Open Challenges in Incremental Coverage of Legacy Software Languages

Vadim Zaytsev
Raincode Labs, Belgium
vadim@grammarware.net

Abstract

Legacy software systems were often written not just in programming languages typically associated with legacy, such as COBOL, JOVIAL and PL/I, but also in decommissioned or deprecated 4GLs. Writing compilers and other migration and renovation tools for such languages is an active business that requires substantial effort but has proven to be a successful strategy for many cases. However, the process of covering such languages (i.e., parsing their close overapproximation and assigning the right assumed semantics to it) is filled with unconventional requirements and limitations: the lack of useful documentation, large scale of codebases, counter-intuitive language engineering principles, buggy reference implementations, fragile workarounds for them, etc.

In this short paper, we motivate the incremental nature of software language engineering when it concerns legacy languages in particular, and outline a few related challenges.

CCS Concepts • **Software and its engineering → Domain specific languages**; **Software reverse engineering**; *Translator writing systems and compiler generators*; *Source code generation*; *Parsers*; Incremental compilers;

Keywords legacy software systems, fourth generation programming languages, compiler construction

ACM Reference Format:
Vadim Zaytsev. 2017. Open Challenges in Incremental Coverage of Legacy Software Languages. In *Proceedings of 3rd ACM SIGPLAN International Workshop on Programming Experience (PX/17.2)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3167105

1 Introduction

There are hundreds of software languages currently in use, and only a few dozens of them are in widespread use. For the rest, it is possible to find systems written in those languages, deployed in ecosystems characteristic for them and being worked on by developers familiar with the fine details of language constructs and context, but it is not realistic to hire new engineers familiar with the language, unless they are switching from one such non-mainstream project to another. Most of these not-widespread languages are prototypical, experimental, esoteric or domain-specific, which means that they were meant to cater to very peculiar needs of a limited group of people solving specific problems. The remaining ones are what people refer to as *legacy* languages.

Many languages that became legacy, were once called *fourth-generation programming languages* (4GLs), the name signifying the fact that they came after the first generation (machine codes), second generation (assemblers) and third generation (compiled ones). Basically, 4GLs are DSLs (domain-specific languages) [6, 10, 16, 28, 34, 39, 41, 44] designed around the 1980s before powerful language workbenches and other software language engineering technologies made the process easy and (to some extent) error-proof. To reflect the reality closer, we borrow the term "software languages" from the domain of software language engineering [2, 33] where it means any language used in creation of software, be it a programming language, a database schema, a markup notation or a type system.

Besides being "DSLs for database applications" [26, 28], many 4GLs can be regarded as *badly designed DSLs* because they come short on the number of issues that are typically associated with DSL benefits [6, 28, 39, 41]:

- **domain-specific notations** [28] are missing or poor;
- **domain-specific abstractions** [28] are not beyond what a library would offer [39];
- **tool support** for analysis, verification, optimisation, transformation [28], simulation and animation [41] is extremely limited;
- **conciseness** [41] and **self-documentation** [6] claims are subpar to modern alternatives;
- **productivity** and **maintainability** boosts [6] are undermined by the impossibility of hiring new developers to use outdated technology;
- **reliability** allegations [6] rest on deployment platforms that are no longer desirable for some reason such as high costs;
- any hope for **portability** [6] is destroyed by vendor lock-in on obsolete frameworks and platforms;
- **conservation and reuse of domain knowledge** [6] are not achieved due to leaky abstractions [39];
- **testability** [6] opportunities are usually missed;

PX/17.2, October 22, 2017, Vancouver, BC, Canada

- the **lifespan** of a DSL is that of months or years [39] while 4GLs outlive their originally intended lifespan by decades.

Programs in a 4GL are typically compiled (essentially desugared [23]) into a 3GL and then fed into a standard compiler, possibly complemented by code fragments written directly in the 3GLs (and sometimes 2GLs [3]) for performance and expressiveness reasons. Each 4GL is usually known by a small and steadily shrinking group of people and used inside companies that have their entire business logic expressed in that particular 4GL. Eventually every 4GL is decommissioned, and then the companies using it are faced with a choice of migration, re-engineering or bankruptcy. The typical technical options are:

- **Language conversion** [37]. The entire existing codebase is converted to conform to a different language. The closer the source and target languages are to each other, the closer this conversion gets to a so-called "syntax swap" which is a fairly straightforward and testable procedure with good chances of success. If the conceptual gap between the two is too great, native constructs of the source language creep in an emulated form into the code in the target language, and pollute the codebase. Code converted in such a way requires expensive specialists versed in both source and target languages in order to perform any maintenance actions on the result.
- **Automated refactoring** [4, 11]. The existing codebase is fed into the 4GL compiler, and the produced code in a 3GL is taken as its representation. Such representation is honest since it is literally the same code that was being deployed all along, but it is also ugly and contains a lot of artefacts that made 3GL code generation easier but are considered bad patterns and code smells in manually written code. For COBOL such artefacts include duplicate code, `PERFORM THRU` clauses, `GO TO` statements, etc [4]. Such code is then refactored automatically to replace harmful patterns with functionally equivalent counterparts of higher quality.
- **Complete reengineering**. The existing codebase is used as a source of information about the business logic, which is then reimplemented with modern tools and frameworks. This path is risky, error-prone and labour-intensive, but feasible for projects with considerably more legacy boilerplate than core logic.
- **Compiler reimplementation**. Instead of reimplementing the actual code of a software asset, it can be preserved in its original state, while the infrastructure around it gets reengineered. Compilers, debuggers, refactoring aiding tools and even IDEs [46] form this infrastructure, and to replace them means to reverse engineer the *language* from the existing tools and documentation, and reimplement it with modern technology. It is one of the most costly options, but also the most stable one in the long run: the newly redeveloped compiler will be under control, full or partial, of the code owner, who can perform or request many sought-after modifications and upgrades that are impossible with legacy language processing tools. Since the problem of incremental coverage of a software language manifests itself the worst in this scenario, and also since the paper author works for a company regularly providing such services (i.e., of compiler reimplementation or automated refactoring) to paying customers, we will focus on it for the rest of the paper.

A new compiler requires a substantial effort to become operational, since even writing a quality parser for a COBOL-like language takes 2–3 years, according to professionals [27]. However, it provides numerous advantages such as modern deployment platforms (e.g., cloud), modern IDEs (e.g., Eclipse or Visual Studio), as well as the opportunity to evolve the language further in a desired direction—the combination of these offers a threatened company a very bright future.

Besides classic compiler design and development challenges elaborately articulated in many books [13, 44], compiler engineering for legacy languages faces others such as the lack of documentation (which either does not exist, or is outdated beyond usefulness, or may not be used for legal reasons) or a mixture of language rules (imposed by the original compiler) and company conventions (imposed by in-house coding style manuals).

The process of incrementally *covering* a legacy software language goes as follows:

- there is a legacy language, represented by a complete codebase of the customer company and complemented by the knowledge of the in-house experts;
- there is a grammar that covers some of the language and is being continuously worked and improved on;
- a parser is generated automatically from the grammar and tried on all available code samples;
- one of the failing code samples is considered by the grammar engineer who changes the grammar to accommodate it;
- the cycle is repeated until the entire codebase can be parsed;
- semantics of a compiler, interpreter, translator, refactorer, etc, are assigned to syntactic constructs that occur in parse trees, also in an iterative way with substantial amount of testing and verification;
- very occasionally the grammar is revisited and refined until the codebase can be comfortably compiled or migrated;

- the life cycle continues naturally, evolution being driven forward by feature requests and bug reports by end users.

The idea of incremental grammar engineering was conceptually proposed by Klint et al. [21], there are case studies reported in detail [1, 42], as well as techniques to converge multiple imperfect grammars [25] and many others that can also be useful. These techniques can be either applied directly or used from a language workbench [7, 9] such as Rascal [22], Spoofax [17], ASF+SDF Meta-Environment [20], JastAdd [14], Xtext [8], Intentional [32] or MPS [40]. As a more recent example of a realistic combination of grammar engineering, tool building and static program analysis, we refer to Vavrová et al [38]. However, the incremental aspect is not researched well enough to provide tool support for it (which would go beyond conceptual process guidelines).

There is substantial research on incremental parsing, which is usually a combination of semiparsing [43] and ad hoc algorithms of selective or lazy parsing. However, they solve a much less painful problem of reducing parse time, as opposed to reducing compiler development time, which has substantially bigger impact on developers' lives. It is possible for powerful parallel parsing techniques in combination with bug prioritisation [19], fault localisation [12] and error clustering [5] to become somewhat useful at some point, but will not change our lives drastically in the near future.

2 R&D Challenges

We identify at least the following research challenges:

2.1 Regression Parsing

Similar to regression testing [31] we would like to be able to quickly and incrementally check if the change in the grammar or its surroundings has had any detrimental effect on the standing coverage of the language. Parsing is understood here in a broad sense [47] as the process of recovering structure, so it can involve, depending on the problem, regular matching, context-free parsing, abstract syntax graph generation, semantic analysis, symbol table population, etc. Reparsing the entire codebase is definitely an option, but the usual scale of legacy portfolios is in the tens or hundreds of millions of lines of code, which unnecessarily brings up scalability and performance challenges toward the very beginning of the project.

During later stages of the project this typically takes place on a separate server running the nightly build system. It is useful to get a daily report emailed to the lead developers, but daily reports neither have the same role nor effect on them as live or on demand feedback.

For example, in one of our projects, compiling a 4GL to .NET, if the regression testing suite involves recompilation of the entire codebase of the one customer company, including preprocessing and performing some basic sanity checks like PEVerify [29] on the resulting DLLs, the duration of one test run is around 48 hours. As a consequence, it only runs once a week on a server, and all involved developers resort in their daily work to manually created tests of limited scope, which take time and effort to create and do not guarantee soundness nor coverage.

2.2 Grammar Inference from Codebase

The field of grammatical inference is vast [36] and full with techniques that are too theoretical and slow to be applied generally, but can become quite attractive if optimised by taking context conditions into account. In particular for incremental coverage of software languages, it can be interesting to infer a grammar that covers *most of* the input language or the one that covers all of it but requires a human engineer to name identified constructs.

The same conceptually but utterly un-automatable process is that of semantic inference. In the case of the total lack of documentation, compiler engineers need to infer the semantics manually from the codebase. This process is complex and error-prone, and involves investigating the vicinity of the construct in question in the code (e.g., assigning input parameters and processing outputs of a library function), leveraging previous knowledge of similar constructs in other languages (e.g., it is safe to assume that a picture clause in any language will work similarly to COBOL's `PIC`s) and working in close collaboration with living language experts (e.g., for date and time processing there is no standard library on the mainframe, so every language does it in a different way incompatible with alternatives).

For example, consider a recently finished project with position-sensitive pattern languages [45]. We needed a very fast parser of a notation that was based on character positions within a line, so we built a custom parser generator, but also used grammatical inference techniques to create the first approximation of a specification covering the language. The specification was further refined to adjust underspecified commitments (e.g., places which were inferred as "three zeroes followed by two digits" while the true specification was "five digits" and accidentally the codebase did not contain any value in that field larger than 100), underspecified bindings (e.g., if one position contains a one-letter code from a limited character range, it is impossible for an automated algorithm to decide to represent it in the AST as a string, a Boolean, an enumeration, etc), and uninformative names (e.g., numeric codes standing for enumerated concepts, or several obviously string names that are used for database access, code and UI bindings—impossible to differentiate automatically).

2.3 Test Suite Inference from Codebase

The current practice of regression parsing is to approximate the codebase with a test suite which must be gradually designed by hand and coevolve [24] with each discovered detail of the language. Alternatively, this can be done automatically by

a some combination of test prioritisation [35] and test suite minimisation [38]. Modern machine learning techniques based on the naturalness of software [15] (the observation that software exhibits many traits typical for natural language corpora), look promising, and were shown to yield positive results in the past with language identification [18].

For example, to start making any claims of correctness of a compiler, we need to create an inventory of all the features of a software language (statements, blocks, expressions, data types, library functions, etc), which is an extremely labour-intensive process that is also error-prone. The current practice is to create this inventory manually from the codebase, co-evolve it with the grammar and at some point share with the customer's language experts to get feedback on its completeness. Eventually each of the constructs mentioned in the inventory, is covered by a few test cases—which are, again, written manually or generated by ad hoc model-based frameworks.

2.4 Dependency Analysis within the Compiler

Many incremental techniques can be based on the knowledge of dependencies existing within the grammar or, speaking broader, within compiler components, since it enables fast and cheap impact analysis of the changes. There is a substantial body of techniques and tools on change impact analysis [30], but to the best of our knowledge, they have not yet been extended to grammars, parsing specifications and other compiler-building models, nor to compilers themselves.

For example, in a freshly made snapshot of the codebase delivered by the customer we see that identifier names can be followed by `()` which signifies that a call is being made to some callable entity represented by the identifier. The grammar is readjusted to accommodate that. Do any other parts of the grammar need to be co-evolved? Do any parts of the code generator need to be redeveloped as well? The state of the art in this issue is randomly poking around and "thinking hard" which is not the most sound software engineering strategy, even if it is the most widely used one.

2.5 Dependency Analysis on the Grammar vs. Samples

Alternatively or complementary, we can investigate and develop algorithms for tracing dependencies between the codebase (or some representation thereof like a repository of parse trees) and the grammar, and updating the parsability metadata incrementally whenever needed or whenever the CPU is idle, to provide the results of regression/impact analysis to the grammar engineer.

For example, a compiler engineer learns that the software language being reimplemented, supports implicit conversion of decimals to integers by means of truncation. The change involves the classic implementation strategy for type conversions (each expression node in the AST having two pre-computed type attributes: the actual type and the expected type), as well the corresponding adjustments in the code generator. Do all tests need to be re-run? Only type checking tests? All semantic analysis tests? In any project of substantial size the question concerns hundreds and thousands of test cases, so the difference in the test runnning time is hundredfold. The state of the art is to go with the feel of the engineer and hope for the best for the next full daily/weekly regression run.

2.6 Neighbour Analysis

A lot of grammar development cycles are lost on tiny tweaks of features: can you only print a variable or can it be any expression? is assignment target always one identifier or can it be multiple? can you call a method on the result of call of another method? is there an `else` clause to the `if`? how does a default clause of the `caseof` look like? The answers to all these questions in mainstream languages are either straightforward or easily obtained from language manuals (or even StackOverflow discussions). For legacy languages, we must follow the trial and error approach. However, it should be possible, given both the original tested grammar and the changed one, and detailed enough information about the language coverage of both, to identify and show examples of "near misses"— cases that were bluntly rejected by the original syntactic or semantic analyser and still fail in the new one but "later" or "better" by some definition. Alternatively, some language constructs may initially appear to be more complex than they actually are, and their grammar specification can be simplified.

Practical compiler specifications contain a lot of failsafes: if one of the statements in a block cannot be parsed but a symbol is known that typically separates adjacent statements from each other, then it will be parsed as an unknown statement just to let the parsing continue. Similarly on the later stages, if a call is made to an undefined function, the compiler will try its hardest to locate the function, but will not fail the compilation altogether, simply because this function might be a built-in library function that remained unknown until that moment. Such implementation strategies are a common way to achieve parseability and compileability very early on in the process of covering a software language (legacy or otherwise) and be able to quantify and examine the remaining bits in high level of detail.

3 Conclusion

In this short paper, we have raised a question of developing a systematic approach of covering legacy software languages incrementally. A few challenges were outlined in § 2 to be solved in developing, testing, validating and deploying parsers and other form of grammarware. There do not seem to be major roadblocks in this research direction, but engineering and architectural challenges are substantial.

References

[1] Tiago Laureano Alves and Joost Visser. 2008. A Case Study in Grammar Engineering. In *Revised Selected Papers of the First International Conference on Software Language Engineering (SLE) (LNCS)*, Vol. 5452. Springer, 285–304. https://doi.org/10.1007/978-3-642-00434-6_18

[2] Anya Helene Bagge and Vadim Zaytsev. 2015. Open and Original Problems in Software Language Engineering 2015 Workshop Report. *SIGSOFT Software Engineering Notes* 40 (May 2015), 32–37. Issue 3. https://doi.org/10.1145/2757308.2757313

[3] Volodymyr Blagodarov, Yves Jaradin, and Vadim Zaytsev. 2016. Tool Demo: Raincode Assembler Compiler. In *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). 221–225. https://doi.org/10.1145/2997364.2997387

[4] Darius Blasband. 2016. *The Rise and Fall of Software Recipes*. Reality Bites Publishing. http://dariusblasband.com

[5] Giulio Concas, Cristina Monni, Matteo Orrù, and Roberto Tonelli. 2014. Clustering of Defects in Java Software Systems. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics (WETSoM)*. ACM, 59–65. https://doi.org/10.1145/2593868.2593879

[6] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Notices* 35, 6 (June 2000), 26–36. https://doi.org/10.1145/352029.352035

[7] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches — Conclusions from the Language Workbench Challenge. In *Proceedings of the Sixth International Conference on Software Language Engineering (SLE) (LNCS)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[8] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 307–309. https://doi.org/10.1145/1869542.1869625

[9] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? MartinFowler.com. (June 2005). https://martinfowler.com/articles/languageWorkbench.html

[10] Martin Fowler. 2010. *Domain Specific Languages*. Addison-Wesley Professional.

[11] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design or Existing Code*. Addison-Wesley.

[12] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. 2012. Diversity maximization speedup for fault localization. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 30–39. https://doi.org/10.1145/2351676.2351682

[13] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. 2012. *Modern Compiler Design* (second ed.). Addison-Wesley. https://dickgrune.com/Books/MCD_2nd_Edition/

[14] Görel Hedin and Eva Magnusson. 2003. JastAdd — An Aspect-Oriented Compiler Construction System. *Science of Computer Programming (LDTA'01 Special Issue)* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0

[15] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Martin Glinz, Gail C. Murphy, and Mauro Pezzé (Eds.). IEEE, 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[16] Gabor Karsai, Holger Krahn, Class Pinkernell, Bernhard Rumpe, Martin Schneider, and Steven Völkel. 2009. Design Guidelines for Domain Specific Languages. In *Proceedings of the Ninth OOPSLA Workshop on Domain-Specific Modeling (DSM 2009)*, Matti Rossi, Jonathan Sprinkle, Jeff Gray, and Juha-Pekka Tolvanen (Eds.). 7–13. https://arxiv.org/abs/1409.2378

[17] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the 25th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 444–463. https://doi.org/10.1145/1869459.1869497

[18] Juriaan Kennedy van Dam and Vadim Zaytsev. 2016. Software Language Identification with Natural Language Classifiers. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering: the Early Research Achievements track (SANER ERA)*, Katsuro Inoue, Yasutaka Kamei, Michele Lanza, and Norihiro Yoshida (Eds.). IEEE, 624–628. https://doi.org/10.1109/SANER.2016.92

[19] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first?. In *Proceedings of the Sixth Joint Meeting of the 11th European Software Engineering Conference and the 15th International Symposium on Foundations of Software Engineering*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 45–54. https://doi.org/10.1145/1287624.1287633

[20] Paul Klint. 1993. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 2 (1993), 176–201. https://doi.org/10.1145/151257.151260

[21] Paul Klint, Ralf Lämmel, and Chris Verhoef. 2005. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (TOSEM)* 14, 3 (2005), 331–380. https://doi.org/10.1145/1072997.1073000

[22] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[23] Shriram Krishnamurthi. 2015. Desugaring in Practice: Opportunities and Challenges. In *Proceedings of the 20th Workshop on Partial Evaluation and Program Manipulation*. ACM, 1–2. https://doi.org/10.1145/2678015.2678016

[24] Ralf Lämmel. 2016. Coupled Software Transformations—Revisited. In *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*. ACM, 239–252. https://doi.org/10.1145/2997364.2997366

[25] Ralf Lämmel and Vadim Zaytsev. 2009. An Introduction to Grammar Convergence. In *Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009) (LNCS)*, Michael Leuschel and Heike Wehrheim (Eds.), Vol. 5423. Springer, 246–260. https://doi.org/10.1007/978-3-642-00255-7_17

[26] James Martin. 1985. *Fourth Generation Languages*. Prentice Hall.

[27] Vadim Maslov. 1998. Re: An Odd Grammar Question. http://compilers.iecc.com/comparch/article/98-05-108. (May 1998).

[28] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892

[29] Ron Petrusha, Maira Wenzel, Luke Latham, Tom Pratt, and Yisheng Jin. 2017. Peverify.exe (PEVerify Tool). (2017). https://docs.microsoft.com/en-us/dotnet/framework/tools/peverify-exe-peverify-tool

[30] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 432–448. https://doi.org/10.1145/1028976.1029012

[31] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[32] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. In *Proceedings of the 21th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Peri L. Tarr and William R. Cook (Eds.). ACM, 451–464. https://doi.org/10.1145/1167473.1167511

[33] SLE. 2008–2017. International Conference on Software Language Engineering. http://www.sleconf.org. (2008–2017).

[34] Diomidis Spinellis. 2001. Notable Design Patterns for Domain-specific Languages. *The Journal of Systems and Software* 56, 1 (2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3

[35] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 97–106. https://doi.org/10.1145/566172.566187

[36] Andrew Stevenson and James R. Cordy. 2014. A Survey of Grammatical Inference in Software Engineering. *Science of Computer Programming* 96 (2014), 444–459. https://doi.org/10.1016/j.scico.2014.05.008

[37] Andrey A. Terekhov and Chris Verhoef. 2000. The Realities of Language Conversions. *IEEE Software* 17, 6 (Nov./Dec. 2000), 111–124. https://doi.org/10.1109/52.895180

[38] Nicole Vavrová and Vadim Zaytsev. 2017. Does Python Smell Like Java? *The Art, Science and Engineering of Programming (‹Programming›)* 1 (April 2017), 11–1–11–29. Issue 2. https://doi.org/10.22152/programming-journal.org/2017/1/11

[39] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[40] Markus Völter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE, 1449–1450. https://doi.org/10.1109/ICSE.2012.6227070

[41] David S. Wile. 2001. Supporting the DSL Spectrum. *Journal of Computing and Information Technology* 9, 4 (2001), 263–287. https://doi.org/10.2498/cit.2001.04.01

[42] Vadim Zaytsev. 2005. Correct C♯ Grammar too Sharp for ISO. In *Participants Workshop, Part II of the Pre-proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*. Technical Report, TR-CCTC/DI-36, Universidade do Minho, Braga, Portugal, 154–155. Extended abstract.

[43] Vadim Zaytsev. 2014. Formal Foundations for Semi-parsing. In *Proceedings of the Software Evolution Week (IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering), Early Research Achievements Track (CSMR-WCRE 2014 ERA)*, Serge Demeyer, Dave Binkley, and Filippo Ricca (Eds.). IEEE, 313–317. https://doi.org/10.1109/CSMR-WCRE.2014.6747184

[44] Vadim Zaytsev. 2017. Language Design with Intent. In *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. IEEE, 45–52. https://doi.org/10.1109/MODELS.2017.16

[45] Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 212–218. https://doi.org/10.1145/3136040.3136058

[46] Vadim Zaytsev. 2017. Parsing @ IDE. In *Pre-proceedings of the Fifth Annual Workshop on Parsing Programming Languages (Parsing@SLE)*. http://grammarware.net/text/2017/parsingatide.pdf

[47] Vadim Zaytsev and Anya Helene Bagge. 2014. Parsing in a Broad Sense. In *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014) (LNCS)*, Jürgen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran (Eds.), Vol. 8767. Springer, 50–67. https://doi.org/10.1007/978-3-319-11653-2_4