A Coupled Transformations Chrestomathy

Technology Showdown Demonstration

Ralf Lämmel
Software Languages Team
http://www.softlang.org/
University of Koblenz-Landau
Germany

Abstract

Many software engineering contexts involve a collection of coupled artifacts, i.e., changing one artifact may challenge consistency between artifacts of the collection. A coupled software transformation (CX) is meant to transform one or more artifacts of such a collection while preserving consistency. There are many forms of coupling—depending on technological space and application domain and solution approach. We collect important forms of coupling in an illustrative manner within the Prolog-based software language repository YAS (Yet Another SLR (Software Language Repository)).

1 Coupled transformations

Many software engineering contexts involve a collection of *coupled* artifacts, i.e., changing one artifact may challenge *consistency* between artifacts of the collection. For instance, coupling may concern i) the model versus the code of a system in model-driven development, ii) the individual source code units making up a system, iii) the database and the web-based view in a web application, or iv) the model versus the metamodel in modeling. A *coupled software transformation* [Läm04] (CX) is meant to transform one or more artifacts of such a collection while preserving consistency. A very similar view on the coupling problem is based on the notions of *bidirectional transformations* (BX) [FHP15] or model synchronization [Dis11].

We collect important forms of coupling in an illustrative manner. The collection can be considered a chrestomathy, i.e., a collection of software systems (here: transformation systems) meant to be useful in learning about or gaining insight into programming and software engineering (here: about software transformations) [Läm15].

2 Software language repositories

In demonstrating CX, we operate in the context of a software language repository (SLR)—this is a software repository for software languages. An SLR features diverse artifacts: language definitions, language implementations, language processors, and sample artifacts. More specifically, we leverage the SLR YAS[1] (Yet Another

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

[1] YAS website: http://www.softlang.org/yas

SLR) which is an SLR targeting teaching and research on the foundations and engineering of software languages. YAS can be compared to other repositories around metalanguages or metaprogramming scenarios or areas in the programming languages field; e.g., Krishnamurthi's textbook on programming languages [Kri03], Batory's Prolog-based work on teaching MDE [BLA13], Zaytsev et al.'s software language processing suite (SLPS) [Z+08], or the by now unavailable repository for prological language processing by this author [LR01]. YAS relies primarily on Prolog as the language for executable language definition. Some of the defined languages are metalanguages.

3 An example: co-transformation on signatures and terms

Consider a simple algebraic signature (or algebraic data types as in Haskell) for binary trees with Peano-style natural numbers at the leafs:

```
symbol leaf : nat -> tree ; // leafs of trees
symbol fork : tree # tree -> tree ; // binary fork of tree
symbol zero : -> nat ; // natural number 0
symbol succ : nat -> nat ; // successor of a natural number
```

Now assume that some renaming should be applied, e.g., the sort `tree` should be renamed to `bintree`, and the constructors `zero` and `succ` should be renamed to `z` and `s`. The result of renaming looks like this:

```
symbol leaf : nat -> bintree ;
symbol fork : bintree # bintree -> bintree ;
symbol z : -> nat ;
symbol s : nat -> nat ;
```

The actual transformation could be presented as a prefix term (as in Prolog, for example) like this:

```
sequ( sequ(
        renameSym(zero, z),
        renameSym(succ, s)),
        renameSort(tree, bintree) ).
```

The actual interpretation of the transformation, subject to an appropriate term-based representation of signatures could be expressed in Prolog as follows:

```
:- module(bstlSig, []).

interpret(sequ(X1, X2), T1, T3) :-
    interpret(X1, T1, T2),
    interpret(X2, T2, T3).
interpret(renameSort(N1, N2), T1, T2) :-
    map(bstlSig:renameSort1(N1, N2), T1, T2).
interpret(renameSym(N1, N2), T1, T2) :-
    map(bstlSig:renameSym(N1, N2), T1, T2).

renameSort1(N1, N2, (F, Ss1, S1), (F, Ss2, S2)) :-
    renameSort2(N1, N2, S1, S2),
    map(bstlSig:renameSort2(N1, N2), Ss1, Ss2).

renameSort2(N1, N2, N3, N4) :-
    N3 == N1 -> N4 = N2 ; N4 = N3.

renameSym(N1, N2, T1, T2) :-
    T1 = (N1, R) -> T2 = (N2, R) ; T2 = T1.
```

The issue of coupling shows up when we consider terms that are supposed to conform to the shown signatures. These terms would need to be co-transformed. Consider the following term which is meant to conform to the first signature:

```
fork(fork(leaf(zero), leaf(zero)), leaf(succ(zero))).
```

For brevity, we leave out the resulting term of co-transformation and the actual interpretation of transformation descriptions for the purpose of co-transformation.

4 Build management and regression testing

CX scenarios are captured by YAS' Ueber language for build management and regression testing which makes it particularly clear what the relevant artifacts and relationships are. For instance, the scenario of the previous section is modeled in Ueber as follows:

```
[ elementOf('trafo1.term',bstl(term)),
  elementOf('term1.term',term),
  elementOf('term2.term',term),
  elementOf('sig1.term',bsl(term)),
  elementOf('sig2.term',bsl(term)),
  relatesTo(conformsTo,['term1.term','sig1.term']),
  relatesTo(conformsTo,['term2.term','sig2.term']),
  mapsTo(interpret,['trafo1.term','term1.term'],['term2.term']),
  mapsTo(interpret,['trafo1.term','sig1.term'],['sig2.term']) ].
```

That is, all the involved artifacts (two signatures, two terms, the transformation) are associated with the relevant languages; see '`elementOf`'. The terms are stated to conform to the respective signatures; see '`relatesTo(conforms,...)`'. Interpretations of the transformations are applied to both a signature and a term; see '`mapsTo(interpret,...)`'. The symbols `conformsTo` and `interpret` are associated with appropriate functionality for signature-based conformance checking and signature- as well as term-level transformation (subject to overloading the symbol `interpret`).

References

[BLA13] Don S. Batory, Eric Latimer, and Maider Azanza. Teaching Model Driven Engineering from a Relational Database Perspective. In *Proc. MODELS 2013*, volume 8107 of *LNCS*, pages 121–137. Springer, 2013.

[Dis11] Zinovy Diskin. Model Synchronization: Mappings, Tiles, and Categories. In *GTTSE 2009, Revised Papers*, volume 6491 of *LNCS*, pages 92–165. Springer, 2011.

[FHP15] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences*, 58(5):1–21, 2015.

[Kri03] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, 2003. `https://cs.brown.edu/~sk/Publications/Books/ProgLangs/`.

[Läm04] Ralf Lämmel. Coupled software transformations. In *Proc. SET 2004 (First International Workshop on Software Evolution Transformations)*, pages 31–35, 2004. Extended Abstract. Available online at `http://post.queensu.ca/~zouy/files/set-2004.pdf\#page=38`.

[Läm15] Ralf Lämmel. Software chrestomathies. *Sci. Comput. Program.*, 97:98–104, 2015.

[LR01] Ralf Lämmel and Günter Riedewald. Prological Language Processing. *ENTCS*, 44(2):132–156, 2001.

[Z+08] Vadim Zaytsev et al. Software Language Processing Suite, 2008. `http://slps.github.io/`.