Two-Faced Data

Vadim Zaytsev, vadim@grammarware.net

Raincode, Belgium

Intent

One data fragment has several alternative structural representations tailored toward specific data manipulation approaches.

Also Known As

- Concrete Syntax and Abstract Syntax
  - simplifying concrete syntax to abstract syntax [20,58]
  - parsing [44], more than parsing [24,30], parsing in a broad sense [61]
  - object grammars [52]
- Interparadigmatic Data Binding
  - COBOL — OO — Relational databases — XML [35]
  - OO — Relational [13]
  - CRUD — OO [49]
- XML Data Binding
  - XML to Java [37]
  - XML to Haskell [6]
  - XML to C# [38]
- GUI Data Binding
  - generic GUIs [1]
  - WebSocket-based data binding [23]
- Intermediate Representation
  - support imperative and declarative idioms [34]
  - multiple languages within one paradigm: FP [27], OO [11]
  - implementation-geared [5,42]
  - validation-geared [9] and analysis-geared [22,31]
- Views
  - integrated personalised views in databases [48]
  - model views [3,4,10]
  - view-based software engineering [7,46]
  - architectural views [15,51,59]

Motivation

When modelling or programming, people tend to think in terms of conceptual constructs: "inheritance" (of classes), "conformance" (of models to metamodels), "conditional statement" (programming), "input" (data flow, side effects) and others. In practice these conceptual entities are represented as concrete elements: in textual form, in graphical diagrams, in memory blocks, etc. Since the actual solution has to be expressed in such elements, this notation exposed to the language end user, has great impact on the effectiveness of both the solution and the process of modelling or programming.

Results from ontological analysis tell us that a mapping between a modelling notation and an underlying domain model (in SE usually the Bunge-Wand-Weber ontology [57]) should be bijective [39,39,40] to avoid the following issues:

- ◇ **Construct deficit**: when something that exists in the ontology (i.e., in the mind of domain experts), has no counterpart in the modelling notation. Notations with construct deficit are called *incomplete* and have their place in environments that are deliberately limited for reasons of security or (sub)domain-specificity.
- ◇ **Construct redundancy**: when one conceptual entity can be modelled with more than one notational construct that are identical or subtly different from one another. Notations with construct overloaded are called *unclear* and are advocated by ontological analysts to be defective. Construct redundancy in programming languages often leads to discussions of taste and conventions being imposed on top of the language. For example, a functional language called Haskell [26] supports comprehensions and higher order functions equally well, so `map (\x->x*x) xs` is equally acceptable, equally performant and equally maintainable as `[x*x | x <- xs]`, and the choice is up to the particular programmer. Other functional languages like Rascal [29] have better support for comprehensions than for explicit mappings, so the choice there has farther going consequences, known only to programmers that reached certain affinity with the language at hand.
- ◇ **Construct overload**: when one notational entity represents several conceptual entities. Notations with this smell are a different kind of *unclear*: they are merely slightly counter-intuitive to domain experts but give wrong impressions to those who learn the domain through this notation. A famous example nowadays is the Git version control framework that bundles unrelated functionality: for instance, `git reset` is a command that, depending on parameters, can simply "unstage" code changes (which means they will not be included in the next commit) or undo several unpushed commits or even irrecoverably wipe any pending changes away.
- ◇ **Construct excess** is said to happen when a modelling notation have elements that do not have any correspondence in the domain model. Notations frequently have construct excesses as practical shortcuts and quick hacks that

solve the problem at hand but are totally alien to the uninvolved domain experts. Excessive constructs are never "designed" into a notation but find their way into it by the time of implementation, especially under deadline pressure.

Success stories from updatable views in databases [8], synchronised model views [4], data integration [43], serialisation [16] and structure editors [25] demonstrate how it can be useful to have several systematic representations of the same underlying constructs [14]. We argue that this pattern is universal to the entire software language engineering and thus can be used across technical spaces anywhere where a language has several user groups or application varieties.

Applicability

Use the Two-Faced Data pattern when

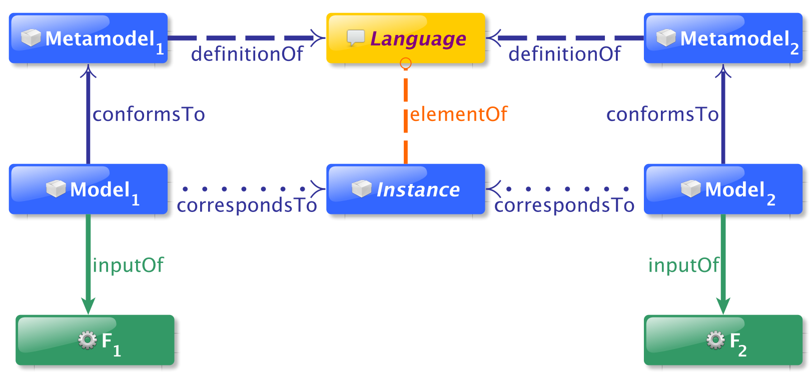
- ◇ You design a software language and must provide functionality in the entire spectrum from parsing the textual input to advanced semantic consistency validation like type checking.
  - ◇ If you make your grammar too close to the desired conceptual representation, you risk making it ambiguous, inefficient for parsing and/or not user friendly for the language users. Projectional language workbenches deliberately choose this path due to their naturally powerful IDE support [25,55,56], other approaches are filled with perils, unless they adopt the same techniques [53].
  - ◇ If you make it too close to the desired way of writing and reading sentences in the language, you risk overburdening your traversals and rewritings with unnecessary details concerning a particular textual representation. Solutions without multiple "faces" usually include conventions that allow to use one representation to mean multiple things at once [28, 54] (e.g., using layout for pretty-printing but ignoring it for parsing/matching).
- ◇ You want your software language to have both textual and visual concrete syntax which are conceptually the same but technically get a different representation each. Due to the "natural" flow of the textual representation (usually left to right, character by character) and a much freer structure of the visual syntax, elements that correspond to the same entities may not only be represented differently individually, but also appear in different order.
  - ◇ The need for several notations of one domain-specific language is widely known and acknowledged in practice [18, 36], but its foundations are lagging somewhat behind.
  - ◇ In general textual information is perceived by humans to be more trustworthy [50] and is faster digestible [45], but with appropriate training visual notations can be more effective and maintainable [41, 47].

- ◇ Graphical models of text that take physical distance between words into account [2] and consider the visual aspect of operations performed on text [12] are an emerging field of research without a readily available cookbook of practically mature recipes.

◇ Structured data that you are working with, needs to be serialised — for storage, communication or backup.

- ◇ Using the existing textual syntax would mean losing the structure and may imply future overhead and/or ambiguity in deserialising such data.
- ◇ In practice people tend to develop a yet another format which conceptually represents the same structure of the same data, but is more suitable for marshalling and unmarshalling. Such a format can be a standalone project but usually is a sublanguage of XML or JSON.

Structure



Participants and Collaborations

The same language (yellow box on the megamodel) can be defined by different, possibly incomplete, metamodels, and thus the models that conform to them, correspond to the same language instances, but belong to different technological stacks and thus can be effectively used with different algorithms. Functions F_k are used in a broad sense and can represent true functions like sorting or traversals, as well as other data manipulation activities such as editing or validation.

Implementation

Consider the following implementation issues:

◇ If the "faces" of the data allow interaction, you need some set of bidirectional update mappings; these imply overhead which might outweigh the advantages of using the faces.

◇ One of the "faces" can be dominant within a domain for historical reasons and so advanced that over the time it developed all necessary algorithms usually associated with other faces — e.g., concrete syntax in metaprogramming [54].

◇ Some mapping need to bridge a semantic gap between "faces" that cannot be fully bridged — e.g., ADT vs OO [13], even though many practically sufficing strategies exist [35].

◇ In scenarios with more than two "faces" it gets too complex to develop direct mappings for each pair; in that case it is better to consider a star-shaped infrastructure with one canonic representation which is capable of synchronising with any of the other ones.

◇ When metamodels are well-defined and their differences are explicitly expressed, we can do coupled transformations [32] — that is, infer model-level mappings from metamodel-level ones. This has been done for various technical spaces: modelware [21], grammarware [60], databases [19], xmlware [33].

Sample Code

Consider the following Rascal [29] code:

```
data A = foo(bool)
       | bar(set[A] xs)
       ;
```

It defines a piece of very simple abstract data type with two constructors. The metaprogramming facilities provided by Rascal allow us to comfortably traverse instances structured in such a way and perform computations:

```
visit(T)
{
    foo(True) : cx += 1
}
```

(even more concise, `len([1 | /foo(True) := T])`), and in place rewritings:

```
visit(T)
{
    bar(_) => foo(False)
}
```

However, writing them to a file can only be done in one fixed notation, and reading back will not be smooth. For such actions, we need concrete syntax — for example, this one:

```
syntax A = foo: "FOO"?
         | bar: "<" {A ":"}+ ">";
```

Parsing any textual input with this concrete syntax definition is trivial in Rascal with the use of `parse(#A, ...)` function. The resulting trees, however, are somewhat clunky, contain too much information (who cares that we used colons as a separator? should we really update the traversal code if the separator changes in the future?) and can only be traversed in their default term form. However, there is a built-in matching function called `implode` that can couple the two:

```
T = implode(A, parse(#A, input))
```

The `implode` function follows grammar production alternative labels and match them to the constructors of the data type. Then, it maps the presence of `FOO` text to a true value and the lack of it to a false value of the Boolean argument of the foo constructor. Parse-guiding anti-ambiguity angle brackets in the concrete syntax carry no structural meaning, so they are disregarded, and the collection of inner entries is mapped to a set because that is what the abstract data type expects (it could have been mapped to a list instead).

Related Patterns

Adapter; Bridge; Visitor; Interpreter [17].

References

1. P. Achten, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Compositional Model-Views with Generic Graphical User Interfaces. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, volume 3057 of *LNCS*, pages 39–55. Springer, 2004.

2. C. C. Aggarwal and P. Zhao. Graphical Models for Text: a New Paradigm for Text Representation and Processing. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 899–900. ACM, 2010.

3. A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr. Efficient Model Synchronization with View Triple Graph Grammars. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications*, volume 8569 of *LNCS*, pages 1–17. Springer, 2014.

4. M. Antkiewicz and K. Czarnecki. Design Space of Heterogeneous Synchronization. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Revised Papers of the Second International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 5235 of *LNCS*, pages 3–46. Springer, 2007.

5. Z. M. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Proceedings of the Fourth Conference on Functional Programming Languages and Computer Architecture*, pages 230–242. ACM Press, 1989.

6. F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema-Haskell Data Binding. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, volume 3057 of *LNCS*, pages 71–85. Springer, 2004.

7. C. Atkinson. Orthographic Software Modelling: A Novel Approach to View-Based Software Engineering. In *Proceedings of the Sixth European Conference on Modelling Foundations and Applications*, volume 6138 of *LNCS*, page 1. Springer, 2010.

8. F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM TODS*, 6(4):557–575, 1981.

9. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems. Volume I*, volume 1708 of *LNCS*, pages 307–327. Springer-Verlag, 1999.

10. R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, Views and Models of UML. In *The Unified Modeling Language — Technical Aspects and Applications*, pages 93–108. Physica-Verlag, 1997.

11. J. Chen and D. Tarditi. A Simple Typed Intermediate Language for Object-Oriented Languages. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd Symposium on Principles of Programming Languages*, pages 38–49. ACM, 2005.

12. S. Conversy. Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. In *Proceedings of the Fourth Symposium on New Ideas in Programming and Reflections on Software*, pages 201–212. ACM, 2014.

13. W. R. Cook. On Understanding Data Abstraction, Revisited. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 557–572. ACM, 2009.

14. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In R. F. Paige, editor, *Proceedings of the Second International Conference on Theory and Practice of Model Transformations*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

15. G. El-Boussaidi, A. B. Belle, S. Vaucher, and H. Mili. Reconstructing Architectural Views from Legacy Systems. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 345–354. IEEE Computer Society, 2012.

16. K. Fisher and R. Gruber. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In V. Sarkar and M. W. Hall, editors, *Proceedings of the 26th Conference on Programming Language Design and Implementation*, pages 295–304. ACM, 2005.

17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

18. M. Gogolla, L. Hamann, J. Xu, and J. Zhang. Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST)*, 41, 2011.

19. M. Gogolla and A. Lindow. Transforming Data Models with UML. In *Knowledge Transformation for the Semantic Web*, pages 18–33. IOS Press, 2003.

20. R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, 35(2):121–130, Feb. 1992.

21. T. Gîrba, J.-M. Favre, and S. Ducasse. Using Meta-Model Transformation to Model Software Evolution. *Electronic Notes in Theoretical Computer Science*, 137(3):57–64, 2005.

22. J. Hayes, W. G. Griswold, and S. Moskovics. Component Design of Retargetable Program Analysis Tools That Reuse Intermediate Representations. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 356–365. ACM, 2000.

23. M. Heinrich and M. Gaedke. Data Binding for Standard-based Web Applications. In S. Ossowski and P. Lecca, editors, *Proceedings of the 27th Symposium on Applied Computing*, pages 652–657. ACM, 2012.

24. A. Herranz and P. Nogueira. More Than Parsing. In F. J. L. Fraguas, editor, *V Jornadas Sobre Programación y Lenguajes, Conferencia Española de Informática (CEDI'05)*, pages 193–202. Thomson Paraninfo, 2005.

25. Z. Hu, S.-C. Mu, and M. Takeichi. A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations. *Higher-Order and Symbolic Computation*, 21(1–2):89–118, 2008.

26. P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2. *SIGPLAN Notices*, 27(5):1–164, May 1992.

27. S. L. P. Jones, M. Shields, J. Launchbury, and A. P. Tolmach. Bridging the Gulf: A Common Intermediate Language for ML and Haskell. In D. B. MacQueen and L. Cardelli, editors, *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1998.

28. L. C. L. Kats, K. T. Kalleberg, and E. Visser. Interactive Disambiguation of Meta Programs with Concrete Object Syntax. In B. A. Malloy, S. Staab, and M. van den Brand, editors, *Revised Selected Papers of the Third International Conference on Software Language Engineering*, volume 6563 of *LNCS*, pages 327–336. Springer, 2010.

29. P. Klint, T. van der Storm, and J. J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Revised Papers of the Third International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 6491 of *LNCS*, pages 222–289. Springer, 2009.

30. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit — System Demonstration. *Electronic Notes in Theoretical Computer Science, LDTA Special Issue*, 65(3):117–123, 2002.

31. R. Koschke and J.-F. Girard. An Intermediate Representation for Reverse Engineering Analyses. In *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 241–250. IEEE Computer Society, 1998.

32. R. Lämmel. Transformations Everywhere. *Science of Computer Programming*, 52:1–8, 2004.

33. R. Lämmel and W. Lohmann. Format Evolution. In *RETIS*, volume 155, pages 113–134. OCG, 2001.

34. R. Leißa, M. Köster, and S. Hack. A Graph-Based Higher-Order Intermediate Representation. In K. Olukotun, A. Smith, R. Hundt, and J. Mars, editors, *Proceedings of the 13th International Symposium on Code Generation and Optimization*, pages 202–212. IEEE Computer Society, 2015.

35. R. Lämmel and E. Meijer. Mappings Make Data Processing Go 'Round. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Revised Papers of the First International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 169–218. Springer, 2005.

36. S. Maro, J.-P. Steghöfer, A. Anjorin, M. Tichy, and L. Gelin. On Integrating Graphical and Textual Editors for a UML Profile Based Domain Specific Language: An Industrial Experience. In R. F. Paige, D. D. Ruscio, and M. Völter, editors, *Proceedings of the Eighth International Conference on Software Language Engineering*, pages 1–12. ACM, 2015.

37. B. McLaughlin. *Java and XML Data Binding. Nutshell handbook.* O'Reilly & Associates, 2002.

38. E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD Conference*, page 706. ACM, 2006.

39. D. L. Moody. The "Physics" of Notations: A Scientific Approach to Designing Visual Notations in Software Engineering. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd International Conference on Software Engineering*, volume 2, pages 485–486. ACM, 2010.

40. D. L. Moody and J. van Hillegersberg. Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UMLFamily of Diagrams. In D. Gašević, R. Lämmel, and E. V. Wyk, editors, *Revised Selected Papers of the First International Conference on Software Language Engineering*, volume 5452 of *LNCS*, pages 16–34. Springer, 2008.

41. B. Mora, F. García, F. Ruiz, and M. Piattini. Graphical Versus Textual Software Measurement Modelling: An Empirical Study. *Software Quality Journal*, 19(1):201–233, 2011.

42. P. A. Nelson. A Comparison of PASCAL Intermediate Languages. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, pages 208–213. ACM, 1979.

43. J. N. Oliveira. Transforming Data by Calculation. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Revised Papers of the Second International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 5235 of *LNCS*, pages 134–195. Springer, 2007.

44. T. Parr and K. Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd Conference on Programming Language Design and Implementation*, pages 425–436. ACM, 2011.

45. Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol, and Y.-G. Guéhéneuc. An Empirical Study on the Efficiency of Graphical versus Textual Representations in Requirements Comprehension. In *Proceedings of the 21st International Conference on Program Comprehension*, pages 33–42, 2013.

46. J. J. Shilling and P. F. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In G. Bosworth, editor, *Proceedings of the Fourth Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 353–361. ACM, 1989.

47. D. Stein and S. Hanenberg. Comparison of a Visual and a Textual Notation to Express Data Constraints in Aspect-Oriented Join Point Selections: A Controlled Experiment. In *Proceedings of the 19th International Conference on Program Comprehension*, pages 141–150. IEEE Computer Society, 2011.

48. D. E. Swartwout and J. P. Fry. Towards the Support of Integrated Views of Multiple Databases: An Aggregate Schema Facility. In E. I. Lowenthal and N. B. Dale, editors, *Proceedings of the Eighth ACM SIGMOD International Conference on Management of Data*, pages 132–143. ACM, 1978.

49. D. Thomas. The Impedance Imperative Tuples + Objects + Infosets = Too Much Stuff! *Journal of Object Technology*, 5(2):7–12, 2003.

50. C. L. Toma. Perceptions of Trustworthiness Online: The Role of Visual and Textual Information. In *Proceedings of the 13th ACM Conference on Computer Supported Cooperative Work*, pages 13–22. ACM, 2010.

51. Q. Tu and M. W. Godfrey. The Build-Time Software Architecture View. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 398–407. IEEE Computer Society, 2001.

52. T. van der Storm, W. R. Cook, and A. Loh. The Design and Implementation of Object Grammars. *Science of Computer Programming*, 96:460–487, 2014.

53. O. van Rest, G. Wachsmuth, J. R. H. Steel, J. G. Süß, and E. Visser. Robust Real-Time Synchronization between Textual and Graphical Editors. In K. Duddy and G. Kappel, editors, *Proceedings of the Sixth International Conference on Theory and Practice of Model Transformations*, volume 7909 of *LNCS*, pages 92–107. Springer, 2013.

54. E. Visser. Meta-programming with Concrete Object Syntax. In *Proceedings of the Second International Conference on Generative Programming and Component Engineering*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.

55. M. Völter. Embedded Software Development with Projectional Language Workbenches. In D. C. Petriu, N. Rouquette, and Øystein Haugen, editors, *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems, Part II*, volume 6395 of *LNCS*, pages 32–46. Springer, 2010.

56. M. Völter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, editors, *Proceedings of the Seventh International Conference on Software Language Engineering*, volume 8706 of *LNCS*, pages 41–61. Springer, 2014.

57. Y. Wand and R. A. Weber. An Ontological Model of an Information System. *IEEE TSE*, 16(11):1282–1292, 1990.

58. D. S. Wile. Abstract Syntax from Concrete Syntax. In W. R. Adrion, A. Fuggetta, R. N. Taylor, and A. I. Wasserman, editors, *Proceedings of the 19th International Conference on Software Engineering*, pages 472–480. ACM, 1997.

59. A. S. Yeh, D. R. Harris, and M. P. Chase. Manipulating Recovered Software Architecture Views. In W. R. Adrion, A. Fuggetta, R. N. Taylor, and A. I. Wasserman, editors, *Proceedings of the 19th International Conference on Software Engineering*, pages 184–194. ACM, 1997.

60. V. Zaytsev. Cotransforming Grammars with Shared Packed Parse Forests. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST)*, Graph Computation Models, 2016. In print.

61. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.