10 Languages in 10 Years

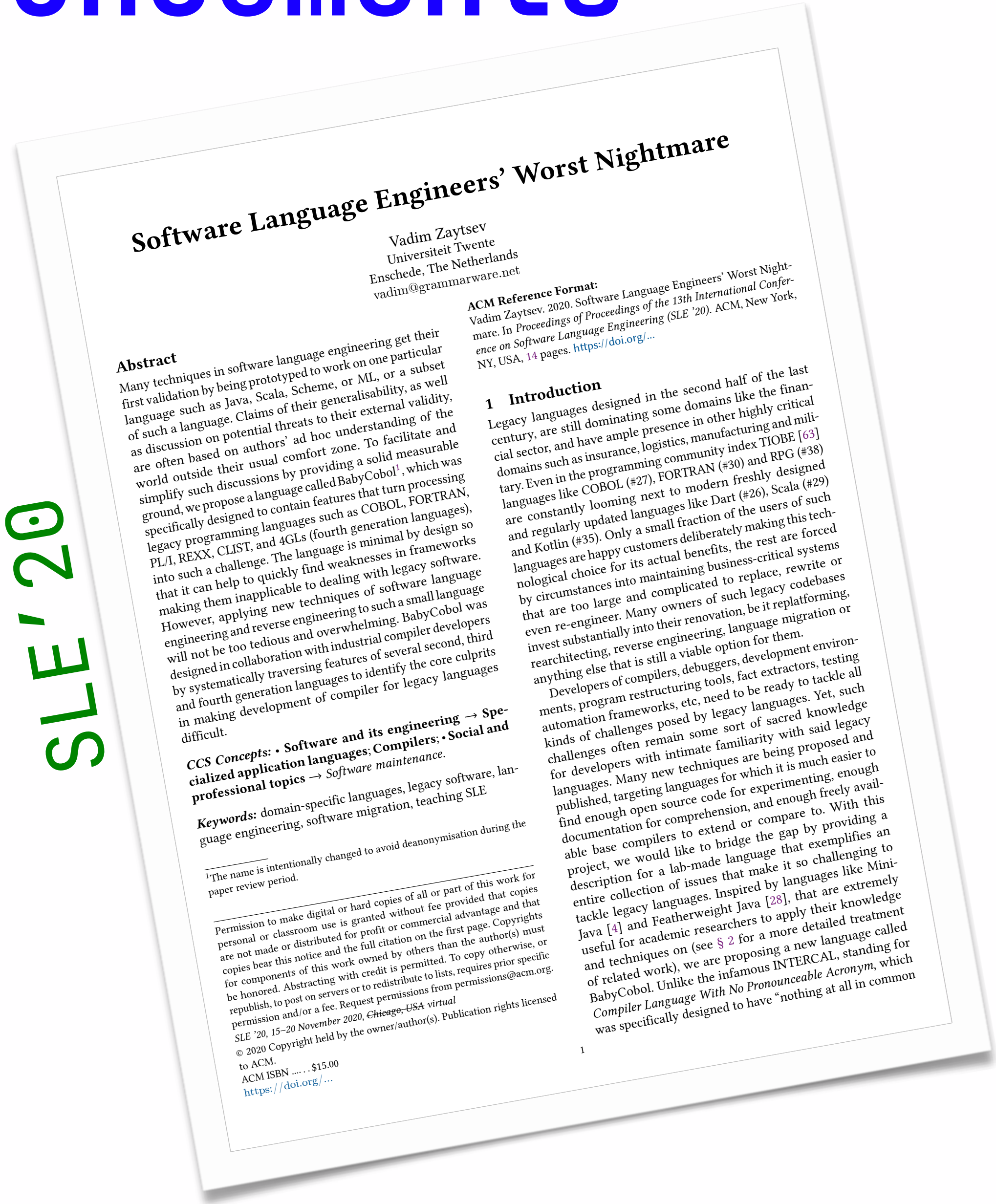
Dr. Vadim Zaytsev aka @grammarware

FMT Colloquium 2020-09-24

UNIVERSITY OF TWENTE.
Formal Methods & Tools

Announcements

SLE'20



Software Language Engineers' Worst Nightmare

Vadim Zaytsev
Universiteit Twente
Enschede, The Netherlands
vadim@grammarware.net

Abstract

Many techniques in software language engineering get their first validation by being prototyped to work on one particular language such as Java, Scala, Scheme, or ML, or a subset of such a language. Claims of their generalisability, as well as discussion on potential threats to their external validity, are often based on authors' ad hoc understanding of the world outside their usual comfort zone. To facilitate and simplify such discussions by providing a solid measurable ground, we propose a language called BabyCobol[1], which was specifically designed to contain features that turn processing legacy programming languages such as COBOL, FORTRAN, PL/I, REXX, CLIST, and 4GLs (fourth generation languages), into such a challenge. The language is minimal by design so that it can help to quickly find weaknesses in frameworks making them inapplicable to dealing with legacy software. However, applying new techniques of software language engineering and reverse engineering to such a small language will not be too tedious and overwhelming. BabyCobol was designed in collaboration with industrial compiler developers by systematically traversing features of several second, third and fourth generation languages to identify the core culprits in making development of compiler for legacy languages difficult.

CCS Concepts: • **Software and its engineering** → **Specialized application languages**; **Compilers**; • **Social and professional topics** → *Software maintenance*.

Keywords: domain-specific languages, legacy software, language engineering, software migration, teaching SLE

[1]The name is intentionally changed to avoid deanonymisation during the paper review period.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '20, 15–20 November 2020, ~~Chicago, USA~~ virtual
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN ....-.... $15.00
https://doi.org/...

ACM Reference Format:
Vadim Zaytsev. 2020. Software Language Engineers' Worst Nightmare. In *Proceedings of Proceedings of the 13th International Conference on Software Language Engineering (SLE '20)*. ACM, New York, NY, USA, 14 pages. https://doi.org/...

1 Introduction

Legacy languages designed in the second half of the last century, are still dominating some domains like the financial sector, and have ample presence in other highly critical domains such as insurance, logistics, manufacturing and military. Even in the programming community index TIOBE [63] languages like COBOL (#27), FORTRAN (#30) and RPG (#38) are constantly looming next to modern freshly designed and regularly updated languages like Dart (#26), Scala (#29) and Kotlin (#35). Only a small fraction of the users of such languages are happy customers deliberately making this technological choice for its actual benefits, the rest are forced by circumstances into maintaining business-critical systems that are too large and complicated to replace, rewrite or even re-engineer. Many owners of such legacy codebases invest substantially into their renovation, be it replatforming, rearchitecting, reverse engineering, language migration or anything else that is still a viable option for them.

Developers of compilers, debuggers, development environments, program restructuring tools, fact extractors, testing automation frameworks, etc, need to be ready to tackle all kinds of challenges posed by legacy languages. Yet, such challenges often remain some sort of sacred knowledge for developers with intimate familiarity with said legacy languages. Many new techniques are being proposed and published, targeting languages for which it is much easier to find enough open source code for experimenting, enough documentation for comprehension, and enough freely available base compilers to extend or compare to. With this project, we would like to bridge the gap by providing an description for a lab-made language that exemplifies an entire collection of issues that make it so challenging to tackle legacy languages. Inspired by languages like MiniJava [4] and Featherweight Java [28], that are extremely useful for academic researchers to apply their knowledge and techniques on (see § 2 for a more detailed treatment of related work), we are proposing a new language called BabyCobol. Unlike the infamous INTERCAL, standing for *Compiler Language With No Pronounceable Acronym*, which was specifically designed to have "nothing at all in common

1

PROFES'20

Improving a Software Modernisation Process by Differencing Migration Logs

Céline Deknop[1,2], Johan Fabry[2], Kim Mens[1], and Vadim Zaytsev[2,3]
[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
[2] Raincode Labs, Brussels, Belgium
[3] Universiteit Twente, Enschede, The Netherlands
celine.deknop@uclouvain.be, johan@raincode.com, kim.mens@uclouvain.be, vadim@grammarware.net

Abstract. Software written in legacy programming languages is notoriously ubiquitous and often comprises business-critical portions of codebases and portfolios. Some of these languages, like COBOL, mature, grow, and acquire modern tooling that makes maintenance activities more bearable. Others, like many *fourth generation languages* (4GLs), stagnate and become obsolete and unmaintained or unmaintainable, which first urges and eventually forces migrating to other languages, if the software needs to be kept in production. In this paper, we dissect a software modernisation process endorsed by Raincode Labs, utilised in particular to migrate software from a 4GL called PACBASE, to pure COBOL. Having migrated upwards of 500 MLOC of production code to COBOL using this process, the company has ample experience with this process. Nevertheless, we identify some improvement points and explain the technical side of a possible solution, based on migration log differencing, that is currently being put to the test by Raincode migration engineers.

Keywords: Software modernisation, legacy programming languages, software migration, software evolution, code differencing, COBOL, PACBASE, 4GL

1 Introduction

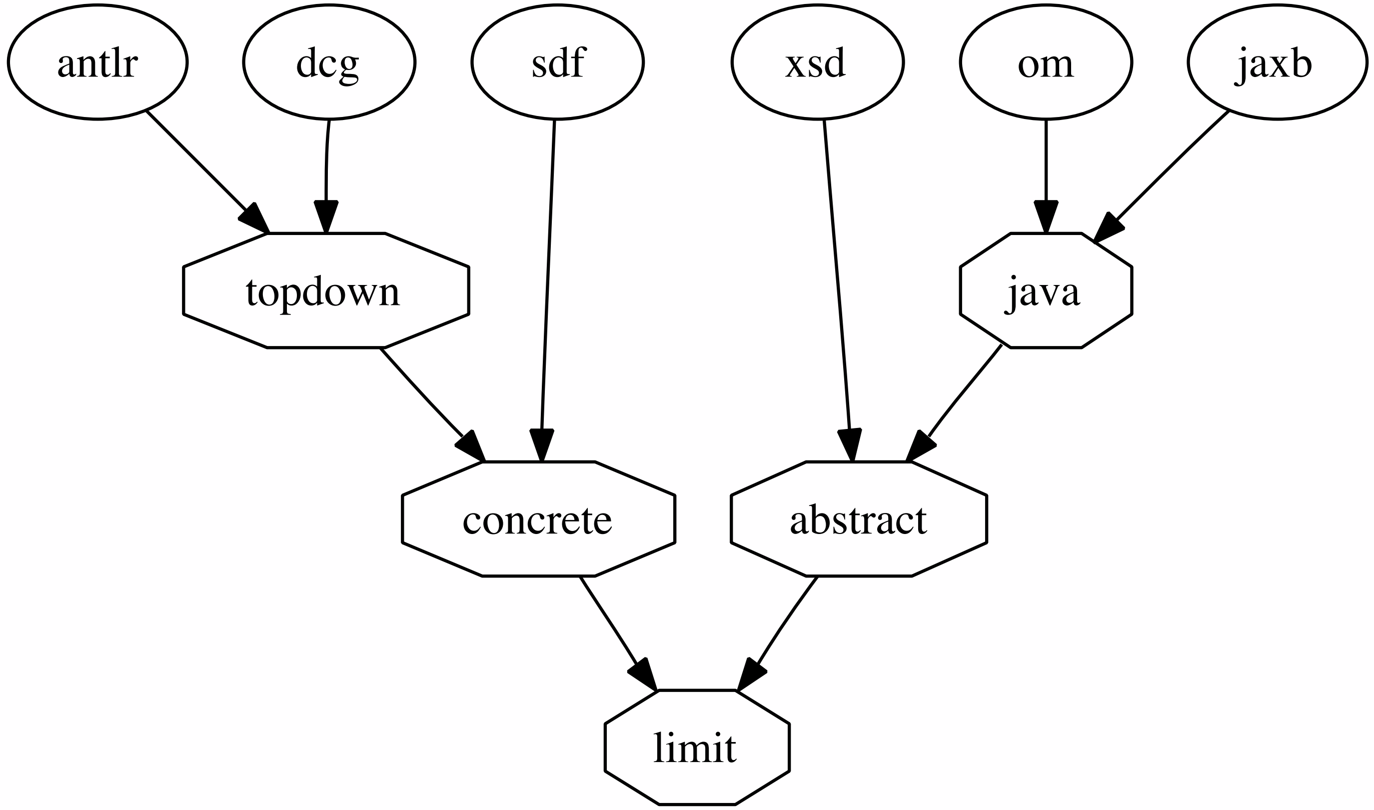
When COBOL was first introduced and published in 1960 [6], it enabled writing software that replaced the manual labour of thousands of people previously performing pen-and-paper bookkeeping or at best manual data entry and manipulation. When 4GLs (fourth generation languages) started emerging, they allowed developers to write significantly shorter programs, and enabled automated generation of dozens pages of COBOL code from a single statement [22,29]. Nowadays, in the era of intentionally designed software languages [18] and domain-specific languages [31], conciseness and brevity is appreciated as much as readability, testability, understandability and ultimately, maintainability [9]. Yet, legacy software continues to exist due to the sheer volume of it: just COBOL alone is estimated to have at least 220 billion lines of code worldwide, according to various

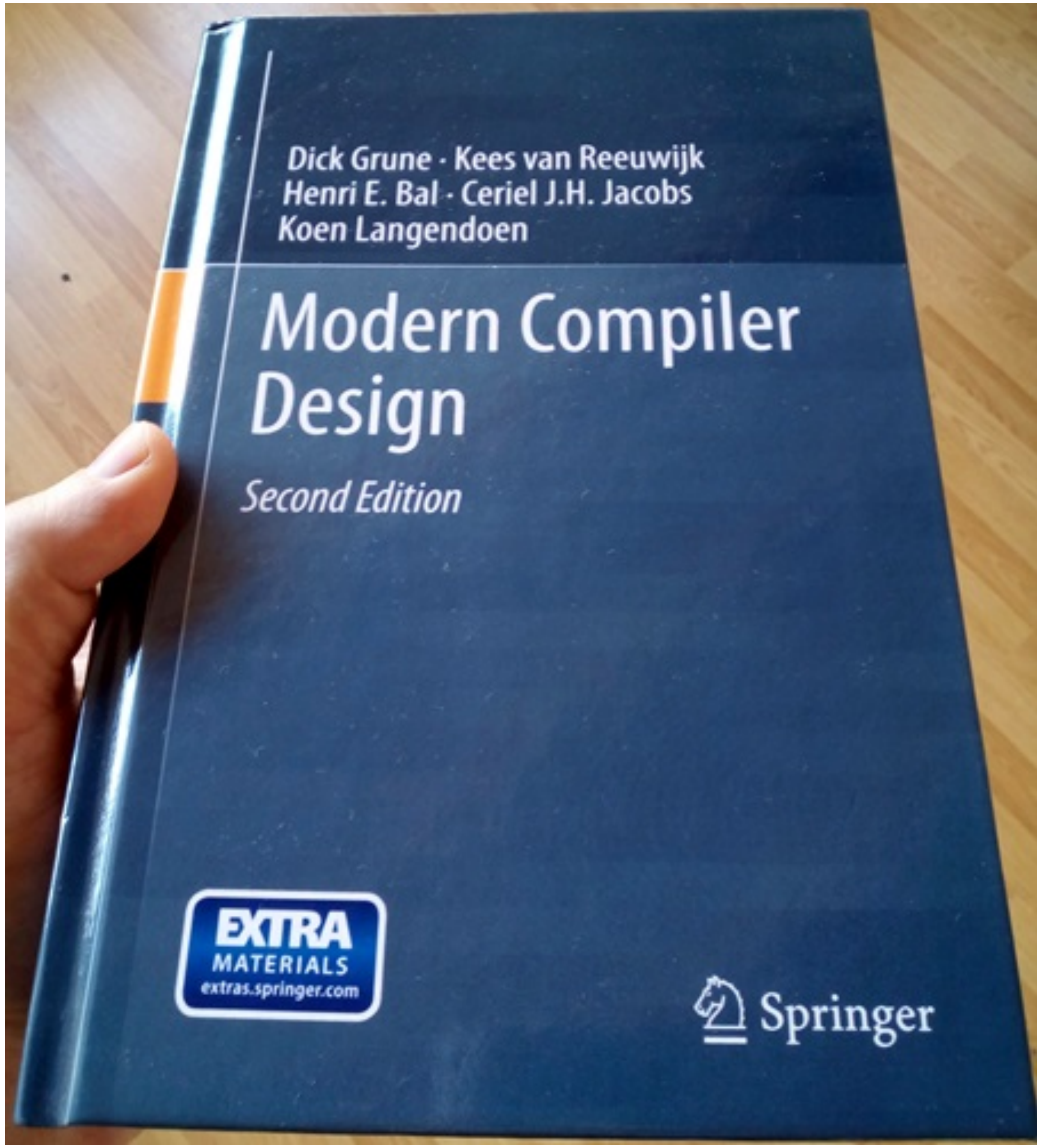
Personal Path

- First paper in 2000
- Real life starts in 2010
- ~⅓ in pure research
  - postdoc @ CWI
- ~⅓ in pure education
  - lecturer @ UvA
- ~⅓ in pure industry
  - developer @ Raincode

http://grammarware.net || grammarware.github.io

1.BGF

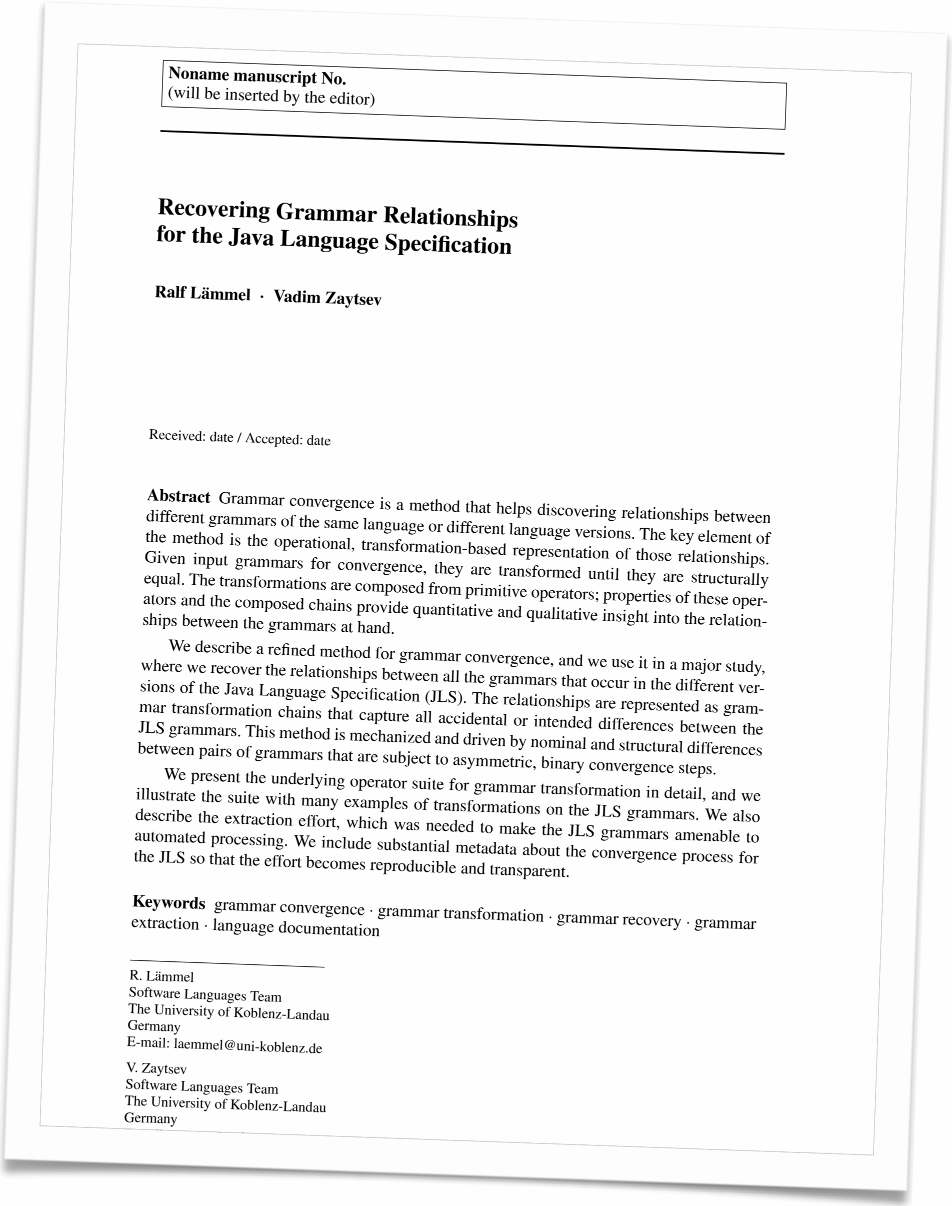




Convergence: [iFM’09] [GTTSE’09] [SQJ’11] [SLE’13]

1.BGF

- Convergence
  - errors in Java language spec



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

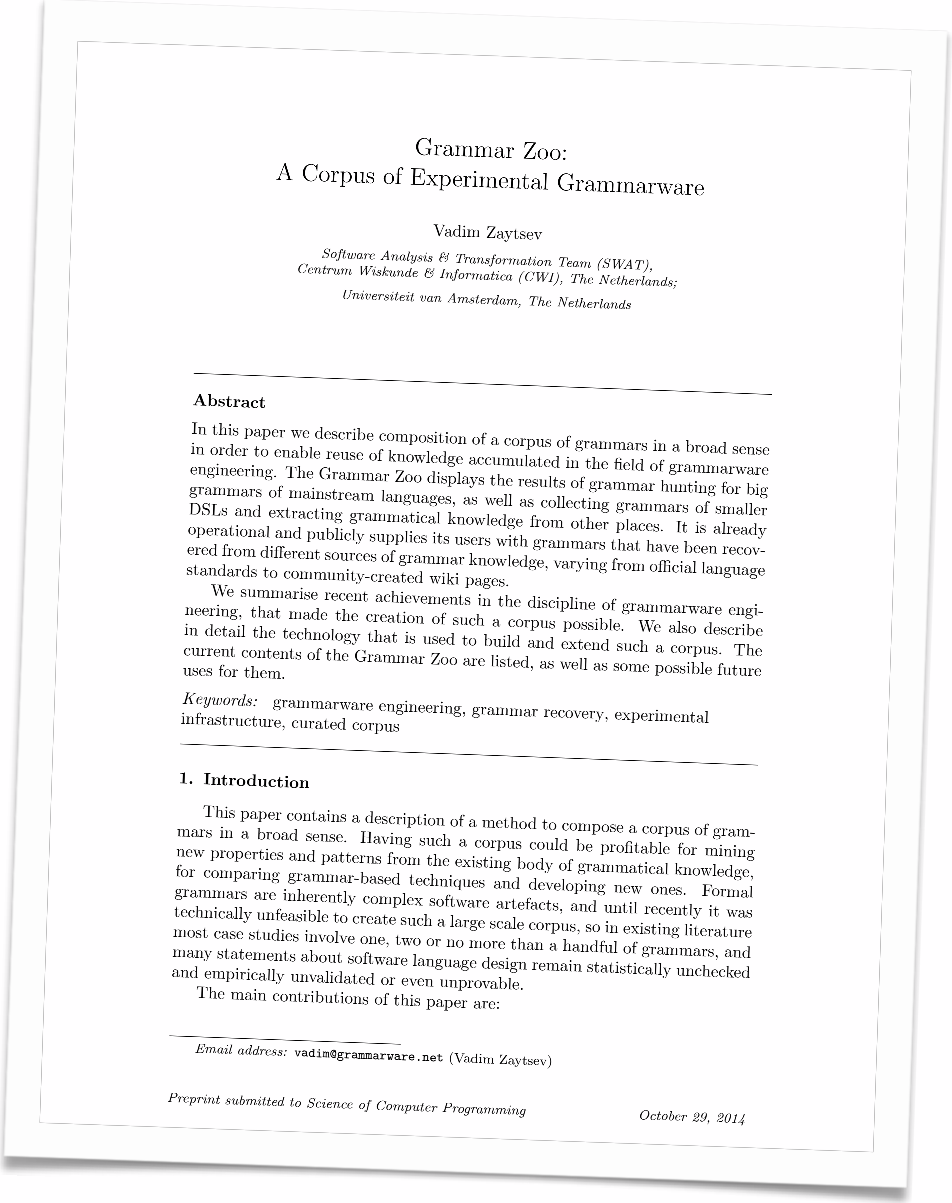
V. Zaytsev
Software Languages Team
The University of Koblenz-Landau
Germany

Recovery: [SQJ'11] [arXiv] [LDTA'12]

UNIVERSITY OF TWENTE. Formal Methods & Tools

1.BGF

- Convergence
  - errors in Java language spec
- Recovery
  - Grammar Zoo



Grammar Zoo:
A Corpus of Experimental Grammarware

Vadim Zaytsev

Software Analysis & Transformation Team (SWAT), Centrum Wiskunde & Informatica (CWI), The Netherlands; Universiteit van Amsterdam, The Netherlands

Abstract

In this paper we describe composition of a corpus of grammars in a broad sense in order to enable reuse of knowledge accumulated in the field of grammarware engineering. The Grammar Zoo displays the results of grammar hunting for big grammars of mainstream languages, as well as collecting grammars of smaller DSLs and extracting grammatical knowledge from other places. It is already operational and publicly supplies its users with grammars that have been recovered from different sources of grammar knowledge, varying from official language standards to community-created wiki pages.

We summarise recent achievements in the discipline of grammarware engineering, that made the creation of such a corpus possible. We also describe in detail the technology that is used to build and extend such a corpus. The current contents of the Grammar Zoo are listed, as well as some possible future uses for them.

Keywords: grammarware engineering, grammar recovery, experimental infrastructure, curated corpus

1. Introduction

This paper contains a description of a method to compose a corpus of grammars in a broad sense. Having such a corpus could be profitable for mining new properties and patterns from the existing body of grammatical knowledge, for comparing grammar-based techniques and developing new ones. Formal grammars are inherently complex software artefacts, and until recently it was technically unfeasible to create such a large scale corpus, so in existing literature most case studies involve one, two or no more than a handful of grammars, and many statements about software language design remain statistically unchecked and empirically unvalidated or even unprovable.

The main contributions of this paper are:

Email address: vadim@grammarware.net (Vadim Zaytsev)

Preprint submitted to Science of Computer Programming *October 29, 2014*

Recovery: [SQJ'11] [arXiv] [LDTA'12]

1.BGF

- Convergence
  - errors in Java language spec
- Recovery
  - Grammar Zoo
- Transformation
  - XBGF, SLEIR, GLUE, …


EASST

ECEASST

Cotransforming Grammars with Shared Packed Parse Forests

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract: SPPF (shared packed parse forest) is the best known graph representation of a parse forest (family of related parse trees) used in parsing with ambiguous/conjunctive grammars. Systematic general purpose transformations of SPPFs have never been investigated and are considered to be an open problem in software language engineering. In this paper, we motivate the necessity of having a transformation operator suite for SPPFs and extend the state of the art grammar transformation operator suite to metamodel/model (grammar/graph) cotransformations.

Keywords: cotransformation, generalised parsing, parse graphs

1 Motivation

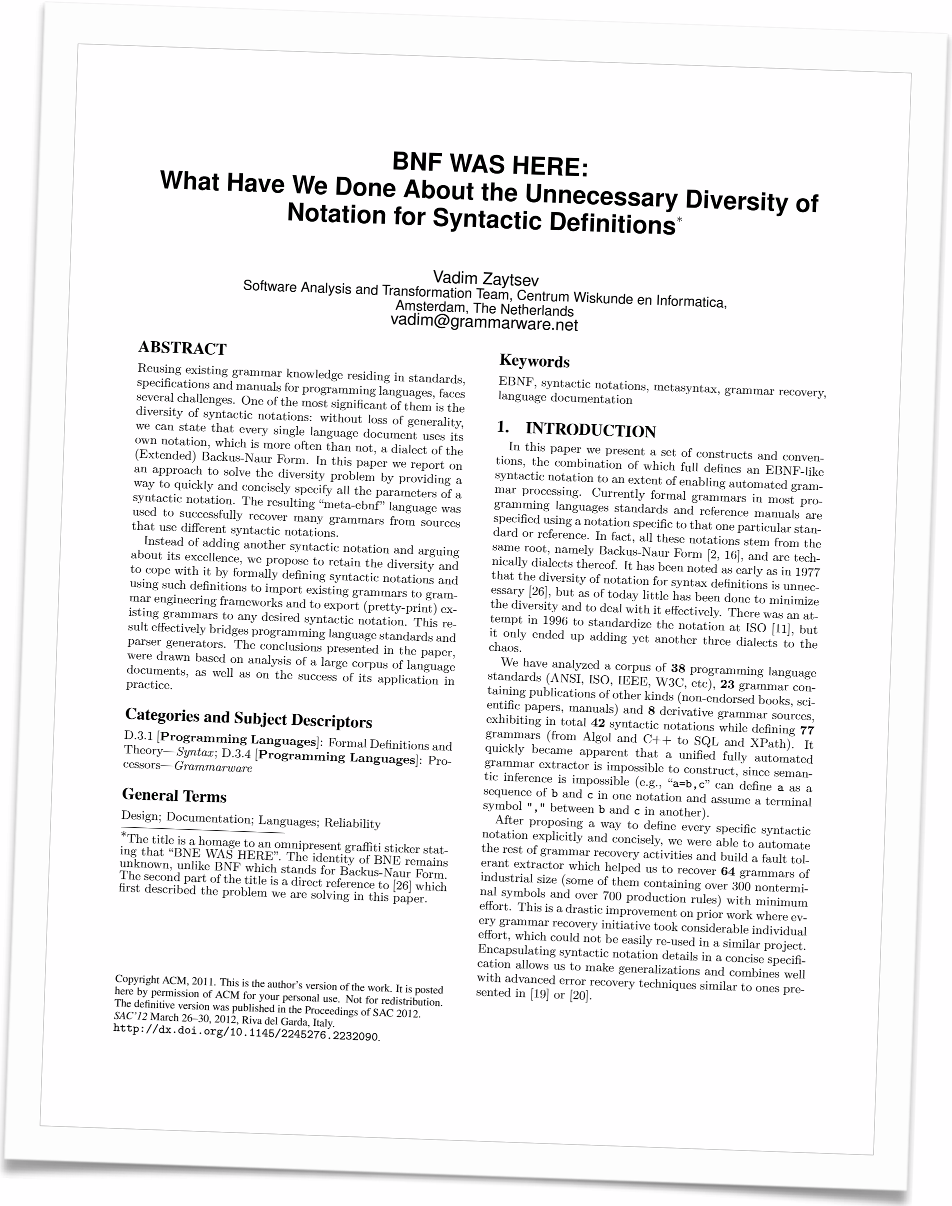
Classically, parsing consumes a string of characters or tokens, recognises its grammatical structure and produces a corresponding parse tree [ASU85]. A more modern perspective is that parsing recognises structure and expresses it explicitly [ZB14]. In many situations, trees appear to be unsatisfactory target data structures: they can express hierarchy easily, but any other structural commitments require special tricks and encodings, which are much less preferable than switching to graphs or pseudographs [SL13]. The most common scenarios include expressing uncertainty (e.g., in generalised parsing), maintaining several structural views (e.g., in the style of Boolean grammars) or manipulating recursive structures (e.g., with structured graphs).

Generalised parsing algorithms (GLR [Tom85], SGLR [Vis97], GLL [SJ10a], RIGLR [SJ05], etc.) differ from their classic counterparts in dealing with ambiguity [BSVV02, BV11]: instead of trying to avoid, ignore or report ambiguous cases, they are expressed explicitly in so called parse forests. Formally, a parse forest is a set of equally grammatically correct parse trees. Some of them may be semantically different, which makes such ambiguity significant and usually undesirable. In practice, such sets usually need to be filtered or ranked in order to make full use of the available tree-based approaches to program analysis and transformation. In *Boolean grammars* [Okh04] and *conjunctive grammars* [Okh01], we can use conjunctive clauses in a grammar to explicitly specify several syntactically different yet equally grammatical views of the same input fragment — they can be semantically equivalent [SC15] or one branch strictly more expressive than the other [Zay13]. Parsing techniques can utilise such specifications to create special kinds of nodes in a parse tree whose descendant subtrees share leaves [Okh13, Ste15]. Shared recursive structures are also facilitated by *parametric higher-order abstract syntax* [PE88, DPS97, Chl08]. It is an advanced method with high expressiveness, but it often requires similarly advanced techniques like multilevel metareasoning [MA03] and demands the use of automated theorem provers [DFH95, RHB01]. For now we will focus on the first two cases, since both kinds of structures defined by those two related approaches conceptually are *parse forests*.

1 / 21 Volume 73 (2016)

Recovery: [SQJ'11] [arXiv] [LDTA'12]

1.BGF



BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions*

Vadim Zaytsev
Software Analysis and Transformation Team, Centrum Wiskunde en Informatica,
Amsterdam, The Netherlands
vadim@grammarware.net

ABSTRACT

Reusing existing grammar knowledge residing in standards, specifications and manuals for programming languages, faces several challenges. One of the most significant of them is the diversity of syntactic notations: without loss of generality, we can state that every single language document uses its own notation, which is more often than not, a dialect of the (Extended) Backus-Naur Form. In this paper we report on an approach to solve the diversity problem by providing a way to quickly and concisely specify all the parameters of a syntactic notation. The resulting "meta-ebnf" language was used to successfully recover many grammars from sources that use different syntactic notations.

Instead of adding another syntactic notation and arguing about its excellence, we propose to retain the diversity and to cope with it by formally defining syntactic notations and using such definitions to import existing grammars to grammar engineering frameworks and to export (pretty-print) existing grammars to any desired syntactic notation. This result effectively bridges programming language standards and parser generators. The conclusions presented in the paper, were drawn based on analysis of a large corpus of language documents, as well as on the success of its application in practice.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; D.3.4 [**Programming Languages**]: Processors—*Grammarware*

General Terms

Design; Documentation; Languages; Reliability

*The title is a homage to an omnipresent graffiti sticker stating that "BNE WAS HERE". The identity of BNE remains unknown, unlike BNF which stands for Backus-Naur Form. The second part of the title is a direct reference to [26] which first described the problem we are solving in this paper.

The definitive version was published in the Proceedings of SAC 2012. *SAC'12* March 26–30, 2012, Riva del Garda, Italy.
http://dx.doi.org/10.1145/2245276.2232090.

Keywords

EBNF, syntactic notations, metasyntax, grammar recovery, language documentation

1. INTRODUCTION

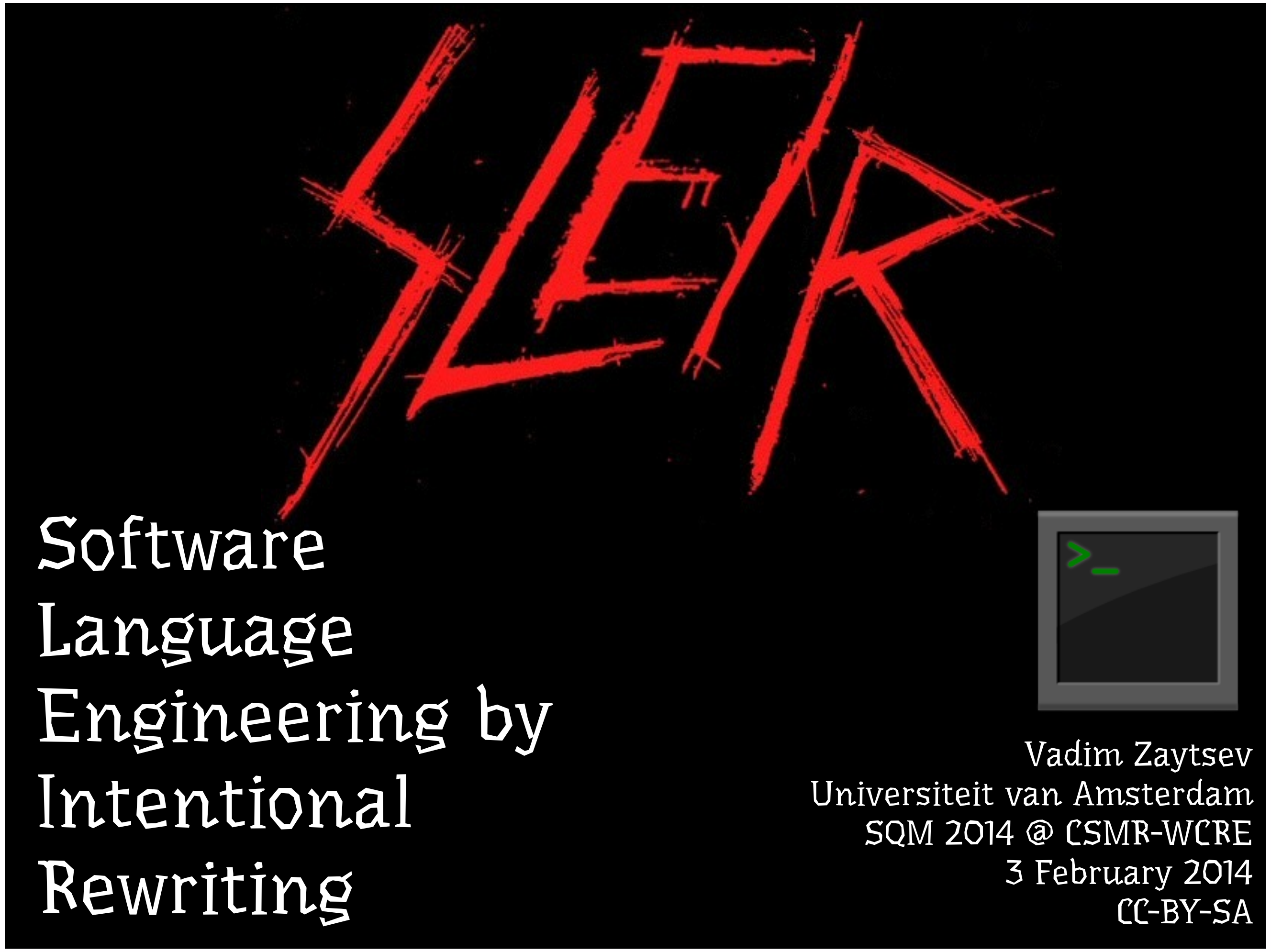
In this paper we present a set of constructs and conventions, the combination of which full defines an EBNF-like syntactic notation to an extent of enabling automated grammar processing. Currently formal grammars in most programming languages standards and reference manuals are specified using a notation specific to that one particular standard or reference. In fact, all these notations stem from the same root, namely Backus-Naur Form [2, 16], and are technically dialects thereof. It has been noted as early as in 1977 that the diversity of notation for syntax definitions is unnecessary [26], but as of today little has been done to minimize the diversity and to deal with it effectively. There was an attempt in 1996 to standardize the notation at ISO [11], but it only ended up adding yet another three dialects to the chaos.

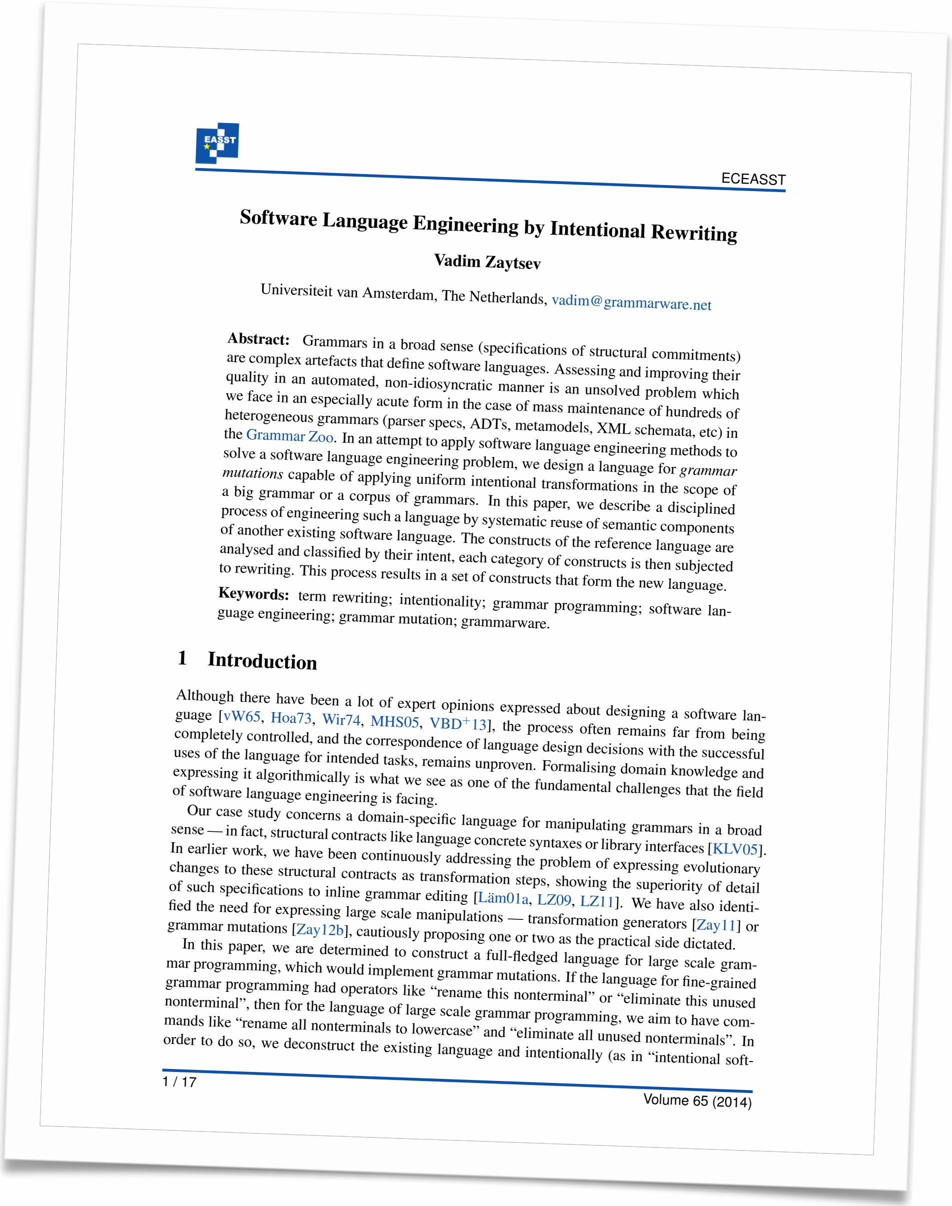
We have analyzed a corpus of **38** programming language standards (ANSI, ISO, IEEE, W3C, etc), **23** grammar containing publications of other kinds (non-endorsed books, scientific papers, manuals) and **8** derivative grammar sources, exhibiting in total **42** syntactic notations while defining **77** grammars (from Algol and C++ to SQL and XPath). It quickly became apparent that a unified fully automated grammar extractor is impossible to construct, since semantic inference is impossible (e.g., "`a=b,c`" can define `a` as a sequence of `b` and `c` in one notation and assume a terminal symbol `","` between `b` and `c` in another).

After proposing a way to define every specific syntactic notation explicitly and concisely, we were able to automate the rest of grammar recovery activities and build a fault tolerant extractor which helped us to recover **64** grammars of industrial size (some of them containing over 300 nonterminal symbols and over 700 production rules) with minimum effort. This is a drastic improvement on prior work where every grammar recovery initiative took considerable individual effort, which could not be easily re-used in a similar project. Encapsulating syntactic notation details in a concise specification allows us to make generalizations and combines well with advanced error recovery techniques similar to ones presented in [19] or [20].

[SAC'12]

1.BGF





ECEASST

Software Language Engineering by Intentional Rewriting

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract: Grammars in a broad sense (specifications of structural commitments) are complex artefacts that define software languages. Assessing and improving their quality in an automated, non-idiosyncratic manner is an unsolved problem which we face in an especially acute form in the case of mass maintenance of hundreds of heterogeneous grammars (parser specs, ADTs, metamodels, XML schemata, etc) in the Grammar Zoo. In an attempt to apply software language engineering methods to solve a software language engineering problem, we design a language for *grammar mutations* capable of applying uniform intentional transformations in the scope of a big grammar or a corpus of grammars. In this paper, we describe a disciplined process of engineering such a language by systematic reuse of semantic components of another existing software language. The constructs of the reference language are analysed and classified by their intent, each category of constructs is then subjected to rewriting. This process results in a set of constructs that form the new language.

Keywords: term rewriting; intentionality; grammar programming; software language engineering; grammar mutation; grammarware.

1 Introduction

Although there have been a lot of expert opinions expressed about designing a software language [vW65, Hoa73, Wir74, MHS05, VBD+13], the process often remains far from being completely controlled, and the correspondence of language design decisions with the successful uses of the language for intended tasks, remains unproven. Formalising domain knowledge and expressing it algorithmically is what we see as one of the fundamental challenges that the field of software language engineering is facing.

Our case study concerns a domain-specific language for manipulating grammars in a broad sense — in fact, structural contracts like language concrete syntaxes or library interfaces [KLV05]. In earlier work, we have been continuously addressing the problem of expressing evolutionary changes to these structural contracts as transformation steps, showing the superiority of detail of such specifications to inline grammar editing [Läm01a, LZ09, LZ11]. We have also identified the need for expressing large scale manipulations — transformation generators [Zay11] or grammar mutations [Zay12b], cautiously proposing one or two as the practical side dictated.

In this paper, we are determined to construct a full-fledged language for large scale grammar programming, which would implement grammar mutations. If the language for fine-grained grammar programming had operators like "rename this nonterminal" or "eliminate this unused nonterminal", then for the language of large scale grammar programming, we aim to have commands like "rename all nonterminals to lowercase" and "eliminate all unused nonterminals". In order to do so, we deconstruct the existing language and intentionally (as in "intentional soft-

1 / 17 Volume 65 (2014)

[SQM'14]

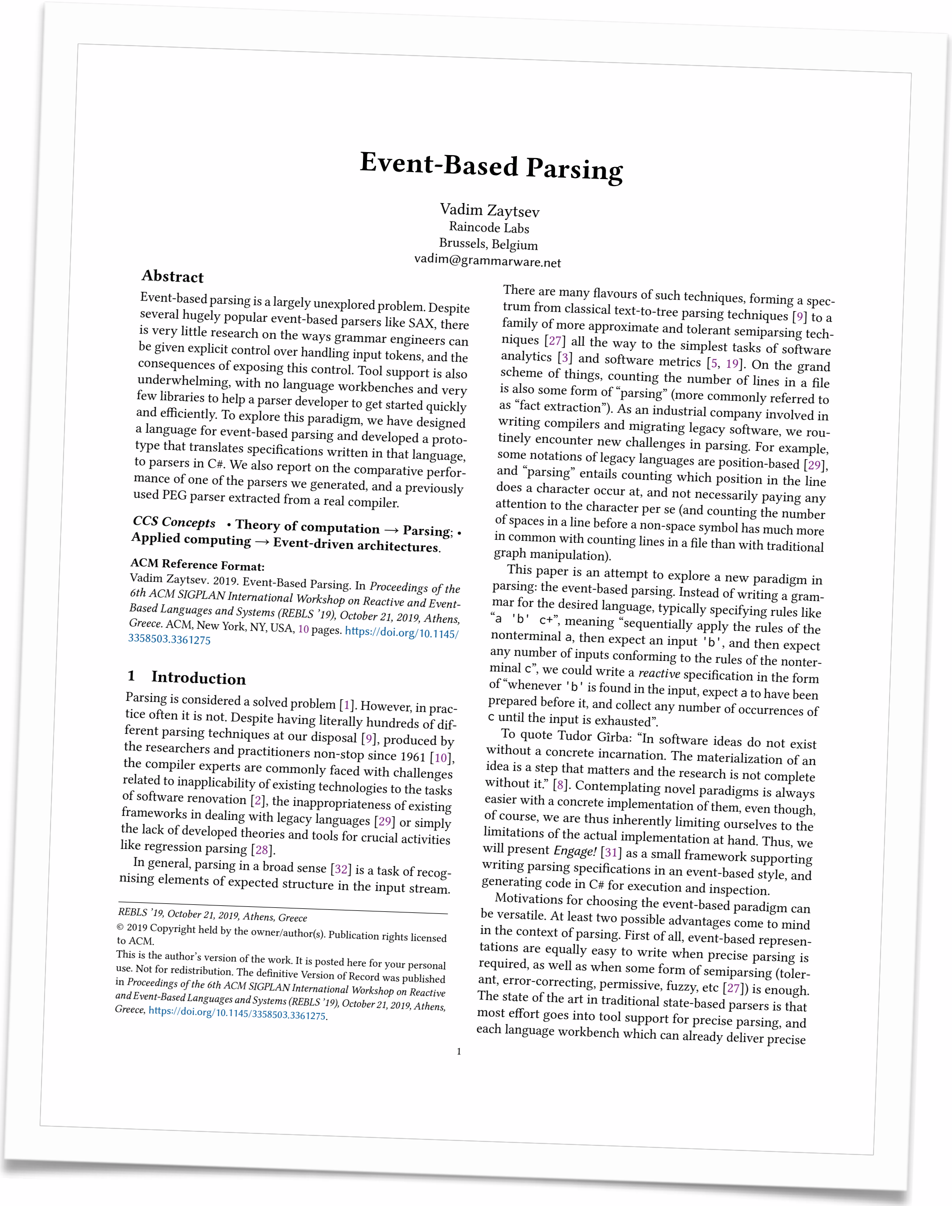
2.Rascal

- Grammar Laboratory
  - Grammar *Library*
- Micropatterns [SLE’13]
- Smells [SLE’17]
- BOOL [NOOL’17]

- Also used externally [SPE]

[SLE’13] [SLE’17] [NOOL’17]

3.Engage!



Event-Based Parsing

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadim@grammarware.net

Abstract

Event-based parsing is a largely unexplored problem. Despite several hugely popular event-based parsers like SAX, there is very little research on the ways grammar engineers can be given explicit control over handling input tokens, and the consequences of exposing this control. Tool support is also underwhelming, with no language workbenches and very few libraries to help a parser developer to get started quickly and efficiently. To explore this paradigm, we have designed a language for event-based parsing and developed a prototype that translates specifications written in that language, to parsers in C#. We also report on the comparative performance of one of the parsers we generated, and a previously used PEG parser extracted from a real compiler.

CCS Concepts • **Theory of computation → Parsing**; • **Applied computing → Event-driven architectures**.

ACM Reference Format:
Vadim Zaytsev. 2019. Event-Based Parsing. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '19), October 21, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358503.3361275

1 Introduction

Parsing is considered a solved problem [1]. However, in practice often it is not. Despite having literally hundreds of different parsing techniques at our disposal [9], produced by the researchers and practitioners non-stop since 1961 [10], the compiler experts are commonly faced with challenges related to inapplicability of existing technologies to the tasks of software renovation [2], the inappropriateness of existing frameworks in dealing with legacy languages [29] or simply the lack of developed theories and tools for crucial activities like regression parsing [28].

In general, parsing in a broad sense [32] is a task of recognising elements of expected structure in the input stream.

REBLS '19, October 21, 2019, Athens, Greece

There are many flavours of such techniques, forming a spectrum from classical text-to-tree parsing techniques [9] to a family of more approximate and tolerant semiparsing techniques [27] all the way to the simplest tasks of software analytics [3] and software metrics [5, 19]. On the grand scheme of things, counting the number of lines in a file is also some form of "parsing" (more commonly referred to as "fact extraction"). As an industrial company involved in writing compilers and migrating legacy software, we routinely encounter new challenges in parsing. For example, some notations of legacy languages are position-based [29], and "parsing" entails counting which position in the line does a character occur at, and not necessarily paying any attention to the character per se (and counting the number of spaces in a line before a non-space symbol has much more in common with counting lines in a file than with traditional graph manipulation).

This paper is an attempt to explore a new paradigm in parsing: the event-based parsing. Instead of writing a grammar for the desired language, typically specifying rules like "`a 'b' c+`", meaning "sequentially apply the rules of the nonterminal `a`, then expect an input `'b'`, and then expect any number of inputs conforming to the rules of the nonterminal `c`", we could write a *reactive* specification in the form of "whenever `'b'` is found in the input, expect `a` to have been prepared before it, and collect any number of occurrences of `c` until the input is exhausted".

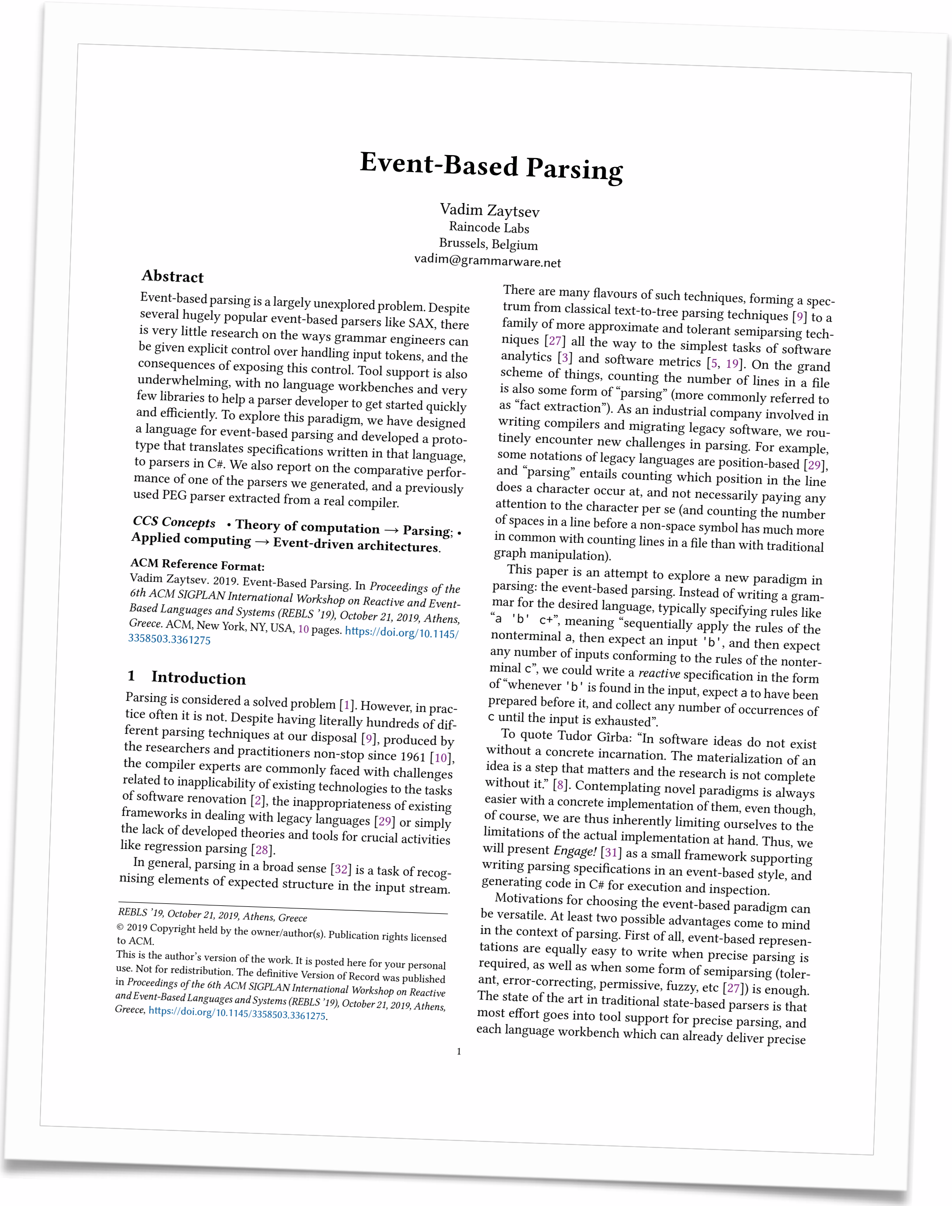
To quote Tudor Gîrba: "In software ideas do not exist without a concrete incarnation. The materialization of an idea is a step that matters and the research is not complete without it." [8]. Contemplating novel paradigms is always easier with a concrete implementation of them, even though, of course, we are thus inherently limiting ourselves to the limitations of the actual implementation at hand. Thus, we will present *Engage!* [31] as a small framework supporting writing parsing specifications in an event-based style, and generating code in C# for execution and inspection.

Motivations for choosing the event-based paradigm can be versatile. At least two possible advantages come to mind in the context of parsing. First of all, event-based representations are equally easy to write when precise parsing is required, as well as when some form of semiparsing (tolerant, error-correcting, permissive, fuzzy, etc [27]) is enough. The state of the art in traditional state-based parsers is that most effort goes into tool support for precise parsing, and each language workbench which can already deliver precise

[REBLS'19]

3.Engage!

```
namespace AB
types
    ABProgram;
    Integer, String, Decimal <: Type;
    Decl;
    Var, Lit <: Expr;
tokens
    ' ', '\r', '\n' :: skip
    ';', '(', ')' :: mark
    'dcl', 'enddcl', 'integer', 'dec' :: word
    number :: Num
    string :: Id
handlers
    EOF                  -> push ABProgram(data,code)
                            where code := pop# Stmt,
                                  data := pop# Decl
    Num                  -> push Lit(this)
    'dcl'                -> lift DCL
    'enddcl'             -> drop DCL
    ';' upon DCL         -> push Decl(v,t)
                            where t := pop Type,
                                  v := pop Var
    'integer' upon DCL -> push Integer
    'dec'     upon DCL -> push Decimal(n)
                            where x := await (Lit upon BRACKET) with DEC,
                                  n := tear x
    '(' upon DEC         -> lift BRACKET
    ')'                  -> drop BRACKET
```



Event-Based Parsing

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadim@grammarware.net

Abstract

Event-based parsing is a largely unexplored problem. Despite several hugely popular event-based parsers like SAX, there is very little research on the ways grammar engineers can be given explicit control over handling input tokens, and the consequences of exposing this control. Tool support is also underwhelming, with no language workbenches and very few libraries to help a parser developer to get started quickly and efficiently. To explore this paradigm, we have designed a language for event-based parsing and developed a prototype that translates specifications written in that language, to parsers in C#. We also report on the comparative performance of one of the parsers we generated, and a previously used PEG parser extracted from a real compiler.

CCS Concepts • **Theory of computation → Parsing**; • **Applied computing → Event-driven architectures.**

ACM Reference Format:
Vadim Zaytsev. 2019. Event-Based Parsing. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '19), October 21, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358503.3361275

1 Introduction

Parsing is considered a solved problem [1]. However, in practice often it is not. Despite having literally hundreds of different parsing techniques at our disposal [9], produced by the researchers and practitioners non-stop since 1961 [10], the compiler experts are commonly faced with challenges related to inapplicability of existing technologies to the tasks of software renovation [2], the inappropriateness of existing frameworks in dealing with legacy languages [29] or simply the lack of developed theories and tools for crucial activities like regression parsing [28].

In general, parsing in a broad sense [32] is a task of recognising elements of expected structure in the input stream.

REBLS '19, October 21, 2019, Athens, Greece

There are many flavours of such techniques, forming a spectrum from classical text-to-tree parsing techniques [9] to a family of more approximate and tolerant semiparsing techniques [27] all the way to the simplest tasks of software analytics [3] and software metrics [5, 19]. On the grand scheme of things, counting the number of lines in a file is also some form of "parsing" (more commonly referred to as "fact extraction"). As an industrial company involved in writing compilers and migrating legacy software, we routinely encounter new challenges in parsing. For example, some notations of legacy languages are position-based [29], and "parsing" entails counting which position in the line does a character occur at, and not necessarily paying any attention to the character per se (and counting the number of spaces in a line before a non-space symbol has much more in common with counting lines in a file than with traditional graph manipulation).

This paper is an attempt to explore a new paradigm in parsing: the event-based parsing. Instead of writing a grammar for the desired language, typically specifying rules like "a 'b' c+", meaning "sequentially apply the rules of the nonterminal a, then expect an input 'b', and then expect any number of inputs conforming to the rules of the nonterminal c", we could write a *reactive* specification in the form of "whenever 'b' is found in the input, expect a to have been prepared before it, and collect any number of occurrences of c until the input is exhausted".

To quote Tudor Gîrba: "In software ideas do not exist without a concrete incarnation. The materialization of an idea is a step that matters and the research is not complete without it." [8]. Contemplating novel paradigms is always easier with a concrete implementation of them, even though, of course, we are thus inherently limiting ourselves to the limitations of the actual implementation at hand. Thus, we will present *Engage!* [31] as a small framework supporting writing parsing specifications in an event-based style, and generating code in C# for execution and inspection.

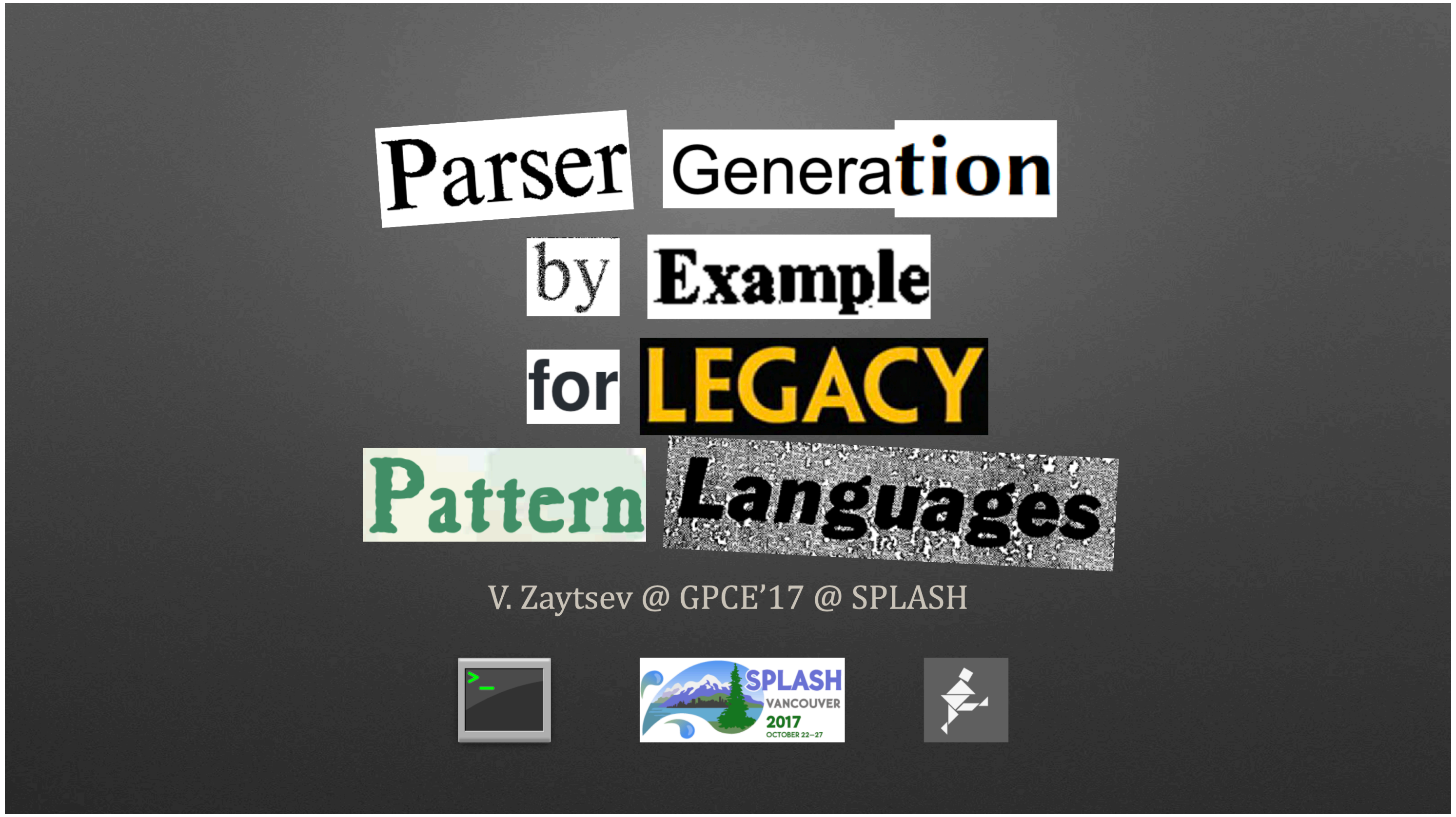
Motivations for choosing the event-based paradigm can be versatile. At least two possible advantages come to mind in the context of parsing. First of all, event-based representations are equally easy to write when precise parsing is required, as well as when some form of semiparsing (tolerant, error-correcting, permissive, fuzzy, etc [27]) is enough. The state of the art in traditional state-based parsers is that most effort goes into tool support for precise parsing, and each language workbench which can already deliver precise

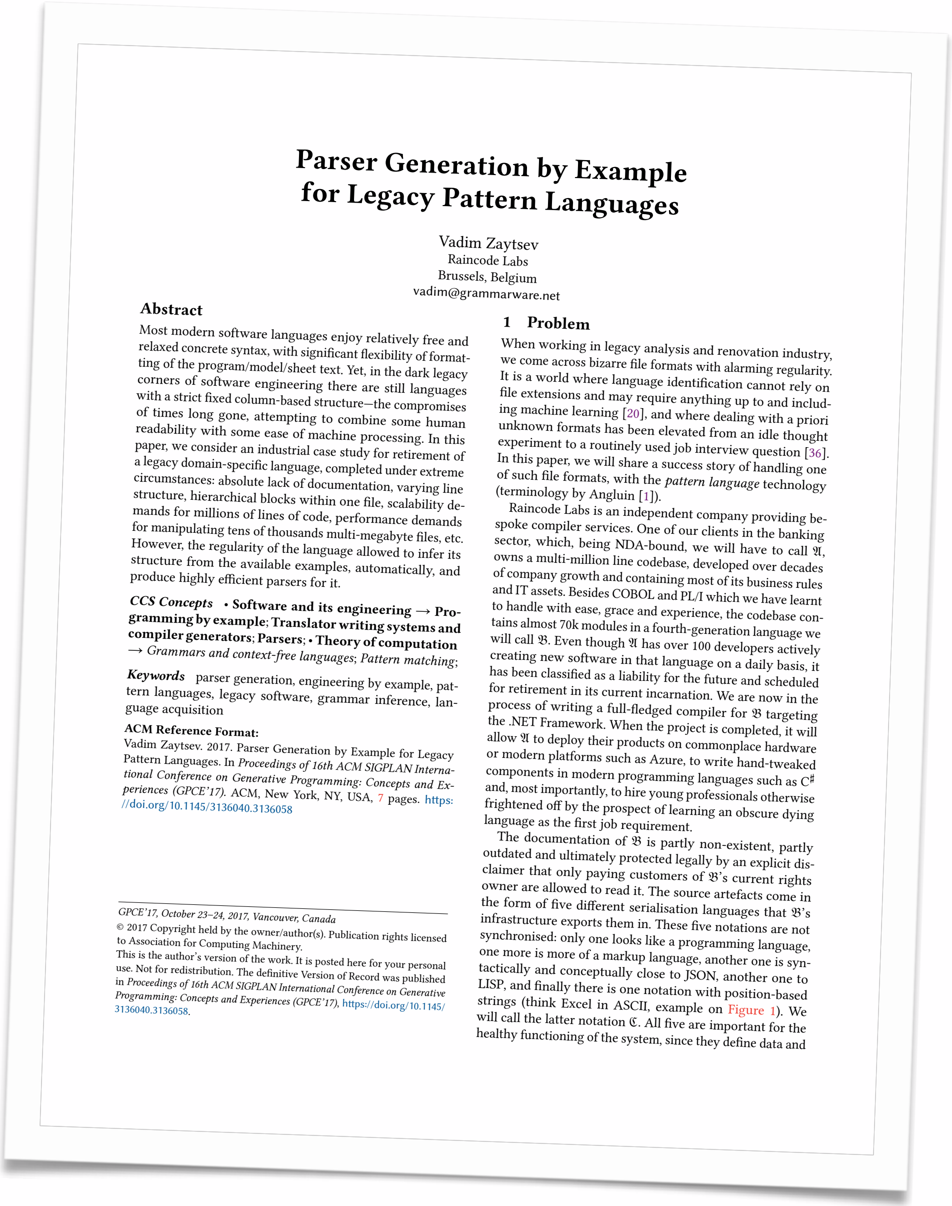
1

[REBLS'19]

UNIVERSITY OF TWENTE. | FMT Formal Methods & Tools

4.PAX





Parser Generation by Example for Legacy Pattern Languages

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadim@grammarware.net

Abstract

Most modern software languages enjoy relatively free and relaxed concrete syntax, with significant flexibility of formatting of the program/model/sheet text. Yet, in the dark legacy corners of software engineering there are still languages with a strict fixed column-based structure—the compromises of times long gone, attempting to combine some human readability with some ease of machine processing. In this paper, we consider an industrial case study for retirement of a legacy domain-specific language, completed under extreme circumstances: absolute lack of documentation, varying line structure, hierarchical blocks within one file, scalability demands for millions of lines of code, performance demands for manipulating tens of thousands multi-megabyte files, etc. However, the regularity of the language allowed to infer its structure from the available examples, automatically, and produce highly efficient parsers for it.

CCS Concepts • **Software and its engineering** → **Programming by example**; **Translator writing systems and compiler generators**; **Parsers**; • **Theory of computation** → *Grammars and context-free languages*; *Pattern matching*;

Keywords parser generation, engineering by example, pattern languages, legacy software, grammar inference, language acquisition

ACM Reference Format:
Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3136040.3136058

1 Problem

When working in legacy analysis and renovation industry, we come across bizarre file formats with alarming regularity. It is a world where language identification cannot rely on file extensions and may require anything up to and including machine learning [20], and where dealing with a priori unknown formats has been elevated from an idle thought experiment to a routinely used job interview question [36]. In this paper, we will share a success story of handling one of such file formats, with the *pattern language* technology (terminology by Angluin [1]).

Raincode Labs is an independent company providing bespoke compiler services. One of our clients in the banking sector, which, being NDA-bound, we will have to call 𝔄, owns a multi-million line codebase, developed over decades of company growth and containing most of its business rules and IT assets. Besides COBOL and PL/I which we have learnt to handle with ease, grace and experience, the codebase contains almost 70k modules in a fourth-generation language we will call 𝔅. Even though 𝔄 has over 100 developers actively creating new software in that language on a daily basis, it has been classified as a liability for the future and scheduled for retirement in its current incarnation. We are now in the process of writing a full-fledged compiler for 𝔅 targeting the .NET Framework. When the project is completed, it will allow 𝔄 to deploy their products on commonplace hardware or modern platforms such as Azure, to write hand-tweaked components in modern programming languages such as C♯ and, most importantly, to hire young professionals otherwise frightened off by the prospect of learning an obscure dying language as the first job requirement.

The documentation of 𝔅 is partly non-existent, partly outdated and ultimately protected legally by an explicit disclaimer that only paying customers of 𝔅's current rights owner are allowed to read it. The source artefacts come in the form of five different serialisation languages that 𝔅's infrastructure exports them in. These five notations are not synchronised: only one looks like a programming language, one more is more of a markup language, another one is syntactically and conceptually close to JSON, another one to LISP, and finally there is one notation with position-based strings (think Excel in ASCII, example on Figure 1). We will call the latter notation ℭ. All five are important for the healthy functioning of the system, since they define data and

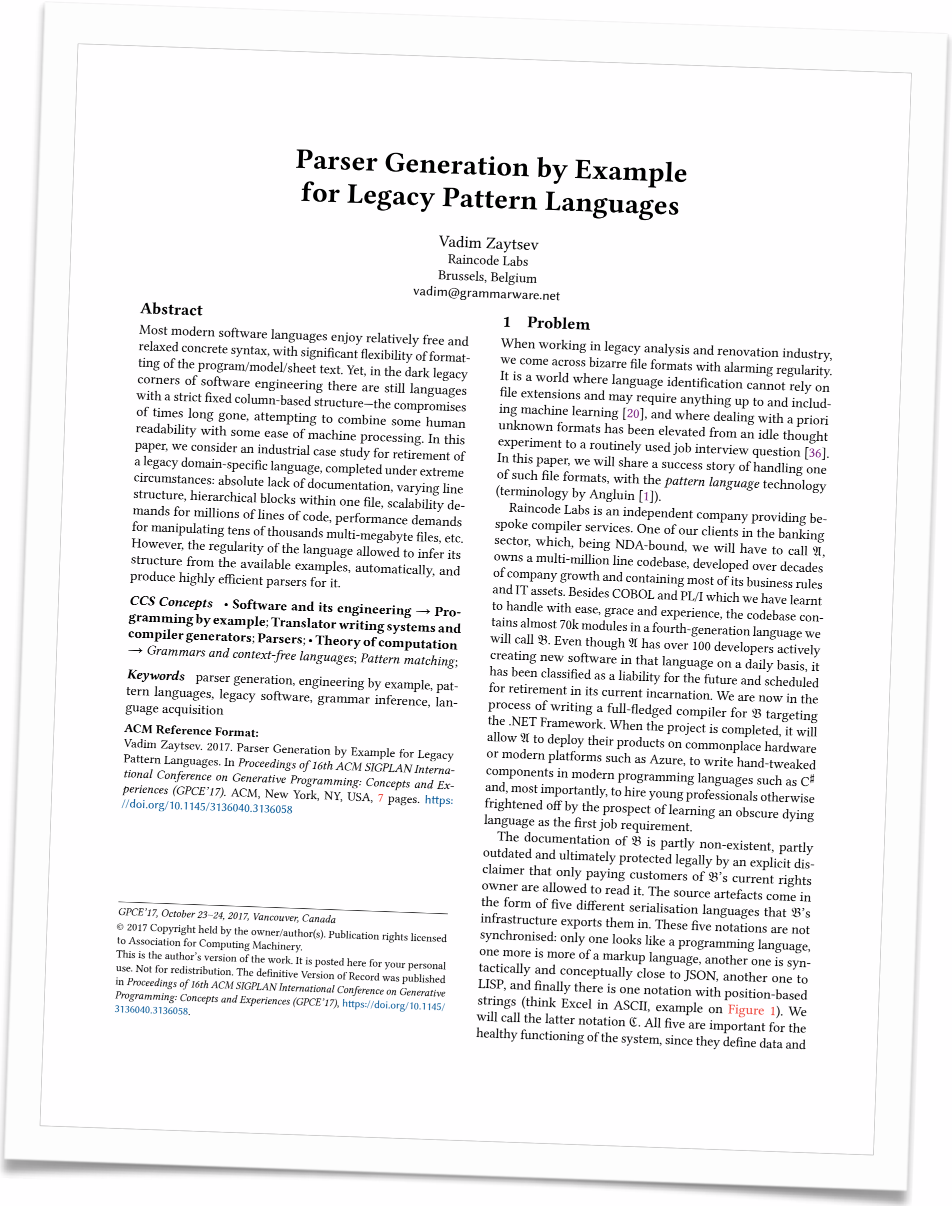
GPCE'17, October 23–24, 2017, Vancouver, Canada

[GPCE'17]

4.PAX

```
$$FILE 06/07/2017 23:59:59
$$FOO    ABCD      Y 06/07/2017 23:59:59 XYZ
 A 1 00010 00 0000 Y Y N Y NAMEA    NAMEB    S
 C 2 00015 02 0000 Y Y Y Y NAMEDDDD NAME EEE S
 F 5 00030 00 0020 Y N N Y NAMEG    NAMEH    S
$$BAR    EFGHKLMN Y 06/07/2017 23:59:59 N/A
 A LONGER_NAME_FOR_ENTITY                999 10.0
 A ANSWER_TO_THE_ULTIMATE_QUESTION        42  7.5
```

- Patterns
- Commitments
- Bindings



Parser Generation by Example for Legacy Pattern Languages

Vadim Zaytsev
Raincode Labs
Brussels, Belgium
vadim@grammarware.net

Abstract

Most modern software languages enjoy relatively free and relaxed concrete syntax, with significant flexibility of formatting of the program/model/sheet text. Yet, in the dark legacy corners of software engineering there are still languages with a strict fixed column-based structure—the compromises of times long gone, attempting to combine some human readability with some ease of machine processing. In this paper, we consider an industrial case study for retirement of a legacy domain-specific language, completed under extreme circumstances: absolute lack of documentation, varying line structure, hierarchical blocks within one file, scalability demands for millions of lines of code, performance demands for manipulating tens of thousands multi-megabyte files, etc. However, the regularity of the language allowed to infer its structure from the available examples, automatically, and produce highly efficient parsers for it.

CCS Concepts • **Software and its engineering → Programming by example; Translator writing systems and compiler generators; Parsers;** • **Theory of computation** → *Grammars and context-free languages; Pattern matching;*

Keywords parser generation, engineering by example, pattern languages, legacy software, grammar inference, language acquisition

ACM Reference Format:
Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3136040.3136058

1 Problem

When working in legacy analysis and renovation industry, we come across bizarre file formats with alarming regularity. It is a world where language identification cannot rely on file extensions and may require anything up to and including machine learning [20], and where dealing with a priori unknown formats has been elevated from an idle thought experiment to a routinely used job interview question [36]. In this paper, we will share a success story of handling one of such file formats, with the *pattern language* technology (terminology by Angluin [1]).

Raincode Labs is an independent company providing bespoke compiler services. One of our clients in the banking sector, which, being NDA-bound, we will have to call $\mathfrak{A}$, owns a multi-million line codebase, developed over decades of company growth and containing most of its business rules and IT assets. Besides COBOL and PL/I which we have learnt to handle with ease, grace and experience, the codebase contains almost 70k modules in a fourth-generation language we will call $\mathfrak{B}$. Even though $\mathfrak{A}$ has over 100 developers actively creating new software in that language on a daily basis, it has been classified as a liability for the future and scheduled for retirement in its current incarnation. We are now in the process of writing a full-fledged compiler for $\mathfrak{B}$ targeting the .NET Framework. When the project is completed, it will allow $\mathfrak{A}$ to deploy their products on commonplace hardware or modern platforms such as Azure, to write hand-tweaked components in modern programming languages such as C♯ and, most importantly, to hire young professionals otherwise frightened off by the prospect of learning an obscure dying language as the first job requirement.

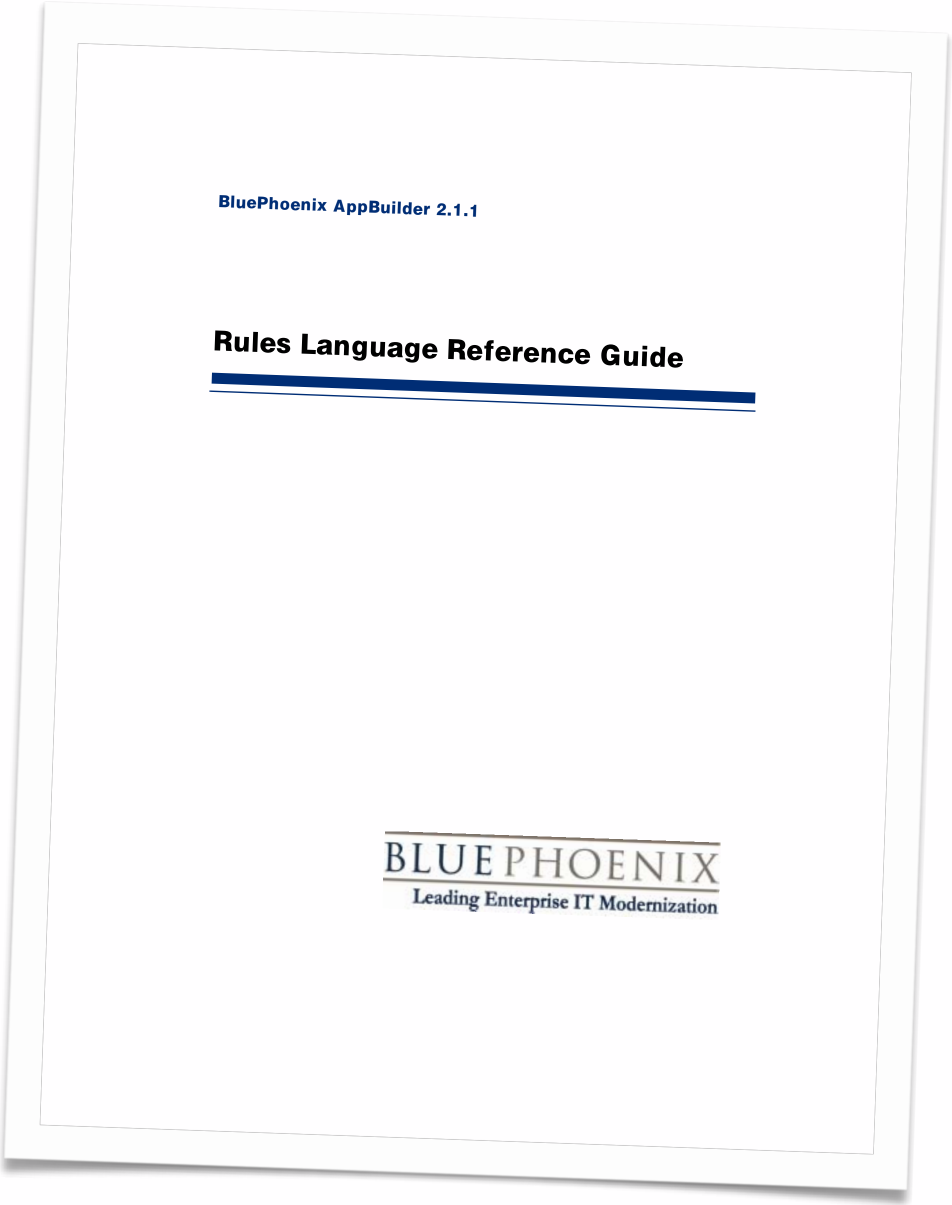
The documentation of $\mathfrak{B}$ is partly non-existent, partly outdated and ultimately protected legally by an explicit disclaimer that only paying customers of $\mathfrak{B}$'s current rights owner are allowed to read it. The source artefacts come in the form of five different serialisation languages that $\mathfrak{B}$'s infrastructure exports them in. These five notations are not synchronised: only one looks like a programming language, one more is more of a markup language, another one is syntactically and conceptually close to JSON, another one to LISP, and finally there is one notation with position-based strings (think Excel in ASCII, example on Figure 1). We will call the latter notation $\mathfrak{C}$. All five are important for the healthy functioning of the system, since they define data and

GPCE'17, October 23–24, 2017, Vancouver, Canada

[GPCE'17]

5.TIALAA

- AppBuilder is a 4GL
  - *“Application Development without Programmers”*
- Tech:
  - compiles to Java & COBOL
  - supported by handmade code
- Business case:
  - ~200 devs, reimplement in .NET


BluePhoenix AppBuilder 2.1.1

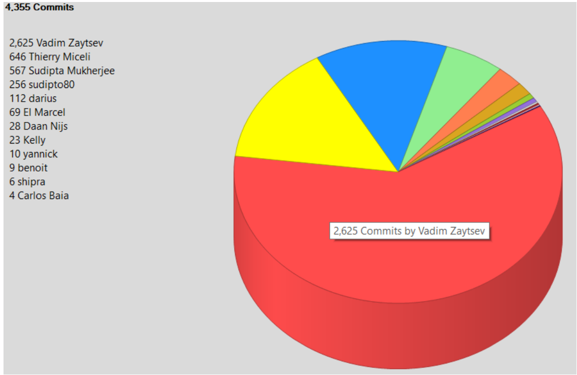
Rules Language Reference Guide

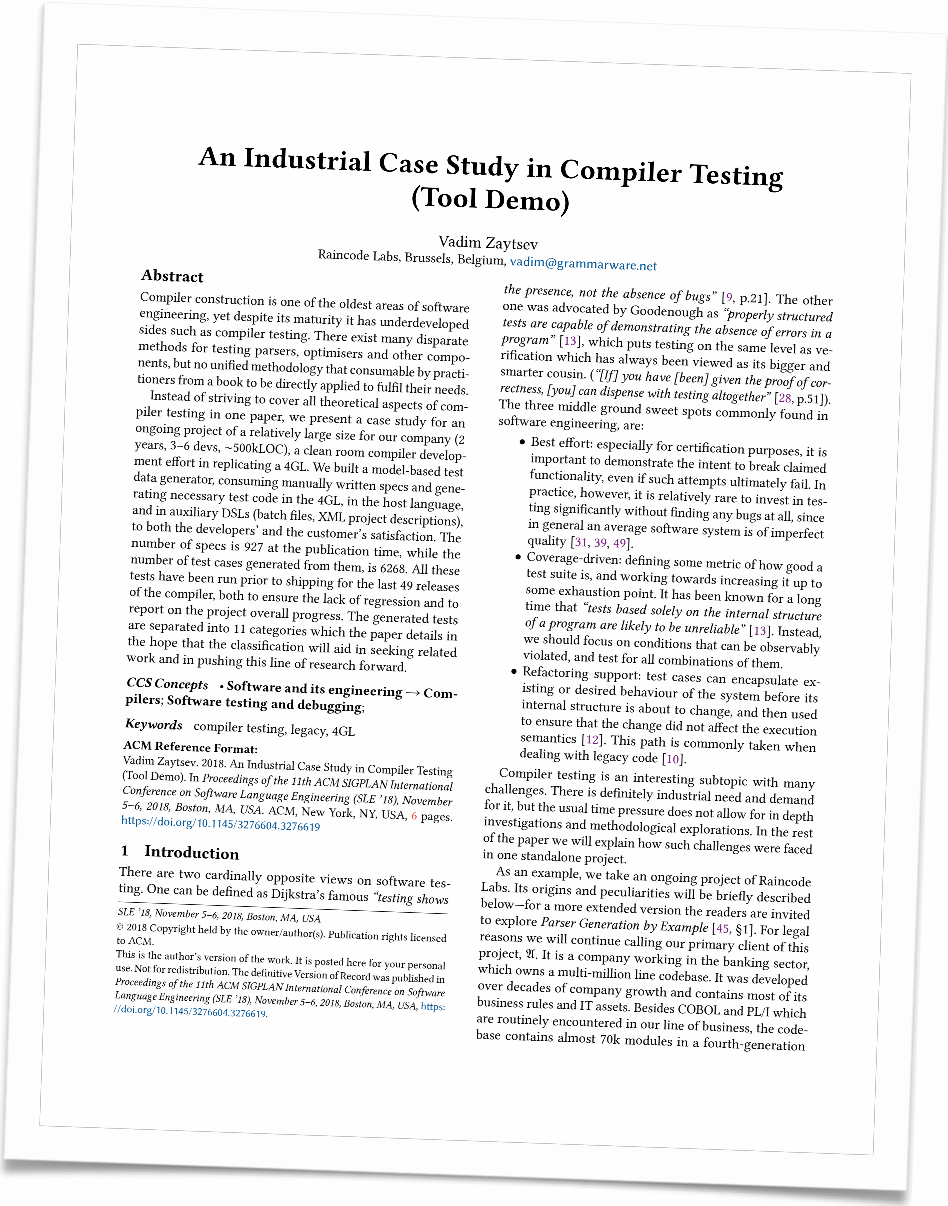
BLUEPHOENIX
Leading Enterprise IT Modernization

~~BluePhoenix AppBuilder documentaton~~

5.TIALAA

- Notations:
  - "rules": non-declarative
  - "sets": key-value lookup tables
  - "views": models in MVC
  - "panels": windows in S-exprs
- Guesswork
  - COBOL & Java
  - .NET/WPF





An Industrial Case Study in Compiler Testing (Tool Demo)

Vadim Zaytsev
Raincode Labs, Brussels, Belgium, vadim@grammarware.net

Abstract

Compiler construction is one of the oldest areas of software engineering, yet despite its maturity it has underdeveloped sides such as compiler testing. There exist many disparate methods for testing parsers, optimisers and other components, but no unified methodology that consumable by practitioners from a book to be directly applied to fulfil their needs.

Instead of striving to cover all theoretical aspects of compiler testing in one paper, we present a case study for an ongoing project of a relatively large size for our company (2 years, 3–6 devs, ~500kLOC), a clean room compiler development effort in replicating a 4GL. We built a model-based test data generator, consuming manually written specs and generating necessary test code in the 4GL, in the host language, and in auxiliary DSLs (batch files, XML project descriptions), to both the developers' and the customer's satisfaction. The number of specs is 927 at the publication time, while the number of test cases generated from them, is 6268. All these tests have been run prior to shipping for the last 49 releases of the compiler, both to ensure the lack of regression and to report on the project overall progress. The generated tests are separated into 11 categories which the paper details in the hope that the classification will aid in seeking related work and in pushing this line of research forward.

CCS Concepts • **Software and its engineering → Compilers**; **Software testing and debugging**;

Keywords compiler testing, legacy, 4GL

ACM Reference Format:
Vadim Zaytsev. 2018. An Industrial Case Study in Compiler Testing (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3276604.3276619

1 Introduction

There are two cardinally opposite views on software testing. One can be defined as Dijkstra's famous *"testing shows the presence, not the absence of bugs"* [9, p.21]. The other one was advocated by Goodenough as *"properly structured tests are capable of demonstrating the absence of errors in a program"* [13], which puts testing on the same level as verification which has always been viewed as its bigger and smarter cousin. (*"[If] you have [been] given the proof of correctness, [you] can dispense with testing altogether"* [28, p.51]). The three middle ground sweet spots commonly found in software engineering, are:

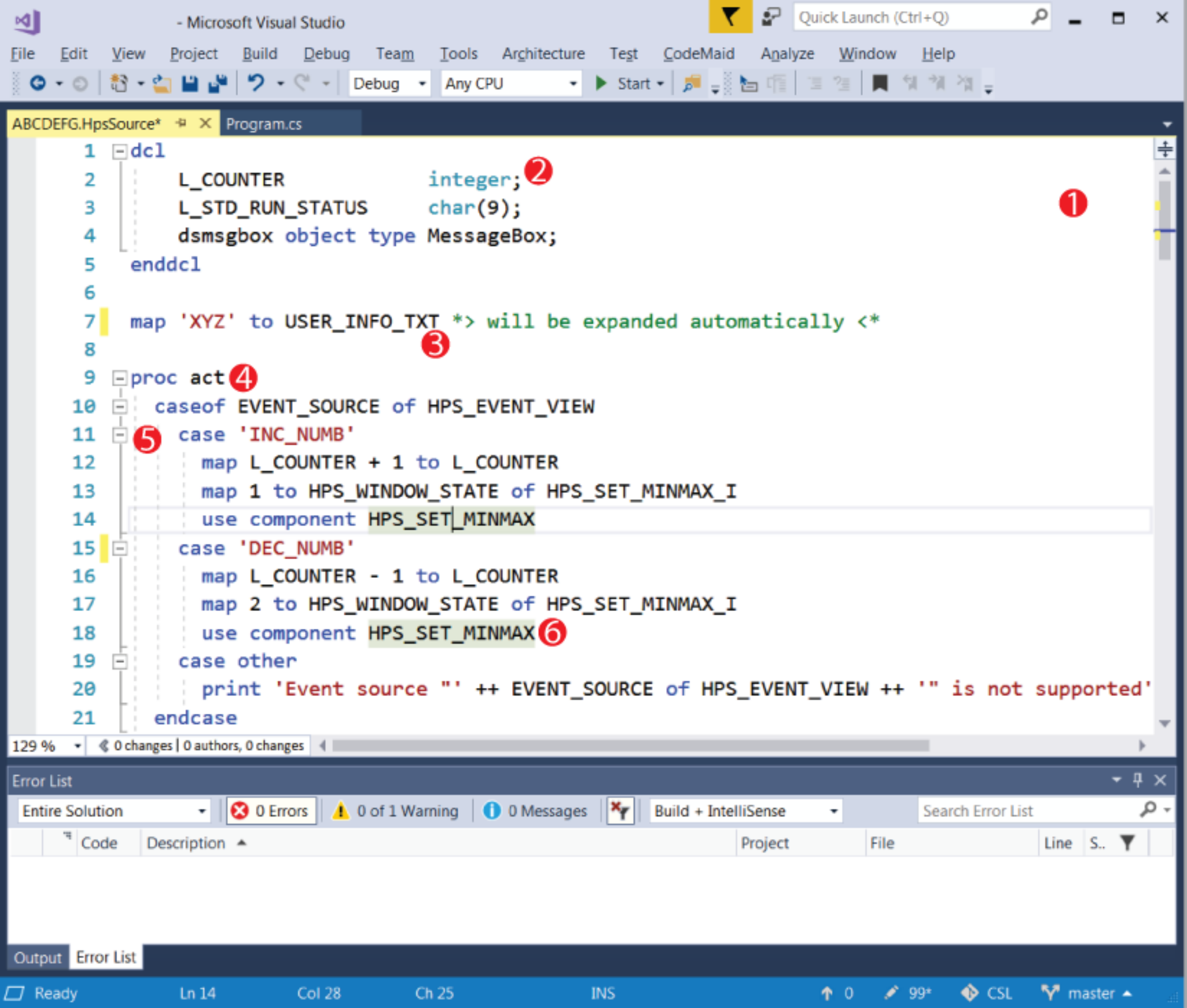
- Best effort: especially for certification purposes, it is important to demonstrate the intent to break claimed functionality, even if such attempts ultimately fail. In practice, however, it is relatively rare to invest in testing significantly without finding any bugs at all, since in general an average software system is of imperfect quality [31, 39, 49].
- Coverage-driven: defining some metric of how good a test suite is, and working towards increasing it up to some exhaustion point. It has been known for a long time that *"tests based solely on the internal structure of a program are likely to be unreliable"* [13]. Instead, we should focus on conditions that can be observably violated, and test for all combinations of them.
- Refactoring support: test cases can encapsulate existing or desired behaviour of the system before its internal structure is about to change, and then used to ensure that the change did not affect the execution semantics [12]. This path is commonly taken when dealing with legacy code [10].

Compiler testing is an interesting subtopic with many challenges. There is definitely industrial need and demand for it, but the usual time pressure does not allow for in depth investigations and methodological explorations. In the rest of the paper we will explain how such challenges were faced in one standalone project.

As an example, we take an ongoing project of Raincode Labs. Its origins and peculiarities will be briefly described below—for a more extended version the readers are invited to explore *Parser Generation by Example* [45, §1]. For legal reasons we will continue calling our primary client of this project, 𝔄. It is a company working in the banking sector, which owns a multi-million line codebase. It was developed over decades of company growth and contains most of its business rules and IT assets. Besides COBOL and PL/I which are routinely encountered in our line of business, the codebase contains almost 70k modules in a fourth-generation

SLE '18, November 5–6, 2018, Boston, MA, USA

[SLE'18] [PX/17.2] [TechDebt'19]

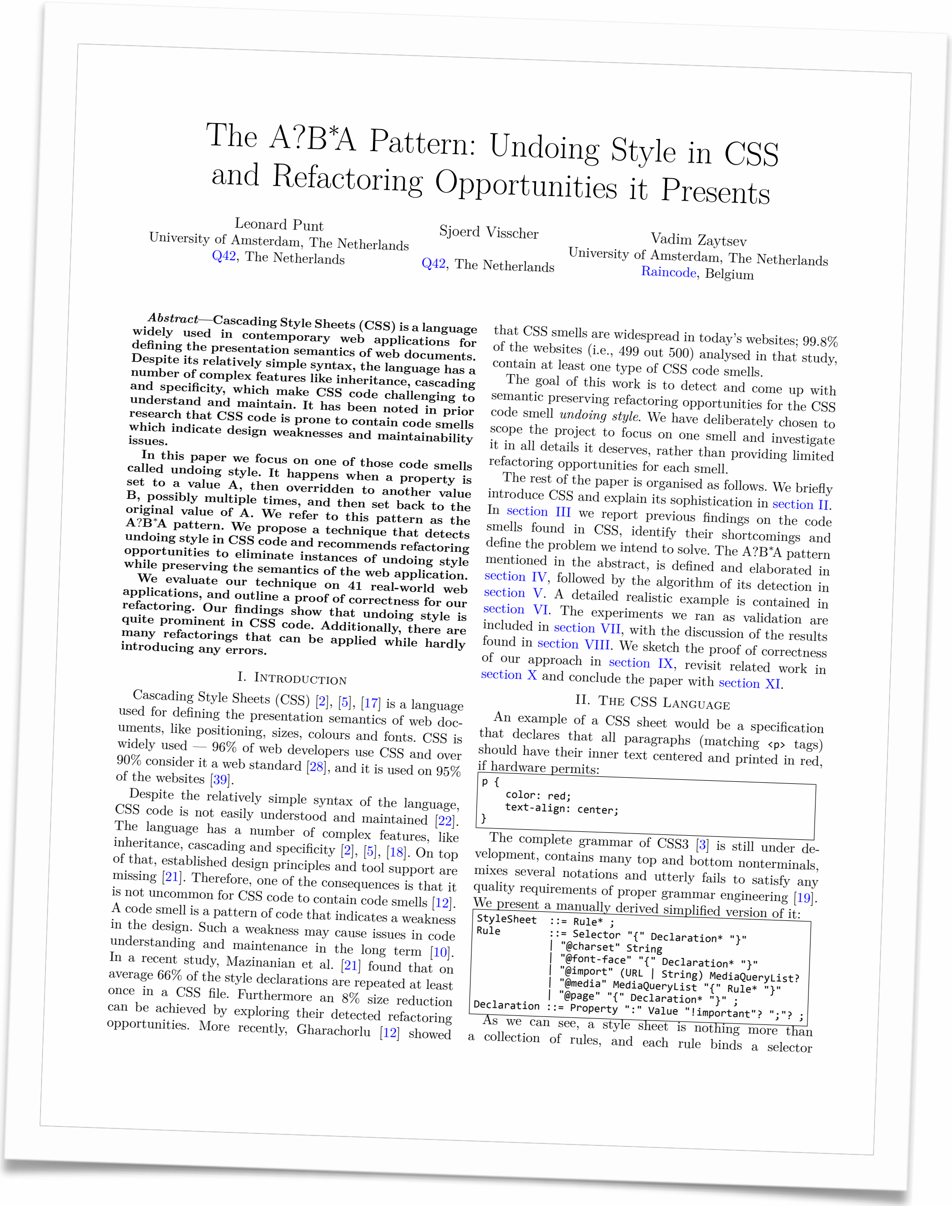


1. Native syntax
2. Data types
3. Symbol expanstion
4. Procedures
5. Code folding
6. System components

Formal Methods & Tools

6. CSS

- Escape the Java bubble!
- Project examples:
  - dead code detection [UvA’15]
  - performance [UvA’16] [SATToSE]
  - refactoring [UvA’15]
  - patterns [UvA’15] [ICSME’16]
  - conventions [UvA’15] [SLE’16]



The A?B*A Pattern: Undoing Style in CSS and Refactoring Opportunities it Presents

Leonard Punt
University of Amsterdam, The Netherlands
Q42, The Netherlands

Sjoerd Visscher
Q42, The Netherlands

Vadim Zaytsev
University of Amsterdam, The Netherlands
Raincode, Belgium

***Abstract*—Cascading Style Sheets (CSS) is a language widely used in contemporary web applications for defining the presentation semantics of web documents. Despite its relatively simple syntax, the language has a number of complex features like inheritance, cascading and specificity, which make CSS code challenging to understand and maintain. It has been noted in prior research that CSS code is prone to contain code smells which indicate design weaknesses and maintainability issues.**

In this paper we focus on one of those code smells called undoing style. It happens when a property is set to a value A, then overridden to another value B, possibly multiple times, and then set back to the original value of A. We refer to this pattern as the A?B*A pattern. We propose a technique that detects undoing style in CSS code and recommends refactoring opportunities to eliminate instances of undoing style while preserving the semantics of the web application.

We evaluate our technique on 41 real-world web applications, and outline a proof of correctness for our refactoring. Our findings show that undoing style is quite prominent in CSS code. Additionally, there are many refactorings that can be applied while hardly introducing any errors.

I. Introduction

Cascading Style Sheets (CSS) [2], [5], [17] is a language used for defining the presentation semantics of web documents, like positioning, sizes, colours and fonts. CSS is widely used — 96% of web developers use CSS and over 90% consider it a web standard [28], and it is used on 95% of the websites [39].

Despite the relatively simple syntax of the language, CSS code is not easily understood and maintained [22]. The language has a number of complex features, like inheritance, cascading and specificity [2], [5], [18]. On top of that, established design principles and tool support are missing [21]. Therefore, one of the consequences is that it is not uncommon for CSS code to contain code smells [12]. A code smell is a pattern of code that indicates a weakness in the design. Such a weakness may cause issues in code understanding and maintenance in the long term [10]. In a recent study, Mazinanian et al. [21] found that on average 66% of the style declarations are repeated at least once in a CSS file. Furthermore an 8% size reduction can be achieved by exploring their detected refactoring opportunities. More recently, Gharachorlu [12] showed that CSS smells are widespread in today’s websites; 99.8% of the websites (i.e., 499 out 500) analysed in that study, contain at least one type of CSS code smells.

The goal of this work is to detect and come up with semantic preserving refactoring opportunities for the CSS code smell *undoing style*. We have deliberately chosen to scope the project to focus on one smell and investigate it in all details it deserves, rather than providing limited refactoring opportunities for each smell.

The rest of the paper is organised as follows. We briefly introduce CSS and explain its sophistication in section II. In section III we report previous findings on the code smells found in CSS, identify their shortcomings and define the problem we intend to solve. The A?B*A pattern mentioned in the abstract, is defined and elaborated in section IV, followed by the algorithm of its detection in section V. A detailed realistic example is contained in section VI. The experiments we ran as validation are included in section VII, with the discussion of the results found in section VIII. We sketch the proof of correctness of our approach in section IX, revisit related work in section X and conclude the paper with section XI.

II. The CSS Language

An example of a CSS sheet would be a specification that declares that all paragraphs (matching `<p>` tags) should have their inner text centered and printed in red, if hardware permits:

```
p {
    color: red;
    text-align: center;
}
```

The complete grammar of CSS3 [3] is still under development, contains many top and bottom nonterminals, mixes several notations and utterly fails to satisfy any quality requirements of proper grammar engineering [19]. We present a manually derived simplified version of it:

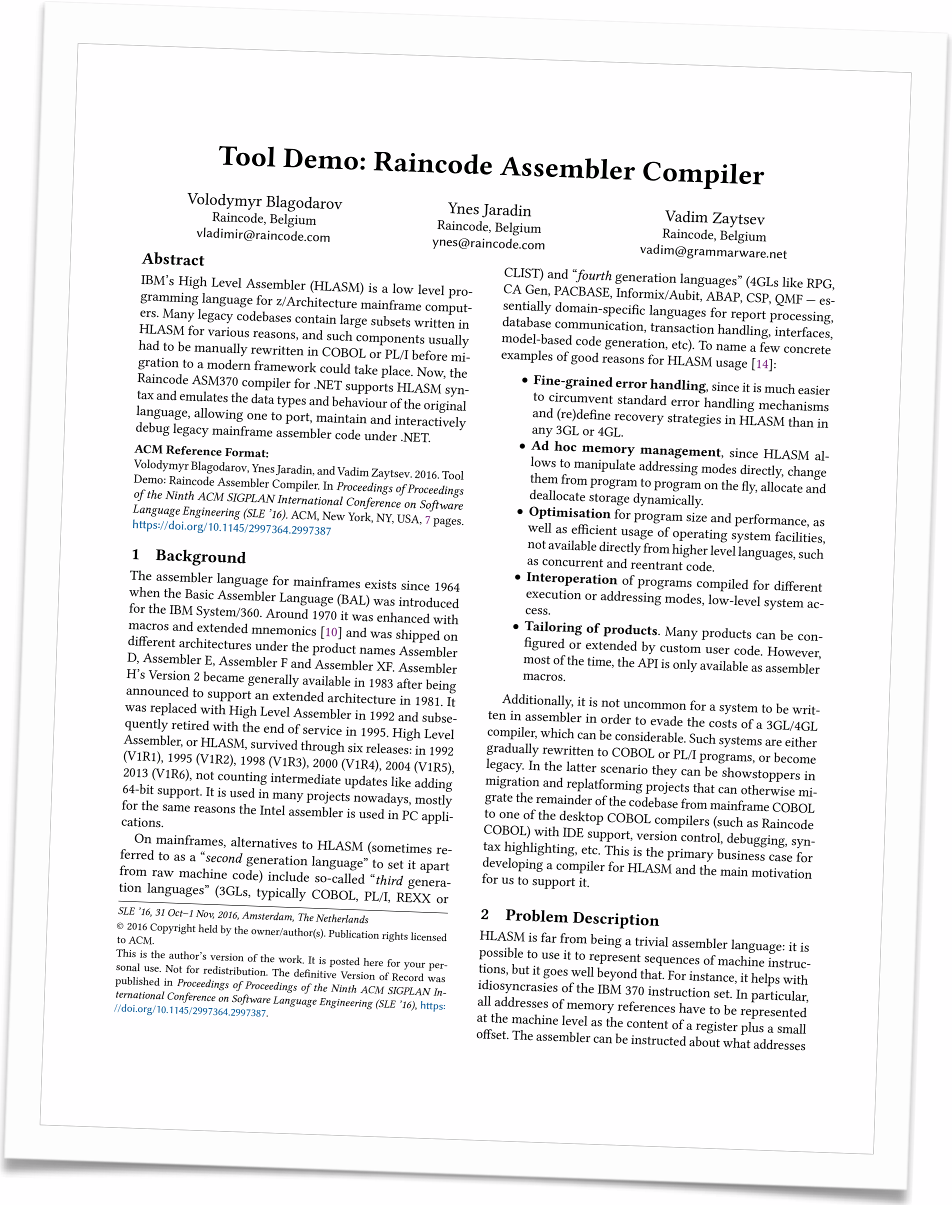
```
StyleSheet  ::= Rule* ;
Rule        ::= Selector "{" Declaration* "}"
              | "@charset" String
              | "@font-face" "{" Declaration* "}"
              | "@import" (URL | String) MediaQueryList?
              | "@media" MediaQueryList "{" Rule* "}"
              | "@page" "{" Declaration* "}" ;
Declaration ::= Property ":" Value "!important"? ";"? ;
```

As we can see, a style sheet is nothing more than a collection of rules, and each rule binds a selector

[ICSME’16] [SLE’16]

7.HLASM

- IBM HLASM is a 2GL
- Non-orthogonal semantics
- Self-modification is glorified
- Errors in documentation
- *Principles of Operation*: 1902 pp
- 953 instructions in the set
- Modelling! Generation! Supercompilaton!



Tool Demo: Raincode Assembler Compiler

Volodymyr Blagodarov
Raincode, Belgium
vladimir@raincode.com

Ynes Jaradin
Raincode, Belgium
ynes@raincode.com

Vadim Zaytsev
Raincode, Belgium
vadim@grammarware.net

Abstract

IBM's High Level Assembler (HLASM) is a low level programming language for z/Architecture mainframe computers. Many legacy codebases contain large subsets written in HLASM for various reasons, and such components usually had to be manually rewritten in COBOL or PL/I before migration to a modern framework could take place. Now, the Raincode ASM370 compiler for .NET supports HLASM syntax and emulates the data types and behaviour of the original language, allowing one to port, maintain and interactively debug legacy mainframe assembler code under .NET.

ACM Reference Format:
Volodymyr Blagodarov, Ynes Jaradin, and Vadim Zaytsev. 2016. Tool Demo: Raincode Assembler Compiler. In *Proceedings of Proceedings of the Ninth ACM SIGPLAN International Conference on Software Language Engineering (SLE '16)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/2997364.2997387

1 Background

The assembler language for mainframes exists since 1964 when the Basic Assembler Language (BAL) was introduced for the IBM System/360. Around 1970 it was enhanced with macros and extended mnemonics [10] and was shipped on different architectures under the product names Assembler D, Assembler E, Assembler F and Assembler XF. Assembler H's Version 2 became generally available in 1983 after being announced to support an extended architecture in 1981. It was replaced with High Level Assembler in 1992 and subsequently retired with the end of service in 1995. High Level Assembler, or HLASM, survived through six releases: in 1992 (V1R1), 1995 (V1R2), 1998 (V1R3), 2000 (V1R4), 2004 (V1R5), 2013 (V1R6), not counting intermediate updates like adding 64-bit support. It is used in many projects nowadays, mostly for the same reasons the Intel assembler is used in PC applications.

On mainframes, alternatives to HLASM (sometimes referred to as a "*second* generation language" to set it apart from raw machine code) include so-called "*third* generation languages" (3GLs, typically COBOL, PL/I, REXX or CLIST) and "*fourth* generation languages" (4GLs like RPG, CA Gen, PACBASE, Informix/Aubit, ABAP, CSP, QMF — essentially domain-specific languages for report processing, database communication, transaction handling, interfaces, model-based code generation, etc). To name a few concrete examples of good reasons for HLASM usage [14]:

- **Fine-grained error handling**, since it is much easier to circumvent standard error handling mechanisms and (re)define recovery strategies in HLASM than in any 3GL or 4GL.
- **Ad hoc memory management**, since HLASM allows to manipulate addressing modes directly, change them from program to program on the fly, allocate and deallocate storage dynamically.
- **Optimisation** for program size and performance, as well as efficient usage of operating system facilities, not available directly from higher level languages, such as concurrent and reentrant code.
- **Interoperation** of programs compiled for different execution or addressing modes, low-level system access.
- **Tailoring of products**. Many products can be configured or extended by custom user code. However, most of the time, the API is only available as assembler macros.

Additionally, it is not uncommon for a system to be written in assembler in order to evade the costs of a 3GL/4GL compiler, which can be considerable. Such systems are either gradually rewritten to COBOL or PL/I programs, or become legacy. In the latter scenario they can be showstoppers in migration and replatforming projects that can otherwise migrate the remainder of the codebase from mainframe COBOL to one of the desktop COBOL compilers (such as Raincode COBOL) with IDE support, version control, debugging, syntax highlighting, etc. This is the primary business case for developing a compiler for HLASM and the main motivation for us to support it.

2 Problem Description

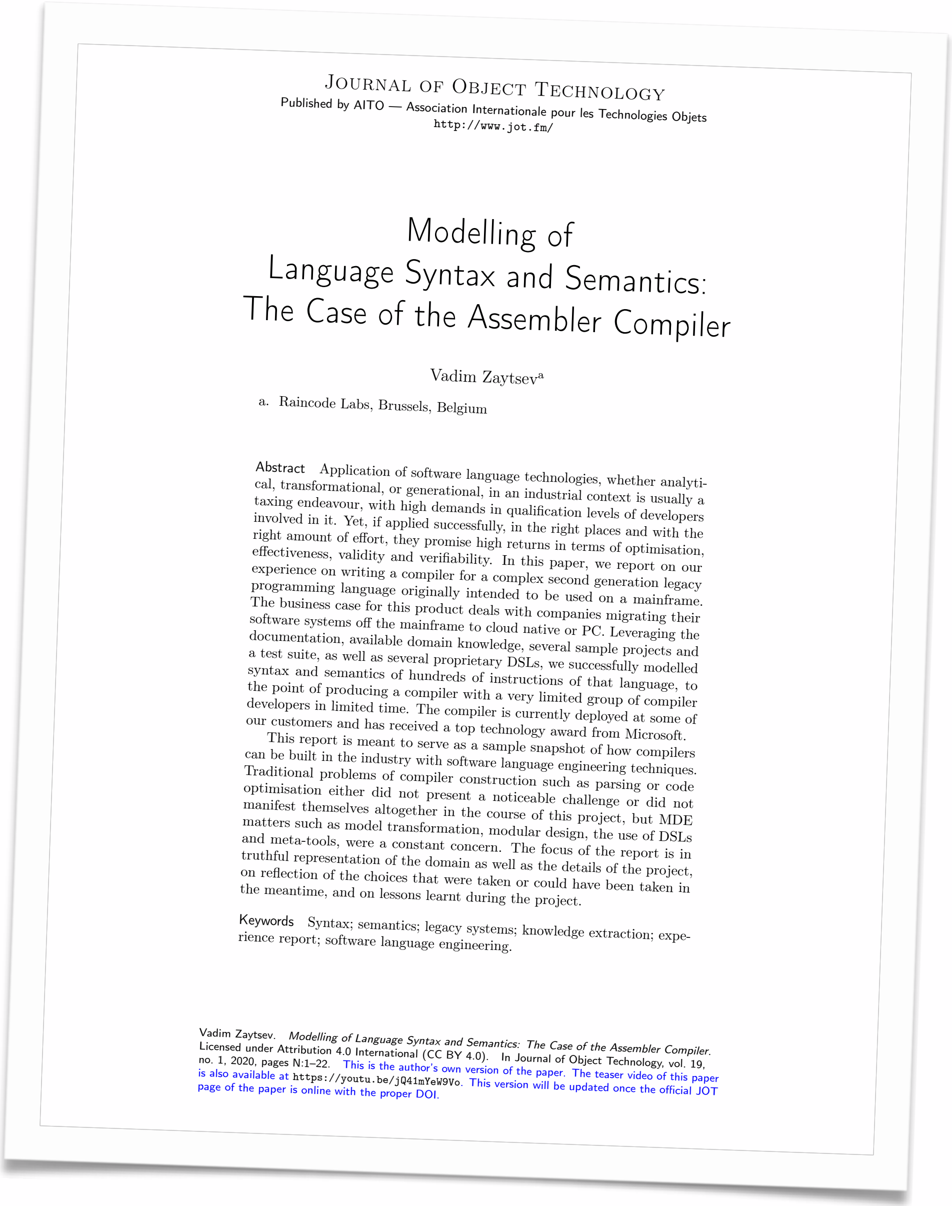
HLASM is far from being a trivial assembler language: it is possible to use it to represent sequences of machine instructions, but it goes well beyond that. For instance, it helps with idiosyncrasies of the IBM 370 instruction set. In particular, all addresses of memory references have to be represented at the machine level as the content of a register plus a small offset. The assembler can be instructed about what addresses

SLE '16, 31 Oct–1 Nov, 2016, Amsterdam, The Netherlands

[SLE'16] [MoreVMs'17] [BENEVOL'20] [ECMFA'20]

7.HLASM

- IBM HLASM is a 2GL
- Non-orthogonal semantics
- Self-modification is glorified
- Errors in documentation
- *Principles of Operation*: 1902 pp
- 953 instructions in the set
- Modelling! Generation! Supercompilaton!



JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler

Vadim Zaytsev[a]
a. Raincode Labs, Brussels, Belgium

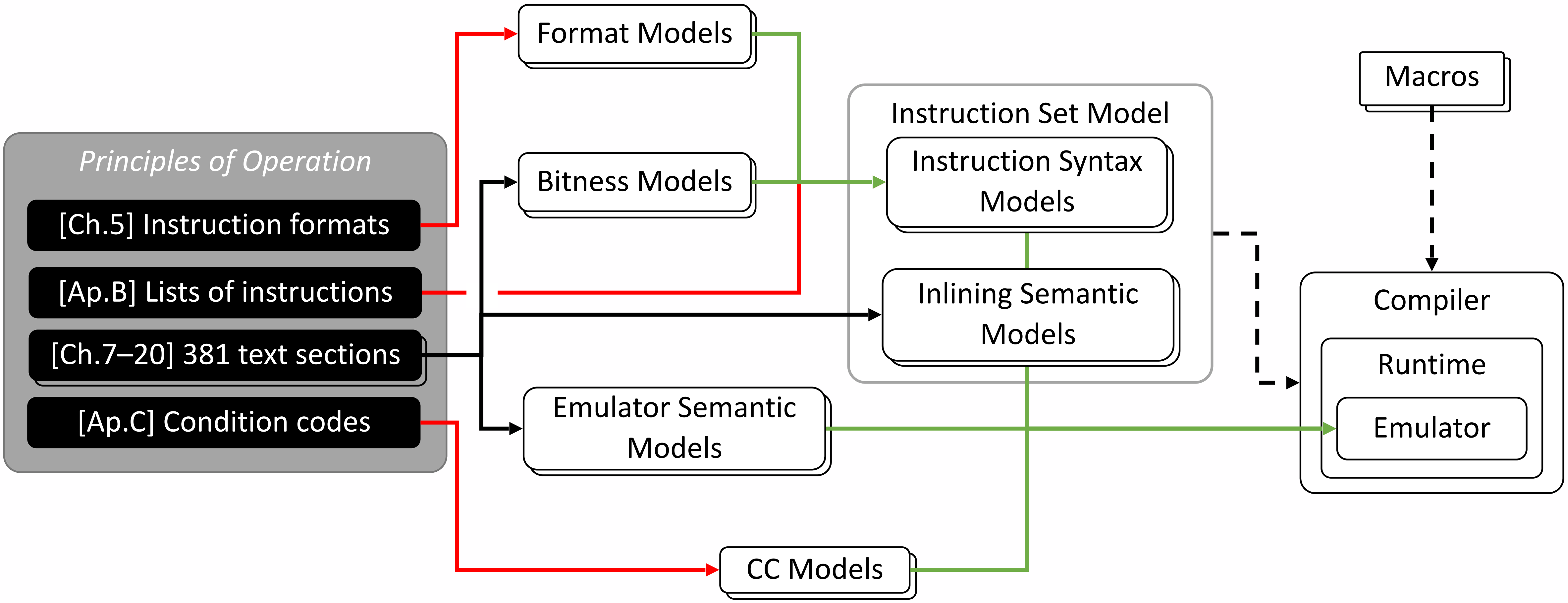
Abstract Application of software language technologies, whether analytical, transformational, or generational, in an industrial context is usually a taxing endeavour, with high demands in qualification levels of developers involved in it. Yet, if applied successfully, in the right places and with the right amount of effort, they promise high returns in terms of optimisation, effectiveness, validity and verifiability. In this paper, we report on our experience on writing a compiler for a complex second generation legacy programming language originally intended to be used on a mainframe. The business case for this product deals with companies migrating their software systems off the mainframe to cloud native or PC. Leveraging the documentation, available domain knowledge, several sample projects and a test suite, as well as several proprietary DSLs, we successfully modelled syntax and semantics of hundreds of instructions of that language, to the point of producing a compiler with a very limited group of compiler developers in limited time. The compiler is currently deployed at some of our customers and has received a top technology award from Microsoft.
This report is meant to serve as a sample snapshot of how compilers can be built in the industry with software language engineering techniques. Traditional problems of compiler construction such as parsing or code optimisation either did not present a noticeable challenge or did not manifest themselves altogether in the course of this project, but MDE matters such as model transformation, modular design, the use of DSLs and meta-tools, were a constant concern. The focus of the report is in truthful representation of the domain as well as the details of the project, on reflection of the choices that were taken or could have been taken in the meantime, and on lessons learnt during the project.

Keywords Syntax; semantics; legacy systems; knowledge extraction; experience report; software language engineering.

Vadim Zaytsev. *Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler.* Licensed under Attribution 4.0 International (CC BY 4.0). In Journal of Object Technology, vol. 19, no. 1, 2020, pages N:1–22. This is the author's own version of the paper. The teaser video of this paper is also available at https://youtu.be/jQ41mYeW9Vo. This version will be updated once the official JOT page of the paper is online with the proper DOI.

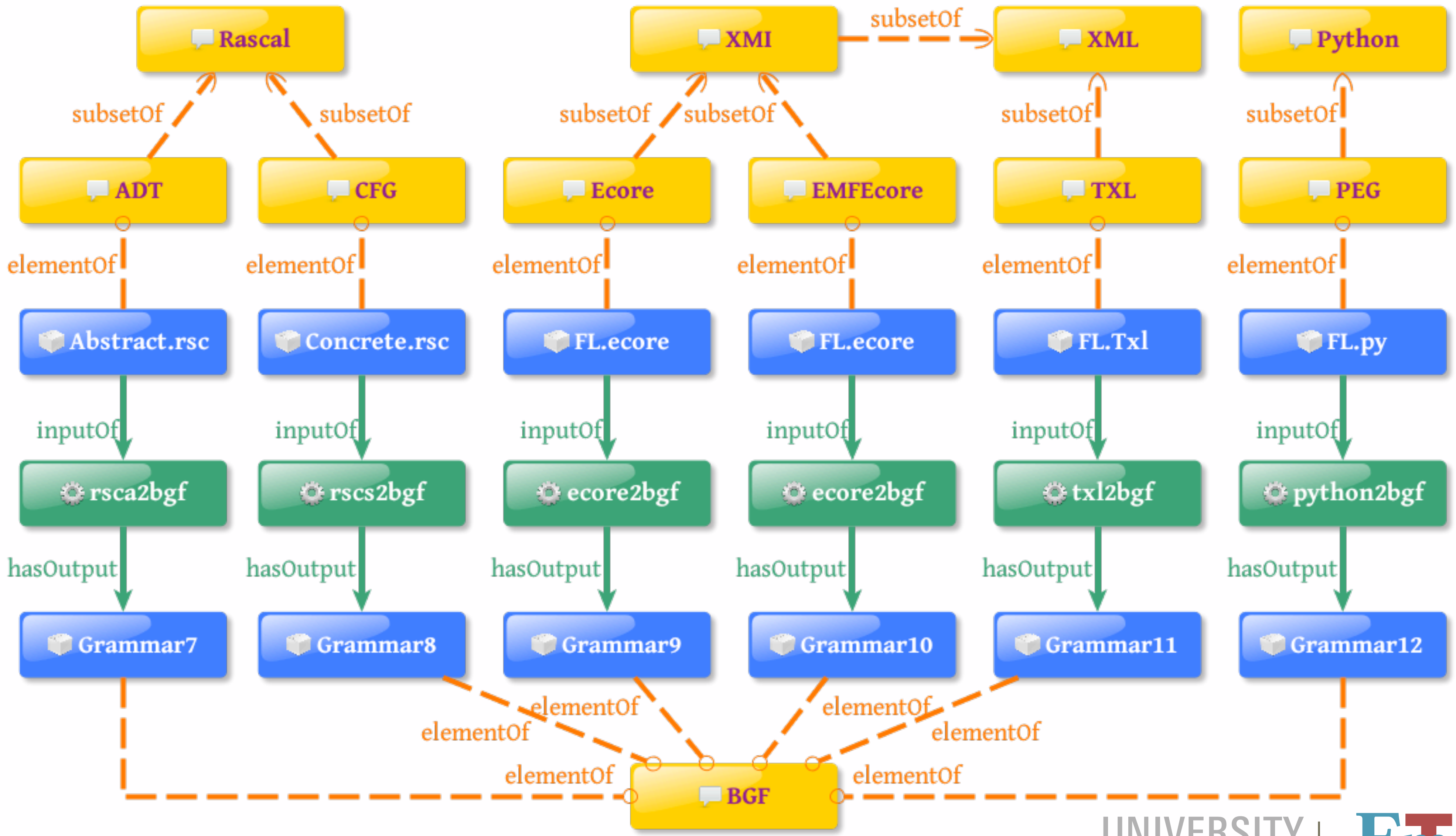
[SLE'16] [MoreVMs'17] [BENEVOL'20] [ECMFA'20]

7.HLASM



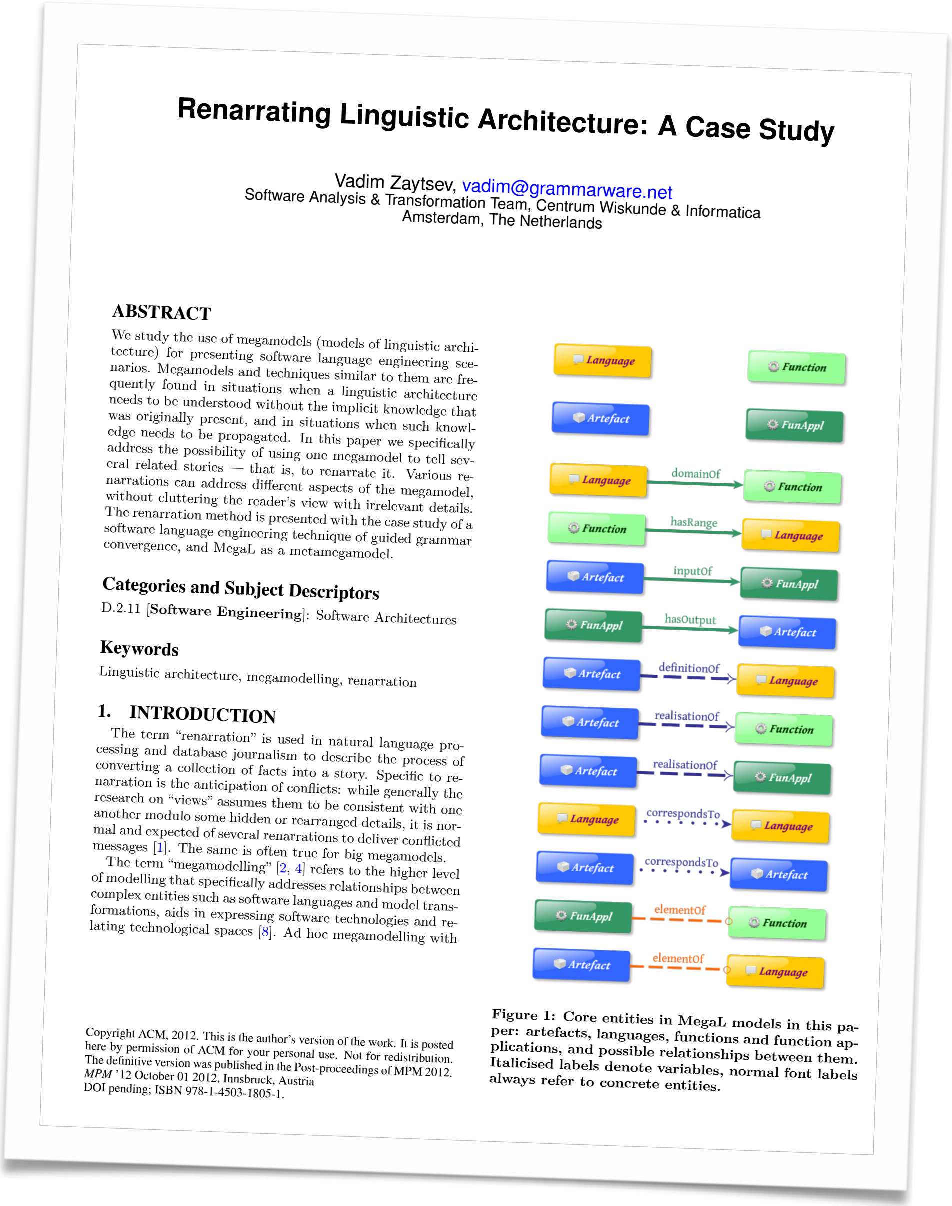
[ECMFA'20]

8.MegaL


Rascal
XMI
subsetOf
XML
Python
subsetOf
subsetOf
subsetOf
subsetOf
subsetOf
subsetOf
ADT
CFG
Ecore
EMFEcore
TXL
PEG
elementOf
elementOf
elementOf
elementOf
elementOf
elementOf
Abstract.rsc
Concrete.rsc
FL.ecore
FL.ecore
FL.Txl
FL.py
inputOf
inputOf
inputOf
inputOf
inputOf
inputOf
rsca2bgf
rscs2bgf
ecore2bgf
ecore2bgf
txl2bgf
python2bgf
hasOutput
hasOutput
hasOutput
hasOutput
hasOutput
hasOutput
Grammar7
Grammar8
Grammar9
Grammar10
Grammar11
Grammar12
elementOf
elementOf
elementOf
elementOf
elementOf
elementOf
BGF

8.MegaL

- Renarration
    - process of converting facts into a story
- Used by Indian storytellers
    - also in database journalism
- Can/must be used
    - to make models less scary



Renarrating Linguistic Architecture: A Case Study

Vadim Zaytsev, vadim@grammarware.net
Software Analysis & Transformation Team, Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

ABSTRACT

We study the use of megamodels (models of linguistic architecture) for presenting software language engineering scenarios. Megamodels and techniques similar to them are frequently found in situations when a linguistic architecture needs to be understood without the implicit knowledge that was originally present, and in situations when such knowledge needs to be propagated. In this paper we specifically address the possibility of using one megamodel to tell several related stories — that is, to renarrate it. Various renarrations can address different aspects of the megamodel, without cluttering the reader's view with irrelevant details. The renarration method is presented with the case study of a software language engineering technique of guided grammar convergence, and MegaL as a metamegamodel.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

Keywords

Linguistic architecture, megamodelling, renarration

1. INTRODUCTION

The term "renarration" is used in natural language processing and database journalism to describe the process of converting a collection of facts into a story. Specific to renarration is the anticipation of conflicts: while generally the research on "views" assumes them to be consistent with one another modulo some hidden or rearranged details, it is normal and expected of several renarrations to deliver conflicted messages [1]. The same is often true for big megamodels.

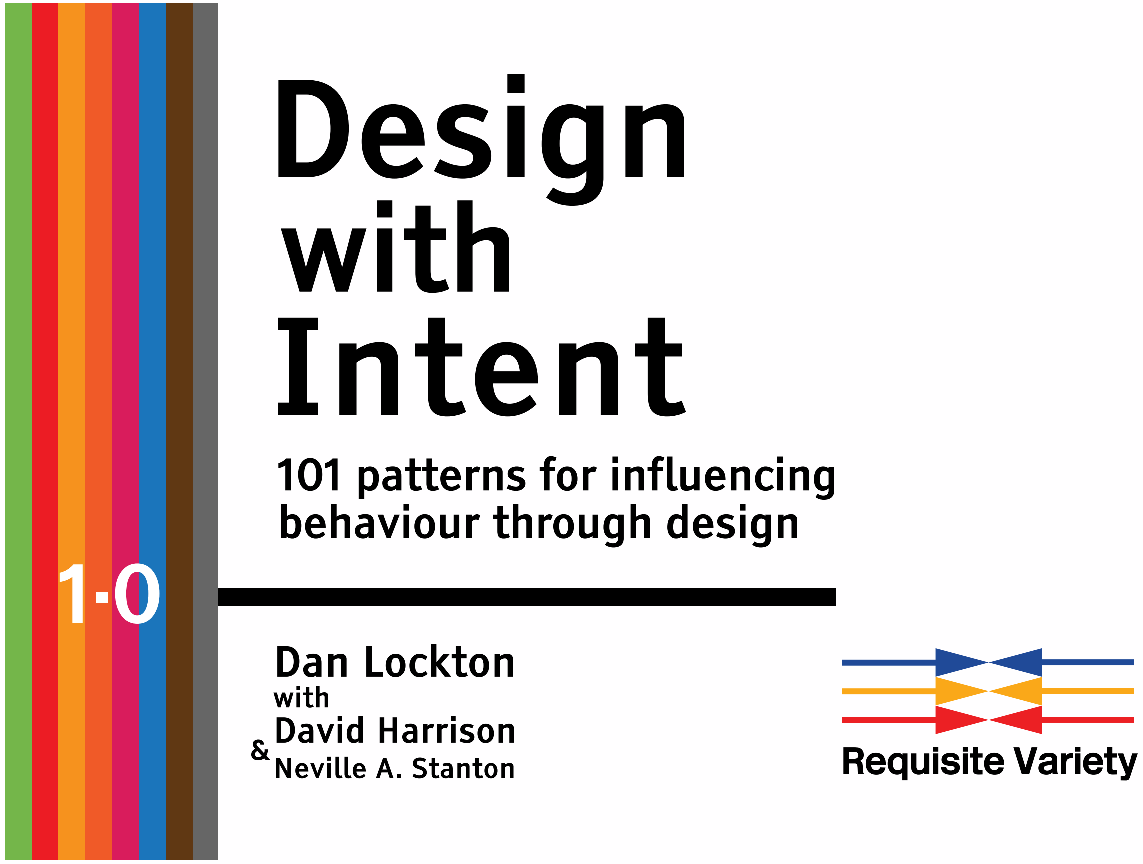
The term "megamodelling" [2, 4] refers to the higher level of modelling that specifically addresses relationships between complex entities such as software languages and model transformations, aids in expressing software technologies and relating technological spaces [8]. Ad hoc megamodelling with

Figure 1: Core entities in MegaL models in this paper: artefacts, languages, functions and function applications, and possible relationships between them. Italicised labels denote variables, normal font labels always refer to concrete entities.

MPM '12 October 01 2012, Innsbruck, Austria
DOI pending; ISBN 978-1-4503-1805-1.

[MPM'12] [XM'13] [GEMOC'14]

9.DYOL



Language Design with Intent

Vadim Zaytsev (http://grammarware.net), Raincode Labs, Brussels, Belgium

Abstract—Software languages have always been an essential component of model-driven engineering. Their importance and popularity has been on the rise thanks to language workbenches, language-oriented development and other methodologies that enable us to quickly and easily create new languages specific for each domain. Unfortunately, language design is largely a form of art and has resisted most attempts to turn it into a form of science or engineering. In this paper we borrow concepts, techniques and principles from the domain of persuasive technology, or wider yet, design with intent — which was developed as a way to influence users behaviour for social and environmental benefit. Similarly, we claim, software language designers can make conscious choices in order to influence the behaviour of language users. The paper describes a process of extracting design components from 24 books of eight categories (dragon books, parsing techniques, compiler construction, compiler design, language implementation, language documentation, programming languages, software languages), as well as from the original set of *Design with Intent* cards and papers on DSL design. The resulting language design card toolkit can be used by DSL designers to cover important design decisions and make them with more confidence.

I. Motivation

First software languages were used in late 1940s[1] as an intermediate step in algorithm design. They allowed programmers of digital computers to bridge the gap between mathematical computations and machine codes. (The codes as such are much older, they were used on punched cards and rolls since 1725 in weaving looms[2] and at least since 1842 in pianolas[3].) A decade later[4] people started developing automated compilers, delegating the task of translating texts written in these languages, to the machine code, to system software components. Another decade passed, and new languages started being developed with specific design aims, targeting a particular problem domain[5] or a particular target audience[6]. By 1969 there were at least 120 widespread software languages [15], [35]. The next two or three decades, the language landscape was becoming more and more populated and — some claim — cluttered with numerous languages designed and implemented for all kinds of goals and purposes. Eventually we all have arrived at the point where creating a new language suitable for the problem at hand, ceased being challenging for engineers. Having, reusing or designing a DSL has been elevated to just a regular MDE problem solving recipe. Now we are focused on making software language creation methods reliable and repeatable [36].

[1]Since von Neumann and the Goldstines' *Flow Diagrams*.
[2]Since Basile Bouchon's silk centre in Lyon.
[3]Since Claude Félix Seytre's French patent no. 8691.
[4]Since Hopper's *MATH-MATIC*.
[5]Since Iverson's *APL*.
[6]Since Papert's *LOGO*, strengthened later by Perlman's *TORTIS*.

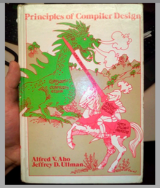
In this paper we assume the standpoint of *software language engineering* and, whenever possible, make no explicit distinction between modelling languages and programming ones, between domain-specific and general-purpose ones, among generations, paradigms, etc. Thus, whenever possible, we say "user" or "language user" instead of "programmer" or "modeller", and use other kinds of neutral terminology. We use the word "model" instead of "language instance" to mean a model, a program, a query, a stylesheet, a spreadsheet, etc. Other principles behind this project are explained in section II.

Languages are designed for following purposes, a.o.:

- to raise the abstraction level (almost universal);
- to improve user experience for languages with known problems but infeasible evolution (C++ for C, Dart for JavaScript, Go for C++, Swift for ObjectiveC, Scala for Java, Hack for PHP, .NET Core for .NET Framework);
- to give domain experts control over executable systems (the goal behind most domain-specific languages);
- to let non-coders structurally communicate with computers (emojis and smileys in most social networks, web forum markup like bbcode, wiki markup, etc);
- to open the usage of tools and services for third party usage (APIs);
- to abstract from irrelevant boilerplate (combinator libraries, languages with built-in constructs for concurrency, error handling, design patterns, etc);
- to explore different ways of human-computer interaction (numerous spreadsheet applications, most languages developed in workbenches like MPS or MetaEdit+);
- to make expressive and robust interchange and storage formats (even JSON and XML work with schemata);
- to build efficient tools by choosing suitable data structures (intermediate representations);
- to redesign legacy languages (VB.NET aligned with C#, XHTML as HTML in XML);
- to evolve existing languages into new versions (coevolution of Java and C# since the initial release of the latter);
- to create attractive language dialects (several industrially applicable extensions of originally educational Pascal, many vendor-specific COBOL compilers incompatible among themselves to prevent users from migrating);
- to experiment with new paradigms and get to know limits of their expressiveness (bidirectional transformation, reversible computation and others).

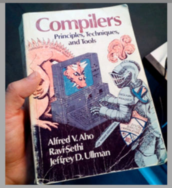
However, "language design is largely an art, not a science" [11, p.67]. There is no clear separation of where the language design starts and where it ends. In practice the work of a software language designer often gets mixed with the

[MoDELS'17] http://slebok.github.io/dyol



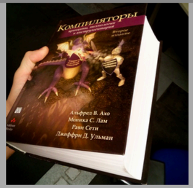
DB-GD

Principles of Compiler Design (Aho, Ullman, 1977)



DB-RD

Compilers: Principles, Techniques, and Tools (Aho, Sethi, Ullman, 1986)

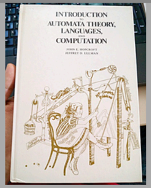


DB-PD

Compilers: Principles, Techniques, & Tools (Aho, Lam, Sethi, Ullman, 2006)

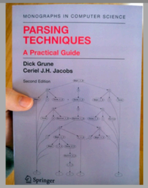
PT-AO

Definition of Programming Languages by Interpreting Automata (Ollongren, 1974)



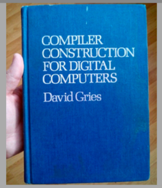
PT-HU

Introduction to Automata Theory, Languages, and Computation (Hopcroft, Ullman, 1979)



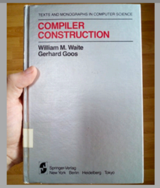
PT-GJ

Parsing Techniques: A Practical Guide (Grune, Jacobs, 2008)



CC-DG

Compiler Construction for Digital Computers (Gries, 1971)

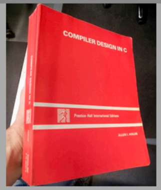


CC-WG

Compiler Construction (Waite, Goos, 1984)

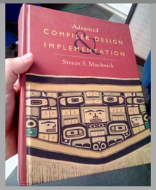
CC-NW

Compiler Construction (Wirth, 2005)



CD-AH

Compiler Design in C (Holub, 1990)

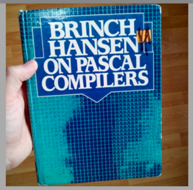


CD-SM

Advanced Compiler Design and Implementation (Muchnick, 1997)

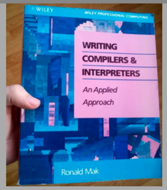
CD-GR

Modern Compiler Design (Grune, van Reeuwijk, Bal, Jacobs, Langendoen, 2012)



LI-BH

Brinch Hansen on Pascal Compilers (Hansen, 1985)



LI-RM

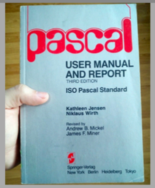
Writing Compilers and Interpreters: An Applied Approach (Mak, 1991)

LI-PZ

Programming Languages: Design and Implementation (Pratt, Zelkowitz, 2001)

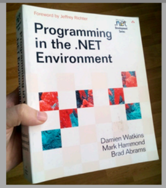
LD-ED

A Primer of ALGOL 60 Programming (Dijkstra, 1962)



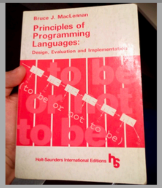
LD-JW

Pascal User Manual and Report (Jensen, Wirth, 1985)



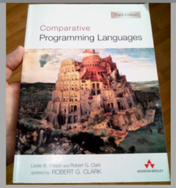
LD-WH

Programming in the .NET Environment (Watkins, Hammond, Abrams, 2003)



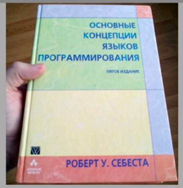
PL-BM

Principles of Programming Languages (MacLennan, 1983)



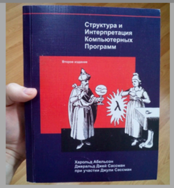
PL-WC

Comparative Programming Languages (Wilson, Clark, 1993)



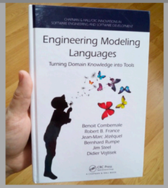
PL-RS

Concepts of Programming Languages (Sebesta, 2001)



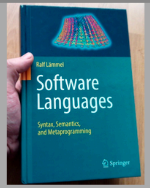
SL-AS

Structure and Interpretation of Computer Programs (Abelson, Sussman, Sussman, 1996)



SL-CF

Engineering Modeling Languages (Combemale, France et al, 2017)



SL-RL

Software Languages: Syntax, Semantics, and Metaprogramming (Lämmel, 2018)

Access Modifier
Annotate components with information about how others are allowed or not allowed to access them. Access can be limited by inheritance (*protected* in C++), modular structure (*internal* in C#), etc. The most popular modifiers are *public* (everyone welcome) and *private* (fully restricted). Similar modifiers can be used to manage scope, such as *global* and *nonlocal* in Python.

Alphabet §
The basic alphabet is often taken for granted, especially for textual languages, but it is an important design aspect. In some languages (APL being the extreme) the alphabet is extremely broad, with specific symbols being used for built-in operators, which shifts the visual feel of the language closer to mathematics. In other languages keywords are taken from English, which limits language appeal to some groups of users (and may lead to reimplementations with translated keywords).

Assignment
Moving a data from one place to another. Some 4GLs have separate statements for straightforward (byte-copying) and composite (pattern-matching) assignments such as Cobol's *MOVE CORRESPONDING* which requires unification. In modern languages the source data structure (and sometimes the target one) can often be created on the fly. Many languages combine assignment with trivial manipulation (such as +=).

Backtracking
A computation strategy commonly found in declarative languages. Every choice in the evaluation path becomes a *save point* to which the computation returns in case of failure. All the changes made between the save point and the point of failure are undone. Backtracking is common in parsers and logic programming, and used for error recovery everywhere else.

Backward Compatibility
In language evolution, introduce new features that should supercede older ones, but ensure the users that their existing code will still run. Ideally, this code should eventually be rewritten and coevolved.

Block §
Viewing a list of statements as a specific (*compound*) kind of statement is a conceptual eye-opener and allows to treat composite constructs in a uniform and orthogonal way (*if … begin … end* and *do … begin … end* instead of *if … endif* and *do … enddo*). Languages either use delimiters (begin/end or curly brackets) or indentation. Blocks can be seen as degenerate subprograms and be useful in optimisation.

Branching
Forking the computation based on conditions known at runtime, is a popular construct. Control flow can be transferred unconditionally (*branch, jump, goto*), or conditionally (based on true/false, zero/positive /negative, explicit condition, exhaustive patterns, etc). In some languages branching can be done by *guarding* statements with constraints.

Character Type
A family of value types that can be used in a language: single characters, special characters, zero-terminated strings, fixed length strings, variable length strings, structured strings, etc.

Class
A class or a trait represents a template that can be followed by objects: a particular collection of properties and methods that can be always relied on. A class can be then *instantiated* with appropriate parameters to form an object that *conforms* to the class definition. Classes are the ultimate form of encapsulation, so they can be inherited from one another to form subclasses.

Client/Server
A language may allow one conceptual model to be split into two intercommunicating components to be executed in parallel: the *server* side which has access to all the necessary system data and runs in a fully controlled environment, and the *client* side which runs closer to the system user's data and has to survive in a much less controllable environment. Client code and server code can be written in different languages or compiled to different languages before deployment.

Code Completion
Many IDEs monitor what the language user is typing and make suggestions based on their knowledge of the language keywords, constructs allowed in the context, variables visible from the current namespace, etc. The list of such suggestions must be short to be useful, otherwise it does nothing but annoy the users.

Code Generation
Generation of machine code, intermediate code, a model in a target language, an output model or a textual result, is the last phase of a classic compiler (before or after optimisation). What is typical for code generation is the richness of the input (generously annotated intermediate graphs) and a deliberate limitedness of the output (which is often platform-specific and/or hardware-specific). In MDE code generation is usually implemented by model-to-text

Code Mining
Besides user surveys and expert opinions, there is a third way to uncover points to improve the language in its next versions: examining existing models created in this language. There are many modern techniques in mining software repositories that can be helpful here: clustering, vocabulary analysis, statistics (especially correlations), natural language processing, information retrieval, machine learning, etc.

Code Ownership
Signing the user's name under a piece of code has the same effect as signing a person's name on an item: caring about what happens to the item later. Comments explaining which dev made which code changes existed since very early on. In modern ecosystems, ownership is tracked automatically by a version control system and can be checked at any time (*git blame*).

Collection
Arrays, lists, tuples, sets and multisets are the most common composite user-defined parametrised types for *collections* of elements. It is up to the language designer to decide which ones are supported and how they are handled — can elements within one collection have different types, are they mutable, passed by name/value/reference, etc.

Comment
Comments are pieces of documentation built directly into the source of the system. Most IDEs support comments visually by presenting them in a completely different colour, usually dimmer than the rest of the model, to focus developers on executable constructs first. In some languages like BibTeX or INTERCAL everything uncompilable is a comment. Some comments like codetags, Javadoc or Documentation Comments are strictly or semi-structured.

Compilation Error
Modern languages have many means of assessing validity of the model before it is actually used. Thus, compilers tend to have a sophisticated error handling facility and try to provide enough information for the language user to fix the problems. Some languages are notoriously known for providing bad error messages. There are many ways to *recover* from an error in order to analyse the rest of the program and report multiple problems at once. Can be provided as a live feedback.

Compilation Warning
When a compiler detects a possibly dangerous situation with extremely limited applicability, it displays a warning message and proceeds with the build process anyway. In many cases there is a special option for disabling a particular warning for a particular piece of code. Warnings can be given when an *anomaly* or a smell is detected, and may involve some form of error *correction*. Can be provided as a live feedback.

Comprehension §
List and set comprehensions are language constructs resembling the mathematical notation for creating a set by its characteristic function (*"for all numbers from 1 to 10, give me their squared values"*), and combine *map* and *filter* classical for functional programming. Comprehensions as a language construct exist in Haskell, Python, Rascal, C# and some other languages.

Concrete Syntax
The way to describe the concrete representation of the programs. The concrete syntax is used by humans to read, write, create and understand sentences of the language. Usually the only languages that do not have concrete syntax are those intended for internal intermediate representation. Some languages have more than one.

Concurrency
Since modern computers and systems are good at multitasking, a language designer may decide to use that. An executable model can then be decomposed into components that are executable in parallel on different CPU cores or different devices. This can be completely undesirable (to avoid deadlocks, overhead, race conditions, etc), or performed automatically, or use the language user's guidance in synchronisation of threads, tasks and processes.

Constraint
Besides languages which programs are expressed only in terms of constraints (OCL, CLR(R), Oz), there are many that have them in one form or another. The most popular form is *assertions*, a non-invasive form of exception handling allows language users to explicitly state (*assert*) invariants, pre-conditions and post-conditions as logic expressions that must universally hold. Such assertions can be easily removed before deploying the system into production.

Cross-compilation
A cross-compiler works on one platform but ultimately targets another. Relying on a cross-compiler allows to separate the development platform from the one where the programs get deployed to — for instance, a mobile app developer can work with a proper keyboard and a big screen. The IDE for a cross-compiled language may include a virtual machine for execution, debugging, etc. A compiler capable of producing code for different targets, is called *retargetable*.

Debugging
The activity of finding and fixing sources of incorrect behaviour is not enjoyed by many language users, but is used by all of them without exception anyway. Declarative and constraint languages are the hardest to debug due to their complex evaluation strategies (unification, backtracking, etc) and imperative ones are the easiest since they specify the algorithm most explicitly. Most modern languages are shipped with a dedicated debugger or have debugging functionality in the IDE.

Default
Unchanged configuration options, uninitialised variables and unspecified optional modifiers are examples of situations when a default value must be used by the compiler. These default values are decided by the language designer and typically represent the *best* option within the paradigm.

Deployment
Once the model written in a language, has been checked, compiled, linked and otherwise prepared for use, it may need to be *deployed*. This happens directly by copying it to the machine of the end user, or by connecting it to the network, or by creating a special installer, etc. In many cases deployment is not viewed as a concern of a language designer, but among practitioners it is perceived as a part of language design.

Deprecated Construct
In language evolution, sometimes a no longer desired construct cannot be simply removed to avoid breaking backward compatibility. However, it can be marked explicitly as deprecated to discourage language users to rely on it.

Design Chart/Diagram
UML distinguishes between structural (class, package, object, component, composite structure, deployment) and behavioural diagrams (activity, sequence, use case, state, communication, interaction overview, timing). The former specify and visualise structure breakdown, the latter — events and interaction. Some languages (e.g., syntactic diagrams) are both.

Documentation
There are two equally important kinds of language manuals: for people learning the language and for its active users — and sometimes these are two disjoint sets of documents. Documentation may contain executable examples and can/should be automatically checked for internal validity and consistency. Some documentation elements must be provided through an IDE, especially if the language is an API.

Encapsulation
Most high level language abstract from low level details like video memory access, memory allocation, register values, caching, etc. Depending on the language design and target audience, these features may be prohibited or just hard to find for beginners. Data structures can also be encapsulated by bundling them into records or classes, and code can be organised in hierarchical modules and subprograms.

Energy Saving
Computationally heavy code requires more CPU or GPU cycles, which consumes more power, which in turn makes the applications spend more energy. Making a compiler of a language especially optimised towards power reduction may increase its appeal for users that intend to run their programs on devices with limited power (mobile phones and smaller). Power reduction and energy saving techniques also contribute towards global sustainability, and can be used/chosen for ethical reasons.

Enumeration Type
An enumeration is a data type that defines a very limited set of possible values which are, nevertheless, represented by their names and not by encoded numbers. The most famous enumeration is the Boolean (logical) type, which contains only two values: *true* and *false*. If the domain permits, the language does not have to support user-defined enumerations.

Esotericism
INTERCAL, Unlambda, Befunge, Malbolge and other esoteric languages are based on paradigms so unconventional that writing even one program puts disproportional strain on the users. This challenging nature makes people engage and compete in programming in such languages as a form of entertainment. LOLCODE, ArnoldC and others are languages developed based on the memes that are circulating among software engineers: the popularity of them piggybacks entirely on the viral nature of

Event
The first implementations of user interfaces were turning the entire program into a giant loop waiting for the user to activate its functionality by choosing the way to communicate (click, tap, edit, etc). Since direct implementations of such an event loop are not green (consume too much energy), event handling can be built natively into the language and implemented efficiently by the compiler and hardware. Events are used for interacting with end users, sensors, threads, etc.

Exception Handling
An emergency sibling of branching used for extraordinary situations — can be slower than the normal branching, but usually more robust in handling situations like a critical failure during the handling of another failure. A less invasive form of exception handling are assertions.

Execution Error
Errors can happen at compile time, but also at run time, due to hardware faults, communication problems, invalid user input or simply bugs that were left undetected at compile time by static analysis. Some languages (Erlang) have very well-designed strategies for handling execution errors, but all others also feature some form of partial recovery from them. The language user controls runtime error handling with exceptions.

Expressivity
There is ultimate expressivity of a software language, typically incorporated in answers to questions like *"is it Turing complete?"* (i.e., does it have enough constructs to emulate a Turing machine?), and there is a much more important and subtle issue of local expressivity in the sense of how small programs can get without sacrificing their readability. Many languages eventually develop shorthand constructs for writing commonly used combinations of constructs shorter and thus faster.

First Class Citizen
It is an important design point to decide which entities within a program have the right to be saved, passed as arguments, transferred through other means, etc. Numbers? Collections? Objects? Functions? Unfinished computations? Data streams? Unfilled templates?

Garbage Collection
Automatic release of memory is impossible for cyclic data structures. Languages that want to support them, have a *garbage collector* — a runtime compiler component that occasionally *marks* data structures that have become inaccessible and then *sweeps* them away, freeing the memory. GC can compromise language responsiveness and performance.

Generation
Tedious, repetitive and error-prone programming tasks can be automated by using templates, wizards, explicit staging/morphing constructs of generative programming, construction workbenches, etc. In many practical cases the language user is allowed to edit the result to fine-tune it. The final generation phase is called code generation.

Heterogeneous Data
Some languages allow considerable freedom in types that makes collections capable of carrying elements of varying structure. Examples: variant records in Modula and Ada, heterogeneous lists in Python, polytypic functions in Haskell, GADTs in Haskell. Allowing heterogeneity empowers the language user but makes the language harder to learn.

IDE
Integrated Development Environments (IDEs) are used to support language users in their common tasks: code navigation, debugging, building, modularising, refactoring, etc. Can take a form of a dedicated standalone editor, a website or a plugin for a universal editor. Needs to have a well-designed UI.

IDE GUI
Most IDEs divide the screen space among areas with different functionality: for navigating through adjacent models, for editing the code, for reviewing the architecture, for watching how values change at runtime, etc. Advanced IDEs like IntelliJ, Eclipse or VS.NET have so many subwindows that the user has to choose which ones to keep open at each given time.

Indentation & Whitespace
The two extremes for this aspect are: treat indentation as something crucial to the program structure (and thus process constructs differently based on columns where they start) and discard all possible indentation (even in the middle of names, as FORTRAN does). Most languages are somewhere in the middle. Normalisation of whitespace use is called pretty-printing.

Inheritance
An "is-a" relation can be represented by a language construct when one class, object or function *inherits* all the properties of its *parent* and possibly adds others exclusive to itself. It is a design consideration which entities can be derived from which, what are the rules for inheriting from several parents, etc.

Input/Output
Most executable models are not self-contained and require input data to run and produce results, which in turn need to be propagated somewhere. There are languages that are volatile with input and output, those that only work with files, those that wrap I/O as a side effect of a monad, etc.

Instruction Set
Instead of freely combinable statements and expressions, low level languages (microcodes, assemblers, virtual machine bytecodes, etc) have limited non-extendable instruction sets. Each of the instructions typically has a mnemonic (name) and a bit-level encoding. Realistic assemblers had to introduce macro expansions to make expressivity and programming experience tolerable.

Iteration
There are many looping constructs, ranging from the imperative classics such as a for loop, to the functional classics such as *map*, *filter* and *fold* (or *reduce*). It is not uncommon for languages to support only some of these constructs. Some older GPLs and 4GLs also have one iterative construct which can be annotated with all kinds of conditions and steppers.

Keyword
Special words in concrete syntax of the language that carry identical meaning across all possible models in the same language. Can be made reserved so that programmers may not redefine them. A language can get new keywords by evolution.

Labelling
Since most engineers know several languages, some language manuals directly assume initial familiarity of their users with other languages. Can refer to paradigms or families (*"this language is strongly typed"*) or directly to other languages (*"inheritance works like in Java"*). Also, by explicitly stating which camp the language is siding with or which key community figures endorse it, the designer can invoke an emotional response directly mappable to language's acceptance and popularity.

Lazy Evaluation
A *lazy* compiler defers evaluation to the latest possible moment. Lazy languages allow infinite data structures (as long as they are processed one chunk at a time) and may have unpredictable outcomes if calculations are allowed to have side effects (like C's ++). Lazy evaluation has many applications from optimisation of generated code to stream data processing.

Live Feedback
An advanced IDE running on modern hardware can utilise its idle cycles to attempt parsing, compilation, dependency analysis and other kinds of checks while the language user is still typing the model. Erroneous and suspicious pieces of code are commonly underlined with red or yellow squiggly lines familiar from natural word processors.

Lock-out/Opt-in
Certain combinations of language features may be disabled (erroneous) by default, with a possibility of enabling them explicitly. For example, redefining a method in a derived class is only allowed in C# when a specific *override* keyword is used, which leaves visual cues to future readers of the piece of code in question.

Macro
A mechanism commonly found in low level languages that allow users to define a piece of syntactic sugar to be *expanded* into a longer sequence of instructions. Advanced parametrised macros resemble subprograms in expressivity but may behave less reliably due to their lexical nature. In bigger languages macros are typically handled by a preprocessor.

Metaphor
Many language constructs names that need no explanation: atom, backtracking, binding, body, build, cloud, collision, compiler, dangling else, debugging, desugaring, dictionary, duck typing, environment, filter, floating point, forest, framework, garbage collection, go to, heap, inheritance, jump, library, linking, map, pointer, pruning, rendezvous, stack, turtle, weaving, window, ...

Module
Large models inevitably outgrow their creators' capabilities to understand them all at once. Comprehension can be aided greatly by the language providing modules, packages, classes, procedures, blocks and other elements to group related code fragments together. Modern IDEs can analyse code for cohesion and coupling to help improve modularisation. Modules are often [one of the possible] *compilation units*.

Natural Pattern
Design patterns, implementation patterns and architecture patterns are used across language boundaries, but many domain-specific languages incorporate well-known patterns as native language constructs: Model-View-Controller, Singleton, State, Visitor, etc.

Numeric Data Type
Often gets overlooked at the early stages of language design, but could significantly shape the application area of the language. There are many integer types, distinguished by their byte sizes and therefore value ranges; also decimal types with fixed scale and precision; and floating point types good for scientific computations but not for handling finances.

Operator Overloading
A language designer may decide to reuse the same symbol for several different operators, usually conceptually related (such as + for arithmetic addition and string concatenation). Using it for totally unrelated operations is considered harmful for readability (such as & for pointer referencing and bit conjunction in C). In some languages (C++, Ada, Fortran) language users can also redefine their own operators that complement their own defined types.

Operator Precedence
To avoid excessive use of parentheses, a language can provide a default convention of binding constructs with 3+ entities bound by binary operators. In arithmetic expressions, the precedence usually follows mathematical laws.

Optimisation
It is always easier and less error-prone to generate intermediate code or machine code with simple and straightforward patterns and subsequently optimise the result in a different phase. The effect on the language users is that they do not need to optimise their models to the fullest, since their own naïve code will be optimised together with the rest. Small efficiences are only relevant 3% of the time, for the rest premature optimisation is considered the root of all evil.

Order
Many languages have ordering constraints: a variable must be declared before its use, a function signature known before its call, etc. Sometimes constructs are grouped and it is the groups that must follow the order: e.g., first all declarations, then all functions, then the rest of the code (COBOL's *divisions* are the extreme example of this).

Orthogonal Design
Independent features should be controlled by independent mechanisms. Things that are similar should look similar and different ones should look different. Regular rules without exceptions are easier to learn. The fewer surprises one has while learning the language, the higher the language quality.

Parameter Passing
There are several strategies in mapping *arguments* that are being passed to a procedure in a call with the *parameters* that procedure expects to get: *call by value* (expose only the values, safe but inefficient for composite data), *call by result* (can return several values at once), *call by value-result* (the caller gets values, updates them, they are passed back), *call by reference* (expose pointers to values, efficient but unsafe), *call by name* (evaluate pointers when they are used inside the caller), etc.

Parametrised Type
Some types can be defined partially by the user and partially by the language designer. For example, the language *designer* knows what a list is, and the language *user* can select any other type for list elements — this will change handling of such elements, but the philosophy behind their collection will stay the same.

Performance
Performance testing and its variations like profiling and stress testing are commonly desired nice-to-have features in IDEs. Languages and their ecosystems greatly vary in the extent to which this aspect is recognised and supported.

Phased Process
Breaking a process into phases is one of the most used divide-and-conquer principles applied in language processing. Most compilers are designed to work in phases, and different competences and skills are required to implement each phase.

Picture Clause
A data type that saves a specially formatted entity (usually a float or a date) that can be used directly in printing statements but also manipulated as data.

Platform Lock-in/out
Supporting a great language only for one particular hardware platform, OS or IDE, implicitly forces people to use them. For example, malware practices of Java installers turned some users agains JVM, which also deprived them of Scala and Clojure. Another example is .NET Core, a redesign of the .NET Framework which allows typically Windows-specific code to run on Linux.

Pointer
A popular data type in low level languages, representing a memory address where the data structure is stored — which is more efficient to pass across functions than the structure itself. The type of the structure needs to be known to decipher its contents, since the pointer itself is nothing more than a number.

Pretty-printing
A language can have a default formatting convention that is not only accepted by the community to improve the representation quality of the models, but also automated and shipped in a form of a tool. Such a tool can be very configurable, have limited feature selection or none at all. A pretty-printer that scans the input and minimises the delimiters in it, is sometimes called a program *compactor*. Pretty-printers are omnipresent in textual languages and may require layout strategies/policies for graphical ones.

Preview
Some features are very useful in general, but implemented in a way that sometimes fails. In this case, the impact of an application of a feature can be explicitly examined by the language user before agreeing to proceed. Common for database queries and object-oriented refactorings.

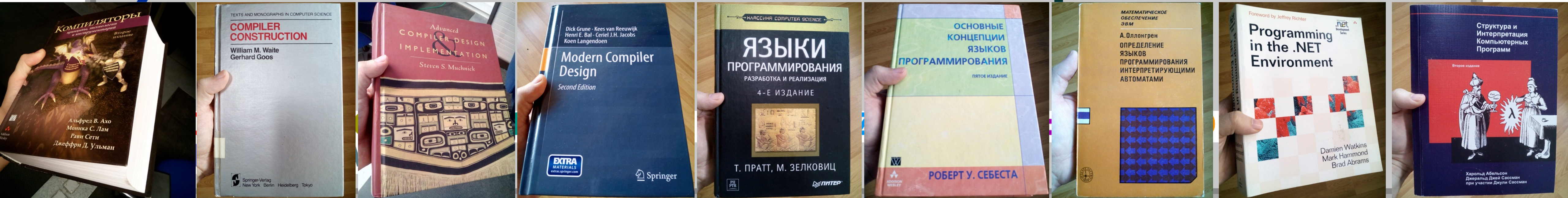
Concurrency

Since modern computers and systems are good at multitasking, a language designer may decide to use that. An executable model can then be decomposed into components that are executable in parallel on different CPU cores or different devices. This can be completely undesirable (to avoid deadlocks, overhead, race conditions, etc), or performed automatically, or use the language user's guidance in synchronisation of threads, tasks and processes.

DB-PD:51, CC-WG:32, CD-SM:571, CD-GR:331, LI-PZ:483, PL-RS:503, PT-AO:254, LD-WH:419, SL-AS:254

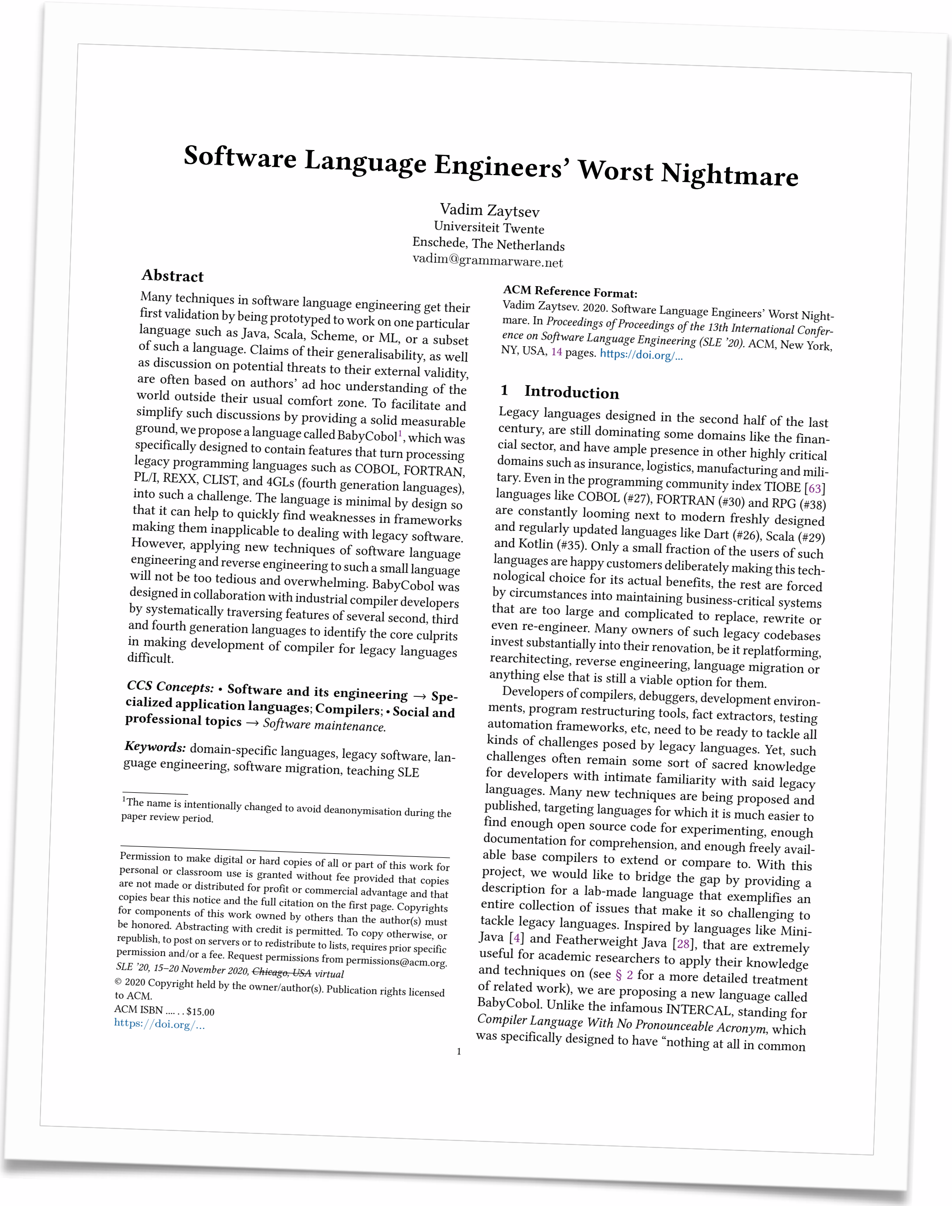
Concurrency

Since modern computers and systems are good at multitasking, a language designer may decide to use that. An executable model can then be decomposed into components that are executable in parallel on different CPU cores or different devices. This can be completely undesirable (to avoid deadlocks, overhead, race conditions, etc), or performed automatically, or use the language user's guidance in synchronisation of threads, tasks and processes.



10. BabyCobol

- Indentation has semantics
- Imports are lexical
- Keywords are not reserved
- Assignments are name-driven
- GO TOs can be ALTERed
- Expressions have contractions
- . . .



Software Language Engineers' Worst Nightmare

Vadim Zaytsev
Universiteit Twente
Enschede, The Netherlands
vadim@grammarware.net

Abstract

Many techniques in software language engineering get their first validation by being prototyped to work on one particular language such as Java, Scala, Scheme, or ML, or a subset of such a language. Claims of their generalisability, as well as discussion on potential threats to their external validity, are often based on authors' ad hoc understanding of the world outside their usual comfort zone. To facilitate and simplify such discussions by providing a solid measurable ground, we propose a language called BabyCobol[1], which was specifically designed to contain features that turn processing legacy programming languages such as COBOL, FORTRAN, PL/I, REXX, CLIST, and 4GLs (fourth generation languages), into such a challenge. The language is minimal by design so that it can help to quickly find weaknesses in frameworks making them inapplicable to dealing with legacy software. However, applying new techniques of software language engineering and reverse engineering to such a small language will not be too tedious and overwhelming. BabyCobol was designed in collaboration with industrial compiler developers by systematically traversing features of several second, third and fourth generation languages to identify the core culprits in making development of compiler for legacy languages difficult.

CCS Concepts: • **Software and its engineering** → **Specialized application languages**; **Compilers**; • **Social and professional topics** → *Software maintenance.*

Keywords: domain-specific languages, legacy software, language engineering, software migration, teaching SLE

[1]The name is intentionally changed to avoid deanonymisation during the paper review period.

SLE '20, 15–20 November 2020, ~~Chicago, USA~~ virtual

ACM ISBN $15.00
https://doi.org/...

ACM Reference Format:
Vadim Zaytsev. 2020. Software Language Engineers' Worst Nightmare. In *Proceedings of Proceedings of the 13th International Conference on Software Language Engineering (SLE '20).* ACM, New York, NY, USA, 14 pages. https://doi.org/...

1 Introduction

Legacy languages designed in the second half of the last century, are still dominating some domains like the financial sector, and have ample presence in other highly critical domains such as insurance, logistics, manufacturing and military. Even in the programming community index TIOBE [63] languages like COBOL (#27), FORTRAN (#30) and RPG (#38) are constantly looming next to modern freshly designed and regularly updated languages like Dart (#26), Scala (#29) and Kotlin (#35). Only a small fraction of the users of such languages are happy customers deliberately making this technological choice for its actual benefits, the rest are forced by circumstances into maintaining business-critical systems that are too large and complicated to replace, rewrite or even re-engineer. Many owners of such legacy codebases invest substantially into their renovation, be it replatforming, rearchitecting, reverse engineering, language migration or anything else that is still a viable option for them.

Developers of compilers, debuggers, development environments, program restructuring tools, fact extractors, testing automation frameworks, etc, need to be ready to tackle all kinds of challenges posed by legacy languages. Yet, such challenges often remain some sort of sacred knowledge for developers with intimate familiarity with said legacy languages. Many new techniques are being proposed and published, targeting languages for which it is much easier to find enough open source code for experimenting, enough documentation for comprehension, and enough freely available base compilers to extend or compare to. With this project, we would like to bridge the gap by providing a description for a lab-made language that exemplifies an entire collection of issues that make it so challenging to tackle legacy languages. Inspired by languages like MiniJava [4] and Featherweight Java [28], that are extremely useful for academic researchers to apply their knowledge and techniques on (see § 2 for a more detailed treatment of related work), we are proposing a new language called BabyCobol. Unlike the infamous INTERCAL, standing for *Compiler Language With No Pronounceable Acronym*, which was specifically designed to have "nothing at all in common

[BENEVOL'19] [PRiML'20] **[SLE'20]** http://slebok.github.io/baby

Conclusion

BGF	Rascal	Engage!
PAX	TIALAA	CSS
HLASM	MegaL	DYOL
	BabyCobol	

http://grammarware.net || grammarware.github.io