Language Convergence Infrastructure

Vadim Zaytsev, zaytsev.vadim@gmail.com
Software Languages Team, Universität Koblenz-Landau, Germany

Abstract. The process of grammar convergence involves grammar extraction and transformation for structural equivalence and contains a range of technical challenges. These need to be addressed in order for the method to deliver useful results. The paper describes a DSL and the infrastructure behind it that automates the convergence process, hides negligible back-end details, aids development/debugging and enables application of grammar convergence technology to large scale projects. The necessity of having a strong framework is explained by listing case studies. Domain elements such as extractors and transformation operators are described to illustrate the issues that were successfully addressed.

1 Introduction

The method of grammar convergence has been presented in [15] and elaborated in a large case study [16], with a journal version being in print. The basic idea behind it is to extract grammars from available grammar artefacts, transform them until they become identical, and draw conclusions from the properties of the transformation chain: its length, the type of steps it consisted of, the correspondence with the properties

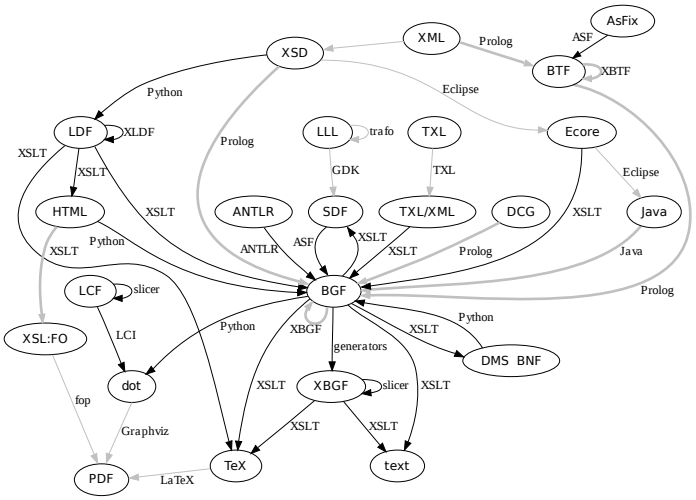

Fig. 1. The megamodel of SLPS: every vertex is a language, every arc is a language transformation. Thin grey lines denote tools present prior to this research: e.g., GDK [13] or TXL [3]. Thick grey edges are for co-authored transformations.

expected a priori from documentation, etc. Grammar convergence can be used among other ways to establish an agreement between a hand-crafted object model for a specific domain and an XML Schema for standard serialisation of the same domain; to prove that various grammarware such as parsers, code analysers and reverse engineering tools agree on the language; to synchronise the language definition in the manual with the reference implementation; to aid in disciplined grammar adaptation.

In this paper we will use the terms "grammar convergence" and "language convergence" almost interchangeably. In fact, language convergence is a broader term that includes convergence of not only the syntax, but also parse trees, documentation, possibly even semantics. We focus on dealing with grammars here, but the reader interested in consistency management for language specifications can imagine additional automated steps like extracting a grammar from the language document before the transformation and inserting it back afterwards [12,14].

Language convergence was developed and implemented as a part of an open source project called SLPS, or Software Language Processing Suite[1]. It comprises several stand-alone scripts targeting comparison, transformation, benchmarking, validation, extraction, pretty-printing. Most of those scripts were written in Python, Prolog, Shell and XSLT. Grammar convergence is a complicated process that can only be automated partially and therefore requires expert knowledge to be used successfully. In order to simplify the work of a grammar engineer, a specific technical infrastructure is needed with a solid transformation operators suite, steadily defined internal notations and a powerful tool support for every stage. This paper presents such a framework and explains both engineering and scientific design choices behind it.

Figure 1 presents a "megamodel" [2] of SLPS. Every arc from this graph is a language transformation tool or a sequence of pipelined tools. Many of the new DSLs developed for this infrastructure are in fact XML: BGF, XBGF, BTF, XBTF, LDF, XLDF, LCF—just an engineering decision that let them profit fully from XMLware facilities like validation against schemata and transformation with pattern matching. (These advantages are not unique for XML, of course). Others are mostly well-known languages that existed prior to this research: ANTLR [18], SDF [9], LLL [13], XSD [5], etc.

The left hand side of the megamodel is mostly dedicated to language documentation-related components: LDF is a Language Document Format [23], an extension of grammar notation that covers most commonly encountered elements of language manuals and specifications. The central part contains the grammar notation itself: the BGF node has a big fan-in since every incoming arc represents a grammar extraction tool (see §4.1). The only outgoing arcs are the main presentation forms: pure text, marked up LaTeX and a graph form, plus transformation generators (see §5.4) and integration tools (see §6.3).

The inherent complexity of the domain and the methodology led to the development of what we call LCI, or Language Convergence Infrastructure. It is the central point of SLPS, it provides full technical support to its functionalities, operating on a DSL called LCF (LCI Configuration Format) in which the input configuration must be expressed. The DSL details are also provided in this paper.

§2 motivates the need for language convergence by giving three example scenarios of its application. §3 starts describing the domain by explaining the DSL, while §4, §5 and §6 address the notions linked to sources, transformations and targets correspondingly.

[1] Software Language Processing Suite: `http://slps.sf.net`

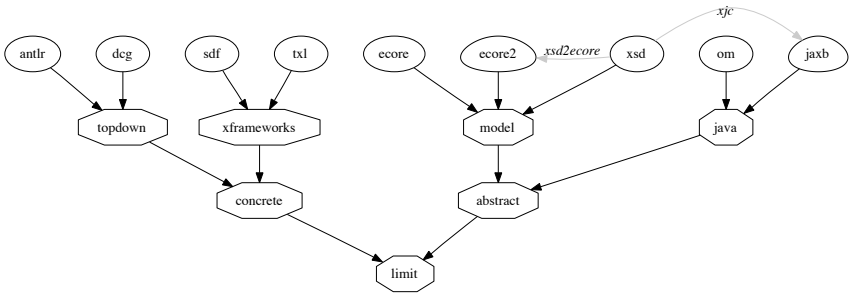

Fig. 2. The overall convergence graph for the Factorial Language. The grey arrows show grammar relations that are expressed in LCF but not performed directly by the convergence infrastructure (the reason is that, for example, generating Ecore from XML Schema cannot be done from command line and must be performed via Eclipse IDE).

2 Motivation

In this section three distinct applications of grammar convergence are briefly presented together with the results acquired from them.

2.1 Same Language, Multiple Implementations: Factorial Language

A trivial functional programming language was defined in [15] to test out the method of grammar convergence, we called it Factorial Language. We modelled the common scenario of one language having several independently developed grammars by writing or generating nine grammar artefacts within various frameworks, as seen on Figure 2:

antlr. A parser description in the input language language of ANTLR [18]. Semantic actions (in Java) are intertwined with EBNF-like productions.
dcg. A logic program written in the style of definite clause grammars.
sdf. A concrete syntax definition in the notation of SDF (Syntax Definition Formalism [9]), a parser description targeted for SGLR parsing.
txl. Another transformational framework that allows for agile development of tools based on language descriptions [3].
ecore. An Ecore model, created manually in Eclipse and represented in XMI [17].
ecore2. An alternative Ecore model, generated automatically by Eclipse, given the XML Schema of the domain.
xsd. An XML schema [5] for the abstract syntax of FL. In fact, this is the schema that served as the input for generating both the object model of the *jaxb* source and the Ecore model of the *ecore2* source.
om. A hand-crafted object model (Java classes) for the abstract syntax of FL. It is used by a Java-based implementation of an FL interpreter.
jaxb. Also an object model, but generated by the JAXB data binding technology [10] from the XML schema for FL.

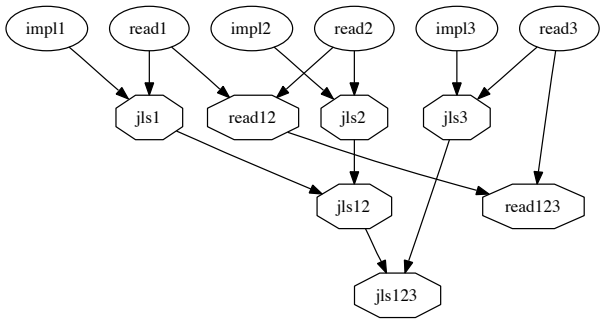

Fig. 3. Binary convergence tree for the JLS grammars—or rather two trees with shared leaves. As usual, the nodes on the top (the leaves) are grammars extracted directly from the JLS. All other nodes are derived by transformation chains denoted as arcs. We use a (cascaded) binary tree here: i.e., each non-leaf node is derived from two grammars.

2.2 Language Evolution: Java Language Specification

In [16] we describe a completed effort to recover the relationships between all the grammars that occur in the different versions of the Java Language Specification (JLS). The case study concerns the 3 different versions of the JLS [6,7,8] where each of the 3 versions contains 2 grammars: one grammar is optimised for readability (i.e., *read1–read3* on Figure 3), and another one is intended to serve as a basis for implementation (i.e., *impl1–impl3* on Figure 3). The JLS is critical to the Java platform — it is a foundation for compilers, code generators, pretty-printers, IDEs, code analysis and manipulation tools and other grammarware for the Java language. One would expect that the different grammars per version are essentially equivalent in terms of the generated language. For implementability reasons one grammar may be more liberal than the other. One would also expect that the grammars for the different versions engage in an inclusion ordering (again, in terms of the generated languages) because of the backwards-compatible evolution of the Java language.

The case study comprised around 17000 transformation steps and has shown that the expected relationships of (liberal) equivalence and inclusion ordering are significantly violated by the JLS grammars. Thus, grammar convergence can be used as a form of consistency management for the JLS in particular, and language specifications in general.

2.3 BNF-like Grammar Format

The abstract syntax of BGF, which is the internal representation for grammars in our infrastructure, is defined by the corresponding XML Schema. There is also a pretty-printer that helps to present BGF grammars for debugging and publishing purposes (let us call this presentation notation "BNF"). This pretty-printer is grammarware, the concrete syntax of its output can be specified by a grammar. How does this grammar relate to the XML Schema of BGF? We applied grammar convergence method to these

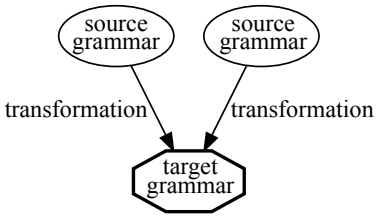

Fig. 4. The abstract view on the grammar convergence process.

two grammars: the hand-crafted one for the concrete syntax and the one derived from the XSD for the abstract syntax. (We had to be satisfied with a manually engineered grammar since grammar inference from an XSL transformation sheet is far from trivial and perhaps even undecidable).

The convergence graph is trivial and thus not shown here. The transformation scripts are also considerably simple for this case study, which allowed us to examine them in detail. The conclusion was that: **BGF allows for empty grammars, BNF does not** (as expected, since BNF is used for presentation); **BNF contains indentation rules and abstract syntax, BGF does not** (as expected due to the abstract nature of BGF); **BGF includes root elements, BNF does not** (as expected, since EBNF dialects never specify starting symbols).

This case study shows that grammar convergence can also be used for validating implicit assumptions within a grammar engineering infrastructure. The cost of such use is low (the case study took no more than an hour), but it brings more discipline to the grammar engineering process. The additional confidence comes from the guarantee that there are no other differences besides those included in our list.

3 Grammar Convergence Domain Overview

Grammar convergence is a method of establishing relationships between language grammars by extracting them and transforming towards equivalence. Thus, we distinguish three core domain elements: the **source grammars** that are obtained from available grammar artefacts; the **target grammars** that are the common denominators of the source grammars; and the **transformation chains** that bind them together and represent their relationships, as shown on Figure 4. The methodology has been presented in [15] and elaborated in a large case study [16].

LCF is a configurational domain specific language that is used by the LCI. Since it encapsulates all crucial domain concepts, we will examine its grammar and explain them while doing so. The grammar is presented in an EBNF dialect specific for SLPS: beside the usual notation it has selectors. By writing `a::b` we refer to a nonterminal `b` but label its particular occurrence as `a`. There are also four built-in symbols: *`string`* for any string, *`xstring`* for a macro expanded string, *`id`* for a unique identifier denoting an entity such as a grammar or a tool and *`refid`* for a reference to such an identifier.

```
scenario:
       shortcut* tools source+ target+ testset*
shortcut:
       name::id expansion::xstring
```

Each convergence scenario contains shortcuts, tools, sources, targets and test sets. **Shortcuts** are macro definitions used mostly for maintainability purposes: for example, it is possible with them to define the path to the main working directory once and refer to it in all necessary places. Shortcuts can be defined based on other shortcuts.

```
tools:
       transformer::tool comparator::tool validator::tool? generator*
tool:
       grammar::xstring tree::xstring?
```

Two **tools** are crucial for grammar convergence and must always be defined: the transformer and the comparator. The **transformer** takes a BGF grammar and an XBGF script and applies the latter to the former, resulting in a transformed BGF grammar (or an error return code, which is handled by the LCI). §5 will address this tool in detail. The **comparator** takes two BGF grammars and returns the verdict on their equivalence. Since the premise of grammar convergence method was to document grammar relationships, the comparator is not expected to do any sophisticated matching besides applying basic algebraic laws. A **validator** tool is optional and can check each one of the many XML files generated in the convergence process for well-formedness and conformance to a schema. Both tools will be described in §6. Each of these tools can consist of a pair of references to external programs: one program that operates on a grammar level and one on a parse tree level. The latter part is optional, but if it is absent, no coupled transformations can take place.

```
generator:
       name::id command::xstring
```

A transformation **generator** is a named tool that takes a BGF grammar as an input and produces an XBGF script applicable to that grammar and containing transformations of a certain nature, see §5.

```
testset:
       name::id command::xstring
```

A **test set** is also used for coupled transformations and for more thorough validation: each test case is tried with a corresponding parser and is co-transformed. This subdomain will not be addressed in this paper since it is a separate big research area and still work in progress for us.

4 Convergence Sources

```
source:
       name::id derived? source-grammar source-tree? test-set::refid*
derived:
       from::refid using::string
source-grammar:
       extraction::xstring parsing::xstring? evaluation::xstring?
source-tree:
       extraction::xstring evaluation::xstring?
```

A convergence **source** is defined at least by a name and the command that will be executed for its **extraction**. Possible additional properties include for a **derived** source its previously known (but invisible for LCI otherwise) relation to another source, which allows LCI to draw grey links in Figure 2. A grammar-based **parser** and **evaluator** detailed in the next subsections, can also be specified. The **extractor** and **evaluator** for the tree level are also optional (the corresponding parser does not make sense since

TXL	BGF
`program`	`bgf:grammar`
`defineStatement`	`bgf:production`
`repeat barLiteralsAndTypes`	`bgf:expression/choice`
`repeat literalOrType`	`bgf:expression/sequence` if length>1
`literalOrType/type/typeSpec` (depending on `opt typeRepeater/typeRepeater`)	`bgf:expression/plus/bgf:expression` or `bgf:expression/star/bgf:expression` or `bgf:expression/optional/bgf:expression` or `bgf:expression`
`literalOrType/literal`	`bgf:expression/terminal`
`typeid/id`	`nonterminal`

Table 1. The mapping between the XML output of the TXL parser and BGF.

a parse tree is stored in a BTF which is either correct by definition or filtered out by a validator). **Test sets** compatible with this source can also be listed here to be used later to find bugs in the source grammar.

One of the crucial parts of our infrastructure is the format for storing grammars. Instead of trying to model all possible peculiar or even idiosyncratic details deployed within grammar artefacts in various frameworks: semantic actions, lexical syntax descriptions, precedence declarations, classes/interfaces or elements/attributes dichotomy, etc—we opted for sacrificing them and storing only the crucial core grammar knowledge. In fact, by abstracting from these details at the extraction stage, we get an XML-based dialect of EBNF.

4.1 Extractors

Extraction happens only once per source even if the source is used more than once. When it succeeds, LCI stores the extracted grammar in order to fall back to the old snapshot if it ever goes wrong in one of the future runs. The extracted grammar is also subject to validation, in case the validator is specified.

An extractor is simply a software component that processes a software artefact and produces a BGF grammar. In the simplest case, extraction boils down to a straightforward mapping defined by a single pass over the input. Extractors are typically implemented within the computational framework of the kind of source, or in its affinity: e.g., in Prolog for DCG, in ASF+SDF for SDF, in ANTLR for ANTLR. Several examples follow.

TXL to BGF mapping. TXL [3] distribution contains a TXL grammar for TXL grammars. By using that, we can parse any correct TXL grammar and serialise the resulting abstract syntax tree in the XML form. After that the mapping becomes trivial and is easily implemented in the form of XSLT templates that match TXL tags and generate BGF tags with the equivalent internal details, as shown on Table 1.

SDF to BGF mapping. The Meta-Environment [11] contains both SDF definition for SDF definitions and the transformational facitilies needed for mapping. After specifying the mapping in ASF in the form of traversal functions and rewriting rules, this sequence of actions is required for extraction:

	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 2. Irregularities resolved by grammar extraction given the HTML source.

- ◇ `pack-sdf` for combining all extractor modules into one definition
- ◇ `sdf2table` for making a parse table out of that definition
- ◇ `eqs-dump` for compiling ASF formulæ
- ◇ `sglr` for parsing the SDF source grammar with the table
- ◇ `asfe` for rewriting the resulting parse tree
- ◇ `unparsePT` for serialising the transformed parse tree into the file

These tools are tied together by appropriate makefiles and shell scripts. The first three steps are performed once and need to be redone only if the extractor itself changes; the last three steps are executed per extracted grammar.

HTML to BGF recovery. 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 whole more implementable grammar is given at once in the last section.

The JLS is available electronically in HTML and PDF format. Neither of these formats was designed with convenient access to the grammars in mind. We have opted for the HTML format here. The grammar format slightly varies across the different JLS grammars and versions; we had to reverse engineer formatting rules from different documents and sections — in particular from [6,7,8, §2.4] and [7,8, §18].

In order to deal with irregularities of the input format, such as liberal use of markup tags, misleading indentation, duplicate definitions as well as numerous smaller issues, we needed to design and implement a non-classic parser to extract and analyse the grammar segments of the documents and to perform a recovery. About 700 fixes were performed that way, as can be seen from Table 2.

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 mech-

anism that performs transformations on each sequence of tokens that corresponds to an alternative.

The rules are implemented in Python by regular expression matching. They are applied until they are no longer applicable. Examples of them include matching up missing parentheses by deriving their absence from the context, converting improperly positioned metasymbols to terminals and removing duplicate definitions. The complete list is given with details and examples in the journal version of [16].

4.2 Parsers and Evaluators

A **parser** is one of the most commonly available grammar artefacts: it is a syntactic analyser that can tell whether some input sequence matches the given grammar. If such a tool is indeed present, it can be referenced in LCF as well and will be used for testing purposes. A compatible test set must also be provided separately.

It is possible to implement all transformation operators to be applicable not only to languages (grammars), but also to instances (parse trees). If this is done and the corresponding tree extractors and parsers are provided in LCF, then LCI is not limited to converging grammars only. For every source that has a test set attached, for every test case in that set, LCI performs coupled extraction, transformation and comparison.

Additionally, **evaluators** can be provided that can execute test cases and compare return values with expected ones (for simplicity our prototype works with integers). Test sets must be present in a unified format for LCI to figure out applicable actions. Test cases will also be validated if the validation tool is specified. The evaluators play a similar role, but their return value is not an error code, but rather the result of evaluating the given expression. The difference between an evaluator listed in the grammar properties and an evaluator given in the instance properties is that the input of the former is a correct program in the original format and the latter takes a parse tree of an instance, presented in BTF (BGF Tree Format).

5 Grammar Transformation

We have developed a suite of sophisticated grammar transformation operators that can be parametrised appropriately and called from a script. The resulting language is called XBGF (X stands for transformation), and is processed by the transformer. Some XBGF commands have been presented in [15,16], we give several examples here as well; the complete language manual is available as [22].

5.1 Unfolding

There are several folding and unfolding transformation operators in XBGF, of which the simplest one is just called **unfold**. It searches the scope for all the instances of the given nonterminal usage and replaces such occurrences with the defining expression of that nonterminal. By default the scope of the transformation is the full grammar, but it can be limited to all the definitions of one nonterminal or to one labelled production. Regardless of the specified scope, unfolding is not applied to the definition of the argument nonterminal.

The definition that is being unfolded is assumed to consist of one single production. When only one of several existing productions is used for unfolding, such a transformation makes the language (as a set of strings generated by the context-free grammar)

smaller. The corresponding XBGF command is called **downgrade**. Other refactoring variants of **unfold** operator include **inline** that unfolds the definition and purges it from the grammar, and **unchain** which removes chain productions (`a: b; b: ...;` with no other use for `b`).

5.2 Massaging

The **massage** operator is used to rewrite the grammar by local transformations such that the language generated by the grammar (or the denotation according to any other semantics for that matter) is preserved. There are two expression arguments: one to be matched, and another one that replaces the matched expression. One of them must be in a "massage relation" to the other. The scope of the transformation can be limited to one labelled production or to all productions for a specific nonterminal symbol.

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

$$
\begin{array}{lll}
x? = (x;\varepsilon) & (x?)? = x? & (x, x^\star) = x^+ \\
x? = (x?;\varepsilon) & (x?)^+ = x^\star & (x^\star, x) = x^+ \\
x^\star = (x^+;\varepsilon) & (x?)^\star = x^\star & (x?, x^\star) = x^\star \\
x^\star = (x^\star;\varepsilon) & (x^+)? = x^\star & (x^\star, x?) = x^\star \\
x? = (x?;x) & (x^+)^+ = x^+ & (x^+, x^\star) = x^+ \\
x^+ = (x^+;x) & (x^+)^\star = x^\star & (x^\star, x^+) = x^+ \\
x^\star = (x^\star;x) & (x^\star)? = x^\star & (x^+, x?) = x^+ \\
x^\star = (x?;x^+) & (x^\star)^+ = x^\star & (x?, x^+) = x^+ \\
x^\star = (x?;x^\star) & (x^\star)^\star = x^\star & (x^\star, x^\star) = x^\star \\
x^\star = (x^+;x^\star) & x = (s_1 :: x; s_2 :: x) &
\end{array}
$$

The selectors are needed in the bottom right formula because a choice between two unnamed x will always be normalized as x, as explained in §6.1.

5.3 Projection and injection

A good example of the transformation operators that do not preserve semantics of a language will be **inject** and **project**. Projection means removing components of a sequential composition, injection means adding them. The operators take one production as a parameter with additional or unnecessary components marked in a special way. For projection the transformation engine checks that the complete production exists in the grammar and replaces it with the new production with fewer components, injection works similarly, but the other way around. If the projected part is nillable, i.e. it can evaluate to ε, the operator is always semantic-decreasing and is called **disappear**. If the projected part corresponds to the concrete syntax, i.e. contains only terminal symbols, the operator preserves abstract semantics and is called **abstractize**.

5.4 Transformation Generators

Grammar convergence research has started with an objective to use programmable grammar transformations to surface the relationships between grammars extracted from sources of different nature. Hence, we mostly aimed to provide a comprehensive

transformation suite, a convergence strategy and an infrastructure support. However, at some point we found it easier to generate the scripts to resolve specific mismatches rather than to program them manually. A full-scale research on this topic remains future work, yet below we present the results obtained so far and the considerations that can serve as foundation for the next research steps.

Consider an example of converging concrete and abstract syntax definitions. This situation requires a transformation that removes all details that are specific to concrete syntax definitions, i.e., first and foremost strips all the terminals away from the grammar. Given the grammar, it is always possible to generate a sequence of transformations that will remove all the terminal symbols. It will take every production in the grammar, search for the terminals in it and if found, produce a corresponding call to **abstractize** (the name refers to going from concrete syntax to abstract syntax). For instance, given the production:

```
[ifThenElse] expr:
       "if" expr "then" expr "else" expr
```

the following transformation will be generated (the angle brackets denote parts that will be projected away):

```
abstractize(
 [ifThenElse] expr:
       ⟨"if"⟩ expr ⟨"then"⟩ expr ⟨"else"⟩ expr
);
```

Other generators we used in the FL case study were meant for removing all selectors from the grammar (works quite similar to removing terminals), for disciplined renamings (e.g., aligning all names to be lower-case) and for automated setting of the root nonterminals by evaluating them to be equal to top nonterminals of the grammar.

Eliminating all unused nonterminals can also be a valuable generator in some cases. For us it was not particularly practical since we wanted to look into each nominal difference (which unused terminal is a subtype of) in order to better align the grammars.

A more aggressive transformation generator example can be the one that **inline**s or **unchain**s all nonterminals that are used only once in the grammar. This can become a powerful tool when converging two slightly different grammars and thus can be considered a form of aggressive normalisation. We did not work out such an application scenario for grammar convergence so far.

Deyaccification [12,20], a well-studied mapping between recursion-based and iteration-based nonterminal definitions, can also be performed in an automated fashion. In general, all grammar transformations that have a precondition enabling their execution, can be generated—we only need to try to apply them everywhere and treat failed preconditions as identical transformations.

On various occasions we also talk about “vertical” and “horizontal” productions. The former means having separate productions for one nonterminal, as in:

```
x:
       foo
x:
       bar
```

The latter (horizontal) means having one production with a top choice, as in:

```
x:
       foo
       bar
```

There are also singleton productions that are neither horizontal nor vertical (as in just "`x: foo`"), and productions that can be made horizontal by distribution (as in "`x: foo | bar`"). According to this classification and to the need of grammar engineers, it is possible to define a range of generators of different aggressiveness levels that would search for horizontal productions and apply **vertical** to them; or search for vertical productions and apply **horizontal** to them; or search for potential horizontal productions and apply **distribute** and **vertical** to them; etc.

It is important to note here that even though complete investigation of the possible generators and their implementation remain future work, this alone will not be enough to replace human expertise. Semi-automation will only be shifted from "choose which transformation to apply" to "choose which generator to apply". A strongly validated strategy for automating the choice is needed, which is not easy to develop, even if possible.

6 Convergence Targets

A target needs a name and one or more branches it consists of:

```
target:
       name::id branch+
branch:
       input::refid preparation::phase? nominal-matching::phase? structural-matching::phase?
              (extension::phase | correction::phase | relaxation::phase)*
```

Each branch is defined as an input node and optionally some phases. The input can refer to a source or to another target, which is then called an **intermediate target**. Phases of convergence have been related to the strategy advised by [16], the notion is used to separate preliminary nominal matching scripts from language-preserving refactorings doing structural matching and from unsafe steps like relaxation, correction or extension. In case of no phases specified, the input grammar is propagated to the output of the branch.

```
phase:
       step::(perform-transformation::string | automated-transformation)+
automated-transformation:
       method::id result::string
```

Any transformation step is either bound to an XBGF file or relates to a generator. The latter is not necessarily a one-to-one relation, in the Java case study some scripts were designed so universally that they were re-used several times for different sources. The re-use requires higher expertise level and better accuracy in grammar manipulation, but pays off in large projects. Transformation *generators* are external tools that take a BGF grammar as an input and produce an XBGF script applicable to that grammar and containing transformations of a certain nature, see §5.4. Generators are defined at the top-level just as transformers or comparators, so that they can be applied in different places. LCI is prepared for a generator to fail or to produce inapplicable scripts.

Whenever a generator or a script fails, that branch is terminated prematurely, implying that all consecutive transformations will fail. For all branches that reach the target, their results are compared pairwise to all others. If all branches fail or the comparator reports a mismatch, the target fails.

The graph that depicts all sources and targets as vertices and all branches as edges, is called a **convergence graph** (or a convergence tree, if it is a tree). The examples have already been provided on Figures 2 and 3.

6.1 Grammar Comparison and Measurement

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 normalising grammars as a post-transformation or pre-comparison activity:

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

The output of the comparator, boiled down to the number of mismatches, is used for measuring the progress when working on huge convergence scenarios like the JLS one. We say that we face a *nominal mismatch* when a nonterminal is defined or referenced in one of the grammars but not in the other. We face a *structural mismatch* when the definitions of a shared nonterminal differ. For every nonterminal, we count the maximum number of unmatched alternatives (of either grammar) as the number of structural mismatches [16].

The Levenshtein distance and similar clone detection metrics used for code analysis [21] can be applied to grammars. The result of such comparison can possibly be suggestive enough for automated generation of transformation scripts—this is our future work in progress at the moment. There is significant related work on schema matching [19] and model diffing [4] as well.

6.2 Validation and Error Control

Validator is an optional tool that is asked to check the XML validity of every grammar produced in the convergence process. Normally all BGF grammars produced during convergence are valid, which means if validation fails, there is something fundamentally wrong with the extractor or another part that produced it. The LCI is ready for any external tool to fail or behave inappropriately. For example, the generators discussed in the previous section can fail; can produce invalid scripts; can produce inapplicable scripts; can produce scripts that produce invalid grammars. The error handling mechanism must be prepared for any of those possibilities and the report of the LCI should be useful for a grammar engineer.

6.3 Pretty-Printing

Strictly speaking, the presentation level itself is not a necessary part in the grammar convergence approach. However, the LCI does not exist in vacuum, and in this section we describe three most important application points for grammar representation. We call the presentation layer "pretty-printing" since it mostly comprises taking a grammar stored in its abstract form and serialising it in a specific concrete notation.

Debugging activities are unavoidable even for the best grammar engineers. When grammars are extracted, they need some kind of cursory examination that is cumbersome in pure XML. When grammars are compared, the comparison results need to

be presented in the most concise and the most expressive way possible. To create a comprehensive test set one needs a way to print out any particular test case in a clear form. For tasks like these in our infrastructure we have uniform pretty-printers from BGF (grammars), XBGF (transformations) and LDF (documentation).

Publishing can take a form of an example included in a paper or in a thesis, or it can be a complete hypertext manual. Somewhat more sophisticated pretty-printers are required at this stage: for instance, a language manual in LDF can be transformed to a PDF document, to an HTML web page, to a LaTeX source, to an XSL:FO sheet, etc.

Connecting to other frameworks is the most complicated form of pretty-printing. When a functionality is required by our research that is already handled by an existing framework, it is better to pretty-print the input that is expected by that framework, use the external tool, and import back the result. For instance, the DMS software engineering toolkit [1] contains much more advanced grammar comparator which we can utilise after pretty-printing BGF as DMS. Another example can be the lack of parser generation facility in our own infrastructure: the MetaEnvironment [11] can generate it for us, if we serialise BGF as SDF. (Naturally, the lexical part could not be derived and had to be added by hand).

7 Conclusion

Essentially the LCI tool set is a model-driven framework for the domain of language recovery, transformation and convergence. LCI Configuration Format is a DSL that allows a language engineer to express the domain concepts in a concise and abstract form. Several other DSLs were designed to be used for expressing grammar knowledge, transformation steps, parse trees, language documents. It has been shown both by argument and by example that utilising these DSLs helps to take on convergence scenarios of considerable size. Our future areas of research interest include both strengthening the automation aspect by providing more generators and introducing inferred transformation steps, on one hand, and widening the application area to full-scale language convergence by working on bidirectional and coupled transformations, on the other hand.

References

1. I. D. Baxter and C. W. Pidgeon. Software Change through Design Maintenance. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 250, Washington, DC, USA, 1997. IEEE Computer Society.
2. J. Bezivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, British Columbia, Canada, October 2004.
3. J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.
4. J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *Proceedings of Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *LNCS*, pages 326–340. Springer, 2008.

5. S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. *W3C Candidate Recommendation*, 30 April 2009.
6. J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
7. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.
8. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.
9. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—Reference Manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.
10. JCP JSR 31. JAXB 2.0/2.1 — Java Architecture for XML Binding, 2008.
11. P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993.
12. S. Klusener and V. Zaytsev. ISO/IEC JTC1/SC22 Document N3977—Language Standardization Needs Grammarware. Available at http://www.open-std.org/jtc1/sc22/open/n3977.pdf, 2005.
13. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M. G. J. van den Brand and R. Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.
14. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.
15. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proceedings of 7th International Conference on Integrated Formal Methods (iFM'09)*, volume 5423 of *LNCS*, pages 246–260. Springer, 2009.
16. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 178–186. IEEE, September 2009. Full version for Software Quality Journal is in print.
17. Object Management Group. *MOF 2.0/XMI Mapping*, 2.1.1 edition, December 2007.
18. T. Parr. ANTLR—ANother Tool for Language Recognition, 2008.
19. E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.
20. A. Sellink and C. Verhoef. Generation of Software Renovation Factories from Compilers. In *Proceedings of 15th International Conference on Software Maintenance (ICSM'99)*, pages 245–255, 1999.
21. R. Tiarks, R. Koschke, and R. Falke. An Assessment of Type-3 Clones as Detected by State-of-the-Art Tools. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 67–76. IEEE, September 2009.
22. V. Zaytsev. *XBGF Manual: BGF Transformation Operator Suite v.1.0*, August 2009. Available at http://slps.sf.net/xbgf.
23. V. Zaytsev and R. Lämmel. Language Documentation: Survey and Synthesis of a Unified Format. Submitted for publication; online since 7 July, 2010.