Coupled Software Transformations—Revisited

Ralf Lämmel

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

Abstract

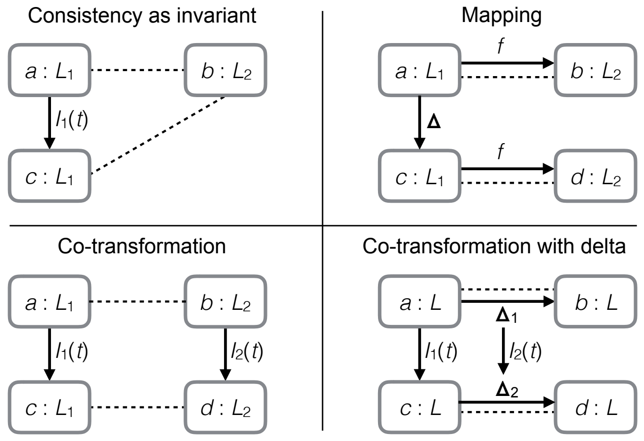
We revisit the notion of coupled software transformations (CX) which is concerned with keeping collections of software artifacts consistent in response to changes of individual artifacts. We model scenarios of CX while we abstract from technological spaces and application domains. Our objective is to mediate between universal consistency properties of CX and test-driven validation of concrete (illustrative) CX implementations. To this end, we leverage an emerging megamodeling language LAL which is based on many- and order-sorted predicate logic with support for reuse by inlining modulo substitution. We provide a simple translation semantics for LAL so that formulae can be rendered as test cases on appropriate interpretations of the megamodel elements. Our approach has been implemented and validated in logic programming; this includes the executable language definition of LAL and test-case execution on top of illustrative CX implementations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.2.5 [*Software Engineering*]: Testing and Debugging; D.1.6 [*Programming Techniques*]: Logic Programming

Keywords Coupled software transformation, CX, Bidirectional transformation, BX, Megamodeling, Linguistic architecture, Predicate logic, Logic programming, Testing

1. Introduction

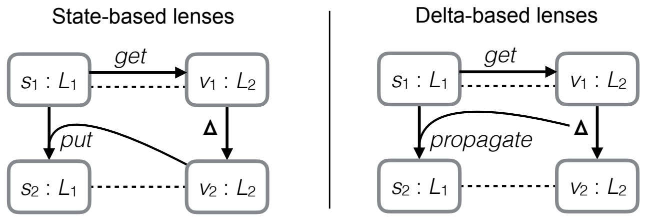
Many software engineering contexts involve a collection of *coupled* artifacts, i.e., changing one artifact may challenge *consistency* between artifacts of the collection. For instance, coupling may concern i) the model versus the code of a system in model-driven development, ii) the individual source 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* [36] (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) [27] or model synchronization [17]. For the purpose of this paper, we mainly stick to the term CX.



Legend: Nodes denote named artifacts 'typed' by languages. Dashed lines denote consistency relationships. Arrows denote changes with labels such as f for a function, Δ, Δ_1, and Δ_2 for deltas, and $I_1(t)$ and $I_2(t)$ for interpretations of a transformation description t.

Figure 1. Some CX patterns (inspired by [36])

The nature and characteristics of artifacts, changes and transformations, consistency relations, synchronization measures, and yet other aspects may be quite different, as evident from surveys on CX/BX or classifications of such transformations [16, 20, 47]. A few forms or 'patterns' of CX/BX are illustrated in Fig. 1–Fig. 2. In this paper, we aim at pattern descriptions which are more precise than those in the figure; we aim at capturing interesting aspects of CX at an appropriate level of abstraction so that the models can be validated in terms of actual CX implementations.



In the first (more basic) BX pattern, *get* maps a source to a view; *put* maps back a changed view to a source while taking into account the original source so that BX can go beyond bijective functions. In the second (more detailed) BX pattern, *put* has been replaced by a decomposition of differencing and change propagation.

Figure 2. Two basic BX patterns

The kind of models that we seek are *megamodels* [5, 7, 8, 35] as pioneered in the technological space 'modelware' or model-driven engineering (MDE). Megamodels have been proposed to manage repositories of models, metamodels, model transformations, and model-based software, e.g., in the sense of models@runtime. Megamodeling is an active research area with challenges related to the formal underpinnings, the generality in terms of application domains and technological spaces, and the validation of the models. [19, 23, 39].

Contributions of the paper

- *A suite of megamodels for CX/BX patterns.* In this manner, non-trivial forms of software transformations are modeled. The paper includes patterns for CX/BX forms such as mapping, co-transformation, and lenses.
- *A predicate logic-based megamodeling approach without commitment to a specific technological space.* To this end, an emerging language LAL ('Linguistic Architecture Language') is described including its executable language definition.
- *A translation semantics for megamodels suitable for testing software transformations.* In this manner, the CX megamodels are shown to abstract in a useful manner. For instance, universally quantified properties are mapped to executable test cases for actual CX implementations on actual artifacts.

Roadmap of the paper Sec. 2 provides background on the notion of CX by surveying research on CX and specifically pointing out different application domains and scenarios. Sec. 3 introduces LAL in a nutshell by capturing basics of software transformation. Sec. 4 develops megamodels for patterns of CX. Sec. 5 describes the translation of megamodels into test cases. Sec. 6 provides the language definition of LAL. Sec. 7 discusses related work. Sec. 8 concludes the paper.

The megamodels of this paper and the implementation of LAL are available online.[1]

2. Background on CX

We survey the literature on CX to discover application domains and scenarios of CX, thereby also motivating the more abstract CX/BX patterns of Fig. 1–Fig. 2. As a matter of scoping this survey, we specifically look at papers that are concerned with CX explicitly. In fact, we considered papers that cited the original CX paper [36][2] and follow-up papers in a few cases.

In a metamodeling context, there is the important problem of model/metamodel co-evolution [30, 54]; this is an instance of 'Co-transformation' as of Fig. 1. In the context of relational databases or XML, there is the very similar problem of instance/schema coevolution [6, 28].

In generalization of instance/schema co-evolution, programs (queries, transformations) may also be involved in co-evolution [13, 14, 29]. Likewise, there are situations of a network of artifacts at the same or different levels of abstraction; see, for example, the co-evolution of GMF editor models [46] or multi-language refactoring [48].

In a parsing context, there is the important problem of concrete versus abstract syntax adaptation [40, 51]. When, for example, the concrete syntax is transformed such that the generated language is not affected, then this is an instance of 'Consistency as invariant' as of Fig. 1. Other forms of CX have been studied in the broader context of syntax or language definition: the coevolution of metamodels and model-to-text transformations [45] and change tracking for DSL programs based on semantically meaningful source code deltas [50].

In a code generation context, as relevant in the areas of domain-specific languages and model-driven engineering, there is the important problem of code customization [41, 59], i.e., as to how to preserve changes to generated code when re-generating the code. In this case, 'Co-transformation with delta' as of Fig. 1 may be applicable.

In a technological space traveling context, in the generalized sense of de-/serialization, there is the important problem of mapping data models from one space to the other as well as instances across these spaces, back and forth; see [37, 38] for a general discussion on Object/Relational/XML mapping; this problem involves 'Mapping' as of Fig. 1.

The large body of research on BX is mentioned here by means of these proxies: bidirectionalization of transformations on trees and graphs [31, 42], model synchronization in the sense of BX and lenses [17]. In Fig. 2, we sketch two patterns for BX with lenses, state-based versus delta-based lenses [18], which differ in whether change discovery and

[1] `https://github.com/softlang/yas/tree/sle16`

[2] `https://scholar.google.com/scholar?cluster=7317986457099942654`

change propagation are separated through the intermediate entity of a delta.

CX occur in yet other contexts of software engineering and development: evolution of spreadsheets [15, 43]; co-evolution in web applications [12, 57]; modernization of component-based systems [26]; co-evolution in requirements managements [22] and viewpoint modeling [58]; the refinement of feature models [55].

3. LAL—in a Nutshell

In the following, we introduce the emerging LAL language, which we use for megamodeling in this paper—specifically for modeling CX patterns. LAL is a logic-based modeling or specification language as follows:

- LAL leverages *first-order predicate logic*. For instance, conformance is a relation (i.e., a predicate).
- LAL leverages *many-sorted logic*—sorts model languages, '∈' models membership tests for languages.
- LAL leverages *order-sorted logic*—'⊆' models subset relationships on languages.
- LAL supports flexible reuse of megamodels ('modules') by *inlining modulo substitution*.

We introduce LAL's constructs by means of examples.

3.1 Languages

Let us express that a language L is a subset of a suitable universe Any (such as 'all' strings, trees, or graphs):

LAL megamodel *language*

```
sort Any // The universe to draw elements from
sort L ⊆ Any // A language as a subset of the universe
```

The names of megamodel elements may be substituted along reuse of a megamodel. This is illustrated here for the case of the concrete XML-based language MathML.

LAL megamodel *language.mathml*

```
reuse language [ L ↦ MathML, Any ↦ XML ]
link MathML to 'https://www.w3.org/TR/MathML3'
link XML to 'https://www.w3.org/XML'
```

Thus, we reuse the megamodel *language* by substituting L and Any by MathML and XML, respectively. At the bottom, we also added 'identity links' to the names (see link XML to ...) so that it is clear that XML and MathML are specific languages as opposed to mere placeholders.

In LAL, the semantics of 'reuse' is inlining modulo substitution of names by names; see the '... ↦ ...' construct. The LAL language processor exposes the result of inlining modulo substitution. For instance, the megamodel *language.mathml*, as shown above, looks as follows—after inlining modulo substitution:

```
sort XML
sort MathML ⊆ XML
link MathML to 'https://www.w3.org/TR/MathML3'
link XML to 'https://www.w3.org/XML'
```

The following megamodel captures the basic pattern of 'demonstrating' a given language in terms of both a positive and negative case for membership:

LAL megamodel *membership*

```
reuse language
constant pos, neg : Any // Candidate elements
axiom member { pos ∈ L } // A member
axiom notMember { ¬ (neg ∈ L) } // A non−member
```

That is, we use (trivial) formulae ('axioms') to express that given constants (nullary functions) are elements or not of a given language. Axioms are optionally labeled for convenience; see member and notMember.

The following megamodel captures the basic pattern of 'conformance': there is a definition language and an actual definition defining a language such that conformance of a instance to the definition holds if and only if the instance is an element of the defined language [24, 25]:

LAL megamodel *conformance*

```
reuse language // The defined language
reuse language [ L ↦ DefL, Any ↦ DefAny ]
constant defL : DefL // The language definition
relation conformsTo : Any × DefL
axiom { ∀x ∈ Any. x ∈ L ⇔ conformsTo(x, defL) }
```

Thus, we reuse the megamodel *language* both for the language under definition and the definition language with possibly different universes. For instance, we may set up 'conformsTo' as XML Schema-based validation and apply it to MathML as follows:

LAL megamodel *conformance.mathml*

```
reuse conformance [
    Any ↦ XML, DefAny ↦ XML,
    L ↦ MathML, DefL ↦ XSD, defL ↦ MathMLSchema ]
link XML to 'https://www.w3.org/XML'
link XSD to 'https://www.w3.org/XML/Schema'
link MathML to 'https://www.w3.org/TR/MathML3'
link MathMLSchema to 'https://www.w3.org/Math/XMLSchema'
```

That is, we use XSD (XML Schema) for language definition with the MathMLSchema as the actual definition of MathML.

3.2 Transformations

Semantically speaking, transformations are simply functions, possibly partial functions because of preconditions. Here is the basic scheme of a transformation from one language L_1 to another language L_2; we use '⇸' to hint at partiality.

LAL megamodel *transformation*

```
reuse language [ L ↦ L1, Any ↦ Any1 ]
reuse language [ L ↦ L2, Any ↦ Any2 ]
function transform : L1 ⇸ L2
```

Here are some possible substitution situations with the purpose of 'transform' to be explained below:

	L_1	Any_1	L_2	Any_2
(1)	XML	XML	XML	XML
(2)	XMI	XML	XMI	XML
(3)	XSD	XML	XSD	XML
(4)	Ecore	XMI	Ecore	XMI
(5)	XSD	XML	Ecore	XMI

(1) corresponds to an ‘untyped’ XML transformation. (2) corresponds to a model transformation operating at the level of the XML-based XMI format for model representation. (3) corresponds to an XML schema transformation. (4) corresponds to a metamodel transformation while assuming the XMI-based representation of EMF/Ecore-based metamodels. (5) corresponds to an XML schema-to-Ecore transformation (i.e., a bridge between the XML and EMF/Ecore technological spaces).

3.3 Interpretations

We may be interested in transformation languages as opposed to actual transformations. Thus, we may need interpreters of transformation descriptions:

LAL megamodel *interpretation*

```
reuse language [ L ↦ L1, Any ↦ Any1 ]
reuse language [ L ↦ L2, Any ↦ Any2 ]
reuse language [ L ↦ XL, Any ↦ XAny ]
function interpret : XL × L1 ⇸ L2
```

Here are some possible substitution situations with the purpose of ‘interpret’ to be explained below:

	L_1	Any_1	L_2	Any_2	XL	XAny
(1)	XML	XML	XML	XML	XSLT	XML
(2)	XMI	XML	XMI	XML	ATL	Text
(3)	Java	Text	SQL	Text	JPA annot.	Java annot.

(1) corresponds to the situation of plain XSLT transformation where the regular XSLT processor transforms a given XML document according to some XSLT ‘transformation’ to another XML document. (2) corresponds to the situation of an ATL [34] model transformation where the ATL interpreter transforms a given XMI-based model into another model without a decomposition into phases like parsing, compilation, and bytecode interpretation. (3) corresponds to the situation of object/relational mapping with JPA (or Hibernate) where Java classes are mapped to SQL CREATE TABLE et al. statements on the grounds of JPA annotations.

3.4 Whole-part Relationships

Software artifacts can be decomposed according to whole-part relationships. For instance, models consist of model elements, parse trees consist of subtrees, etc. These relationships are ‘exercised’ by software transformations in that they, for example, recurse into parts.

LAL megamodel *composition*

```
reuse language
relation partOf, partOf+, partOf*: L × L
axiom partAsym { ∀x, y ∈ L. partOf(x, y) ⇒ ¬ partOf(y, x) }
axiom partReflexive { ∀x ∈ L. partOf*(x, x) }
axiom partTransitive { ∀x, y ∈ L.
  (partOf(x, y) ⇒ partOf+(x, y))
  ∧ (partOf+(x, y) ⇒ partOf*(x, y))
  ∧ (∀ z ∈ L. partOf+(x, z) ∧ partOf+(z, y) ⇒ partOf+(x, y)) }
```

We need to assume here that ‘wholes’ and ‘parts’ are of the same language.

3.5 Correspondence

When transformations perform some sort of systematic mapping where parts of the source correspond to parts of the target ‘more or less’ in a one-to-one manner, possibly in a recursive fashion, then we may speak of ‘correspondence’ [23, 39]. Such correspondence may serve as the consistency relation in CX.

As a concrete example, consider object/relational/XML mapping [37] when two type- or instance-level artifacts are similarly composed from parts. We introduce a corresponding relation and provide an axiomatization of an ‘extreme’ (say, practically unrealistic) case with perfect 1:1 correspondence:

LAL megamodel *correspondence*

```
reuse composition [ L ↦ L1, Any ↦ Any1 ]
reuse composition [ L ↦ L2, Any ↦ Any2 ]
relation correspondsTo : L1 × L2
```

LAL megamodel *correspondence.oneToOne*

```
reuse correspondence
relation related : L1 × L2
axiom { ∀a1 ∈ L1. ∀a2 ∈ L2.
  related(a1, a2)
    ∧ (∀ b1 ∈ L1. partOf(b1, a1) ⇒
          ∃! b2 ∈ L2. partOf(b2, a2) ∧ correspondsTo(b1, b2))
    ∧ (∀ b2 ∈ L2. partOf(b2, a2) ⇒
          ∃! b1 ∈ L1. partOf(b1, a1) ∧ correspondsTo(b1, b2))
  ⇒ correspondsTo(a1, a2) }
```

The axiomatization assumes an ingredient for identifying ‘related’ parts on each side; this identification could be based on matching names, for example. In practice, either side of the correspondence may involve parts or levels of composition that cannot be associated with the other side in a 1:1 manner.

3.6 Differencing

Changes due to manual or automated transformation may be represented as a diff (a delta) inferred from two ‘versions’ of an artifact; see the function diff. Diffs may be represented in appropriate diff languages [10, 11]; see the language DiffL. A diff can be applied very much like a transformation description is interpreted; see the function applyDiff. Diffs are obviously needed in modeling CX patterns, as evident from Fig. 1–Fig. 2.

LAL megamodel *differencing*

```
reuse language // The language of artifacts to be diffed
reuse language [ L ↦ DiffL, Any ↦ DiffAny ] // Differences
function diff : L × L → DiffL // The differencing algorithm
function applyDiff : DiffL × L → L // Application of differences
function invDiff : DiffL → DiffL // Inversion of differences
constant emptyDiff : DiffL // The unit for differences
axiom apply { ∀x, y ∈ L. ∀d ∈ DiffL. applyDiff(diff(x, y), x) = y }
axiom inv { ∀x, y ∈ L. invDiff(diff(x,y)) = diff(y,x) }
axiom diffEmpty { ∀x ∈ L. diff(x, x) = emptyDiff }
axiom empty { ∀x ∈ L. applyDiff(emptyDiff, x) = x }
axiom invEmpty { invDiff(emptyDiff) = emptyDiff }
axiom invTwice { ∀d ∈ DiffL. invDiff(invDiff(d)) = d }
```

4. Selected CX Patterns

We capture the patterns of Fig. 1–Fig. 2 in LAL. We set up the basic scheme of coupling by assuming two languages and a consistency relationship between artifacts of the two languages. Thus:

LAL megamodel *coupling*

```
reuse language [ L ↦ L₁, Any ↦ Any₁ ]
reuse language [ L ↦ L₂, Any ↦ Any₂ ]
relation consistent : L₁ × L₂ // The consistency relationship
```

The assumption is that consistency could be defined in different ways depending on application scenarios. For instance, consistency may correspond to *conformance* (Sec. 3.1), *correspondence* (Sec. 3.5), or some form of *interface compatibility* (such as two code units providing the same interface).

4.1 The 'Mapping' Pattern

The 'Mapping' pattern, as expressed by the following axiom, assumes that consistency is re-established by mapping a possibly changed source to a new target:

LAL megamodel *cx.mapping*

```
reuse coupling
function mapping : L₁ → L₂ // Mapping between languages
axiom { ∀a ∈ L₁. ∀b ∈ L₂. mapping(a) = b ⇒ consistent(a, b) }
```

An example of 'Mapping' is XML-schema-to-object-model mapping, where a suitable object model (e.g., Java classes) is derived from a given XML schema.

The pattern can be advanced to enable incremental mapping, i.e., propagating changes of the source rather than producing a completely new target; see the online resources for the paper.

4.2 The 'Consistency as Invariant' Pattern

The following axiom requires that any interpretation of a transformation description of the transformation language XL is consistency-preserving:

LAL megamodel *cx.invariant*

```
reuse coupling
reuse interpretation [ L₂ ↦ L₁, Any₂ ↦ Any₁ ]
axiom { ∀t ∈ XL. ∀a, c ∈ L₁. ∀b ∈ L₂.
  consistent(a, b) ∧ interpret(t, a) = c
    ⇒ consistent(c, b) }
```

An example of 'Consistency as invariant' is grammar refactoring or grammar extension without affecting or extending the generated language so that available elements of the language remain consistent with the grammar. Ultimately, consistency preservation may also rely on constraints on a and b.

4.3 The 'Co-transformation' Pattern

The following axiom requires that any transformation t, when applied to consistent sources $a \in L_1$ and $b \in L_2$, returns consistent targets $c \in L_1$ and $d \in L_2$:

LAL megamodel *cx.cotransformation*

```
reuse coupling
reuse interpretation [ L₂ ↦ L₁, Any₂ ↦ Any₁ ]
reuse interpretation [ L₁ ↦ L₂, Any₁ ↦ Any₂ ]
axiom consistency { ∀t ∈ XL. ∀a, c ∈ L₁. ∀b, d ∈ L₂.
  consistent(a, b)
  ∧ interpret(t, a) = c
  ∧ interpret(t, b) = d ⇒ consistent(c, d) }
```

'Co-transformation' is relevant, for example, in the context of model/metamodel co-evolution. That is, L_1 would be a language of models whereas L_2 would be a language of metamodels. When a metamodel evolves, existing models have to co-evolve to reestablish conformance. In Section 5, we will consider an illustrative co-transformation; it is concerned with term/signature co-evolution.

4.4 The 'Co-transformation with Delta' Pattern

In the basic 'Co-transformation' pattern, a transformation description t is interpreted at both ends of coupling. If we assume that one end deals with deltas (diffs) rather than ordinary artifacts, then the interpretation of the transformation serves change propagation on that end. In the following megamodel, we assume a relation compatible to constrain the kind of changes that can be dealt with consistently.

LAL megamodel *cx.delta*

```
reuse differencing
reuse cx.cotransformation [
  L₁ ↦ L, Any₁ ↦ Any,
  L₂ ↦ DiffL, Any₂ ↦ DiffAny ]
relation compatible : L × L
axiom { ∀x, y ∈ L. ∀delta ∈ DiffL.
  compatible(x, y) ∧ diff(x, y) = delta ⇒ consistent(x, delta) }
axiom { ∀a, b ∈ L. ∀delta₁ ∈ DiffL.
  applyDiff(delta₁, a) = b ∧ compatible(a, b) ⇒
    (∀ t ∈ XL. ∀c ∈ L. ∀delta₂ ∈ DiffL.
      interpret(t, a) = c ∧ interpret(t, delta₁) = delta₂ ⇒
        (∃ d ∈ L. applyDiff(delta₂, c) = d ∧ compatible(c, d))) }
```

The first axiom simply captures that consistency is meant here in the sense of changing an artifact in a compatible sense while being able to capture that change as a diff. The second axiom assumes such a compatible change from a to b, as captured by diff delta_1; the axiom states that any transformation t, when interpreted on a and delta_1, yields c at one end and delta_2 at the other end such that the transformed diff can be used to change c to a compatible d. The pattern 'Co-transformation with delta' is useful in code generation when changes to generated code are to be preserved along regeneration.

4.5 The 'State-based Lenses' Pattern

Basic lenses enhance the 'Mapping' pattern; see the substitution of mapping by what is called get in the case of lenses. There is put for the opposite direction, which we mark here as being possibly partial. In the terminology of lenses, L_1 is the language of the *source* and L_2 is the language of the *view*.

LAL megamodel *bx.state*

```
reuse cx.mapping [ mapping ↦ get ]
function get : L₁ → L₂
function put : L₁ × L₂ ⇸ L₁
axiom GetPut { ∀s ∈ L₁.
  put(s, get(s)) = s }
axiom PutGet { ∀s₁, s₂ ∈ L₁. ∀v ∈ L₂.
  put(s₁, v) = s₂ ⇒ get(s₂) = v }
```

The axioms GetPut and PutGet are the most basic ones in the theory on lenses. The specific formulation of PutGet accounts for partiality of put: we do not assume that all conceivable changes of the view can be put back.

4.6 The 'Delta-based Lenses' Pattern

The following axiomatization imposes more structure on state-based lenses to arrive at the delta-based form. Differences on views as well as their propagation on sources are taken into account.

LAL megamodel *bx.delta*

```
reuse bx.state
reuse differencing [ L ↦ L₂, Any ↦ Any₂ ]
function propagate : L₁ × DiffL ⇸ L₁
axiom { ∀s₁, s₂ ∈ L₁. ∀v₁, v₂ ∈ L₂. ∀delta ∈ DiffL.
  get(s₁) = v₁
  ∧ diff(v₁, v₂) = delta
  ∧ propagate(s₁, delta) = s₂ ⇒
       put(s₁, v₂) = s₂ ∧ get(s₂) = v₂ }
```

The axiom models that put can be regarded as a composition of diffing and diff propagation. The overall idea of delta-based lenses is indeed that they decompose change propagation into parts that may be controlled and reused independently. We could even carry on and decompose propagation into diff transformation and normal diff application with applyDiff.

5. Translation of Megamodels

Megamodels reside at a high level of abstraction, giving rise to the overall problem of megamodel 'adequacy'. That is, how to gain confidence about a megamodel's correctness or appropriateness or usefulness? The language processing model of LAL with its translation semantics to test cases addresses the adequacy problem in a particular manner.

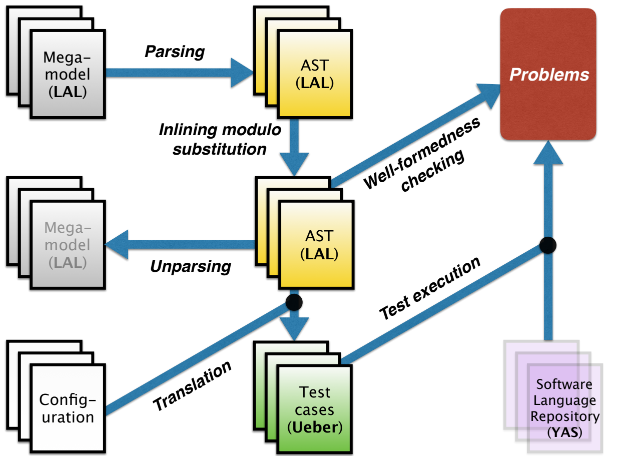


Figure 3. Megamodel processing for LAL.

5.1 Megamodel Processing for LAL

The various aspects of processing LAL's megamodels are shown in Fig. 3. LAL's concrete syntax is parsed into an abstract syntax. Inlining modulo substitution is applied then. Well-formedness checking is applied to megamodels after such inlining. Well-formedness checking is concerned with the integrity of the megamodel such that all referenced names are declared and yet other conditions which are comparable to a programming language's type system or static semantics. There is also an unparser so that the result of inlining can be inspected by the user, which may help with understanding. A translation is applied to the megamodel to derive test cases (descriptions thereof) so that available interpretations of languages, relations, and functions can be tested in terms of the formulae in the megamodels.

Megamodel-based testing is applied to artifacts available in YAS—Yet Another SLR (Software Language Repository)[3]. That is, YAS is a collection of executable language definitions and language processing components including software transformations. Megamodel-based testing is specifically applied to the logic programming-based slice of YAS. The derived test cases are represented in a lower level megamodeling notation, UEBER, which serves for build management and regression testing in YAS.

5.2 An illustrative CX

We set up an illustrative CX; it is concerned with term/signature co-evolution. YAS supports a 'Basic Signature Language' (BSL) inspired by algebraic signatures. Binary trees with Peano-like natural numbers (*zero*, *succ*(*zero*), *succ*(*succ*(*zero*)), ...) at the leafs are modeled by the following signature:

Signature *languages/BSTL/tests/**sig1.bsl***

```
symbol leaf : nat → tree ; // leaf in a tree
symbol fork : tree × tree → tree ; // binary fork in a tree
```

[3] http://www.softlang.org/yas

```
symbol zero : → nat ; // natural number 0
symbol succ : nat → nat ; // successor of a natural number
```

Now imagine that we wish to rename the sort tree to bintree and the symbols zero and succ to z and s. The resulting signature looks like this:

Signature *languages/BSTL/tests/**sig2.bsl***

```
symbol leaf : nat → bintree ;
symbol fork : bintree × bintree → bintree ;
symbol z : → nat ;
symbol s : nat → nat ;
```

Here is a term that conforms to the initial signature:

Term *languages/BSTL/tests/**term1.term***

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

Here is the co-transformed term which conforms to the transformed signature:

Term *languages/BSTL/tests/**term2.term***

```
fork(fork(leaf(z), leaf(z)), leaf(s(z))).
```

We use a transformation language BSTL—Basic Signature Transformation Language. The described transformation can be expressed by a BSTL term like this:

Transformation *languages/BSTL/tests/**trafo1.term***

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

That term represents the sequential composition (see sequ) of three simpler transformations (see two applications of renameSym and one application of renameSort). The syntax of the corresponding language for signature and term transformation is described by the following signature; we use YAS' 'Extended Signature Language' (ESL) here. (ESL is an extension of BSL. For instance, ESL features primitive types like strings.)

Syntax of BSTL *languages/BSTL/**as.esl***

```
symbol sequ : trafo × trafo → trafo ;
symbol renameSort : sort × sort → trafo ;
symbol renameSym : sym × sym → trafo ;
...
type sort = string ;
type sym = string ;
```

Let us also provide the (ESL-based) signature of (BSL-based) signatures which we need when expressing transformations of signatures:

Syntax of signatures *languages/BSL/**as.esl***

```
type signature = profile∗ ;
type profile = sym × sort∗ × sort ;
type sym = string ;
type sort = string ;
```

For instance, the signature of binary trees, as presented at the top of the section, is rendered in abstract syntax as follows:

Signature *languages/BSTL/tests/**sig1.term***

```
[ (leaf, [nat], tree),
  (fork, [tree, tree], tree),
  (zero, [], nat),
  (succ, [nat], nat) ].
```

We are now at the point that we implement BSTL by interpreters. We begin with the interpretation on terms. (We use the higher-order predicate map/3 for list processing.)

Prolog module *bstlTerm.pro*

```
interpret(sequ(X1, X2), T1, T3) ⇐
    interpret(X1, T1, T2),
    interpret(X2, T2, T3).
interpret(renameSort(_, _), T, T).
interpret(renameSym(N1, N2), T1, T2) ⇐
    T1 =.. [N3|Ts1],
    ( N3 = N1 −> N4 = N2; N4 = N3 ),
    map(bstlTerm:interpret(renameSym(N1, N2)), Ts1, Ts2),
    T2 =.. [N4|Ts2].
```

Thus, renaming of sorts is a no-op at the level of terms; renaming of symbols is applied to the functors. Here is also the interpretation on signatures:

Prolog module *bstlSig.pro*

```
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.
```

Thus, renaming of sorts and symbols is realized by iterating over the structure of a signature in terms of individual symbol declarations and the list of argument sorts for each symbol.

5.3 Testing the CX

YAS uses a lower-level megamodeling language, UEBER, for build management and regression testing. As far as the translation of LAL is concerned, the following declaration forms of UEBER are relevant:

elementOf Associate a file with a language.

membership Associate a language with a logic programming predicate for a membership test.

relation/function Declare a relation or a function on files of specific languages as implemented by a logic programming predicate.

relatesTo/mapsTo Apply some relation or function on actual files.

Transformation according to the BSTL language is set up by the following declarations:

UEBER megamodel *languages/BSTL/**framework.ueber***

```
[ language(bstl(term)),
  membership(bstl(term), eslLanguage, ['as.term']),
  function(interpret,
    [bstl(term), bsl(term)], [bsl(term)], bstlSig:interpret, []),
  function(interpret,
     [bstl(term), term], [term], bstlTerm:interpret, []) ].
```

The shown declarations register i) the BSTL language assuming the term-based representation bstl(term), ii) a membership test for BSTL based on the term-based representation ('as.term') of the signature for BSTL (shown in textual syntax earlier), and iii) two function overloads for interpret which are declared to operate on different argument and result types. The function overloads are linked to the Prolog predicates bstlSig:interpret and bstlTerm:interpret for signature and term transformation.

The actual application of the CX can be expressed by the following UEBER declarations; this is what should be considered a test case:

UEBER megamodel *languages/BSTL/tests/**trafo1.ueber***

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

That is, the signatures and terms are associated with the relevant languages. Further, the functions for interpreting transformation descriptions are applied to the relevant files.

5.4 Megamodel-to-test Translation

The test case, as shown just above, is generated directly from the megamodel for the 'Co-transformation' pattern, from the consistency axiom, specifically. For convenience's sake, we repeat here the megamodel for the pattern—after inlining modulo substitution:

```
sort Any₁
sort L₁ ⊆ Any₁
sort Any₂
sort L₂ ⊆ Any₂
relation consistent : L₁ × L₂
sort XAny
sort XL ⊆ XAny
function interpret : XL × L₁ ⇸ L₁
function interpret : XL × L₂ ⇸ L₂
axiom consistency {
 ∀t ∈ XL. ∀a ∈ L₁. ∀c ∈ L₁. ∀b ∈ L₂. ∀d ∈ L₂.
        consistent(a, b)
        ∧ interpret(t, a) = c
        ∧ interpret(t, b) = d ⇒ consistent(c, d)
}
```

All the symbols of the megamodel including the variables from the axiom are to be bound to actual interpretations: files, languages, relations, and functions. Universal quantifications are exercised in a 'pointwise' manner by picking representatives. Interpretations are assigned by a configuration file:

LAL configuration
*languages/LAL/lib/cx/**cotransformation.lalconfig***

```
[ sort('L1', term),
  sort('Any1', term),
  sort('L2', bsl(term)),
  sort('Any2', term),
  sort('XL', bstl(term)),
  sort('XAny', term),
  relation(consistent, conformsTo),
  axiom(consistency, [
    (t, 'trafo1.term'),
    (a, 'term1.term'),
    (b, 'sig1.term'),
    (c, 'term2.term'),
    (d, 'sig2.term') ])].
```

The first few lines map the sorts of the LAL megamodel to implemented languages of YAS. The interpretation functions of the LAL megamodel do not need to be mapped explicitly because the name 'interpret' is used on both sides; see again the UEBER declarations for BSTL. Consistency of coupling is mapped to conformance checking with signatures. The variables of the consistency axiom are mapped to the files with the terms and signatures of our illustrative example.

6. LAL—Language Definition

We define LAL's syntax, well-formedness constraints (comparable to a type system), inlining reused megamodels modulo substitution (comparable to preprocessing), and a translation to test cases (comparable, in a limited manner, to a compilation semantics). The syntax is specified by a grammar for the concrete syntax and a signature for the abstract syntax. The remaining language definition components are specified as logic programs representing a deductive system for well-formedness and a rewrite system for inlining modulo substitution and translation.

6.1 Syntax

The concrete syntax is defined in YAS' 'Extended Grammar Language' (EGL; reminiscent of EBNF). The abstract syntax is defined in YAS' ESL, which we encountered earlier already; we omit the mapping from concrete to abstract synax for brevity.

Grammar *languages/LAL/**cs.egl***

```
// Megamodels
model : { decl }* ;

// Declarations
[reuse] decl : 'reuse' mname { substs }? ;
mname : name { '.' name }* ;
```

```
substs : '[' subst { ',' subst }∗ ']' ;
subst : name '|−>' name ;
[sort] decl : 'sort' name { '<=' name }? ;
[relation] decl : 'relation' names ':' types ;
names : name { ',' name }∗ ;
[function] decl : 'function' names ':' types arrow type ;
[total] arrow : '−>' ;
[partial] arrow : '~>' ;
[constant] decl : 'constant' names ':' type ;
[axiom] decl : 'axiom' { name }? '{' formula '}' ;
[link] decl : 'link' name 'to' url ;

// Types
types : type { '#' type }∗ ;
[star] type : typeterm { '∗' }∗ ;
[ref] typeterm : name ;
[product] typeterm : '(' types ')' ;

// Formulae
[forall] formula : 'forall' vars '<−' type '.' formula ;
[foreach] formula : 'foreach' var ':' expr '.' formula ;
[exists] formula : 'exists' vars '<−' type '.' formula ;
[existsu] formula : 'exists!' vars '<−' type '.' formula ;
[ifetal] formula : orforumula { ifetal formula }? ;
[iff] ifetal : '<=>' ;
[ifthen] ifetal : '=>' ;
[or] orforumula : andformula { '\/' orforumula }? ;
[and] andformula : basicformula { '/\' andformula }? ;
[not] basicformula : '~' basicformula ;
[relapp] basicformula : name '(' expr { ',' expr }∗ ')' ;
[eq] basicformula : expr '=' expr ;
[element] basicformula : expr '<−' type ;
basicformula : '(' formula ')' ;

// Expressions
[funapp] expr : name '(' expr { ',' expr }∗ ')' ;
[var] expr : name ;

// Variables and tuple patterns
vars : var { ',' var }∗ ;
[bindv] var : name ;
[bindt] var : '(' var { ',' var }+ ')' ;
```

Signature *languages/LAL/**as.esl***

```
// Megamodels
type model = decl∗ ;

// Declarations
symbol reuse : mname × subst∗ → decl ;
type mname = name+ ;
type name = string ;
type subst = name × name ;
symbol sort : name × name? → decl ;
symbol relation : name × types → decl ;
symbol function : name × types × arrow × type → decl ;
symbol total : → arrow ;
symbol partial : → arrow ;
symbol constant : name × type → decl ;
symbol axiom : name? × formula → decl ;
symbol link : name × url → decl ;
type url = string ;

// Type expressions
type types = type+ ;
symbol ref : name → type ;
symbol star : type → type ;
symbol product : types → type ;

// Formulae
symbol forall : var × type × formula → formula ;
symbol foreach : var × expr × formula → formula ;
symbol exists : var × type × formula → formula ;
symbol existsu : var × type × formula → formula ;
symbol relapp : name × expr+ → formula ;
symbol element : expr × type → formula ;
symbol eq : expr × expr → formula ;
symbol and : formula × formula → formula ;
symbol or : formula × formula → formula ;
symbol not : formula → formula ;
symbol iff : formula × formula → formula ;
symbol ifthen : formula × formula → formula ;

// Expressions
symbol funapp : name × expr+ → expr ;
symbol var : name → expr ;

// Variables and tuple patterns
symbol bindv : name → var ;
symbol bindt : var+ → var ;
```

Thus, a megamodel is a collection of declarations for sorts (languages), relations (predicates), functions (with constants as a special case), and axioms (formulae). Also, reuse declarations point to other megamodels to be inlined modulo substitution. We may also associate URIs as 'identity links' with declared names. The actual forms of formulae and expressions (terms) resemble predicate logic with some special forms due to the choice of a many- and order-sorted logic and the addition of convenience notation for sequences and products; see production labels *element* (for membership tests), *foreach* (for conjunctions over elements in a sequence), and *bindt* (for tuple patterns).

6.2 Inlining Modulo Substitution

Inlining entails replacement of reuse declarations by the corresponding megamodels modulo substitution. Substitution is not limited to consistent renaming; two names are allowed to be resolved to one.

Prolog module *lalReuse.pro*

```
% Case for megamodels (lists of declarations)
inline(Ds1, Ds2) ⇐
    map(lalReuse:inline, Ds1, Dss),
    concat(Dss, Ds2).

% Case for non−reuse declarations
inline(D, [D]) ⇐
    \+ D = reuse(_, _).

% Case for reuse declarations
inline(reuse(MN, Ss), Ds2) ⇐
    lalDeps:filename(MN, F),
    readTermFile(F, Ds1),
    substs(Ss, Ds1, Ds2).

% Apply a list of substitutions
substs([], Ds, Ds).
substs([(N1, N2)|Ss], Ds1, Ds3) ⇐
    topdown(try(lalReuse:subst(N1, N2)), Ds1, Ds2),
    \+ Ds1 == Ds2,
    substs(Ss, Ds2, Ds3).
```

```
% Patterns relevant for substitution
subst(N1, N2, sort(N1, [N1]), sort(N2, [N2])).
subst(N1, N2, sort(N1, N3), sort(N2, N3)).
subst(N1, N2, sort(N3, [N1]), sort(N3, [N2])).
subst(N1, N2, ref(N1), ref(N2)).
subst(N1, N2, relation(N1, Ts), relation(N2, Ts)).
subst(N1, N2, relapp(N1, Es), relapp(N2, Es)).
subst(N1, N2, function(N1, Ts, A, T), function(N2, Ts, A, T)).
subst(N1, N2, funapp(N1, Es), funapp(N2, Es)).
subst(N1, N2, constant(N1, T), constant(N2, T)).
subst(N1, N2, var(N1), var(N2)).
subst(N1, N2, axiom([N1], F), axiom([N2], F)).
subst(N1, N2, link(N1, U), link(N2, U)).
```

Thus, we iterate over the declarations of a megamodel, non-reuse declarations are preserved, reuse declarations are replaced by the referenced megamodels after performing substitution. Each substitution is checked to have an effect (see the test for non-equality). Each substitution is carried out by a top-down traversal (see the higher-order predicate topdown/3) with simple rules for all the relevant patterns. Referenced megamodels are retrieved by the predicate readTermFile/2.

6.3 Well-formedness

Well-formedness is modeled with one predicate per syntactic category and one clause per syntactic form with some details elided for brevity:

Prolog module *lalOk.pro*

```
model(Ds) ⇐
     typesOfNames(Ds),
     map(lalOk:decl(Ds), Ds).

% Each name is used for just one type of declaration
typesOfNames(Ds) ⇐ \+ (
     member(D1, Ds),
     member(D2, Ds),
     declToName(D1, F1, N),
     declToName(D2, F2, N),
     \+ F1 == F2 ).

declToName(sort(N, _), sort, N).
declToName(relation(N, _), relation, N).
declToName(function(N, _, _, _), function, N).
declToName(constant(N, _), constant, N).
declToName(axiom([N], _), axiom, N).

% Well−formedess of types
type(Ds, ref(N)).
type(Ds, star(T)) ⇐ type(Ds, T).
type(Ds, product(Ts)) ⇐ map(lalOk:type(Ds), Ts).

% Subtyping relationship
subTypeOf(_, T, T).
subTypeOf(Ds, ref(N1), ref(N2)) ⇐
     member(sort(N1, [N2]), Ds).

% Well−formedness of declarations
decl(Ds, sort(N, X)) ⇐ ...
decl(Ds, relation(_, Ts)) ⇐ ...
decl(Ds, function(_, Ts, _, T)) ⇐ ...
decl(Ds, constant(_, T)) ⇐ ...
decl(Ds, axiom(_, F)) ⇐ formula(Ds, [], F).
decl(Ds, link(N, _)) ⇐ ...
```

```
% Well−formedness of formulae
formula(Ds, M, relapp(N, Es)) ⇐
     member(relation(N, Ts), Ds),
     map(lalOk:expr(Ds, M), Es, Ts).
formula(Ds, M1, forall(V, T, F)) ⇐
     type(Ds, T),
     bind(Ds, V, T, M1, M2),
     formula(Ds, M2, F).
...
formula(Ds, M, ifthen(F1, F2)) ⇐
     formula(Ds, M, F1),
     formula(Ds, M, F2).

% Binding of variables for quantifiers
bind(Ds, bindv(N), T, M, [(N, T)|M]) ⇐
     \+ member((N, _), M),
     \+ member(constant(N, _), Ds).
bind(Ds, bindt([]), product([]), M, M).
bind(Ds, bindt([V|Vs]), product([T|Ts]), M1, M3) ⇐
     bind(Ds, V, T, M1, M2),
     bind(Ds, bindt(Vs), product(Ts), M2, M3).

% Well−formedness of expressions
expr(Ds, M, funapp(N, Es), T) ⇐
     member(function(N, Ts1, _, T), Ds),
     map(lalOk:expr(Ds, M), Es, Ts2),
     map(lalOk:subTypeOf(Ds), Ts2, Ts1).
expr(Ds, M, var(N), T) ⇐
     member((N, T), M);
     member(constant(N, T), Ds).
```

The constraint *typesOfNames/1* checks that names are unique across the different declaration forms. An axiom declaration is well-formed, if its formula is well-formed in terms of correctly applying relation and function symbols as well as variables bound by quantifiers. Both relations and functions may be overloaded and subtyping (see *subType/3*) for languages is applied for comparing formal and actual argument and result types.

6.4 Megamodel-to-test Translation

We translate LAL's megamodels to test cases which are described in terms of constructs of the lower-level megamodeling language UEBER tailored towards building and testing language processing components.

Relevant part of UEBER's abstract syntax

```
type model = decl∗ ;
symbol elementOf : file × lang → decl ;
symbol relatesTo : rela × file∗ → decl ;
symbol mapsTo : func × file∗ × file∗ → decl ;
type file = string ; // filenames
type rela = string ; // names of relations
type func = string ; // names of functions
type lang = term ; // names of languages
...
```

That is, declarations either associate files with languages, or relate files in terms of a relation, or map files to other files in terms of function applications. The translation relies on a configuration to map languages, relations, functions, and variables to actual manifestations. Here is the signature of configurations:

Signature *languages/LAL/config/**as.esl***

```
type config = entry∗ ;
symbol sort : string × term → entry ;
symbol constant : string × string → entry ;
symbol function : string × string → entry ;
symbol relation : string × string → entry ;
symbol axiom : string × map → entry ;
type map = (string × string)∗ ;
```

That is, LAL's language names (i.e., strings) are mapped to UEBER's language names (i.e., terms); LAL's relations and functions are mapped to UEBER's counterparts; LAL's axioms are associated with maps that map existentially or universally quantified variables to files.

Here is the translation of LAL to UEBER:

Prolog module *lalUeber.pro*

```
% Map LAL specification to Ueber declarations
translate(Lals, C, Ues) ⇐
    map(lalUeber:entry(Lals, C), C, Uess),
    concat(Uess, Ues).

% Map configuration entries to Ueber declarations
entry(_, _, sort(_, _), []).
entry(_, _, function(_, _), []).
entry(_, _, relation(_, _), []).
entry(Lals, C, constant(N1, F), [elementOf(F, L)]) ⇐
    member(constant(N1, ref(N2)), Lals),
    member(sort(N2, L), C).
entry(Lals, C, axiom(N, M), Ues) ⇐
    member(axiom([N], X), Lals),
    formula(C, M, X, Ues).

% Map LAL formulae to Ueber declarations
formula(C, M, X0, [elementOf(F, L)|Ues]) ⇐
    X0 =.. [Op, bindv(V), ref(N), X1],
    member(Op, [forall, exists, existsu]),
    member(sort(N, L), C),
    member((V, F), M),
    formula(C, M, X1, Ues).
formula(I, M, X0, Ues3) ⇐
    X0 =.. [Op, X1, X2],
    member(Op, [and, ifthen, iff]),
    formula(I, M, X1, Ues1),
    formula(I, M, X2, Ues2),
    append(Ues1, Ues2, Ues3).
formula(I, M, relapp(N1, Ts1), [relatesTo(N2, Ts2)]) ⇐
    ( member(relation(N1, N2), I) −> true; N2 = N1 ),
    map(lalUeber:argument(I, M), Ts1, Ts2).
formula(I, M, T0, [mapsTo(N2, Ts2, [T2])]) ⇐
    ( T0 = eq(funapp(N1, Ts1), T1);
      T0 = eq(T1, funapp(N1, Ts1)) ),
    ( member(function(N1, N2), I) −> true; N2 = N1 ),
    map(lalUeber:argument(I, M), Ts1, Ts2),
    argument(I, M, T1, T2).

% Map constants and variables to files
argument(I, _, var(V), F) ⇐ member(constant(V, F), I).
argument(_, M, var(V), F) ⇐ member((V, F), M).
```

In essence, the translation maps logic formulae to sequences of membership tests (exists etc. maps to elementOf) and applications of relations (relapp maps to relatesTo) and functions (funapp maps to mapsTo).

One should note that universal and existential quantification are translated in the same manner. That is, testing 'exercises' both forms of formulae in a 'point-wise' manner—for one specific element. Logically, we assume '$(\forall x \in L.\, P(x)) \Rightarrow (\exists x \in L.\, P(x))$'. Clearly, this is only sound if we assume L to be a non-empty set and if we limit the patterns of formulae, as this is the case in the translation. One should also note that implication (ifthen) and equivalence (iff) are treated exactly like conjunction (and). Again, this is only sound because we limit the patterns of formulae. In particular, at this stage, disjunction and negation are omitted from the translation; neither is sequence-related expressiveness covered here.

7. Related Work

We discuss related work on i) classification of CX/BX, ii) formalization of transformations, and iii) megamodeling (and megamodeling languages, specifically).

7.1 Classification of CX/BX

In [32], within the MDE context, the space of design choices for bidirectional transformations is clarified and visualized in the form of a feature model. There are these top-level features: technological space, correspondence (such as forms of defining consistency), changes (such as support for different kinds of changes and representation of changes), and execution (such as distinguishing checking from enforcement or resolving choices automatically or with the help of the user). It may be worthwhile to use the feature model as a foundation for capturing more patterns of CX and exercising them by megamodel-to-test translation. We consider YAS with its support for higher- and lower-level megamodeling languages LAL and UEBER to be a suitable sandbox for such an endeavor.

In [20], based on the mathematical model of delta lenses, 16 types and 44 more concrete forms of bidirectional model synchronization are identified based on three overall questions. (i) Does either model (one of two sides) have non-trivial private updates (i.e., updates that destroy consistency)? (ii) How many types of updates on one model are propagated to the other model? (iii) Is update propagation incremental? The actual taxonomy is based on a 3D space with dimensions for organizational symmetry (such as the matter of direction), informational symmetry (such as the matter of source versus view), and incrementality. A comparison with our patterns is not straightforward; the two patterns for co-transformations with intended applications to coevolution seem to be out of scope of the taxonomy.

We also refer to [3, 49] for earlier semantics-informed discussions of options and issues in model synchronization. Overall, our work does not classify model synchronization options at any level of detail; it merely provides a modeling approach for transformation patterns and exercising them by testing.

There are also efforts underway to collect BX examples in a systematic, case-based manner [2, 9, 56] so that the examples (cases) can form a foundation for comparing (e.g., 'benchmarking') BX approaches and technologies. We hope to contribute to such efforts by suggesting elements of specification and automation.

7.2 Formalization of Transformations

Software or program or model transformations are formally studied in software engineering, software re- and reverse engineering, programming language theory, compiler construction, model-driven engineering, and yet elsewhere. See [1] for a survey on the formalization of transformations: classification is based on three dimensions: the transformations involved, the transformation properties of interest, and the verification techniques used to establish properties. CX and BX are discussed in terms of the properties for consistency that they typically involve. Our work may be considered as an original approach towards capturing transformation properties and validating them by testing. The original aspect is that we approach formalization from a megamodeling (i.e., macroscopic) perspective while also helping with reusing formalized properties for testing. This is not so much meant to be useful for engineering of 'industrial strength' transformations, but it should help with providing 'understandable' models of representative (illustrative) transformations.

7.3 Megamodeling

Most of the megamodels in the literature leverage notations that are essentially invented for the presentational purposes in research publications without introducing a proper modeling language to be reused by others; see [4] for many pointers to such uses of megamodeling; see these papers [21, 44, 60] for some concrete examples.

Let us discuss notable examples of actual megamodeling languages. MoScript [35] is a DSL (a scripting language) for querying and manipulating model repositories. The megamodeling language of [53] supports feedback loops in self-adaptive systems. Both of these languages are tailored towards the MDE technological space; they essentially focus on the execution of model transformations. MegaF [33] is a megamodeling-based architectural description framework for managing views, stakeholders, correspondences, concerns, and yet other models. In our ongoing work on the MEGAL language [23, 39], we focus on a specific vocabulary of entities and relationships that is useful in documenting language and technology usage in software systems; its dynamic semantics is tailored towards Java-/Eclipse-based systems. The emerging LAL language of the present paper is distinctively logic-based; it aims at capturing abstract transformation properties without commitment to application domains and technological spaces.

In [19], the edges of megamodels (such as edges for conformance or transformation relationships) are formalized as structured collections of links based on the notions of graphs and graph mappings in the framework of category theory. The specification language at hand, LAL, could be used to specify the underlying structure: references into compound structures, collections of references due to part-whole relationships, and collections of pairs of references due to correspondence relationships between parts.

In [52], a predicative dependently typed calculus is used to formalize well-typedness of a model repository with models, metamodels, transformations, and accompanying conformance and transformation relationships. Our work focuses on properties of CX based on custom predicate logic without the complexity of dependent typing. Our approach does not commit to any technological space. In YAS, we cover text, tree, and graph languages and corresponding transformations.

8. Conclusion

We used predicate logic to capture patterns of coupled transformations (CX). The resulting axiomatization succinctly conveys the principle artifacts, the relevant interpreters of transformations, and the consistency properties of transformation. We believe that this formulation is particularly effective in capturing the essence of CX without in-depth formal treatment of specific forms or aspects of CX, without commitment to specific technological spaces or application domains. By separating out the phase of megamodel-to-test translation, our megamodels can be validated in a specific technological context nevertheless.

To quote [19]: "To be independent of a particular modeling language, typical megamodels reduce relationships between models to unstructured edges encoding nothing but a labeled pair of models." However, the present work shows how a logic megamodeling language (i.e., LAL) can capture abstract properties in a megamodeling domain and how these properties can be demonstrated to be adequate in the sense of testing with the help a (logic programming-based) software language repository (i.e., YAS), subject to an assignment of interpretations to megamodel elements.

In future work, we plan to generalize the megamodel-based testing approach to include metamodel-based test-data generation. On a more routine front, we continue to include CX illustrations into YAS.

Acknowledgment

Much of what is developed in the paper benefitted from collaboration with colleagues and my research students. I would like to thank, in particular, Jean-Marie Favre, Vadim Zaytsev, Andrei Varanovich, Marcel Heinz, and Lukas Härtel. I am also grateful to the ICMT 2016 reviewers who provided valuable input on an earlier version. Finally, I thank the SLE 2016 reviewers' advice which I used in finishing the paper.

References

[1] M. Amrani, B. Combemale, L. Lucio, G. M. K. Selim, J. Dingel, Y. L. Traon, H. Vangheluwe, and J. R. Cordy. Formal Verification Techniques for Model Transformations: A Tridimensional Classification. *Journal of Object Technology*, 14 (3):1:1–43, 2015.

[2] A. Anjorin, A. Cunha, H. Giese, F. Hermann, A. Rensink, and A. Schürr. Benchmarx. In *Proc. Workshops of the EDBT/ICDT 2014*, volume 1133 of *CEUR Workshop Proceedings*, pages 82–86. CEUR-WS.org, 2014.

[3] M. Antkiewicz and K. Czarnecki. Design Space of Heterogeneous Synchronization. In *GTTSE 2007, Revised Papers*, volume 5235 of *LNCS*, pages 3–46. Springer, 2008.

[4] A. H. Bagge and V. Zaytsev. Languages, Models and Megamodels. In *Post-proc. of SATToSE 2014*, volume 1354 of *CEUR Workshop Proceedings*, pages 132–143. CEUR-WS.org, 2015.

[5] M. Barbero, F. Jouault, and J. Bézivin. Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In *Proc. ECBS 2008*, pages 277–286. IEEE, 2008.

[6] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled Schema Transformation and Data Conversion for XML and SQL. In *Proc. PADL 2007*, volume 4354 of *LNCS*, pages 290–304. Springer, 2007.

[7] J. Bézivin, F. Jouault, and P. Valduriez. On the need for Megamodels. In *Proc. of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop*, 2004.

[8] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *European MDA Workshops MDAFA 2003 and MDAFA 2004, Revised Selected Papers*, volume 3599 of *LNCS*, pages 33–46. Springer, 2005.

[9] J. Cheney, J. McKinna, P. Stevens, and J. Gibbons. Towards a Repository of Bx Examples. In *Proc. Workshops of the EDBT/ICDT 2014*, volume 1133 of *CEUR Workshop Proceedings*, pages 87–91. CEUR-WS.org, 2014.

[10] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[11] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proc. ECOC 2008*, pages 222–231. IEEE, 2008.

[12] A. Cicchetti, D. D. Ruscio, L. Iovino, and A. Pierantonio. Managing the evolution of data-intensive Web applications by model-driven techniques. *Software and System Modeling*, 12 (1):53–83, 2013.

[13] A. Cleve and J. Hainaut. Co-transformations in Database Applications Evolution. In *GTTSE 2005, Revised Papers*, volume 4143 of *LNCS*, pages 409–421. Springer, 2006.

[14] A. Cunha and J. Visser. Strongly Typed Rewriting For Coupled Software Transformation. *ENTCS*, 174(1):17–34, 2007.

[15] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva. Embedding, Evolution, and Validation of Model-Driven Spreadsheets. *IEEE Trans. Software Eng.*, 41(3):241–263, 2015.

[16] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Proc. ICMT 2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

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

[18] Z. Diskin, Y. Xiong, and K. Czarnecki. From State- to Delta-Based Bidirectional Model Transformations. In *Proc. ICMT 2010*, volume 6142 of *LNCS*, pages 61–76. Springer, 2010.

[19] Z. Diskin, S. Kokaly, and T. Maibaum. Mapping-Aware Megamodeling: Design Patterns and Laws. In *Proc. SLE 2013*, volume 8225 of *LNCS*, pages 322–343. Springer, 2013.

[20] Z. Diskin, H. Gholizadeh, A. Wider, and K. Czarnecki. A three-dimensional taxonomy for bidirectional model synchronization. *Journal of Systems and Software*, 111:298–322, 2016.

[21] D. Djuric, D. Gasevic, and V. Devedzic. The Tao of Modeling Spaces. *Journal of Object Technology*, 5(8):125–147, 2006.

[22] A. Etien and C. Salinesi. Managing Requirements in a Co-evolution Context. In *Proc. RE 2005*, pages 125–134. IEEE, 2005.

[23] J. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.

[24] J.-M. Favre. Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[25] J.-M. Favre. Foundations of meta-pyramids: Languages vs. metamodels – Episode II: Story of thotus the baboon. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[26] R. Fentes-Fernández, J. Pavón, and F. J. Garijo. A model-driven process for the modernization of component-based systems. *Sci. Comput. Program.*, pages 247–269, 2012.

[27] S. Fischer, Z. Hu, and H. Pacheco. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences*, 58(5):1–21, 2015.

[28] J. Hainaut. The Transformational Approach to Database Engineering. In *GTTSE 2005, Revised Papers*, volume 4143 of *LNCS*, pages 95–143. Springer, 2006.

[29] J. Hainaut, A. Cleve, J. Henrard, and J. Hick. Migration of Legacy Information Systems. In *Software Evolution*, pages 105–138. Springer, 2008.

[30] M. Herrmannsdoerfer, S. Benz, and E. Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In *Proc. ECOOP 2009*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[31] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *Proc. ASE 2011*, pages 480–483. IEEE, 2011.

[32] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based classification of bidirectional transformation approaches. *Software & Systems Modeling*, pages 1–22, 2015.

[33] R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing architecture frameworks through megamodelling techniques. In *Proc. ASE 2010*, pages 305–308. ACM, 2010.

[34] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[35] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. MoScript: A DSL for Querying and Manipulating Model Repositories. In *Proc. SLE 2011*, volume 6940 of *LNCS*, pages 180–200. Springer, 2012.

[36] R. 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`.

[37] R. Lämmel and E. Meijer. Mappings Make Data Processing Go 'Round. In *GTTSE 2005, Revised Papers*, volume 4143 of *LNCS*, pages 169–218. Springer, 2006.

[38] R. Lämmel and E. Meijer. Revealing the X/O Impedance Mismatch - (Changing Lead into Gold). In *Datatype-Generic Programming - International Spring School, SSDGP 2006, Revised Lectures*, volume 4719 of *LNCS*, pages 285–367. Springer, 2007.

[39] R. Lämmel and A. Varanovich. Interpretation of Linguistic Architecture. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 67–82. Springer, 2014.

[40] R. Lämmel and V. Zaytsev. Recovering grammar relationships for the Java Language Specification. *Software Quality Journal*, 19(2):333–378, 2011.

[41] T. Massoni, R. Gheyi, and P. Borba. Synchronizing Model and Program Refactoring. In *Revised Selected Papers SBMF 2010*, volume 6527 of *LNCS*, pages 96–111. Springer, 2011.

[42] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proc. ICFP 2007*, pages 47–58. ACM, 2007.

[43] J. Mendes. Coupled evolution of model-driven spreadsheets. In *Proc. ICSE 2012*, pages 1616–1618. IEEE, 2012.

[44] B. Meyers and H. Vangheluwe. A framework for evolution of modelling languages. *Sci. Comput. Program.*, 76(12):1223–1246, 2011.

[45] J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Dealing with the Coupled Evolution of Metamodels and Model-to-text Transformations. In *Proc. Workshop on Models and Evolution*, volume 1331 of *CEUR Workshop Proceedings*, pages 22–31. CEUR-WS.org, 2015.

[46] D. D. Ruscio, R. Lämmel, and A. Pierantonio. Automated Co-evolution of GMF Editor Models. In *Proc. SLE 2010*, volume 6563 of *LNCS*, pages 143–162. Springer, 2011.

[47] D. D. Ruscio, L. Iovino, and A. Pierantonio. Coupled evolution in model-driven engineering. *IEEE Software*, 29(6): 78–84, 2012. doi: 10.1109/MS.2012.153. URL `http://dx.doi.org/10.1109/MS.2012.153`.

[48] H. Schink, M. Kuhlemann, G. Saake, and R. Lämmel. Hurdles in Multi-language Refactoring of Hibernate Applications. In *Proc. ICSOFT 2011*, pages 129–134. SciTePress, 2011.

[49] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

[50] T. van der Storm. Semantic deltas for live DSL environments. In *Proc. LIVE 2013*, pages 35–38. IEEE, 2013.

[51] S. Vermolen and E. Visser. Heterogeneous Coupled Evolution of Software Languages. In *Proc. MoDELS 2008*, volume 5301 of *LNCS*, pages 630–644. Springer, 2008.

[52] A. Vignaga, F. Jouault, M. Bastarrica, and H. Brunelière. Typing Artifacts in Megamodeling. *Software and Systems Modeling*, pages 1–15, 2011. ISSN 1619-1366.

[53] T. Vogel and H. Giese. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *Proc. SEAMS 2012*, pages 129–138. IEEE, 2012.

[54] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proc. ECOOP 2008*, volume 4609 of *LNCS*, pages 600–624. Springer, 2007.

[55] B. Wang, Z. Hu, Q. Sun, H. Zhao, Y. Xiong, W. Zhang, and H. Mei. Supporting feature model refinement with updatable view. *Frontiers of Computer Science*, 7(2):257–271, 2013.

[56] B. Westfechtel. Case-based exploration of bidirectional transformations in QVT Relations. *Software & Systems Modeling*, pages 1–41, 2016.

[57] M. Wimmer, N. Moreno, and A. Vallecillo. Systematic Evolution of WebML Models by Coupled Transformations. In *Proc. ICWE 2012*, volume 7387 of *LNCS*, pages 185–199. Springer, 2012.

[58] M. Wimmer, N. Moreno, and A. Vallecillo. Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations. In *Proc. TOOLS 2012*, volume 7304 of *LNCS*, pages 336–352. Springer, 2012.

[59] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In *Proc. ICSE 2012*, pages 540–550. IEEE, 2012.

[60] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *Proc. MODELS 2014*, volume 8767 of *LNCS*, pages 50–67. Springer, 2014.