaration );
```

read2’’ (Intermediate grammar fragment)

```
ClassDeclaration:
        NormalClassDeclaration
        EnumDeclaration
NormalClassDeclaration:
        ClassModifiers? "class" Identifier Super? Interfaces? ClassBody
EnumDeclaration:
        ...
```

Using a grammar transformation operator to insert a nillable symbol

```
appear(
 NormalClassDeclaration:
        ClassModifiers? "class" Identifier <TypeParameters?> Super? Interfaces? ClassBody );
```

Fig. 2 Transforming the grammar and proving (**chain** $\circ$ **introduce** $\circ$ **addV** $\circ$ **appear**)-equality.

- Nominal difference:
  - read2’: nonterminal *EnumDeclaration* missing.
- Structural difference: nonterminal *NormalClassDeclaration*
  - read2’: *ClassModifiers*? `"class"` *Identifier Super*? *Interfaces*? *ClassBody*
  - read3: *ClassModifiers*? `"class"` *Identifier TypeParameters*? *Super*? *Interfaces*? *ClassBody*
- Structural difference: nonterminal *ClassDeclaration*
  - Unmatched alternatives of read2’: none
  - Unmatched alternatives of read3: *EnumDeclaration*

We see that enumerations are missing entirely from *read2'*, and hence a definition has to be introduced, and a corresponding alternative has to be added to *ClassDeclaration*. Once we are done, the result is again compared to *read3*:

- Structural difference: nonterminal *NormalClassDeclaration*
  - read2'': *ClassModifiers*? `"class"` *Identifier Super*? *Interfaces*? *ClassBody*
  - read3: *ClassModifiers*? `"class"` *Identifier TypeParameters*? *Super*? *Interfaces*? *ClassBody*

Again, this difference is suggestive. Obviously, the definition of *NormalClassDeclaration* according to *read2"* does not cover the full generality of the construct, as it occurs in *read3*. The structural position for the type parameters of a class has to be added. (This has to do with Java generics which were added in the 3rd edition of the JLS.) There is a designated transformation operator that makes new components **appear** (such as type parameters) in existing productions; the newly inserted part is marked on Figure 2 with angle brackets. This is a downward-compatible change since type parameters are optional. Once these small transformations have been completed, all the discussed differences are resolved, and the comparator attests structural equality.

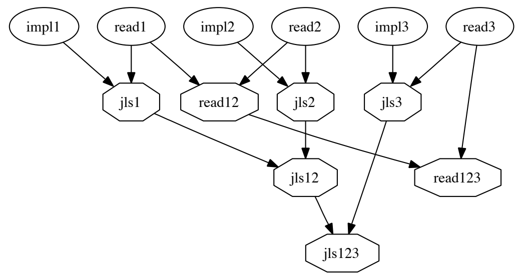
2.3 Convergence graphs

Grammar convergence always starts from the grammars that were extracted from the given software artifacts, to which we refer as *source grammars* or *sources* subsequently. In the present JLS study, we face 6 sources; we use *read1–read3* to refer to the "more readable" grammars, and *impl1–impl3* to refer to the "more implementable" grammars. It is reasonable to relate grammars through an additional grammar of which we think as the common denominator of the original grammars. We refer to such additional grammars as *targets*. The "distance" between source and target grammars may differ. In fact, it is not unusual, that one source—modulo minor transformations only—serves as common denominator.

The idea of the common denominator can be generalized such that we actually devise a *directed acyclic graph* with grammars as the nodes and transformations as the edges. In the trivial case with each target being a result of transforming two sources or targets, we will have a *binary tree*. The root of such a tree (the final target) is the common denominator of all grammars, but there may be additional intermediate targets that already serve as common denominators for some of the grammars. (We use arrows to express the direction of the transformation, and hence the trees appear inverted, when compared to common sense of drawing trees.) The source grammars are the leaves of such a tree.

Figure 3 shows the "convergence tree" for the present JLS case study. The original grammars from the JLS documents are located at the top. The tree states 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 further "converged" to account for inter-version differences—the extensions to the Java language in particular (see the nodes *jls12* and *jls123* as well as *read12* and *read123* in the figure). For the JLS we use a binary tree, which means that we always limit the focus to two grammars, and hence a cascade is needed, if more than two grammars need to be converged.

When deriving *jls1–3*, we favor the "more implementable" grammar as the target of convergence, i.e., as the common denominator—except that some corrections may need to be applied, or some minimum restructuring is applied for the sake a more favorable grammar structure. This preference reflects the general rule that an implementation-oriented artifact



impl1	JLS1, §19
read1	JLS1, §§4.1–15.27
impl2	JLS2, §18
read2	JLS2, §§4.1–15.28
impl3	JLS3, §18
read3	JLS3, §§4.1–15.28

Fig. 3 The convergence graph for the JLS grammars consists of two binary trees with shared leaves. 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 forking node is derived from two grammars. The *implX* leaves are "implementable" grammars, the *readX* ones are "readable".

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*) as a sort of sanity check. It turns out that *read1–3* are structurally quite similar, and accordingly, the additional cascade requires little effort.

2.4 **Convergence process**

As we were discussing grammar comparison and transformation, we already alluded to a basic compare/transform cycle—this cycle is indeed the spline of the convergence process. We identify *phases* for the convergence process in order to impose more structure and discipline onto the process. These convergence phases assume asymmetric, binary convergence trees where one of the two grammars is favored as (near-to) common denominator—as we discussed above. There are five consecutive convergence phases: the initial extraction phase involves a mapping from an external grammar format and is therefore implemented as a standalone tool in our infrastructure; the other four convergence phases are directly concerned with transformation.

Extraction: A starting point for grammar extraction is always a set of real grammar artifacts. A mapping is required for each kind of artifact so that grammar knowledge can be extracted and represented in a uniform grammar format. (In in the case of our infrastructure, we use BGF—a BNF-like Grammar Format.) Each extractor may implement particular design decisions in terms of any normalization or abstraction to be performed along with extraction. Once extraction is completed, a (possibly incorrect or not fully interconnected) grammar is ready for transformation.

Convergence preparation: This convergence phase involves correcting immediately obvious or a priori known errors in the given sources. These corrections are represented as grammar transformations so that they can be easily revisited or re-applied in the case when the extractor is modified or the source changes. In the JLS case, we incorporated an available bug list at this stage[3]. Some inaccuracies caused by representation anomalies in the HTML input were also resolved a this stage. Further, we added some missing definitions the lack of which was discovered through an early inspection; see the discussion of bottom nonterminals in §3.5.

Nominal matching: We perform asymmetric compare/transform steps. That is, the non-favored grammar is compared with the (prepared) favored grammar, which is the baseline for the (intermediate) target of convergence. The objective of this convergence phase is to align the syntactic categories of the grammars in terms of their nonterminals. The nominal differences, as identified by comparison, guide the grammar engineer in drafting transformations for renaming as well as extraction and inlining such that the transformations immediately reduce the number of nominal differences. It is important to notice that we restrict ourselves to operators for renaming, inlining, and extraction. These operators convey our intuition of (initial) nominal alignment. We make these assumptions:

- *When a nonterminal occurs in both grammars, then it models the same syntactic category (conceptually).* If the assumption does not hold, then this will become evident later through considerable structural differences, which will trigger a renaming to resolve the name clash. Such corrective renaming may be pushed back to the phase of convergence preparation.
- *Any renaming for nonterminals serves the purpose of giving the same name to the same syntactic category (in an conceptual sense).* If a grammar engineer makes a mistake, then this will become evident later, again, through considerable structural differences. In this case, we assume that the grammar engineer returns to the name matching phase to revise the incorrect match.

Structural matching: We continue with asymmetric compare/transform steps. This convergence phase dominates the transformation effort; it aligns the definitions of the nonterminals in a structural sense. The structural differences, as identified by comparison, guide the grammar engineer in drafting transformations for refactoring such that they immediately reduce the number of structural differences. As we continue to limit ourselves to refactoring, the order of the individual transformations does not matter due to its commutativity. The grammar engineer can simply pick any applicable refactoring operator, but the firm requirement is that the number of structural and nominal differences declines, which is automatically verified by our infrastructure.

Resolution: This convergence phase consists of three kinds of steps, as discussed in more detail in §4: *extension*, *relaxation* and *correction*. In the case of semantics-increasing operators, it is up to the grammar engineer to perform the classification. Semantics-decreasing operators serve correction on the grounds of a convention. That is, we assume a directed process of convergence where the grammars of extended (relaxed) languages are derived from the grammars of "sublanguages". However, if the grammars violate such an intended sublanguage relationship, then correction must be expressed through semantics-decreasing operators.

The correctness of the process relies on one assumption regarding the limited use of non-semantics-preserving operators. In particular, non-semantics-preserving operators should

[3] 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 was also endorsed by Oracle: `http://www.cmis.brighton.ac.uk/staff/rnb/bosware/javaSyntax/syntaxV2.html`. We refer to this list as "known bugs" in our process.

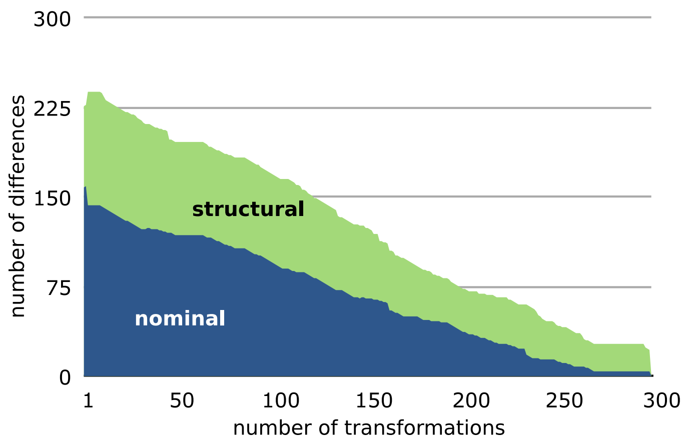


Fig. 4 Difference reduction for *read2* towards the convergence target *jls2* in the convergence tree of Figure 3.

only be used, if the given grammars are not equivalent. Making equivalent grammars non-equivalent is clearly not desirable. Currently, we cannot verify this assumption, and in fact, it is generally impossible because of undecidability of grammar equivalence. However, a heuristic approach may be feasible, and provides an interesting subject for future work. Even when the given grammars are non-equivalent, we still need to limit the use of non-semantics-preserving operators for correctness' sake. That is, we should disallow zigzag transformations such that semantics-increasing and -decreasing transformations partially cancel each other.

We use the number of nominal and structural differences as means to **track progress** of grammar convergence. Each unmatched nonterminal symbol of either grammar counts as a nominal difference. For every nominally matched nonterminal, we add the maximum number of unmatched alternatives (of either grammar), if any, to the number of structural differences.

The main guiding principle for grammar convergence is to consistently reduce the number of grammar differences throughout the two matching convergence phases as well as the final resolution phase. Figure 4 illustrates this principle for one edge in the convergence graph of the present JLS study. The figure also visualizes that nominal differences tend to be resolved earlier than structural differences.

Our transformation infrastructure is aware of the different phases of convergence, and it checks the incremental reduction of differences at runtime. As a concession to a simple design of the operator suite for grammar transformations, restructuring steps may also slightly increase structural differences as long as they are explicitly grouped in "transactions" whose completion achieves reduction.

3 Grammar extraction

The previous section decomposed grammar convergence essentially into grammar extraction and a compare/transform cycle. The present section will focus on extraction, whereas the next section covers grammar transformations to be used in the compare/transform cycle.

(Here we assume that our grammar comparison approach is currently trivial and not worth a designated, detailed description.)

The central objective of grammar extraction is to map software artifacts of a given kind (such as parser descriptions, language documentation, or XML schemata) to the uniform grammar format that is used in a convergence effort. Several engineering issues arise in this context:

Domain analysis. One needs to understand the software artifacts at hand. One dimension of understanding may be based on metadata about the grammars at hand: version, style, completeness. Such information can be often obtained by inspecting the given grammar artifacts, e.g., language documents. Consider, for example, encountering of a left-recursive style. Awareness of this style is beneficial for the transformations to be devised eventually.

Source selection. There may be multiple potential candidates (think of PDF vs. HTML for language documentation, or Java sources vs. byte code for object models). Hence, trade-offs regarding the simplicity, robustness, correctness, and completeness of extraction must be considered. Either one contender is chosen, or multiple options are explored in parallel, or a fallback is considered, if the contender fails, eventually.

Extractor implementation. In our experience, most extractors are unique programs—they require particular programming techniques, specifically parsing techniques. Seeking the right implementation strategy is a matter of trial-and-error, but, ultimately, it is important to be able to describe a lucid implementation strategy so that one can have trust in the robustness, correctness, and completeness of extraction.

Metrics assessment. One should evaluate the initial quality of the extracted grammar based on common grammar metrics for bottom nonterminals ("undefined nonterminals") and top nonterminals ("unused nonterminals"). Such quality properties are helpful in guiding subsequent transformation efforts. Also, quality issues may indicate flaws in the extractor logic, and hence trigger reconsideration and revision.

These issues will be addressed in the following text.

3.1 **JLS domain analysis**

We have already begun capturing JLS terminology, recall the notions of "more readable" and "more implementable". These notions are not sharply defined, but one can think of, for example, *left factoring* (to help with look ahead) as being used in the more implementable grammars but not in the more readable grammars. Let us extract related characteristics of the grammars from the JLS documents on a per-grammar basis:

JLS1 It is stated (Gosling et al, 1996, §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 (Gosling et al, 1996, §2.3) that *read1* is "*very similar to*" *impl1* "*but more readable*".

JLS2 The second edition of the JLS (Gosling et al, 2000, "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

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

Table 1 Basic properties of the JLS grammars.

as different developments. Also, the LALR(1) claim for *impl1* is not matched by *impl2* which does not list a grammar-class claim. However, *impl2* is said (Gosling et al, 2000, §18) to be “*the basis for the reference implementation*”.

JLS3 JLS3 extends JLS2 in numerous ways (Gosling et al, 2005, 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 (Gosling et al, 2005, §18) to be “*not an LL(1) grammar, though [...] it minimizes the necessary look ahead.*” Hence, *impl3* has definitely departed from *impl1* with its associated grammar class LALR(1).

In addition to grammar class claims for the JLS grammars we have also recorded iteration styles during cursory examination; see Table 1. 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.

3.2 **JLS source selection**

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. After some deliberation, we have opted for the HTML format because parsing seemed relatively straightforward.

Obviously, the JLS grammars have been implemented by different parties in various ways. For instance, there exist parser descriptions whose authors have consulted the JLS. However, as a matter of principle, none of these options was considered appropriate in the present JLS study because we wanted to make sure to perform consistency checking for the primary JLS as opposed to any derived artifact. Hence, we started from the JLS (and its HMTL documents, in particular), even if such a source selection required more effort than a path that reuses third-party Java grammars.

3.3 **JLS grammar notation**

The extractor needs to identify grammar portions within general HTML markup. The used grammar format slightly varies across the different JLS grammars and versions; there are relevant formatting rules in different documents and sections—in particular from Gosling et al (1996, §2.4), Gosling et al (2000, §2.4, §18) and Gosling et al (2005, §2.4, §18).

Grammar fragments are hosted by `<pre>...</pre>` blocks in the JLS documents. According to Gosling et al (1996, 2000, 2005, §2.4): terminal symbols are shown in fixed font (as in `<code>class</code>`); nonterminal symbols are shown in italic type (as in `<i>Expression</i>`); a subscripted suffix "opt" indicates an optional symbol (as in `Expression<sub>opt</sub>`); alternatives start in a new line and they are indented; "one of" marks a top-level choice with atomic branches. (We have also observed that nonterminals are expected to be alphanumeric and start in upper case.) Further notation and expressiveness is described in Gosling et al (2000, 2005, §18): $[x]$ denotes zero or one occurrences of x; $\{x\}$ denotes zero or more occurrences of x; $x_1|\cdots|x_n$ forms a choice over the x_i. The JLS documents consistently suffice with "*" lists (zero or more occurrences); there are no uses of "+" lists. Refer to Figure 5 for a summary of the assumed source grammar notation. Refer to Figure 6 for a summary of the notation we use for the examples in this paper. All examples presented here were obtained from their XML (BGF) form in an automated generative manner as in Kort et al (2002).

We should also mention line continuation; it allows to spread one alternative over several lines (Gosling et al, 2005, §2.4): "*A very long right-hand side may be continued on a second line by substantially indenting this second line*". In our notation we double the indentation for every continued line.

Example 1 A grammar fragment as of Gosling et al (2000, §4.2):

```
<i>NumericType:
      IntegralType
      FloatingPointType

IntegralType: one of</i>
      <code>byte short int long char
</code>
```

It should be parsed as:

```
NumericType:
      IntegralType
      FloatingPointType

IntegralType:
      "byte"
      "short"
      "int"
      "long"
      "char"
```

The fragment illustrates two different kinds of "choices", i.e., multiplicity of vertical alternatives, and "one of" choices. (The third form, which is based on "|", is not illustrated.) The fragment also clarifies that markup tags are used rather liberally. The "nonterminal" tag (i.e., `<i>...</i>`) spans more than one production. The terminal tag (i.e., `<code>...</code>`) spans several terminals and the closing tags ends up on a new line.

```
Production:
        Nonterminal ":" [ "one" "of" ] CR Line { Line } CR
Line:
        Indent Symbols CR
Symbols:
        Symbol { Symbol }
Symbol:
        Nonterminal
        Terminal
        "(" Symbols "|" Symbols { "|" Symbols } ")"
        "[" Symbols "]"
        "{" Symbols "}"
CR:
        ... carriage return ...
Indent:
        ... indentation ...
```

Fig. 5 Relevant expressiveness of the JLS grammar notation, given in a self-descriptive manner; for clarity, terminals are enclosed in double quotes as opposed to the use of markup; the markup-based form of optional symbols is also omitted.

```
grammar:
        root::STRING* production*
label:
        "[" STRING "]"
production:
        label::label? nonterminal::STRING ":" CR right−hand−side
right−hand−side:
        (INDENT symbol+ CR)+ CR
symbol:
        "ε "
        "EMPTY"
        "ANY"
        "STRING"
        "INT"
        terminal::(""" STRING """)
        nonterminal::STRING
        selectable::(selector::STRING "::" symbol)
        sequence::("(" symbol+ ")")
        choice::("(" (symbol ("|" symbol)*) ")")
        optional::(symbol "?")
        plus::(symbol "+")
        star::(symbol "*")
        marked::("<" symbol ">")
STRING:
        ... any letter sequence ...
CR:
        ... carriage return ...
INDENT:
        ... indentation ...
```

Fig. 6 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; operators “?”, “*” and “+” serve for optionality and lists; elisions are shown as “...”.

Example 2 Not only the indentation is incorrect in the following fragment, it is also the only place where the subscript “opt” is capitalized (Gosling et al, 2005, §4.5.1):

```
Wildcard:
? WildcardBounds<sub>Opt</sub>
```

3.4 JLS extractor implementation

The tiny Example 1 is a good indication of the many irregularities that are found in the HTML representation, such as volatile use of markup tags, liberal indentation, duplicate definitions. We needed to design and implement a non-classic grammar parser to extract and analyze the grammar segments of the documents and to perform recovery. Our extractor therefore deals with the expected irregularities in several phases.

- Phase 1—Preprocessing: the tool takes an HTML formatted text and filters out all hypertext tags and indentation, extracting all possible information from them in the process.
- Phase 2—Error recovery: the recovery rules are applied until they are no longer applicable. There are rules for transforming a terminal symbol to a nonterminal symbol or the other way around, matching up parentheses, splitting/combining sibling symbols, etc.
- Phase 3—Removal of doubles: duplicate definitions are purged. This could not happen during earlier extraction phases because clones could differ in markup.
- Phase 4—Precise parsing: the extracted grammar is serialized to some parseable form. We use the XML-based interchange format called BGF, or BNF-like Grammar Format.

The extraction phases are discussed in detail in the following subsections.

Extraction phase 1—Preprocessing

The first extraction phase, which we call a preprocessing phase, has the following I/O behavior:

- Input: the `<pre>...</pre>` blocks.
- Output: a dictionary
  - Keys: Left-hand side nonterminals
  - Values: Arrays of top-level alternatives

The phase is subject to the following requirements:

Tag elimination. The input notation interleaves tags with proper grammar structure. In order to prepare for classic parsing, we need to eliminate the tags in the process of constructing properly typed lexemes for terminals and nonterminals.

Indentation elimination. The input notation relies on indentation to express top-level choices and line continuation. The output format stores top-level choices in arrays, and fuses multi-line alternatives.

Robustness. The inner structure of top-level alternatives is parsed simply as a sequence of tokens in the interest of robustness so that recovery rules can be applied separately, before, finally, the precise grammar structure is parsed.

	italic	`fixed`	default
Alphanumeric	N (2341)	T (173)	T? (194)
|	M (2)	T (2)	M? (29)
{,},[,],(,)	M (708)	T (174)	T? (200)
otherwise	T (198)	T (165)	T (205)

(T — terminal, N — nonterminal, M — metasymbol)

Table 2 Decision table of the extractor's scanner. Classes of strings are rows, scanner states are columns.

The preprocessor relies on a stateful scanner (to meet "tag elimination") and a robust parser (to meet "robustness"). The parser recognizes sequences of productions, each one essentially consisting of a sequence of alternatives; it parses alternatives as sequences of tokens terminated by CR. The scanner uses three states:

– **italic** upon opening `<i>` tag (or `<em>`)
– **fixed** upon opening `<code>` tag
– **default** when no tag is open

That is, we treat each tag as a special token that changes the global state of the scanner, which in turn can be observed when creating morphemes for terminals and nonterminals. We also deal with violations of XML and HTML well-formedness in this manner. The decision table of the scanner is presented in section 3.4 along with the number of times each decision is taken for all JLS documents.

Most of these decisions are inevitable, even though some of them pinpoint markup errors. An example of an "error-free" decision is to map an alphanumeric string in the italic mode to a nonterminal. An example of an "error-recovering" decision is to map a non-alphanumeric token (that does not match any metasymbol) to a terminal—even when it is tagged with `<i>...</i>`. Several decisions in the "default" column involve an element of choice (as indicated by "?"). The shown decisions give the best results, that is, they require the least subsequent transformations of the extracted grammar. For instance, it turned out that bars without markup were supposed to be BNF bars, but other metasymbols were better mapped to terminals, whenever markup was missing. Also, alphanumeric strings without markup turned out to be mostly terminals, and hence that preference was implemented as a decision by the scanner.

Extraction phase 2—Error recovery

We face a few syntax errors with regard to the syntax of the grammar notation. We also face a number of "obvious" semantic errors in the sense of the language generated by the grammar. We call them obvious errors because they can be spotted by simple, generic grammar analyses that involve only very little Java knowledge, if any. We have opted for an error-recovery approach that relies on a uniform, rule-based mechanism that performs transformations on each sequence of tokens that corresponds to an alternative. The rules are applied until they are no longer applicable. We describe the rules informally; they are implemented in Python by regular expression matching.

Rule 1 (Match up parentheses) *When there is a group (a bar-based choice) that misses an opening or closing parenthesis, such as in "* `(a|b`*", then a nearby terminal "(" or ")" (if available) is to be converted to the parenthesis, as in Example 3. If there is still a closing parenthesis that cannot be matched, then it is dropped, as in Example 4. We have not* *seen the case of an opening parenthesis to remain unmatched, but the rule is implemented symmetrically for the sake of completeness.*

Example 3 A grammar production from Gosling et al (2005, §18.1): the symbols for closing bracket and parenthesis need to be converted to metasymbols to match the opening bracket and parenthesis:

```
TypeArgument:
        Type
        "?" [ ( "extends" | "super" ")" "Type" "]"
```

Example 4 A grammar production from Gosling et al (2000, §18.1) and Gosling et al (2005, §18.1): a non-matching square bracket has to be removed:

```
Expression:
        Expression1 [ AssignmentOperator Expression1 ] ]
```

Rule 2 (Metasymbol to terminal) *(a) When "|" was scanned as a BNF metasymbol, but it is not used in the context of a group, then it is converted to a terminal, as in Example 5.*

(b) When "[" and "]" occur next to each other as BNF symbols, then they are converted to terminals, as in Example 6.

(c) When "{" and "}" occur next to each other as BNF symbols, then they are converted to terminals. (Not encountered so far, implemented for the sake of consistency).

(d) When an alternative makes use of the metasymbols for grouping, but there is no occurrence of the metasymbol "|", then the parentheses are converted to terminals, as in Example 7.

Example 5 A grammar production from Gosling et al (2000, §15.22): there is no group, so the bar here is not a metasymbol, but a terminal:

```
InclusiveOrExpression:
        ExclusiveOrExpression
        InclusiveOrExpression | ExclusiveOrExpression
```

Example 6 A grammar production from Gosling et al (2000, §8.3): there is nothing to be made optional, so the square brackets here are not metasymbols, but terminals:

```
VariableDeclaratorId:
        Identifier
        VariableDeclaratorId [ ]
```

Example 7 A grammar production from Gosling et al (2000, §14.19) and Gosling et al (2005, §18.1): there is no choice inside the group so the parentheses here are not metasymbols, but terminals:

```
CatchClause:
        "catch" ( FormalParameter ) Block
```

Rule 3 (Compose sibling symbols) *When two alphanumeric nonterminal or terminal tokens are next to each other where one of the symbols is of length 1, then they are composed as one symbol, as in Example 8 and Example 9.*

Example 8 Multiple terminals to compose (Gosling et al, 1996, §19.11):

```
<code>continu</code><i>e
```

Example 9 Multiple nonterminals to compose (Gosling et al, 1996, §14.9):

```
S<i>witchBlockStatementGroups</i>
```

Rule 4 (Decompose compound terminals) *When a terminal consists of an alphanumeric prefix, followed by ".", possibly followed by a postfix, then the terminal is taken apart into several ones, as in Example 10.*

Example 10 Consider this phrase (Gosling et al, 2000, §15.9):

```
Primary.new Identifier ( ArgumentListopt ) ClassBodyopt
```

The decomposition results in the following:

```
Primary . new Identifier ( ArgumentListopt ) ClassBodyopt
```

Rule 5 (Nonterminal to terminal) *Lower-case nonterminals that are not defined by the grammar (i.e., that do not occur as a key in the dictionary produced during extraction phase 1), and are in lower case, are converted to terminals, as in Example 11.*

Example 11 A grammar production from Gosling et al (2000, §14.11): `default` needs to be converted to a terminal:

```
SwitchLabel:
      </em>case<em> ConstantExpression :
      default :
```

The same error is present in the later version of the specification (Gosling et al, 2005, §14.11):

```
SwitchLabel:</em>
      case <em>ConstantExpression </em>:
      case <em>EnumConstantName </em>:<em>
      default :
```

Note the changes in JLS3: a new alternative was added and the colon was correctly marked up as a terminal symbol. However, "`default`" is still incorrectly marked up as a nonterminal.

Rule 6 (Terminal to nonterminal) *Alphanumeric terminals that start in upper case, and are defined by the grammar (when considered as nonterminals) are converted, as in Example 12.*

Example 12 A grammar production from Gosling et al (2000, §7.5):

```
<em>ImportDeclaration</em>:
      SingleTypeImportDeclaration
      TypeImportOnDemandDeclaration
```

The decisive definitions are found in Gosling et al (2000, §7.5.1, §7.5.2):

```
SingleTypeImportDeclaration:
      "import" TypeName ";"
TypeImportOnDemandDeclaration:
      "import" PackageOrTypeName "." "*" ";"
```

Rule 7 (Recover optionality) *When a nonterminal's name ends on "opt", as in "fooopt", and the grammar defines a nonterminal "foo", then the nonterminal "fooopt" is replaced by* [*foo*]. *(Hence, markup for the subscript "opt" was missing.)*

Example 13 Consider again the result of Example 10:

```
Primary . new Identifier ( ArgumentListopt ) ClassBodyopt
```

After recovery it will be parsed as:

```
ClassInstanceCreationExpression:
      Primary "." "new" Identifier "(" ArgumentList? ")" ClassBody?
```

These are all the rules that have stabilized over the project's duration. Several other rules where investigated but eventually abandoned because the corresponding issues could be efficiently addressed by grammar transformations. We used experimental rules to test for the recurrence of any issue we had spotted. We quantify the use of the rules shortly.

Extraction phase 3—Removal of doubles

The JLS documents (deliberately) repeat grammar parts. Hence, we have added a trivial extraction phase for removal of double alternatives. That is, when a given right-hand side nonterminal is encountered several times in a source, then extraction phase 1 accumulates all the alternatives via one entry of the dictionary, and extraction phase 3 compares alternatives (i.e., sequences of tokens) to remove any doubles.

Example 14 Recall the following definition from Example 6 (Gosling et al, 2000, §8.3):

```
VariableDeclaratorId:
      Identifier
      VariableDeclaratorId [ ]
```

The same definition appears elsewhere in the document, even though the markup is different, but these differences were already neutralized during extraction phase 1 (Gosling et al, 2000, §14.4):

	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 3 Basic metrics of the JLS grammars.

```
<em>VariableDeclaratorId:
      Identifier
      VariableDeclaratorId</em> [ ]
```

Extraction phase 3 preserves 2 alternatives out of 4. As an aside, this particular example also required the application of Rule 2.b because `[ ]` must be converted to terminals.

Extraction phase 4—Precise parsing

Finally, the dictionary structure of extraction phase 1, after the recovery of extraction phase 2, and double removal of extraction phase 3, is trivially parsed according to the (E)BNF for the grammar notation, as presented on Figure 5. In fact, our implementation dumps the extracted grammar immediately in an XML-based grammar interchange format so that generic grammar tools for comparison and transformation can take over (Lämmel and Zaytsev, 2009).

3.5 **JLS grammar metrics**

Table 3 displays simple grammar metrics for the extracted JLS grammars. A *top nonterminal* is a nonterminal that is defined but never used; a *bottom nonterminal* is a nonterminal that is used but never defined (Lämmel and Verhoef, 2001b; Sellink and Verhoef, 2000). Through continued domain analysis, we have understood that the major differences between the numbers of productions and nonterminals for the two grammars of any given version are 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 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.

	impl1	impl2	impl3	read1	read2	read3	Total
Arbitrary lexical decisions	2	109	60	1	90	161	423
Well-formedness violations	5	0	7	4	11	4	31
Indentation violations	1	2	7	1	4	8	23
Recovery rules	3	12	18	2	59	47	141
◦ Match parentheses	0	3	6	0	0	0	9
◦ Metasymbol to terminal	0	1	7	0	27	7	42
◦ Merge adjacent symbols	1	0	0	1	1	0	3
◦ Split compound symbol	0	1	1	0	3	8	13
◦ Nonterminal to terminal	0	7	3	0	8	11	29
◦ Terminal to nonterminal	1	0	1	1	17	13	33
◦ Recover optionality	1	0	0	0	3	8	12
Purge duplicate definitions	0	0	0	16	17	18	51
Total	11	123	92	24	181	238	669

Table 4 Irregularities resolved by grammar extraction.

3.6 **JLS extractor statistics**

Consider Table 4 as an attempt to measure either the effort needed to complete extraction (manually) or the degree of inconsistency of the input format. The table summarizes the frequencies of using recovery rules, handling "unusual" continuation lines[4], and removal of doubles. The extractor has fixed 669 problems that otherwise would have prevented straightforward parsing to succeed with extraction, or implied loss of information, or triggered substantial grammar transformations.

4 **Grammar transformation**

In this section we provide an overview of the major language-independent operators that are needed for the transformation of concrete syntax definitions in the context of grammar convergence, we illustrate intended and accidental differences between the JLS grammars and their representation as operational grammar transformations, and we summarize the application of our operator suite to the present JLS study.

4.1 **Operator suite**

Just as in Lämmel and Zaytsev (2009), we distinguish here among semantics-preserving, semantics-increasing, semantics-decreasing and semantics-editing operators. The term semantics refers to the language generated by the grammar—when considered as a set of strings.

The complete grammar of our grammar transformation language is presented on Figure 7. As we see, many operators have the form of either $f(n)$, where f is the operator in question, and n is the nonterminal to be affected, or $f(x, y)$, where f is the operator in question, x is the grammar expression to be located in the input, and y is the corresponding replacement. There are also several operators that are parametrized with a so called "marked

[4] Our initial guess was that "substantially indenting" means more spaces or tabs than the previous line, but some cases were discovered when continuation lines were not indented at all.

```
sequence:
    (transformation | atomic)⋆
atomic:
    transformation+
transformation:
    folding−unfolding−transformation
    refactoring−transformation
    increasing−transformation
    decreasing−transformation
    concrete−revising−transformation
    abstract−revising−transformation
    renaming−transformation
    decorative−transformation
    reroot::(root::nonterminal⋆)
    dump::ε
folding−unfolding−transformation:
    unfold::(nonterminal in::scope?)
    fold::(nonterminal in::scope?)
    inline::nonterminal
    extract::(production in::scope?)
    abridge::production
    detour::production
    unchain::production
    chain::production
refactoring−transformation:
    massage::(expression expression in::scope?)
    distribute::scope
    factor::(expression expression in::scope?)
    deyaccify::nonterminal
    yaccify::(production+)
    eliminate::nonterminal
    introduce::(production+)
    import::(production+)
    vertical::scope
    horizontal::nonterminal
    equate::(align::nonterminal with::nonterminal)
    rassoc::production
    lassoc::production
increasing−transformation:
    addV::production
    addH::marked−production
    appear::marked−production
    widen::(expression expression in::scope?)
    upgrade::(marked−production production)
    unite::(add::nonterminal to::nonterminal)
decreasing−transformation:
    removeV::production
    removeH::marked−production
    disappear::marked−production
    narrow::(expression expression in::scope?)
    downgrade::(marked−production production)
concrete−revising−transformation:
    abstractize::marked−production
    concretize::marked−production
    permute::production
abstract−revising−transformation:
    define::(production+)
    undefine::(nonterminal+)
    redefine::(production+)
    inject::marked−production
    project::marked−production
    replace::(expression expression in::scope?)
renaming−transformation:
    renameL::(from::label to::label)
    renameN::(from::nonterminal to::nonterminal)
    renameS::(in::label? from::selector to::selector)
    renameT::(from::terminal to::terminal)
decorative−transformation:
    designate::production
    unlabel::label
    deanonymize::marked−production
    anonymize::marked−production
marked−production:
    production
scope:
    label | nonterminal
```

Fig. 7 The grammar of the transformation language. Metasyntax nonterminals like *production* and *expression* are deliberately omitted for clarity.

production"—a grammar production that has parts of it specifically marked as local transformation targets.

Folding, unfolding and refactoring operators preserve the string-oriented semantics. Many of them are implemented as conditional replacements. For instance, **massage** has a pre-condition that the two expressions that parametrize it are $massage$-equal. If the pre-condition is satisfied, the former expression is replaced with the latter; if it fails, the transformation chain stops with an error reported. The $massage$-equality is defined by a set of algebraic laws listed in Appendix B for completeness' sake.

Below we briefly explain the operators that are needed to understand the present JLS study; see also Table 7.

– **unfold** replaces all occurrences of a given nonterminal by its definition, possibly limited by scope.
– **fold** replaces all occurrences of a given nonterminal's definition by the nonterminal itself, possibly limited by scope.

- **inline** is the same as **unfold**, but it also removes the nonterminal from the grammar after unfolding (if some recursion prevents it, the operator fails).
- **extract** introduces a new nonterminal to the grammar and **fold**s its definition.
- **chain** introduces a chain production by performing a local **extract** on the whole definition of a given nonterminal (i.e., it goes from $a : xyz$ to $a : b$ and $b : xyz$).
- **massage** allows for rewriting grammar expressions according to the algebraic laws listed in Appendix B.
- **distribute** pulls the choices that occur inside a grammar expression outwards.
- **factor** rewrites an expression to an equivalent but differently factored one. Its common use is to push choices deeper since an automated **distribute** operator is more useful in other cases.
- **deyaccify** converts a recursive definition-based style of iteration to the use of the EBNF operators "+" and "*". The operator name is justified by de Jonge and Monajemi (2001) and Lämmel (2001).
- **yaccify** is the intended inverse of **deyaccify**.
- **eliminate** removes a production from the grammar if it is not used anywhere, otherwise fails.
- **introduce** adds a free nonterminal with its definition to the grammar.
- **import** works as multiple **introduce** operators for introducing possibly interconnected productions.
- **vertical** converts a "horizontal" production with top-level choices (previously called "flat" by Lämmel and Wachsmuth (2001)) to several productions.
- **horizontal** converts a "vertical" nonterminal definition consisting of multiple productions (previously called "non-flat" by Lämmel and Wachsmuth (2001)) to a single production with top-level choices.
- **addV** adds a production to the existing nonterminal definition.
- **addH** replaces any expression with a choice involving that expression and something else (i.e., it goes from $a : xyz$ to $a : x(y|b)z$).
- **appear** injects a nillable symbol (the one that can be evaluated to ε).
- **widen** replaces an expression with a more general one (e.g., x^+ with $x^\star$).
- **upgrade** replaces an expression with a nonterminal that can possibly be evaluated to it.
- **unite** merges the definitions and the uses of two nonterminals.
- **removeV** removes a production from the existing nonterminal definition such that the nonterminal does not become undefined.
- **removeH** replaces any choice with a similar choice with less alternatives.
- **disappear** projects a nillable symbol (the one that can be evaluated to ε).
- **narrow** replaces an expression with a refined one.
- **downgrade** replaces a nonterminal with one of its definitions.
- **define** adds a definition to a bottom (used but undefined) nonterminal.
- **undefine** removes a definition of a nonterminal, forcing it to become a bottom sort.
- **redefine** replaces the existing definition of a nonterminal with the given one.
- **inject** inserts possibly non-nillable symbols to a sequence.
- **project** removes possibly non-nillable symbols from a sequence.
- **replace** replaces any subexpression with another one.
- **unlabel** strips a production of its label.

4.2 **Examples of grammar transformations**

The examples that illustrate the uses of the XBGF operators are presented below in the same order they are encountered in the transformation scripts with respect to the convergence phases that we listed and motivated in §2.4.

*4.2.1 **Convergence preparation***

The following 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 shown below.

impl3 (Gosling et al, 2005, §18.1)

```
Block:
        BlockStatements⋆
```

Semantics-revising transformation

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

Corrected syntax

```
Block:
        "{" BlockStatements "}"
```

The source format defines curly brackets to express iteration. However, in the example at hand, taken from *impl3*, they were meant as terminal symbols, and were not recognized due to missing markup. The incorrect list construct is replaced accordingly.

Another activity frequently undertaken during the convergence preparation phase is initial correction. Unlike correction that happens during convergence resolution phase (see §4.2.6), it is triggered by an external bug report and not by a grammar comparator. Such a bug report can be produced by another tool or taken from a third party. For instance, a misnamed nonterminal is found in *impl2* when examining the list of bottom nonterminals before the later phases of convergence process start:

impl2 (Gosling et al, 2000, §18.1)

```
Expression3:
        "(" (Expr | Type) ")" Expression3
```

Semantics-revising transformation

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

Corrected syntax

```
Expression3:
        "(" (Expression | Type) ")" Expression3
```

4.2.2 ***Nominal matching***

In *impl2*, the built-in variable types are defined by a nonterminal called *BasicType* which contains all the alternatives. In the more readable counterpart the same nonterminal is called *PrimitiveType*, and it requires several intermediate nonterminals because types are explained in two different language document sections.

read2 (Gosling et al, 2000, §4.1, §4.2)

```
PrimitiveType:
        NumericType
        "boolean"
NumericType:
        IntegralType
        FloatingPointType
IntegralType:
        "byte"
        "short"
        "int"
        "long"
        "char"
FloatingPointType:
        "float"
        "double"
```

impl2 (Gosling et al, 2000, §18.1)

```
BasicType:
        "byte"
        "short"
        "char"
        "int"
        "long"
        "float"
        "double"
        "boolean"
```

Semantics-preserving transformation

renameN*(PrimitiveType, BasicType);*
inline*(IntegralType);*
inline*(FloatingPointType);*
inline*(NumericType);*
distribute*(BasicType);*

The last **distribute** transformation is needed to normalize the definition of *BasicType* to a top-level choice.

4.2.3 ***Structural matching***

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 in the following example to factor out the shared reference to *Block*. Then, the **massage** operator changes the style of expressing optionality of the keyword “`static`”.

read2 (Gosling et al, 2000, §8.1.5, §8.6, §8.7)

```
ClassBodyDeclaration:
        InstanceInitializer
        StaticInitializer
        ...
StaticInitializer:
        "static" Block
InstanceInitializer:
        Block
```

impl2 (Gosling et al, 2000, §18.1)

```
ClassBodyDeclaration:
        "static"? Block
```

Semantics-preserving transformation

```
inline(StaticInitializer);
inline(InstanceInitializer);
factor(
  (Block | ("static" Block)) ,
  ((ε | "static") Block) );
massage(
  (ε | "static"),
  "static"? );
```

Like the previous transformation sample, the following one is taken from a refactoring script that aligns *read2* with *impl2*. The JLS case involves many hundreds of such small refactoring steps; see §4.3.

In *read2*, the 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 "*".

read2 (Gosling et al, 2000, §8.1.5)

```
ClassBody:
        "{" ClassBodyDeclarations? "}"
ClassBodyDeclarations:
        ClassBodyDeclaration
        ClassBodyDeclarations ClassBodyDeclaration
```

impl2 (Gosling et al, 2000, §18.1)

```
ClassBody:
        "{" ClassBodyDeclaration* "}"
```

Semantics-preserving transformation

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

4.2.4 ***Extension***

The following transformation is part of a chain that captures the difference 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* (recall Figure 7). 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 following example) or productions (specified by a production label).

jls1 after many transformation steps (Gosling et al, 1996)

```
ClassBodyDeclaration:
      "static" Block
```

impl2 (Gosling et al, 2000, §9.3)

```
ClassBodyDeclaration:
      "static"? Block
```

Semantics-increasing transformation

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

The following transformation is part of a script that captures the difference between JLS2 and JLS3, where the latter offers *Annotation* as the additional option.

read2 (Gosling et al, 2000, §9.3)

```
ConstantModifier:
      "public"
      "static"
      "final"
```

read3 (Gosling et al, 2005, §9.3)

```
ConstantModifier:
      Annotation
      "public"
      "static"
      "final"
```

Semantics-increasing transformation

```
addV(
 ConstantModifier:
         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 difference 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.

4.2.5 ***Relaxation***

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 respectively) imply the need for relaxation. Similar issues arise with relationships between abstract and concrete syntaxes (Lämmel and Zaytsev, 2009; Wile, 1997).

In *impl2*, there is only one category of (arbitrary) modifiers. In contrast, in *read2*, there are various precise categories of modifiers for classes, fields, methods, constructors, interfaces, constants and abstract methods. Accordingly, the *impl2* grammar is more permissive as far as modifiers are concerned. The grammar fragments in the following example are deliberately pretty-printed as horizontal productions for the sake of readability. In reality the extractor produces only vertical ones as usual.

read2 (Gosling et al, 2000, §8.1.1, §8.3.1, §8.4.3, §8.8.3, §9.1.1, §9.3, §9.4)

```
ClassModifier:
        "public" | "protected" | "private" | "abstract" | "static"
              | "final" | "strictfp"
FieldModifier:
        "public" | "protected" | "private" | "static" | "final" |
              "transient" | "volatile"
MethodModifier:
        "public" | "protected" | "private" | "abstract" | "static"
              | "final" | "synchronized" | "native" | "strictfp"
ConstructorModifier:
        "public" | "protected" | "private"
InterfaceModifier:
        "public" | "protected" | "private" | "abstract" | "static"
              | "strictfp"
ConstantModifier:
        "public" | "static" | "final"
AbstractMethodModifier:
        "public" | "abstract"
```

impl2 (Gosling et al, 2000, §18.1)

```
Modifier:
      "public" | "protected" | "private" | "static" | "abstract"
            | "final" | "native" | "synchronized" | "transient"
            | "volatile" | "strictfp"
```

Semantics-increasing transformation

```
renameN(ClassModifier, Modifier);
unite(FieldModifier, Modifier);
unite(MethodModifier, Modifier);
unite(ConstructorModifier, Modifier);
unite(InterfaceModifier, Modifier);
unite(ConstantModifier, Modifier);
unite(AbstractMethodModifier, Modifier);
```

Constructor declarations are defined very differently in *impl3* and *read3*. The following example shows the last steps of their convergence, where *ConstructorBody* must be replaced by *MethodBody*. However, the definition of *ConstructorBody* (at this stage) is equal to the definition of *Block*, while *MethodBody* can be *Block* or `";"`. By letting the terminal `";"` appear instead of *Block*, we make the grammar of the language more permissive.

read3 after many transformation steps

```
ConstructorDeclaratorRest:
      FormalParameters Throws? ConstructorBody
ConstructorBody:
      "{" BlockStatements "}"
MethodBody:
      Block
MethodBody:
      ";"
Block:
      "{" BlockStatements "}"
```

impl3 (Gosling et al, 2005, §18.1)

```
ConstructorDeclaratorRest:
      FormalParameters ("throws" QualifiedIdentifierList)? MethodBody
```

Semantics-increasing transformation

```
fold(Block);
upgrade(
 ConstructorBody:
      <MethodBody>
 MethodBody:
      Block
);
inline(ConstructorBody);
inline(Throws);
```

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

4.2.6 ***Correction***

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.

What Oracle SDN (Sun Developer Network) Bug Database reports[5] as “the master bug for errors in the Java grammar” is the fact that §18.1 of Gosling et al (2005) does not permit the obsolescent array syntax in a method declaration of an annotation type.

Incorrect syntax in *impl3*

```
AnnotationMethodRest:
        "(" ")" DefaultValue?
```

Semantics-increasing transformation

```
appear(
 AnnotationMethodRest:
        "(" ")" <("[" "]")*> DefaultValue?
);
```

Corrected syntax

```
AnnotationMethodRest:
        "(" ")" ("[" "]")* DefaultValue?
```

Not all corrections may be expressed in terms of semantics-increasing/-decreasing operators. If that is not possible, we have to use less disciplined operators. For example, the production for the *break* statement in *impl2* lacks the semicolon which is injected accordingly (left unnoticed in Bosworth’s bug list, but obvious when converging with *read2*).

impl2 (Gosling et al, 2000, §18.1)

```
Statement:
        "break" Identifier?
```

Semantics-revising transformation

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

Corrected syntax

```
Statement:
        "break" Identifier? ";"
```

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).

[5] `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6442525`

	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 5 Simple metrics for the derived JLS grammars.

Incorrect expression syntax in *impl2* and *impl3*

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

Semantics-revising transformation

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

Corrected syntax

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

4.3 Postmortem of the JLS case

We recall that Table 3 lists simple metrics for the leaves of JLS' convergence tree. The new Table 5 shows the same data for the derived grammars. It is easily seen that 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.

Table 6 measures the extraction effort and the involved grammar transformations. Matching phases (§4.2.2 and §4.2.3) consist of semantics-preserving transformations, the measurement for other phases is presented in the table directly. This information was obtained in an automated manner but it relies on some amount of semantic annotation of the transformations for the classifications and convergence phases.

The number of transformations directly refers to the number of *applications of transformation operators*. As one can infer from Table 7, 33 different operators are used in the JLS

	jls1	jls12	jls123	jls2	jls3	read12	read123	Total
Number of lines	682	5114	2847	6774	10721	1639	3082	30859
Number of transformations	67	290	111	387	544	77	135	1611
◦ Semantics-preserving	45	231	80	275	381	31	78	1121
◦ Semantics-increasing/-decreasing	22	58	31	102	150	39	53	455
◦ Semantics-revising	—	1	—	10	13	7	4	35
Convergence preparation phase (§4.2.1)	1	—	—	15	24	11	14	65
◦ Known bugs	—	—	—	1	11	—	4	16
◦ Post-extraction	—	—	—	7	8	7	5	27
◦ Initial correction	1	—	—	7	5	4	5	22
Resolution phase	21	59	31	97	139	35	43	425
◦ Extension (§4.2.4)	—	17	26	—	—	31	38	112
◦ Relaxation (§4.2.5)	18	39	5	75	112	—	2	251
◦ Correction (§4.2.6)	3	3	—	22	27	4	3	62

Table 6 Transformation of the JLS grammars—effort metrics and categorization.

case; most of them were introduced in §4. About three quarters of the transformations are semantics-preserving. The remaining quarter is mainly dedicated to semantics-increasing or -decreasing transformations with only 2% left for semantics-revising transformations.

In Table 6, 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 Oracle'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.

5 Related work

We organize the related work discussion in the following manner:

– grammar recovery (including grammar inference);
– programmable grammar transformations;
– other grammar engineering work;
– coupled transformations of grammar- or schema- or metamodel-like artifacts and grammar- or schema- or metamodel-dependent artifacts;
– comparison (including matching) of schemas or metamodels.

5.1 Grammar recovery

The main objective of the present JLS study is to discover grammar relationships, but an "important byproduct" of the study is a consolidated Java grammar[6]. Hence, this particular instance of grammar convergence (perhaps more than grammar convergence in general) relates strongly to other efforts on grammar recovery. This topic has seen substantial interest over the last decade because of the need for grammars in various software engineering scenarios. We categorize this work in the following.

[6] See also Software Language Processing Suite Grammar Zoo at `http://slps.sf.net/zoo`.

	jls1	**jls12**	**jls123**	**jls2**	**jls3**	**read12**	**read123**	**Total**
∘ *rename*	9	4	2	9	10	—	2	36
∘ *reroot*	2	—	—	2	2	2	1	9
∘ *unfold*	1	10	8	11	13	2	3	48
∘ *fold*	4	11	4	11	13	2	5	50
∘ *inline*	3	67	8	71	100	—	1	250
∘ *extract*	—	17	5	18	30	—	5	75
∘ *chain*	1	—	2	—	—	1	4	8
∘ *massage*	2	13	—	15	32	5	3	70
∘ *distribute*	3	4	2	3	6	—	—	18
∘ *factor*	1	7	3	5	24	3	1	44
∘ *deyaccify*	2	20	—	25	33	4	3	87
∘ *yaccify*	—	—	—	—	1	—	1	2
∘ *eliminate*	1	8	1	14	22	—	—	46
∘ *introduce*	—	1	30	4	13	3	34	85
∘ *import*	—	—	2	—	—	—	1	3
∘ *vertical*	5	7	7	8	22	5	8	62
∘ *horizontal*	4	19	5	17	31	4	4	84
∘ *add*	1	14	13	7	20	28	20	103
∘ *appear*	—	8	11	8	25	2	17	71
∘ *widen*	1	3	—	1	8	1	3	17
∘ *upgrade*	—	8	—	14	20	2	2	46
∘ *unite*	18	2	—	18	21	5	4	68
∘ *remove*	—	10	1	11	18	—	1	41
∘ *disappear*	—	7	4	11	11	—	—	33
∘ *narrow*	—	—	1	—	4	—	—	5
∘ *downgrade*	—	2	—	8	3	—	—	13
∘ *define*	—	6	—	4	9	1	6	26
∘ *undefine*	—	3	—	5	3	—	—	11
∘ *redefine*	—	3	—	8	7	6	2	26
∘ *inject*	—	—	—	2	4	—	1	7
∘ *project*	—	1	—	1	2	—	—	4
∘ *replace*	3	1	2	3	6	1	1	17
∘ *unlabel*	—	—	—	—	—	—	2	2

Table 7 XBGF operators usage for JLS convergence.

Recovery option 1: Parser-based testing and improvement cycle

A by now classical approach to grammar recovery is to start from some sort of documentation that contains a raw grammar, which can be extracted, and then to improve the raw grammar through parser-based testing until all sources of interest can be parsed (such as test programs, or entire software projects): Sellink and Verhoef (2000); Lämmel and Verhoef (2001a,b); de Jonge and Monajemi (2001); Alves and Visser (2009). The actual improvement steps may be carried out manually (Sellink and Verhoef, 2000; de Jonge and Monajemi, 2001; Alves and Visser, 2009) or by means of programmable grammar transformations (Lämmel and Verhoef, 2001a,b), as discussed in more detail in §5.2.

The present JLS study, in particular, and the basic paradigm of grammar convergence, in general, do not involve parser-based testing. Instead, the similarity between two or more given grammars is used as the criterion for possibly improving correctness. Of course, it would be a viable scenario to actually try deriving a useful parser description from the converged Java grammar, and if additional problems were found, then the parser-based testing and improvement cycle of grammar recovery may be applied.

Recovery option 2: Grammar recovery from ASTs

Generally, raw grammars (as discussed above) may also be extracted from compilers. This is relatively straightforward, if the compiler uses a parser description to implement the parser. Duffy and Malloy (2007); Kraft et al (2009) present another option, which relies on access to the parse trees or ASTs of a compiler. A grammar can be extracted from the ASTs for given sample programs. This approach is specifically meant to help with the recovery of language dialects for which precise grammars are often missing. In order to derive the grammar for the concrete syntax, one must discover the mapping between AST schema and concrete syntax. To this end, the approach also involves some verification infrastructure. If we assume that a baseline grammar is available (as opposed to a grammar for the specific dialect at hand), then grammar convergence may also be useful in providing the mapping between AST schema and concrete syntax.

Recovery option 3: Grammar inference

Different authors have approached grammar recovery for software languages through grammar inference techniques: Mernik et al (2003); Črepinšek et al (2005); Dubey et al (2005); Di Penta and Taneja (2005); Dubey et al (2006a,b); Di Penta et al (2008); Dubey et al (2008). Inference relies on language samples, typically on both positive and negative examples. Different inference scenarios have been addressed. Mernik et al (2003); Črepinšek et al (2005) infer more or less complete grammars, which is a very difficult problem. The approach applies to small languages, e.g., small domain-specific languages. Di Penta and Taneja (2005); Di Penta et al (2008) start from a baseline grammar, and infer modifications to the grammar so that all sources of interest can be parsed. This search-based inference approach addresses the dialect problem in software engineering, where a grammar for the language of interest may be available, but not for the specific dialect at hand. Both of the approaches use *genetic algorithms*. Dubey et al (2005, 2006a,b, 2008) use a mix of advanced parsing and inference techniques instead.

Just as in the case of Option 1, the approach uses parser-based testing as the correctness criterion, whereas grammar convergence leverages the similarity between two or more given grammars as the criterion for possibly improving correctness. It is quite conceivable and interesting to combine grammar inference and grammar convergence. For instance, grammar inference techniques could be used to inform a semi-automatic grammar transformation approach. Also, it is interesting to understand whether transformation operators for convergence can usefully represent the modifications of the inference approach of Di Penta and Taneja (2005); Di Penta et al (2008).

Recovery option 4: Special-purpose grammars

Rather than trying to recover the (full) grammar for a given language, one may also limit the recovery effort to specific samples, and more potently, to the specific purpose of the grammar. For instance, when the grammar is needed for a simple fact extractor, then there is no need to parse the full language, or to be fully aware of the dialect at hand. Moonen (2001, 2002) suggests so-called island grammars to only define as much syntactical structure as needed for the purpose and to liberally consume all other structure essentially as a token stream. Synytskyy et al (2003) also pursue this approach specifically in the context of multilingual parsing. Nierstrasz et al (2007) also pursue a variant of special-purpose grammars, where sample programs are essentially modeled, and a grammar is computed from the samples. A disciplined and productivity-tuned, iterative approach is used to rapidly parse all the samples of interest. The approach also produces the right metamodel (object model) to represent parse trees tailored to the specific purpose at hand.

5.2 **Programmable grammar transformations**

Grammar convergence, and some forms of grammar recovery, but also some other software engineering problems rely on grammar transformations. In fact, we would like to limit the focus here to *programmable* grammar transformations. We are not interested in “hidden” transformations as they may be performed implicitly by some software tools such as a parser generator which removes left recursion automatically.

Cordy, Dean and collaborators have invented the notion of agile parsing (Dean et al, 2003; Cordy, 2003; Dean and Synytskyy, 2005) and the paradigm of grammar programming (Dean et al, 2002) in this context. Both concepts rely on language embedding of a grammar formalism into a programming language (TXL, in their case). Agile parsing basically suggests the customization of a baseline grammar for a specific use case (such as components for reverse engineering or re-engineering). The simpler programmable grammar transformations, which are sufficient for some scenarios, are **redefine** (to redefine a nonterminal), and **define** with the ability to extend the previous definition.

In Dean et al (2002), a range of additional grammar programming techniques is discussed, where some of these techniques can be naturally modeled as grammar transformations (or more generally, as program transformations). These are the techniques: rule abstraction (so that grammar rules may be parametrized), grammar specialization (so that the semantics of specific uses cases can be incorporated into the grammar), grammar categorization (so that the resulting parser can effectively deal with context-free ambiguities), union grammars (so that one can have multiple grammars in the same namespace, perhaps even with a non-empty intersection), and markup (i.e., the use of markup syntax in combination with regular textual syntax).

In our own work (the one reported here, as well as in Lämmel (2001); Lämmel and Wachsmuth (2001); Lämmel (2005)), we have been interested in operator suites for (programmable) grammar transformations. The idea is basically to view the possible evolution of a grammar (along recovery or convergence) as a disciplined editing process such that each editing step is described in terms of an appropriate transformation operator. The use of an operator immediately documents a certain intention, and is subject to precondition checking—just like in other domains of program transformation. Wile (1997) has also suggested a small set of operators to specifically address the problem of computing abstract from concrete syntax.

The Amsterdam/Koblenz school of grammar transformation

To better understand the design space of programmable grammar transformations based on operator suites, we would like to compare several efforts in which at least one of the authors has been involved; see Table 8 for an overview. The figure summarizes known grammar transformation operators, and compares operator suites for grammar transformations:

VSC2 (Lämmel, 2001; Lämmel and Verhoef, 2001b)
The suite used for recovery of a Cobol grammar.[7]

[7] `http://homepages.cwi.nl/~ralf/fme01`

	VSC2	FST	GDK	GRK	XBGF
add a definition for a bottom nonterminal	`resolve`	`resolve`	`resolve`		`define`
add a new definition	`introduce`	`introduce`	`introduce`		`introduce`
add a new definition & fold it	`extract`			`extract`	`extract`
add a production to any nonterminal definition	`include`	`include`	`include`		`addV`
add a production to the grammar	`add`	`add`		`add`	
add alternatives to a choice					`addH`
change the order in a sequence	`permute`			`permute`	`permute`
do nothing	`id`	`id`			
give a production a label					`designate`
give a subexpression a selectable name					`deanonymize`
inject a nillable symbol					`appear`
inject a terminal symbol					`concretize`
inject symbols to a sequence					`inject`
inline a chain production					`unchain`
introduce a chain production					`chain`
introduce a reflexive chain production					`detour`
introduce several possibly interconnected definitions					`import`
merge two nonterminals					`unite`
merge two nonterminals if their definitions are equal		`equate`			`equate`
merge two nonterminals, one of which is bottom	`unify`	`unify`	`unify`		`unite*`
move a production across modules/sections		`move`			
perform factoring transformation			`preserve*`		`factor`
perform folding transformation	`fold`	`fold`	`fold`	`fold`	`fold`
perform massaging transformation	`preserve`	`simplify`	`preserve`		`massage`
perform narrowing transformation (as in x? to x)	`restrict*`	`restrict*`	`restrict*`		`narrow`
perform specialized automated factoring transformation					`distribute`
perform unfolding transformation	`unfold`	`unfold`	`unfold`	`unfold`	`unfold`
perform widening transformation (as in x+ to x*)	`generalise`	`generalise`	`generalize`		`widen`
project a nillable symbol	`restrict*`	`restrict*`	`restrict*`		`disappear`
project a terminal symbol					`abstractize`
project symbols from a sequence					`project`
remove a definition of a possibly used nonterminal	`reject`	`reject`	`reject`		`undefine`
remove a label from a production					`unlabel`
remove a production from the grammar	`sub`	`sub`			
remove a reflexive chain production					`abridge`
remove a selector in a subexpression					`anonymize`
remove alternatives from a choice					`removeH`
remove any part of a grammar	`reset`	`reset`			
remove unused definition	`eliminate`	`eliminate`	`eliminate`	`reject`	`eliminate`
removes one production of a nonterminal (not the last one)	`exclude`	`exclude`	`exclude`		`removeV`
rename a label					`renameL`
rename a nonterminal	`rename`	`rename`	`rename`		`renameN`
rename a nonterminal in a limited scope	`substitute`	`substitute`			`replace*`
rename a selector					`renameS`
replace a nonterminal with one of its definitions					`downgrade`
replace a nonterminal with ε	`delete`		`delete`		`replace*`
replace a terminal with another terminal					`renameT`
replace an expression by a nonterminal that can be evaluated to it					`upgrade`
replace any expression with another expression	`replace`	`replace`		`replace`	`replace`
replace iteration with left-associative equivalent					`lassoc`
replace iteration with recursion			`preserve*`		`yaccify`
replace iteration with right-associative equivalent					`rassoc`
replace recursion with iteration			`preserve*`		`deyaccify`
replace the current definition by a new one			`redefine`		`redefine`
separate one nonterminal into several (reverse of merge)	`separate`	`seperate`	`separate`		
terminate transformation sequence	`fail`	`fail`	`write`		`dump`
transpose a multi-production definition to the one with top-level choices					`horizontal`
transpose top-level choices to multiple productions					`vertical`
unfold & eliminate					`inline`

Table 8 Systematic comparison of grammar transformation operators provided by different frameworks

FST (Lämmel and Wachsmuth, 2001)
  A design experiment to define a comprehensive suite for SDF (Visser, 1997).[8]
GDK (Kort et al, 2002)
  A suite that is part of a grammar-deployment infrastructure.[9]
GRK (Lämmel, 2005)
  A suite that is part of an effort to reproduce our Cobol recovery case.[10]
XBGF
  The suite of the present paper; see §4.1.[11]

A starred name in the figure (as in "restrict*") means that the given operator covers the function at hand, but it is more general.

XBGF, the transformation language of the present paper, provides clearly the most comprehensive suite. There are a few empty cells in the XBGF column. Reasons for non-inclusion differ; either the operator is considered too low-level for the XBGF surface syntax (e.g., **substitute**, **reset**), or it is too low-level in the sense that all major application scenarios are covered by more specialized operators (e.g., **add**, **sub**), or it is not currently implementable (e.g., **move**—modules are not fully supported in our infrastructure), or it was simply not needed and perhaps debated so far (e.g., **delete**, **id**, **separate**, also known as FST's **seperate**; see the table for the typo).

There is generally a tension between the number of transformation operators vs. the achievable precision of a transformational program in terms of expressing intentions, and thereby enabling extra sanity checks by the transformation engine. Consider, for example, the line "add a production to the grammar". This low-level idiom may be used to **include** another production into an existing definition, or to add one or more productions in an effort to **resolve** a missing definition, or to **introduce** a definition for a so-far fresh nonterminal. In GRK, all these idioms are modeled by **add**, and hence no intentions are documented, and no extra checks can be performed automatically. In the case of XBGF, we have indeed tried to separate idioms aggressively. This approach also helps us with predicting the formal properties of each application of transformation operators (i.e., semantics-preserving, -increasing, -decreasing, -revising), and chains thereof.

5.3 **Grammar engineering**

Let us also discuss some additional related work on grammar engineering (Klint et al, 2005) in a broader sense. We begin with metrics which are used by various recovery approaches and other work on grammar engineering. We want to highlight Alves and Visser (2009); Malloy et al (2002); Duffy and Malloy (2007); Julien et al (2009); Kraft et al (2009). Our work leverages simple grammar metrics (numbers of bottom and top nonterminals) and grammar-comparison metrics (numbers of nominal and structural differences) for providing guidance in a grammar convergence context.

An interesting blend of recovery and convergence (or consistency checking) is described in (Bouwers et al, 2008) where *precedence rules* are recovered from *multiple* grammars and checked for consistency. At this point, grammar convergence (in our sense) does not cover such sophisticated convergence issues. In fact, our approach is, as yet, oblivious to

[8] `http://www.cs.vu.nl/grammarware/fst`
[9] `http://gdk.sf.net`
[10] `http://slps.sf.net/grk`
[11] `http://slps.sf.net/xbgf`

technology-specific representations of priority rules (as used in, say YACC or SDF). We could potentially detect priority layers in plain grammars, though.

An alternative to grammar recovery is the use of a flexible parsing scheme based on advanced error handling (Barnard, 1981; Barnard and Holt, 1982; Klusener and Lämmel, 2003), subject to a baseline grammar. Because of flexible parsing, the grammar could also be used to parse a dialect; no precise grammar is needed. Also, code with syntax errors can be handled, which is important in some application areas such as reverse or re-engineering of legacy code.

There are approaches to connect the technical spaces of grammarware and modelware in a manner that can be viewed as a form of grammar convergence. That is, the parser may be obtained from the (meta)model based on appropriate metadata and mapping rules, using a generative approach (Jouault et al, 2006; Nierstrasz et al, 2007). We also use the term model-driven parser development for these approaches. The point of grammar convergence is that it provides a very flexible means to represent relationships between grammar-like artifacts from different technical spaces—without enforcing a particular scheme of designing grammar-based artifacts or mappings.

5.4 **Schema/metamodel comparison**

Grammar comparison, as it is part of grammar convergence, can be loosely compared with *schema matching* in ER/relational modeling (Do and Rahm, 2007; Rahm and Bernstein, 2001) as well as model and *metamodel matching or comparison* in model-driven engineering (Falleri et al, 2008; Wenzel and Kelter, 2008; Xing and Stroulia, 2006) (specifically in the context of model/metamodel evolution). However, our current approach to comparison (as of §2.1) is relatively trivial, and does not make any contribution to this subject, not even remotely. A simple comparison approach was sufficient so far for two reasons. First, the metamodel of grammars is relatively simple. Second, we only require to determine nominal differences (subject to the comparison of defined nonterminal names) and structural differences (subject to matching alternatives). We will need a more advanced comparison machinery once we aim at the partial inference of grammar transformations. In this case, grammar convergence should benefit from previous work on schema matching and metamodel comparison.

5.5 **Coupled transformations**

Grammar convergence relates to mappings in data processing (Thomas, 2003; Lämmel and Meijer, 2006), specifically to the underlying theory of data refinement, and applications thereof (Hoare, 1972; Morgan, 1990; Alves et al, 2005; Oliveira, 2008; Cunha et al, 2008). In data refinement, one also considers certain well-defined operators for transforming data models. These operators must be defined immediately in a way that they can be also interpreted as mappings at the data level so that instance data can be converted back and forth between the data models that are related by the transformation.

Inspired by data refinement, all semantics-preserving and -decreasing operators for grammar transformation can also be interpreted at the AST level, and we experiment with such an interpretation, which opens up new applications for grammar convergence. For instance, one could replace the parser of a given program with another parser, even when their

AST types are different. That is, the convergence transformations would be executed at the AST-level as a conversion.

Data refinement is actually a specific and highly disciplined instance of so-called coupled transformations, which are characterized to involve multiple kinds of software artifacts (such as types vs. instance data vs. programs over those types) that depend on each other in the sense that the transformation of one entity (of one kind) necessitates a transformation of another entity (of another kind, potentially) (Lämmel, 2004). For instance, Hainaut et al (1994); Lämmel and Lohmann (2001); Wachsmuth (2007); Vermolen and Visser (2008); Cicchetti et al (2008); Berdaguer et al (2007) are concerned with coupling for data models or metamodels vs. instance data or models; Cleve and Hainaut (2006) are concerned with coupling for data models and programs over these data models. Again, we suggest that grammar convergence should be generalized to cover coupled transformations. As a result, the convergence method will find new application areas.

6 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 just coding and debugging 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 (e.g., de-/yaccification in de Jonge and Monajemi (2001); Lämmel (2001)).

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 can promise 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 (i.e., heuristics).

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.

Finally, a more strategic goal shall be to reconnect to standardization bodies, and to examine potential for industrial deployment of the method of grammar convergence. An early attempt, preceding the development of grammar convergence, is documented in Klusener and Zaytsev (2005). The challenge is here to understand what sort of tools and refined methods would be acceptable for those users in practice. This is an entirely non-trivial problem, but its solution is critical to the value proposition of grammar convergence. Oracle, ISO, and other stakeholders will not adopt grammar convergence tools and methodology, unless they can measure the added value in terms of productivity and correctness, and they do not need to engage with uncomfortable dependencies on tool providers. For instance, they may not like to completely overhaul their current methodology; they will not use an experimental, academic, open-source project; neither will they invest into major development of grammar-convergence tools; nor will they hire a designated computer scientist with a PhD on grammar engineering.

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

Alves TL, Visser J (2009) A Case Study in Grammar Engineering. In: Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers, Springer, LNCS, vol 5452, pp 285–304

Alves TL, Silva PF, Visser J, Oliveira JN (2005) Strategic Term Rewriting and Its Application to a VDMSL to SQL Conversion. In: FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings, Springer, LNCS, vol 3582, pp 399–414

Barnard D (1981) Syntax Error Handling Techniques. Tech. Rep. Technical Report 81-125, Queen's University, Department of Computing and Information Science, 23 pages

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

Berdaguer P, Cunha A, Pacheco H, Visser J (2007) Coupled Schema Transformation and Data Conversion for XML and SQL. In: Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007, Springer, LNCS, vol 4354, pp 290–304

Bouwers E, Bravenboer M, Visser E (2008) Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. ENTCS 203(2):85–101

Cicchetti A, Ruscio DD, Eramo R, Pierantonio A (2008) Automating Co-evolution in Model-Driven Engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, IEEE Computer Society, pp 222–231

Cleve A, Hainaut JL (2006) Co-transformations in Database Applications Evolution. In: Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers, Springer, LNCS, vol 4143, pp 409–421

Cordy JR (2003) Generalized Selective XML Markup of Source Code Using Agile Parsing. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC), Portland, Oregon, pp 144–153

Cunha J, Saraiva J, Visser J (2008) From Spreadsheets to Relational Databases and Back. In: PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, ACM, New York, NY, USA, pp 179–188

Dean T, Synytskyy M (2005) Agile Parsing Techniques for Web Applications. In: Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering, Part II, Technology Presentations, Braga, Portugal, pp 29–38

Dean T, Cordy J, Malton A, Schneider K (2002) Grammar Programming in TXL. In: Proceedings, Source Code Analysis and Manipulation (SCAM'02), IEEE

Dean T, Cordy J, Malton A, Schneider K (2003) Agile Parsing in TXL. Journal of Automated Software Engineering 10(4):311–336

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

Di Penta M, Lombardi P, Taneja K, Troiano L (2008) Search-based Inference of Dialect Grammars. Soft Computing — A Fusion of Foundations, Methodologies and Applications 12(1):51–66

Do HH, Rahm E (2007) Matching Large Schemas: Approaches and Evaluation. Information Systems 32(6):857–885

Dubey A, Aggarwal SK, Jalote P (2005) 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, pp 217–225

Dubey A, Jalote P, Aggarwal SK (2006a) A Deterministic Technique for Extracting Keyword Based Grammar Rules from Programs. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, ACM, pp 1631–1632, DOI http://doi.acm.org/10.1145/1141277.1141659

Dubey A, Jalote P, Aggarwal SK (2006b) Inferring Grammar Rules of Programming Language Dialects. In: Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings, Springer, Lecture Notes in Computer Science, vol 4201, pp 201–213

Dubey A, Jalote P, Aggarwal SK (2008) Learning Context-Free Grammar Rules from a Set of Program. IET Software 2(3):223–240

Duffy EB, Malloy BA (2007) An Automated Approach to Grammar Recovery for a Dialect of the C++ Language. In: Proceedings, 14th Working Conference on Reverse Engineering

(WCRE 2007), IEEE, pp 11–20

Falleri JR, Huchard M, Lafourcade M, Nebut C (2008) Metamodel Matching for Automatic Model Transformation Generation. In: Proceedings of Model Driven Engineering Languages and Systems (MoDELS 2008), Springer, LNCS, vol 5301, pp 326–340

Gosling J, Joy B, Steele GL (1996) The Java Language Specification. Addison-Wesley, available at `java.sun.com/docs/books/jls`

Gosling J, Joy B, Steele GL, Bracha G (2000) The Java Language Specification, 2nd edn. Addison-Wesley, available at `java.sun.com/docs/books/jls`

Gosling J, Joy B, Steele GL, Bracha G (2005) The Java Language Specification, 3rd edn. Addison-Wesley, available at `java.sun.com/docs/books/jls`

Hainaut JL, Tonneau C, Joris M, Chandelon M (1994) Schema Transformation Techniques for Database Reverse Engineering. In: Entity-Relationship Approach - ER'93, 12th International Conference on the Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993, Proceedings, Springer, LNCS, vol 823, pp 364–375

Hoare CAR (1972) Proof of Correctness of Data Representations. Acta Informatica 1(4):271–281

de Jonge M, Monajemi R (2001) Cost-Effective Maintenance Tools for Proprietary Languages. In: Proceedings, International Conference on Software Maintenance (ICSM'01), IEEE, pp 240–249

Jouault F, Bézivin J, Kurtev I (2006) 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, pp 249–254

Julien C, Črepinšek M, Forax R, Kosar T, Mernik M, Roussel G (2009) On Defining Quality Based Grammar Metrics. In: Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT 2009, pp 651–658

Klint P, Lämmel R, Verhoef C (2005) Toward an Engineering Discipline for Grammarware. ACM Transactions on Software Engineering Methodology (TOSEM) 14(3):331–380

Klusener A, Lämmel R (2003) Deriving Tolerant Grammars from a Base-line Grammar. In: Proceedings, International Conference on Software Maintenance (ICSM'03), IEEE, pp 179–189

Klusener S, Zaytsev V (2005) ISO/IEC JTC1/SC22 Document N3977—Language Standardization Needs Grammarware. Available at `http://www.open-std.org/jtc1/sc22/open/n3977.pdf`

Kort J, Lämmel R, Verhoef C (2002) The Grammar Deployment Kit. In: Proceedings, Language Descriptions, Tools, and Applications (LDTA'02), Elsevier Science, ENTCS, vol 65, 7 pages

Kraft NA, Duffy EB, Malloy BA (2009) Grammar Recovery from Parse Trees and Metrics-Guided Grammar Refactoring. IEEE Trans Software Eng 35(6):780–794

Lämmel R (2001) Grammar Adaptation. In: Proceedings, Formal Methods Europe (FME) 2001, Springer, LNCS, vol 2021, pp 550–570

Lämmel R (2004) Coupled Software Transformations (Extended Abstract). In: First International Workshop on Software Evolution Transformations

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), Elsevier Science, ENTCS

Lämmel R, Lohmann W (2001) Format Evolution. In: Kouloumdjian J, Mayr H, Erkollar A (eds) Proceedings, Re-Technologies for Information Systems (RETIS'01), OCG, books@ocg.at, vol 155, pp 113–134

Lämmel R, Meijer E (2006) Mappings Make Data Processing Go 'round. In: Lämmel R, Saraiva J, Visser J (eds) Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers, Springer, LNCS, vol 4143, pp 169–218

Lämmel R, Verhoef C (2001a) Cracking the 500-Language Problem. IEEE Software pp 78–88

Lämmel R, Verhoef C (2001b) Semi-automatic Grammar Recovery. Software—Practice & Experience 31(15):1395–1438

Lämmel R, Wachsmuth G (2001) Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In: Proceedings, Language Descriptions, Tools and Applications (LDTA'01), Elsevier Science, ENTCS, vol 44

Lämmel R, Zaytsev V (2009) An Introduction to Grammar Convergence. In: Integrated Formal Methods, 7th International Conference, IFM 2009, Proceedings, Springer, LNCS, vol 5423, pp 246–260

Malloy B, Power J, Waldron J (2002) 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, pp 75–82, in cooperation with ACM Press

Mernik M, Gerlic G, Zumer V, Bryant BR (2003) Can a Parser be Generated from Examples? In: Proceedings of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA, ACM, pp 1063–1067

Moonen L (2001) Generating Robust Parsers Using Island Grammars. In: Proceedings, Working Conference on Reverse Engineering (WCRE'01), IEEE, pp 13–22

Moonen L (2002) Lightweight Impact Analysis Using Island Grammars. In: Proceedings, International Workshop on Program Comprehension (IWPC'02), IEEE

Morgan C (1990) Programming from Specifications. Prentice Hall International

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

Oliveira J (2008) Transforming Data By Calculation. In: Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2007, Revised Papers, Springer, LNCS, vol 5235, pp 134–195

Rahm E, Bernstein PA (2001) A Survey of Approaches to Automatic Schema Matching. VLDB Journal 10(4):334–350

Sellink M, Verhoef C (2000) Development, Assessment, and Reengineering of Language Descriptions. In: Proceedings, Conference on Software Maintenance and Reengineering (CSMR'00), IEEE, pp 151–160

Synytskyy N, Cordy J, Dean T (2003) Robust Multilingual Parsing using Island Grammars. In: Proceedings CASCON'03, 13th IBM Centres for Advanced Studies Conference, Toronto, pp 149–161

Thomas DA (2003) The Impedance Imperative — Tuples + Objects + Infosets = Too Much Stuff! Journal of Object Technology 2(5):7–12

Črepinšek M, Mernik M, Javed F, Bryant BR, Sprague A (2005) Extracting Grammar from Programs: Evolutionary Approach. SIGPLAN Notices 40(4):39–46

Vermolen S, Visser E (2008) Heterogeneous Coupled Evolution of Software Languages. In: Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings, Springer, LNCS, vol 5301, pp 630–644

Visser E (1997) Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam

Wachsmuth G (2007) Metamodel Adaptation and Model Co-adaptation. In: Ernst E (ed) ECOOP'07, Springer, LNCS, vol 4609, pp 600–624

Wenzel S, Kelter U (2008) Analyzing Model Evolution. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, ACM, pp 831–834

Wile D (1997) Abstract Syntax From Concrete Syntax. In: Proceedings, International Conference on Software Engineering (ICSE'97), ACM Press, pp 472–480

Xing Z, Stroulia E (2006) Refactoring Detection based on UMLDiff Change-Facts Queries. In: WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering, IEEE, pp 263–274

A Grammar normalization

If (x, y) represents sequential composition of symbols x and y, and $(x; y)$ represents a choice with x and y as alternatives, then the following formulæ are used for normalizing grammars within our framework:

$$
\begin{array}{rl}
(,) \Rightarrow \varepsilon & (;) \Rightarrow \mathit{fail} \\
(\ldots,(x,\ldots,z),\ldots) \Rightarrow (\ldots,x,\ldots,z,\ldots) & (x,) \Rightarrow x \\
(\ldots,x,\varepsilon,z,\ldots) \Rightarrow (\ldots,x,z,\ldots) & (x;) \Rightarrow x \\
(\ldots;(x;\ldots;z);\ldots) \Rightarrow (\ldots;x;\ldots;z;\ldots) & \varepsilon^{+} \Rightarrow \varepsilon \\
(\ldots;x;\mathit{fail};z;\ldots) \Rightarrow (\ldots;x;z;\ldots) & \varepsilon^{\star} \Rightarrow \varepsilon \\
(\ldots;x;\ldots;x;z;\ldots) \Rightarrow (\ldots;x;\ldots;z;\ldots) & \varepsilon? \Rightarrow \varepsilon
\end{array}
$$

B Massage-equality

The massage-equality relation is defined by these algebraic laws:

$$
\begin{array}{lll}
x? = (x;\varepsilon) & (x?)? = x? & (x, x^{\star}) = x^{+} \\
x? = (x?;\varepsilon) & (x?)^{+} = x^{\star} & (x^{\star}, x) = x^{+} \\
x^{\star} = (x^{+};\varepsilon) & (x?)^{\star} = x^{\star} & (x?, x^{\star}) = x^{\star} \\
x^{\star} = (x^{\star};\varepsilon) & (x^{+})? = x^{\star} & (x^{\star}, x?) = x^{\star} \\
x? = (x?;x) & (x^{+})^{+} = x^{+} & (x^{+}, x^{\star}) = x^{+} \\
x^{+} = (x^{+};x) & (x^{+})^{\star} = x^{\star} & (x^{\star}, x^{+}) = x^{+} \\
x^{\star} = (x^{\star};x) & (x^{\star})? = x^{\star} & (x^{+}, x?) = x^{+} \\
x^{\star} = (x?;x^{+}) & (x^{\star})^{+} = x^{\star} & (x?, x^{+}) = x^{+} \\
x^{\star} = (x?;x^{\star}) & (x^{\star})^{\star} = x^{\star} & (x^{\star}, x^{\star}) = x^{\star} \\
x^{\star} = (x^{+};x^{\star}) & &
\end{array}
$$

$$x = (s_1 :: x; s_2 :: x)$$

The infix operator "::" in the last formula denotes selectors (named addressable subexpressions). They are needed because a choice between two unnamed x will always be normalized as x, as explained in Appendix A.