Taxonomy of Flexible Linguistic Commitments

Vadim Zaytsev

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

Abstract. Beside strict linguistic commitments of models to metamodels, documents to schemata, programs to grammars and activities to protocols, we often require or crave more flexible commitments. Some of such types of flexible commitments are well-understood and even formalised, others are considered harmful. In this paper, we make an attempt to classify these types from the language theory point of view, provide concrete examples from software language engineering for each of them and estimate usefulness and possible harm.

1 Introduction

In software language engineering people often speak of linguistic commitment, structural commitment or commitment to grammatical structure [4]. Examples include:

- well-formed schema-less XML documents that commit to containing tags with allowed names and in proper combinations;
- XML documents that commit to the structure explicitly specified in the DTD or XSD;
- models that conform to their metamodel;
- programs in high level programming languages that commit to the structure specified by a grammar of such language and by other constraints of the designated compiler;
- programs that rely on some library and have to commit to calling its functions in proper order and with properly typed and ordered arguments;
- communicating entities that commit to sending and receiving messages according to the chosen protocol and must respect it in order to achieve compatibility and interoperability.

Any form of flexibility in such commitments is either not present or not considered. In this paper we propose to model it explicitly and classify its forms into several categories, varying in usefulness and the ability to lead to robust systems or to catastrophes.

The paper is organised as follows: §2 state the problem more clearly and formally. Based on that, §3 introduces three kinds of flexible language variations. Then, §4 classifies and explains all flexibly committing mappings within the framework. §5 proposes five kinds of streamlining helper mappings around the same language. §6 gives preliminary outlook at higher order manipulations such as composition of mappings,extension/restriction of them and constructing inverse functions. §7 concludes the paper.

2 Problem Statement

Consider a mapping $f : L_1 \to L_2$ which domain is a software language L_1 and which codomain is a software language L_2 (they can be the same in many cases, but let us look at the general case). Following the principle of least surprise, we could assume that f is surjective and total (i.e., its image fully coincides with its codomain and its preimage with its domain) so that it maps every element of L_1 to some element of L_2 and that every element of L_2 is reachable. By *flexible linguistic commitment* we will understand a situation when this expectation is violated.

3 Variations in Software Languages

Consider three variants based on a language L:

- ⋄ By $\check{L}$ we will denote a language that is strictly smaller than L: it has more constraints and less elements. Committing to a subset of a language is in general not that harmful and means limited applicability of the tool or a mapping.
- ⋄ By $\hat{L}$ we will denote a similarly strict superset of L which has then more elements and less constraints. Committing to a bigger language than intended is considered good practice in some areas that value robustness highly [7]. However, here we consider only "accidental" violations that extend the intended language in a way that preserves the semantics of the language instances up to heuristics that make sense for the problem domain.
- ⋄ By $\tilde{L}$ we will denote another superset of L that refers to semantic changes — instances from $\tilde{L}$ are not just outside L, but they also allow interpretations that are incorrect according to the semantics of L. The existence of $\tilde{L}$ is the main cause for critique on robustness guidelines [1].

In order to stay universally applicable, we consider all definitions to be specific to the problem domain of the software language (without loss of generality, so called general purpose programming and modelling languages are assumed to belong to problem domains of Turing-complete algorithms and the everything-is-a-model paradigm respectfully). For example, unclosed tags are indicators of $\widetilde{HTML}$ since the HTML language is meant to be processed in a soft and extremely flexible way, yet they are indicators of $\widetilde{XML}$ in most cases except for the absolutely trivial ones — say, one could argue that an XML document which is well-formed except one place with the construction like `<a><b></a>` is in $\widetilde{XML}$ since it can be treated as `<a><b/></a>`. However, even `<a><b>c</a>` is already in $\widetilde{XML}$ since it has two intuitively good resolutions: `<a><b/>c</a>` and `<a><b>c</b></a>`. Yet, in a subdomain of XML that deals with exclusively empty or complex elements but never with cdata or mixed content, `<a><b>c</a>` would also belong to $\widetilde{XML}'$.

mapping to	mapping from $\check{L}_1 \to \ldots$	$L_1 \to \ldots$	$\hat{L}_1 \to \ldots$	$\tilde{L}_1 \to \ldots$
$\ldots \to \check{L}_2$	correct program wrong language	non-surjectivity conservativeness	robustness	overrobustness
$\ldots \to L_2$	partial applicability	perfect assumed	fault recovery	overrecovery
$\ldots \to \hat{L}_2$	antirobustness	liberality	fault tolerance	overtolerance semirecovery
$\ldots \to \tilde{L}_2$	fault introduction		shotgun effect	linguistic ignorance

Table 1. Forms of linguistic commitments of mappings with a domain L_1 and codomain L_2, classified by their preimages $\check{L}_1 \subset L_1 \subset \hat{L}_1$ and images $\check{L}_2 \subset L_2 \subset \hat{L}_2$. Additionally, $\tilde{L}$ is a special case where $\tilde{L}_i \supset L_i$, but the difference $\tilde{L}_1 \setminus L_1$ is unanticipated and $\tilde{L}_2 \setminus L_2$ is unintended.

4 Variations in First Order Mappings

Table 1 presents all possible preimage/image combinations. Let us consider each of them in turn more closely.

- $f : \check{L}_1 \to \check{L}_2$ (correct program, wrong language)
  Some mappings that look like having flexible linguistic commitments, are in fact (possibly) correct mappings working on a different language than expected. This kind of "flexibility" is easily fixable by correcting the specification.
- $f : \check{L}_1 \to L_2$ (partial mapping, limited applicability)
  Theoretically this scenario corresponds to the notion of partial function. In practice it can be well disguised in a wrapper that extends f to $L_1 \setminus \check{L}_1$ to behave like an identity function. If this is not done, this kind of flexibility is not flexible at all: it actually means that in order to use f, one needs to normalise its inputs in a more strict way than documented. In certain cases it manifests itself as a works-on-my-machine antipattern when such undocumented constraints concern configuration management and deployment details.
- $f : \check{L}_1 \to \hat{L}_2$ (antirobustness, ungratefulness)
  If robustness (see below) is to expect less and provide more, then antirobustness is its exact opposite: the demands on the input in this case are higher than officially specified, and even when they are met, the output is ungratefully relaxed.
- $f : \check{L}_1 \to \tilde{L}_2$ (fault introduction)
  If antirobust mappings can damage syntactic conformity, this variant can also do semantic damage. Such a tool is flexible beyond reasonable, it is inherently faulty: while holding surprisingly high expectations on its inputs, it generates outputs that are outright wrong.
- $f : L_1 \to \check{L}_2$ (non-surjective mapping, conservativeness)
  Concervative mappings that transform a language instance into an instance

with an even stronger linguistic commitment, are not surjective. For cases of $L_1 = L_2$ they are called language reductions and represent a form of traditional normalisers without extra tolerance for input violations. Most code generators are conservative: they cover the entire input language but there exist possible target language programs that will never be generated. Classic canonical normal forms in grammarware and databases are also this kind of conservative mappings: they are proven to exist for all inputs and infer standardised provably equivalent counterparts of them. Formally, this is one of the best kinds of flexibility.

- $f : L_1 \to L_2$ (assumed / perfect)
  We list this form of commitment for the sake of completeness, but in fact it represents no flexibility: this is the standard commitment to grammatical structure [4], more or less precisely defined and precisely obeyed.
- $f : L_1 \to \hat{L}_2$ (liberality)
  In mathematics, a function is under no circumstances allowed to return a value outside its codomain, but from our point of view this variant is a conceptual sibling of the conservativeness discussed above which corresponds to the lack of surjectivity. For the case of $L_1 = L_2$ this is called language extension: and it may be intentional — consider a situation of a coupled transformation sequence representing language evolution (if the language is backwards compatible, it will only contain language preserving and extending operators) or some kind of refinement/enrichment plan that recognises patterns of language use and transforms them into constructs of some more sophisticated language.
- $f : L_1 \to \tilde{L}_2$ (fault introduction)
  What classifies mappings of this category is the lack of anticipation: it was probably meant to be a $\check{L}_1 \to \check{L}_2$ mapping, and as it turns out, it does not work correctly on unanticipated instances from $L \setminus \check{L}$. One of the typical examples is refactoring engines that claim to cover some programming language but in fact work correctly only on its subset while yielding irregular results whenever certain concurrency or subtyping patterns are involved [3].
- $f : \hat{L}_1 \to \check{L}_2$ (robustness)
  This is precisely the model of the "be conservative in what you do, be liberal in what you expect from others", also known as Postel's Robustness Principle [7], or at least its harmless implementations. The input language is extended to a safe superset of L_1, but the output language is limited to a subset of L_2. Many normalisers work this way, accepting mostly valid entities of a language and mapping them onto a strict subset of the same language. For example, BibSLEIGH [15], a project involving a large JSON database, has a normaliser that traverses all JSON files, including those that have trailing commas in lists or dictionaries, as well as those with naïve quoting, and transforms each of them into an LRJ file (Lexically Reliable JSON) which is basically valid JSON with extra guarantees of one key-value pair per line and lines being sorted lexicographically.
- $f : \hat{L}_1 \to L_2$ (recovery)
  A mapping that accepts slightly more than obligatory yet remains true to its

output language, represents error recovery: in the simplest case of $L_1 = L_2$ this may be the only purpose of the mapping, but it does not have to be. For example, consider notation-parametric grammar recovery, a technique that takes a human-written error-containing language document and a definition of the format it is supposed to have, and yields a well-formed grammar extracted from it [12]. Most of its heuristics are such mappings. Approximate pattern matching [6] and semiparsing [13] also belong to the same group, as do screen-scraping libraries like Beautiful Soup [8].

- $f : \hat{L}_1 \to \hat{L}_2$ (tolerance)
  In the terminology of negotiated mappings, the previous kind of flexible commitment represented adaptation through adjustment; this one is adaptation through tolerance [14]. Such a mapping still needs to cope with the unexpected outputs, but has an option of propagating the unexpected parts further down the pipeline instead of fixing them.
- $f : \hat{L}_1 \to \tilde{L}_2$ (shotgun effect)
  This kind of mapping has been identified in the technological space of internet security and named "shotgun parsing" there [2]. A shotgun parser is a program that is meant to check its input for linguistic commitment and sanitise it, yet in the interest of performance optimisations does not perform a thorough check and limits itself to fundamentally flawed approaches such as regular means of treating context-free languages or using plain string substitution for escaping; as a result, the system becomes vulnerable to malevolent input (comment bombs, SQL injections, etc). Every shotgun parser in the pipeline increases the span of possible treatments of data, hence the shotgun metaphor.
- $f : \check{L}_1 \to \check{L}_2$ (overrobustness)
  A few paragraphs above we reintroduced robustness as input type contravariance and output type covariance. Overrobustness does the same but crosses the syntax-semantics border and hence becomes dangerously ambiguous, nondeterministic and in general error-prone. Many examples of overrobustness leading to security bugs were given by Meredith Patterson et al. at `http://langsec.org` [2,9].
- $f : \check{L}_1 \to L_2$ (overrecovery)
  Overrecovery is a process of applying heuristic-based fixes to semantically incorrect language instances with the goal to return to the intended output language. Some aggressive heuristics like parenthesis matching from notation-parametric grammar recovery [12] fall into this category, as well as desperate matches in semiparsing [13].
- $f : \tilde{L}_1 \to \hat{L}_2$ (overtolerance, semirecovery)
  Overtolerance is a form of harmful error handling when semantic errors in the input are presumably fixed, but the output is still not always guaranteed to be syntactically correct.
- $f : \tilde{L}_1 \to \tilde{L}_2$ (ignorance)
  The ultimate form of flexible linguistic commitment is complete linguistic ignorance: our inputs can be broken beyond any hope of unambiguous repair, and the outputs are not much better in that respect. All lexical analysis

	$L \to \ldots$	$\hat{L} \to \ldots$	$\tilde{L} \to \ldots$
$\ldots \to \check{L}$	$\backsimeq$ *canoniser*	$\hat{\backsimeq}$ *normaliser*	$\tilde{\backsimeq}$ *regulator*
$\ldots \to L$	id	$\hat{\simeq}$ *codifier*	$\tilde{\simeq}$ *calibrator*

Table 2. Symbols and names for streamlining mappings.

methods belong to this category and are still being often use due to their extreme speed, ease of development and virtually unlimited flexibility, despite limited expressiveness and abundant false positives and negatives. Some migration and analysis projects of considerable size have been reported being completed with language ignorant lexical methods [5,10].

5 Streamlining Mappings

Before we try to compose flexibly committed mappings and consider higher order combinators, we need to introduce a special subcategory of streamlining mappings that help us "get back" to L or even $\check{L}$ whenever we deviate. In particular, we have a need for the following five (summarised on Table 2:

- *Canoniser* ($\backsimeq : L \to \check{L}$) has a normal (precise) commitment to L and produces a strict subset of it. Typically this is some kind of canonical normal form that makes sense for the chosen technological space; if it is strictly canonical, there is usually a proof of its uniqueness up to L.
- *Codifier* ($\hat{\simeq} : \hat{L} \to L$) is flexible with its input because it applies certain rules for recovery which are applied until the output conforms to all expectations of L.
- *Normaliser* ($\hat{\backsimeq} : \hat{L} \to \check{L}$) implements a classic Postel-style normalisation scheme that we have briefly discussed before: it is liberal with respect to its input and conservative with respect to its output. In our formalisation it also means that $\hat{\backsimeq} \equiv \backsimeq \circ \hat{\simeq}$ (a normaliser is equivalent to the superposition of a codifier and a canoniser), and indeed, any normaliser can be conceptually studied as a composition of two parts implementing the liberality and the conservativeness accordingly. We will consider superposition of flexibly committing functions in the next section in more detail.
- *Calibrators* ($\tilde{\simeq} : \tilde{L} \to L$) and *regulators* ($\tilde{\backsimeq} : \tilde{L} \to \check{L}$) can be decomposed similarly in various ways, but the key when considering them is remembering that the main distinction between $\hat{L}$ and $\tilde{L}$ is that $\hat{L} \setminus L$ is anticipated and $\tilde{L} \setminus L$ is not. Hence, when something is not anticipated, it cannot be casually accounted for by an automated streamliner. In practice developing calibrators and regulators requires the same steps as developing codifiers and normalisers, with additional effort dedicated to testing and documenting recovery heuristics, which are usually unreliable.

$f \circledcirc g$		g, applied first			
		$\ldots \to \check{L}$	$\ldots \to L$	$\ldots \to \hat{L}$	$\ldots \to \tilde{L}$
f, applied second	$\check{L}' \to \ldots$	$f \circ g$, if $\check{L} \subseteq \check{L}'$ $f \circ \eqsim \circ g$	$f \circ \eqsim \circ g$	$f \circ \approx \circ g$	$f \circ \approx \circ g$
	$L \to \ldots$	$f \circ g$	$f \circ g$	$f \circ \simeq \circ g$	$f \circ \simeq \circ g$
	$\hat{L}' \to \ldots$	$f \circ g$	$f \circ g$	$f \circ g$, if $\hat{L} \subseteq \hat{L}'$ $f \circ \simeq \circ g$	$f \circ \simeq \circ g$
	$\tilde{L}' \to \ldots$	$f \circ g$	$f \circ g$	$f \circ \simeq \circ g$	$f \circ \simeq \circ g$

Table 3. Superposition of two mappings with flexible commitments ($f \circledcirc g$) expressed as normal superposition of f, g and possibly some streamliners. Dashes simply denote non-uniqueness: for example, $\hat{L}$ may be a *different* superset of L than $\hat{L}'$.

6 Manipulating Flexible Mappings

6.1 Extension and Restriction

For every possible combination of input and output types (strictly speaking, for any input type since output type is irrelevant), a restriction on L is defined straightforwardly as a function which returns whatever the original function would have returned, for all inputs from the restricted set, and is underfined otherwise. An extension of a function deliberately committing to a subset of the intended language, on the other hand, cannot be defined in a general fashion. Hence, for any f which preimage is L, $\hat{L}$ or $\tilde{L}$, we get a $f|_L$ for free, but the extension $f|_L$ for $f : \check{L} \to L'$ is, generally speaking, unknown.

A more interesting observation is that restriction can affect the flexibility of the linguistic commitment to the output language as well: for example, if $f : \hat{L}_1 \to L_2$, then $f|_{L_1} : L \to L_2$ or possibly $f|_{L_1} : L \to \check{L}_2$. This has bad consequences on calculating inverses of restrictions.

6.2 Inverse Functions

Due to the nature of our formalisation that considers preimages and images instead of domains and codomains, all functions that map between L_1, $\check{L}_1$, $\hat{L}_1$ and L_2, $\check{L}_2$, $\hat{L}_2$, have straightforward inverse mappings. Their non-unique existence is guaranteed (the proof is a direct consequence of the definitions of a premiga and an image). For example, for any $f : \check{L}_1 \to \hat{L}_2$ there is at least one $f^{-1} : \hat{L}_2 \to \check{L}_1$. Inverses of restrictions can be more desirable, but less reliable, since there is no guarantee that for an arbitrary $f : \hat{L}_1 \to \hat{L}_2$, the image of $f|_{L_1}(L_1) \cap L_2 \neq \varnothing$.

6.3 Composition

Table 3 summarises the resulting superpositions of two flexible mappings with respect to all possible kinds of mappings. The most obvious case is the following:

$$\circledcirc : (L_2 \to L_3) \to (L_1 \to L_2) \to (L_1 \to L_3)$$

However, the following compositions are also universally safe (variations of L_2 correspond to variations of L in the table):

$$\circledcirc : (L_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\hat{L}_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\tilde{L}_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\hat{L}_2 \to L_3) \to (L_1 \to L_2) \to (L_1 \to L_3)$$
$$\circledcirc : (\tilde{L}_2 \to L_3) \to (L_1 \to L_2) \to (L_1 \to L_3)$$

The result type of such compositions given above is a possible overapproximation, because technically we combine g with $f|_{L_2}$ or $f|_{\check{L}_2}$, not with f itself, so it is possible for $f \circledcirc g$ with the codomain L_3 to have image $\check{L}_3$.

The rest often require streamliners, with two special cases stemming from the fact that in our notation $\hat{L}$ expresses *any* superset of L and not a particular one. Thus, it can be the case that $\check{L}_2 \subseteq \check{L}'_2$ and then $\circledcirc : (\check{L}'_2 \to L_3) \to (L_1 \to \check{L}_2) \to (L_1 \to L_3)$ is expressible with a simple superposition: $f \circledcirc g = f \circ g$, but otherwise we need an $\check{L}'_2$-canoniser to glue them together: $f \circledcirc g = f \circ \overline{\check{}} \circ g$. Similarly to the safe case discussed above, the image of such composition can be $\check{L}_3$ if $\check{L}_2 \neq \check{L}'_2$.

7 Conclusion

This paper is an attempt to investigate Postel's Robustness Principle ("be conservative in what you do, be liberal in what you expect from others") [7,1], in particular to follow the Postel's Principle Patch ("be definite about what you accept") [9] and the formal language theoretical approach to computer (in)security, in the general software language engineering view (not limited to internet protocols and even data languages). The result was a taxonomy of several families of language mappings depending on the inclusion relations between their domains and preimages, as well as codomains and images. The taxonomy (Table 1) contains two forms of precise commitment and several forms of harmful and profitable flexible commitments. Streamlining mappings (Table 2), mapping superpositions (Table 3) and other details of manipulating mappings with such flexible commitments, have also been considered. This classification is a refinement with respect to any previously existing approach, and provides means to identify safe combinations of mappings and streamliners. Formal treatment of tolerance [11] remains future work, and in general a categorical approach should provide even more solid foundation than a set theoretical one, which can also be refined further to yield interesting and useful properties.

References

1. E. Allman. The Robustness Principle Reconsidered. *Communications of the ACM*, 54(8):40–45, Aug. 2011.
2. S. Bratus and M. L. Patterson. Shotgun Parsers in the Cross-hairs. In *Brucon*, 2012. http://langsec.org.
3. M. Gouseti. A General Framework for Concurrency Aware Refactorings. Master's thesis, UvA, Mar. 2015. http://dare.uva.nl/en/scriptie/502196.
4. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM ToSEM*, 14(3):331–380, 2005.
5. S. Klusener, R. Lämmel, and C. Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.
6. V. Laurikari. TRE. http://github.com/laurikari/tre, 2005.
7. J. Postel. DoD Standard Internet Protocol. RFC 0760, 1980.
8. L. Richardson. Beautiful Soup. http://www.crummy.com/software/BeautifulSoup, 2012.
9. L. Sassaman, M. L. Patterson, and S. Bratus. A Patch for Postel's Robustness Principle. *IEEE Security and Privacy*, 10(2):87–91, Mar. 2012.
10. D. Spinellis. Differential Debugging. *IEEE Software*, 30(5):19–21, 2013.
11. P. Stevens. Bidirectionally Tolerating Inconsistency: Partial Transformations. In *FASE*, volume 8411 of *LNCS*, pages 32–46. Springer, 2014.
12. V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *LDTA*. ACM DL, June 2012.
13. V. Zaytsev. Formal Foundations for Semi-parsing. In S. Demeyer, D. Binkley, and F. Ricca, editors, *CSMR-WCRE ERA*, pages 313–317. IEEE, Feb. 2014.
14. V. Zaytsev. Negotiated Grammar Evolution. *JOT*, 13(3):1:1–22, July 2014.
15. V. Zaytsev. BibSLEIGH: Bibliography of Software Language Engineering in Generated Hypertext. In A. H. Bagge, editor, *SATToSE*, pages 59–62, July 2015.