Case Studies in Bidirectionalisation

Research Paper Extended Abstract

Vadim Zaytsev

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

Abstract. Suppose we have two algebraic data types related in such a way that functions can be defined to map instances of one type to instances of the other to maintain consistency. Given the definitions of these types and these functions, how do we establish a reversible or at least a correct and hippocratic bidirectional mapping to synchronise the system of two instances, one of each type, to be able to propagate changes in any direction? Currently there are no methods yet that can answer these questions in a general and useful way. In this paper we study concrete cases of bidirectionalisation: one turning unidirectional functions into bijective ones by iteratively completing them with more information; and one implementing synchronisation strategies as traversals of the ADT instances.

1 Motivation

Suppose we have two algebraic data types related in some way so that functions can be defined to map entities of one type to entities of the other. In practical software engineering, ADTs can be defined as context-free grammars or attribute grammars, as classes or data models, as schemata or ontologies, etc, and the mappings themselves can be implemented in an imperative way with significant side effects, but we will try to abstract from these details. Given the definitions of these types and these functions, how do we establish a bidirectional mapping [1,9,10] to synchronise the system of two entities, one of each type, to be able to propagate changes in any direction? In particular,

- Given L, R and $f : L \to R$, how to change f to g such that $\forall x \in L, g(x) = f(x)$, but $\exists g^{-1} : R \to L$ such that $\forall x \in L, (g^{-1} \circ g)(x) = x$?
- Given L, R, $f : L \to R$ and $g : R \to L$, how to derive $\triangleright : L \times R \to R$ and $\triangleleft : L \times R \to L$ such that $\forall x \in L, \forall y \in R, x \triangleright f(x) = f(x)$ and $g(y) \triangleleft y = g(y)$, as well as $\forall x \in L, \forall y \in R, x \triangleleft (x \triangleright y) = x$ and $(x \triangleleft y) \triangleright y = y$?

Currently there are no methods yet that can answer these questions in a general and useful way. Hence, we study concrete cases of bidirectionalisation found in practical software engineering, in an endeavour to learn from them.

2 Convergence case study

2.1 Problem description

Complex transformation scenarios may often be understood as directed, acyclic graphs with nodes corresponding to models and edges corresponding to model transformations. One strong instance of this notion is grammar convergence [6] with the graph being a tree with grammars as nodes and grammar transformations as edges. For example, the convergence scenario of six Java grammars from several versions of the corresponding language specification was represented as a graph with six starting nodes, two final nodes (one for the "readable" converged target and one for the "implementable" one, according to Java language creators), 91 intermediate nodes and 70 transformation scripts (some used multiple times within the transformation graph) comprising in total 1611 transformation operator applications [7].

Rearranging such transformation graphs (e.g., adjusting the convergence tree) or changing their topology (e.g., deriving a coevolution scenario from a convergence tree) is difficult unless certain important graph refactorings are effectively supported, in particular inverting edges and propagating nodes through the tree. This objective becomes feasible, if relations between adjacent nodes are bidirectional.

2.2 Technical details

XBGF [6] is a language for specifying grammar transformation steps. In fact, it is a library of functions such as **define**(p^+) or **renameN**(n_1, n_2), which serve as parameterised operators. The transformation script in XBGF is a superposition of function calls to them. ΞBGF[1] [11] is a bidirectional counterpart of XBGF. It contains a bijective subset of operators, which are made to be reversible — removed are abstract algorithms like **distribute**(n) and shortcutting like **fold**(p)/**unfold**(n) (i.e., the folding operator in XBGF requires a production rule that needs to be folded, while the unfolding operator requires only the nonterminal name; in ΞBGF both need a production rule to work).

The complete operator set of ΞBGF follows, with its implementation[2] available in the open source repository of the Software Language Processing Suite [13]. In operator parameters, e is an expression, n is a nonterminal, s is a selector, t is a terminal, p is a production, p_m is a marked production.

- **abridge-detour**(p)
  $\rightarrow$ xbgf:abridge(p)
  $\leftarrow$ xbgf:detour(p)
- **abstractize-concretize**(p_m)
  $\rightarrow$ xbgf:abstractize(p_m)
  $\leftarrow$ xbgf:concretize(p_m)
- **add-removeH**(p_m)
  $\rightarrow$ xbgf:addH(p_m)
  $\leftarrow$ xbgf:removeH(p_m)
- **add-removeV**(p)
  $\rightarrow$ xbgf:addV(p)
  $\leftarrow$ xbgf:removeV(p)

[1] ΞBGF is pronounced "ksi bi gi ef".
[2] The implementation of ΞBGF uses a combination of Prolog and XSLT.

- **anonymize-deanonymize**(p_m)
  $\rightarrow$ xbgf:anonymize(p_m)
  $\leftarrow$ xbgf:deanonymize(p_m)
- **appear-disappear**(p_m)
  $\rightarrow$ xbgf:appear(p_m)
  $\leftarrow$ xbgf:disappear(p_m)
- **assoc-iterate**(p_1, p_2)
  $\rightarrow$ xbgf:lassoc(p_2)|rassoc(p_2)
  $\leftarrow$ xbgf:removeV$(p_2)\circ$addV(p_1)
- **chain-unchain**(p)
  $\rightarrow$ xbgf:chain(p)
  $\leftarrow$ xbgf:unchain(p)
- **clone-equate**(p^+, n, c^*)
  $\rightarrow$ xbgf:introduce$(p^+)\circ$replace$(n, p.n,$ in $c_i)$
  $\leftarrow$ xbgf:equate$(p.n, n)$
- **concretize-abstractize**(p_m)
  $\rightarrow$ xbgf:concretize(p_m)
  $\leftarrow$ xbgf:abstractize(p_m)
- **deanonymize-anonymize**(p_m)
  $\rightarrow$ xbgf:deanonymize(p_m)
  $\leftarrow$ xbgf:anonymize(p_m)
- **define-undefine**(p^+)
  $\rightarrow$ xbgf:define(p^+)
  $\leftarrow$ xbgf:undefine$(p.n)$
- **designate-unlabel**(p)
  $\rightarrow$ xbgf:designate(p)
  $\leftarrow$ xbgf:unlabel$(p.l)$
- **detour-abridge**(p)
  $\rightarrow$ xbgf:detour(p)
  $\leftarrow$ xbgf:abridge(p)
- **deyaccify-yaccify**(p^+)
  $\rightarrow$ xbgf:deyaccify$(p.n)$
  $\leftarrow$ xbgf:yaccify(p^+)
- **disappear-appear**(p_m)
  $\rightarrow$ xbgf:disappear(p_m)
  $\leftarrow$ xbgf:appear(p_m)
- **downgrade-upgrade**(p_1, p_2)
  $\rightarrow$ xbgf:downgrade(p_1, p_2)
  $\leftarrow$ xbgf:upgrade(p_1, p_2)
- **eliminate-introduce**(p^+)
  $\rightarrow$ xbgf:eliminate$(p.n)$
  $\leftarrow$ xbgf:introduce(p^+)
- **equate-clone**(p^+, n, c^*)
  $\rightarrow$ xbgf:equate$(p.n, n)$
  $\leftarrow$ xbgf:introduce$(p^+)\circ$replace$(n, p.n,$ in $c_i)$
- **extract-inline**(p)
  $\rightarrow$ xbgf:extract(p)
  $\leftarrow$ xbgf:inline$(p.n)$
- **factor-factor**(e_1, e_2)
  $\rightarrow$ xbgf:factor(e_1, e_2)
  $\leftarrow$ xbgf:factor(e_2, e_1)
- **fold-unfold**(n)
  $\rightarrow$ xbgf:fold(n)
  $\leftarrow$ xbgf:unfold(n)
- **horizontal-vertical**(n)
  $\rightarrow$ xbgf:horizontal(n)
  $\leftarrow$ xbgf:vertical(n)
- **inject-project**(p_m)
  $\rightarrow$ xbgf:inject(p_m)
  $\leftarrow$ xbgf:project(p_m)
- **inline-extract**(p)
  $\rightarrow$ xbgf:inline$(p.n)$
  $\leftarrow$ xbgf:extract(p)
- **introduce-eliminate**(p^+)
  $\rightarrow$ xbgf:introduce(p^+)
  $\leftarrow$ xbgf:eliminate$(p.n)$
- **iterate-assoc**(p_1, p_2)
  $\rightarrow$ xbgf:removeV$(p_1)\circ$addV(p_2)
  $\leftarrow$ xbgf:lassoc(p_1)|rassoc(p_1)
- **massage-massage**(e_1, e_2)
  $\rightarrow$ xbgf:massage(e_1, e_2)
  $\leftarrow$ xbgf:massage(e_2, e_1)
- **narrow-widen**(e_1, e_2)
  $\rightarrow$ xbgf:narrow(e_1, e_2)
  $\leftarrow$ xbgf:widen(e_2, e_1)
- **permute-permute**(p_1, p_2)
  $\rightarrow$ xbgf:permute(p_2)
  $\leftarrow$ xbgf:permute(p_1)
- **project-inject**(p_m)
  $\rightarrow$ xbgf:project(p_m)
  $\leftarrow$ xbgf:inject(p_m)
- **remove-addH**(p_m)
  $\rightarrow$ xbgf:removeH(p_m)
  $\leftarrow$ xbgf:addH(p_m)
- **remove-addV**(p)
  $\rightarrow$ xbgf:removeV(p)
  $\leftarrow$ xbgf:addV(p)
- **rename-renameN**(n_1, n_2)
  $\rightarrow$ xbgf:renameN(n_1, n_2)
  $\leftarrow$ xbgf:renameN(n_2, n_1)
- **rename-renameS**(s_1, s_2)
  $\rightarrow$ xbgf:renameS(s_1, s_2)
  $\leftarrow$ xbgf:renameS(s_2, s_1)
- **rename-renameT**(t_1, t_2)
  $\rightarrow$ xbgf:renameT(t_1, t_2)
  $\leftarrow$ xbgf:renameT(t_2, t_1)

- **replace-replace**(e_1, e_2)
  $\rightarrow$ xbgf:replace(e_1, e_2)
  $\leftarrow$ xbgf:replace(e_2, e_1)
- **reroot-reroot**(n_1^*, n_2^*)
  $\rightarrow$ xbgf:reroot(n_2^*)
  $\leftarrow$ xbgf:reroot(n_1^*)
- **split-unite**(p^+, p'^+, c^*)
  $\rightarrow$ xbgf:introduce(p^+)
  $\circ$ removeV$(p_i|_{p.n:=p'.n})$
  $\circ$ replace$(p'.n, p.n, \text{in } c_i)$
  $\leftarrow$ xbgf:unite$(p.n, p'.n)$
- **unchain-chain**(p)
  $\rightarrow$ xbgf:unchain(p)
  $\leftarrow$ xbgf:chain(p)
- **undefine-define**(p^+)
  $\rightarrow$ xbgf:undefine$(p.n)$
  $\leftarrow$ xbgf:define(p^+)
- **unfold-fold**(n)
  $\rightarrow$ xbgf:unfold(n)
  $\leftarrow$ xbgf:fold(n)
- **unite-split**(p^+, p'^+, c^*)
  $\rightarrow$ xbgf:unite$(p.n, p'.n)$
  $\leftarrow$ xbgf:introduce(p^+)
  $\circ$ removeV$(p_i|_{p.n:=p'.n})$
  $\circ$ replace$(p'.n, p.n, \text{in } c_i)$
- **unlabel-designate**(p)
  $\rightarrow$ xbgf:unlabel$(p.l)$
  $\leftarrow$ xbgf:designate(p)
- **upgrade-downgrade**(p_1, p_2)
  $\rightarrow$ xbgf:upgrade(p_1, p_2)
  $\leftarrow$ xbgf:downgrade(p_1, p_2)
- **vertical-horizontal**(n)
  $\rightarrow$ xbgf:vertical(n)
  $\leftarrow$ xbgf:horizontal(n)
- **widen-narrow**(e_1, e_2)
  $\rightarrow$ xbgf:widen(e_1, e_2)
  $\leftarrow$ xbgf:narrow(e_2, e_1)
- **yaccify-deyaccify**(p^+)
  $\rightarrow$ xbgf:yaccify(p^+)
  $\leftarrow$ xbgf:deyaccify$(p.n)$

2.3 Migration to bidirectionality

Migrating most XBGF transformation steps to their bidirectional counterparts is rather straightforward, but in general case, impossible to automate due to the following cases:

- **xbgf:deyaccify**$(n) \rightarrow$ **ξbgf:deyaccify-yaccify**$(?)$
- **xbgf:distribute**$(n) \rightarrow$ **ξbgf:factor-factor**$(?,?)$
- **xbgf:eliminate**$(n) \rightarrow$ **ξbgf:eliminate-introduce**$(?)$
- **xbgf:equate**$(n_1, n_2) \rightarrow$ **ξbgf:equate-clone**$(?, n_2, ?)$
- **xbgf:inline**$(n) \rightarrow$ **ξbgf:inline-extract**$(?)$
- **xbgf:permute**$(p) \rightarrow$ **ξbgf:permute-permute**$(?, p)$
- **xbgf:redefine**$(p^+) \rightarrow$ **ξbgf:undefine-define**$(?)\circ$**define-undefine**(p^+)
- **xbgf:reroot**$(r^*) \rightarrow$ **ξbgf:reroot-reroot**$(?, r^*)$
- **xbgf:undefine**$(n) \rightarrow$ **ξbgf:undefine-define**$(?)$
- **xbgf:unlabel**$(n) \rightarrow$ **ξbgf:unlabel-designate**$(?)$

In order to obtain missing information, in our solution we execute all transformation steps up to the problematic point infer the remaining bits from the intermediate grammar. To demonstrate this process, we have developed a prototype tool called `xbgf2ξbgf`, which is freely available via SLPS mentioned above [13].

When executed on the FL case, the original case study of grammar convergence [6] with a tree consisting of 22 XBGF files, our tool migrates 14 of them by straightforward mapping and infers additional details in remaining 8 cases.

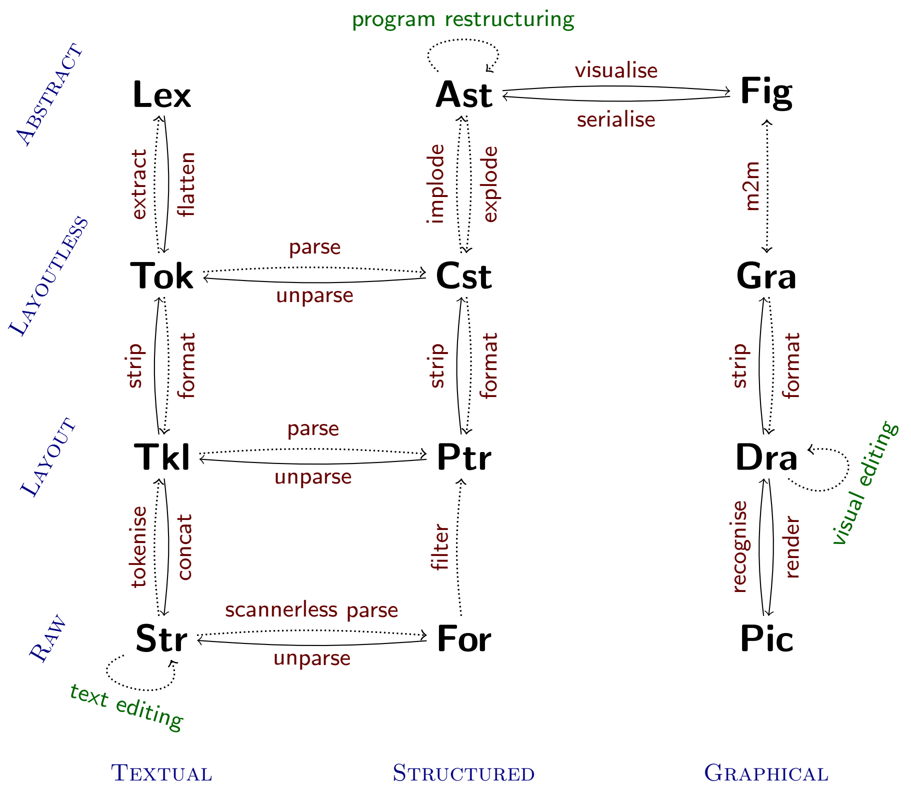

Fig. 1. Megamodel of various kinds of parsing and unparsing. Dotted lines denote mappings that rely on either lexical or syntactic definitions; solid lines denote universally defined mappings [12]; loops are examples of transformations we consider in this paper.

3 (Un)parsing case study

3.1 Problem description

Different approaches and phases of software language processing feature different kinds of artefacts, which can be considered to fit into one of twelve categories, depicted on Figure 1 [12]. Consider three of them:

Str — a purely textual flat string-like representation of a program, easy to edit, transfer and maintain and familiar to what mainstream programmers are used to for the last six decades. There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different entity (such as a parse tree).

Ast — an abstract syntax tree, a conceptual representation of a program which is the most suitable for automated program analysis and assigning semantics. It lacks certain details specific to **Str** such as line numbers and indentation of language expressions, but it encapsulates the structure of the program very well and can contain computed annotations.

Dra — a drawing of a program in some graphical notation, a visualisation found useful for domain experts, business analysts, process modellers, software auditors or any other kinds of language users more comfortable with using graphical notation instead of a textual one. Such a drawing contains lots of extra information about interface elements, their coordinates, colours, line styles, etc, but can leave some entities unnamed or having no direct correspondence to language constructions.

3.2 Technical details

As we can see, each of the three artefacts has some information that the other two lack: **Str** has indentation, **Ast** has names and annotations, **Gra** has visualisation details. This information is of local significance and thus prevents achieving bijection. Ideally, we would like to preserve this locally important information through coevolution updates. A typical non-BX-aware toolchain would include paths such as **Str** $\rightarrow$ **For** $\rightarrow$ **Ptr** $\rightarrow$ **Cst** $\rightarrow$ **Ast** $\rightarrow$ **Fig** $\rightarrow$ **Gra** $\rightarrow$ **Dra** and **Dra** $\rightarrow$ **Gra** $\rightarrow$ **Fig** $\rightarrow$ **Ast** $\rightarrow$ **Cst** $\rightarrow$ **Ptr** $\rightarrow$ **Tkl** $\rightarrow$ **Str**. This, due to the lack of bidirectionality, loses locally important information on every update.

One of the popular approaches, first proposed by Foster et al [3], is to use a *putback* function. For example, we could have a *get* function to obtain **Ast** from **Str**, but its counterpart would be *putback*: **Ast** $\times$ **Str** $\rightarrow$ **Str**, which would put the changes done by an automated refactoring tool back into the text of the program, respecting the original indentation, coding standards and comments. That way, a refactoring tool can masquerade as acting on **Str** while in fact it is only *lowered* to **Str** from **Ast**. Conversely, some edits on **Dra** are purely local (e.g., repositioning an element), while some need to be *lifted* to **Fig** (e.g., removing an element). In situations when both ends of a relation are of equal importance, we can use maintainers [8,10] instead of lenses — this means essentially having two *putback* functions instead of a *get*/*putback* pair.

3.3 Migration to bidirectionality

Suppose that we have all twelve definitions of algebraic data types, and enough unidirectional mappings between them, already defined. Then, the implementation of each semi-maintainer (each "putback") is a function that takes two parameters: the old instance that needs to be updated and the updated instance of a different type. The process is two-phase: the first step is to convert the updated instance to the target type, and the second step is to traverse both the old and the updated entities which now have the same type. The result of this traversal is a composite instance that has all the shared information from the updated instance and as much local information as possible from the old instance.

Str $\triangleright$ **Ast** discards those parts of the abstract syntax tree that do not correspond to any fragments of the updated code, and recalculates all inferred annotations for the fragments that are new. A truly efficient implementation of it would feature iterative parsing, for which an old instance of **Ptr** would need to be stored as well.

Ast $\triangleright$ **Dra** displays the abstract model of a program in such a way that all already recognised elements are placed at their old positions, and the rest are rendered by default.

Dra $\triangleright$ **Ast** checks if any parts of a model have been added or removed, recalculates annotations for added ones and disregards the parts of the **Ast** related to the removed ones.

Ast $\triangleright$ **Str** unparses the abstract syntax tree by preserving the indentation of all recognised fragments of the old code, and pretty-printing the rest.

The code has around 3000 line of documented code in Rascal [4], a functional language for program analysis and transformation. It is released as open source and is publicly available from a dedicated repository:
http://github.com/grammarware/bx-parsing.

4 Conclusion

Two practical cases of bidirectionalisation have been described in this abstract. In §2, a convergence graph with nodes-grammars and edges-transformations was bidirectionalised by iteratively adding more information to the transformations until the mapping became bijective. In §3, bidirectional maintainers were implemented as traversals of ADT instances operating on the result of the superposition of the existing unidirectional mappings and thus achieving preservation of locally significant information while updating the changed fragments. The results are of practical and engineering nature, but their generalisation may be useful for systematic development of new methods of automated and semi-automated bidirectionalisation in the future.

The source code of all discussed prototypes is released as open source through two repositories referenced above. It mostly consists of documented Prolog [5] and Rascal [4] code.

One of the open questions left unanswered is dealing with sustainers [12] instead of maintainers: $\blacktriangleright: L \times R \rightarrow L \times R$ instead of $\triangleright$ and $\blacktriangleleft: L \times R \rightarrow L \times R$ instead of $\triangleleft$. The sustainers are much more interesting because they can model error-correcting strategies. However, it is yet unclear how to prove termination of the update strategy in the case of more than two instances (like in §3), which is required for the general case of model synchronisation [2].

References

1. K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.

2. Z. Diskin. Algebraic Models for Bidirectional Model Synchronization. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MoDELS'08*, volume 5301 of *LNCS*, pages 21–36. Springer, 2008.
3. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM TOPLAS*, 29, May 2007.
4. P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM*, pages 168–177. IEEE Computer Society, 2009.
5. R. Lämmel and G. Riedewald. Prological Language Processing. In M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, Apr. 2001.
6. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In M. Leuschel and H. Wehrheim, editors, *Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009)*, volume 5423 of *LNCS*, pages 246–260, Berlin, Heidelberg, Feb. 2009. Springer-Verlag.
7. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ); Section on Source Code Analysis and Manipulation*, 19(2):333–378, Mar. 2011.
8. L. Meertens. Designing Constraint Maintainers for User Interaction. Manuscript, June 1998.
9. P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *MoDELS, LNCS 4735*, pages 1–15. Springer, 2007.
10. P. Stevens. A Landscape of Bidirectional Model Transformations. In *GTTSE'07, LNCS 5235*, pages 408–424. Springer, 2008.
11. V. Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST); Bidirectional Transformations*, 49, 2012.
12. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. Submitted to the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014). Pending reviews, Mar. 2014.
13. V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, R. Hahn, and G. Wachsmuth. Software Language Processing Suite[3], 2008–2014. http://slps.github.io.

[3] The authors are given according to the list at http://github.com/grammarware/slps/graphs/contributors.