Coupled Software Transformations

— Extended Abstract —

Ralf Lämmel

VUA & CWI, Amsterdam, The Netherlands

Abstract

We identify the category of coupled software transformations, which comprises transformation scenarios involving two or more artifacts that are coupled in the following sense: transformation at one end necessitates reconciling transformations at other ends such that global consistency is reestablished. We describe the essence of coupled transformations. We substantiate that coupled transformation problems are widespread and diverse.

1. De nition of the subject matter

Everyone is used to two common categories of software transformations: type-preserving transformations (also called rephrasing in [15], but other terms are around as well) vs. type-changing transformations (mostly called translations in the literature). We use the term *type* as a placeholder for format, grammar, schema, language, meta-model, and others. (Here, we do not necessarily restrict ourselves to context-free structure.) A *type-preserving transformation* asserts the same type for input and output. For instance, program optimisers and program normalisers are of that kind. A *type-changing transformation* maps data according to one type to data according to another type. For instance, language compilers, application generators, and PIM-to-PSM transformations in MDA are of that kind.

Many transformation scenarios are of a more complex shape. Most notably, one often ends up wanting coupled transformations — the subject of this paper. By this, we mean that ...

> *... two or more artifacts of potentially different types are involved, while transformation at one end necessitates reconciling transformations at other ends such that global consistency is reestablished.*

We note that any kind of coupled transformation problem must instantiate the notion of *consistency* for the involved kinds of artifacts. A coupled transformation starts from and nishes with a consistent conglomeration of artifacts. Any kind of coupled transformation problem must also instantiate the notion of *reconciliation*, which de nes how transformations of one artifact affect all the other artifacts.

We emphasise that we use the term *software transformation* rather than the more restrictive term program transformation. That is, we do not restrict ourselves to the transformation of source code. Hence, transformations of data, non-executable speci cations, grammars, meta-models, documentation, and other artifacts are included.

We also emphasise that we do not restrict ourselves to information-preserving (or semantics-preserving) transformations because enhancements or reductions should be included as well. For instance, *evolutionary transformations* require such generality; see the various transformation properties for (rule-based) programs in [19].

2. An example related to software evolution

Consider an information system that uses a relational database for data management, and all functionality is implemented in 2nd – 4th generation languages ("2–4GLs"). We are faced with artifacts such as the following: the relational model, which is implemented as the database schema in the database; 4GL forms for user-interface components; 4GL reports; SQL or PL/SQL fragments that contribute to 4GL sources; embedded SQL code in the 2–3GL code; 2–3GL programs with data structures that rehash some of the relational model.

There are various evolution scenarios that call for coupled transformations. For instance, we might face a change request that is phrased as a modi cation at the level of the database schema. This primary modi cation must be completed by a database instance mapping. Furthermore, all the 2–4GL code is likely to require modi cation as well. So we have to adapt SQL and PL/SQL snippets, we have to adapt embedded SQL code, and even native 2–4GL code because of the way it commits to the database schema.

All these adaptations are coupled. Consistency is here about the use of the *same* relational model in the different code artifacts. So this is de nitely a coupled transformation *problem*. It is a different question whether or not one succeeds to provide an effective implementation of the scenario, which would need to include an operational reconciliation for all the code artifacts relative to an evolving database schema.

3. The purpose of this text

The general category of coupled software transformations has not been identi ed previously — even though speci c transformation techniques do exist, and quite some amount of related research is being pursued; see, e.g., [5, 11, 26, 29, 22, 17, 10]. So giving nally a name to this category is perhaps useful as such. We go further than that: in Sec. 4, we will describe the essence of coupled transformations.

Coupled transformation problems are ubiquitous; they are encountered in various disciplines of computer science, e.g., in language processing, generative programming, automated software engineering, software re-engineering, model-driven architecture, and database re-engineering. In Sec. 5, we will enumerate some problem domains in which coupled transformations are relevant. The understanding of coupling differs radically for these domains.

4. The essence of coupled transformations

Without loss of generality, we will consider reconciliation for *two* artifacts. Let A and B be the types of the artifacts. We assume a consistency relation c on A and B. We are given two concrete artifacts $a : A$ and $b : B$ such that $c(a, b)$ holds. We consider a type-preserving transformation on A, denoted by g, and we apply this transformation to a such that we obtain $a' = g(a)$. Then, the reconciliation issue is about determining a suitable b' such that $c(a', b')$ holds. We summarise selected reconciliation options in Fig. 1; continuous arrows visualise transformations; dashed arrows visualise consistency claims.

The rst option in the gure, *no reconciliation*, is merely there to provide a good starting point for the discussion. If g is known to be restricted such that a is changed without challenging consistency, then we can just keep b — as is. For instance, using SQL's data manipulation language on a database instance does not affect the underlying database schema. So this sort of restricted instance transformation does not trigger a schema transformation. Clearly, the inverted situation, where the database schema is transformed, is not covered by this trivial option.

The second option in the gure, *degenerated reconciliation*, is still trivial. We assume that the concrete artifacts of type B are derivable from the concrete artifacts of type

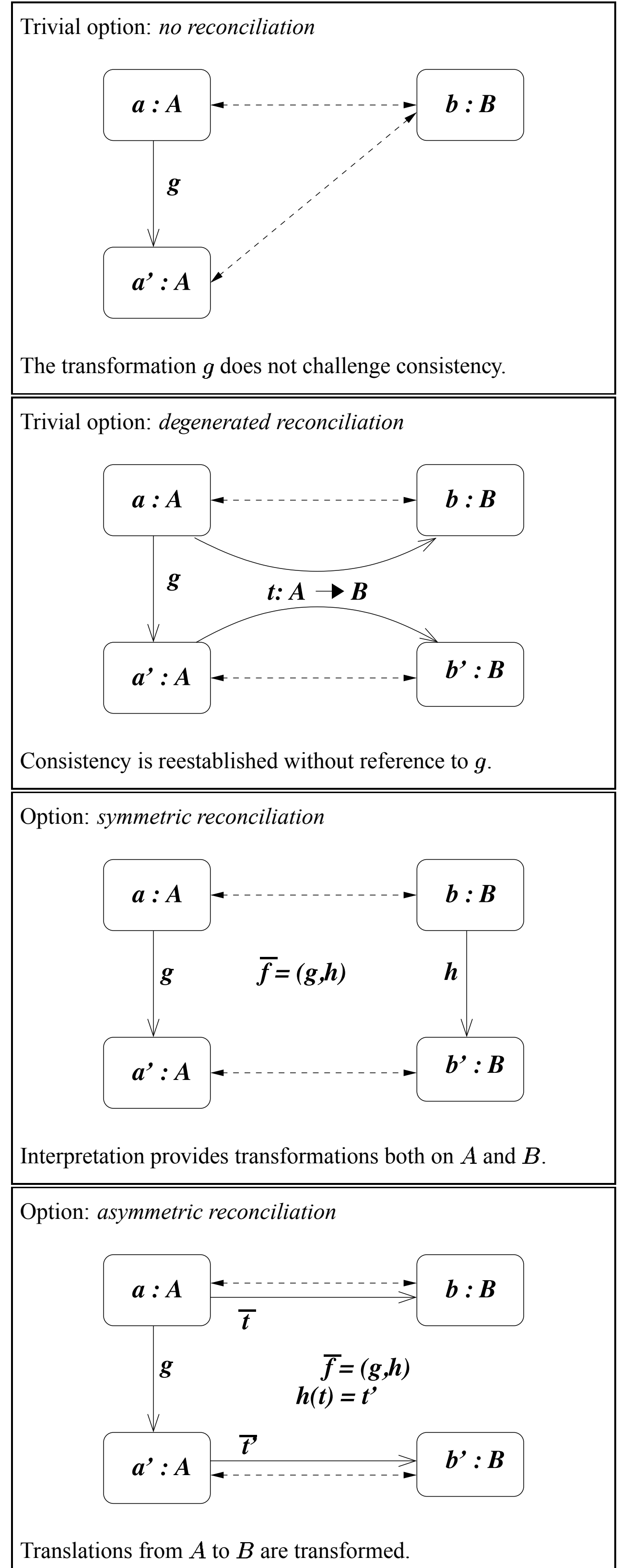

Figure 1. Selected reconciliation options

A — by means of a translation t. For instance, consider the implementation of a domain-speci c language (DSL) via code generation. In a simple case, we can transform the DSL code, and the generated code is simply regenerated. However, if the generated code can be customised, and such adaptations are required to survive regeneration, then proper reconciliation has to be faced.

The third option in the gure, *symmetric reconciliation*, covers a meaningful subcategory of coupled transformations. Here we start from a transformation *description* f, which is phrased in a transformation language. The interpretation of the description f, denoted as $\overline{f}$, provides two actual transformations, one on A, and another on B. For instance, the reconciliation of a database instance in reply to adaptations of a database schema can be covered by this option [12]. (Likewise, XML documents must be updated when the underlying DTDs or XML schemas are adapted [21].) The applicability of this option requires a genuine de nition of the transformation language. A critical issue is the inclusion of all controls into f that are eventually needed for either A or B. For instance, transforming a database schema, such that a NOT NULL column is added, requires the speci cation of a default value — even though this value is not essential for the schema transformation, but only for the instance mapping.

The fourth option in the gure, *asymmetric reconciliation*, is based on the assumption that we have access to an actual translation of a to b. In fact, we require again a *description* t of this translation in terms of a dedicated transformation language. Then, the actual translation from a to b is the interpretation $\overline{t}$ of the description t. The description t can be seen as a means to capture the derivation history of b when related to a. Furthermore, this translation t manifests the consistency of a and b. As in the case of symmetric reconciliation, f is again phrased in a transformation language. This time, the interpretation of f, provides an actual transformation on A and a transformation on translation descriptions (such as t). The aforementioned survival problem of customisations in code that is generated from a DSL program is handled by this option. That is, we use a translation description to maintain a link between DSL code and generated code. The translation description records the customisation of the generated code. When the DSL program is adapted, the recorded customisations can be re-executed (modulo adaptation) on the regenerated code. This option involves two critical issues. Firstly, we have to identify a language for translation descriptions that is t for modelling the derivation of artifacts according to the problem domain at hand. Secondly, we have to de ne the primary transformation language such that its effect can also be transposed to the translation descriptions.

We make no claim of completeness regarding this list of options. For instance, one can think of *reconciliation by matching*, where the mere consistency relation is complemented by a metric that measures the consistency "distance" between two artifacts. That is, given a', and its consistency distance from b, we could gradually adapt b until full consistency is reestablished. We favour symmetric and asymmetric reconciliation because these options inherently employ the structure of the primary transformation. However, even this criterion might be amenable to other realisations. Also, there are presumably a number of possible re nements for the two favoured options, and they might also be mixed. Future work is needed to deliver a comprehensive set of more detailed options.

5. Typical problem domains

We will now list scenarios for coupled transformations. We identify the artifacts and transformations of interest as well as the relevant instances of the notions consistency and reconciliation. This list is by no means complete. The degree of the formal and technical mastery of coupled transformations differs very much per problem domain. An example of a well-understood kind of coupled transformation is the joint transformation of database schema and database instance during database re-engineering [12]. By contrast, the techniques for updating language processors for programming languages in reply to an evolving syntax or semantics are still to be discovered; see [20] for some ideas.

A list of coupled transformations scenarios follows.

Consistency maintenance in cooperative editing Distributed editing of the same content requires synchronisation [7, 28]. Depending on details, either the local copies of the content or the local session state are considered the artifacts subject to coupled transformation. There can be any number of such artifacts, but they happen to be all of the same type. Editing actions de ne the primary transformation language. Reconciliation of a given user view means to incorporate all pending editing actions that were emitted by other users. Consistency means that all views agree on the content. A speci c challenge is that remote editing actions should be incorporated only at de nite points in time, when their effect on the local view and any necessary con ict resolution will be acceptable for the user.

Consistency maintenance in software modelling In software modelling with UML one uses different kinds of structural and behavioural diagrams such as use case diagrams, class diagrams, state diagrams, and sequence diagrams. Consistency of a multi-diagram software model means that the different diagrams do not disagree on each other in those areas where they overlap [18, 14]. Consistency must be maintained along the evolution of UML

models. As a simple form of transformation, one can consider refactorings of UML class diagrams. (Refactorings were originally introduced for the transformation of object-oriented programs, but they have been instantiated for UML diagrams as well [4, 25].)

Co-evolution of design and implementation The system design and the actual implementation should be coupled throughout continuous system maintenance and enhancement. The idea of a methodology and technology for co-evolution is that the coupling must be operationalised or at least checked [6, 31, 8] because design and implementation diverge otherwise. One way to operationalise the coupling is to generate the implementation from the design, modulo provisions for allowing editing at the source-code level and for pushing back implementational changes into the design. The operationalisation can also work the other way around if the additional design information becomes an integral part of the actual code.

View-update translation When updates are allowed at the level of database views, then such updates need to be translated back to the underlying database [2, 9]. Such view-update translation is clearly an instance of asymmetric reconciliation. The view-update problem for databases has been generalised in the recent work on bidirectional transformations between data representations of different levels of abstraction [10, 13] (cf. concrete and abstract views). The view-update problem has also been encountered, in some form, in functional programming, when pattern matching and building is to be provided for abstract datatypes rather than concrete algebraic datatypes [30, 3, 23].

Intentional programming and uid AOP In Simonyi's intentional programming [27, 1], the programmer can specify domain-speci c abstraction forms, while simultaneously recording domain-speci c optimisations that may apply to such new abstractions. Programmers can browse and edit (or transform) programs using different syntaxes enabled by the competing abstraction forms. An intentional programming system would take care of the coupling between the external syntaxes and the internal abstract syntax. A similar situation applies to uid AOP [16] — a strong form of aspect-oriented programming, where programs cannot just be edited to ful l crosscutting concerns, but programs can even be re-sliced according to different views.

Reconcilable model transformation According to OMG's model-driven architecture (MDA [24]), software development starts from a platform-independent software model (PIM), which is then re ned into a platform-speci c model (PSM) by semi-automatic transformations. To this end, model-driven approaches employ meta-models for platform-independent models, for platforms, for platform-speci c models. MDA approaches also tend to employ annotations for driving the the PIM-to-PSM mappings. The basic MDA approach emphasises the operationalisation of the PIM-to-PSM mapping, which is a type-changing transformation. A strong version of MDA would require coupling between all involved models and meta-models [8]. For instance, the modi cation of a platform model should allow for the reconciliation of all actual PIMs that refer to this platform.

Representations in software re-/reverse engineering Software reverse engineering employs problem-oriented abstraction layers, starting from a low-level source-code model, with less code- or language-speci c representations in between, and possibly complemented by high-level architectural descriptions at the top. Coupling concerns the mappings between the layers. These mappings need to be traceable in order to enable the navigation between layers. Likewise, software re-engineering can take advantage of extra intermediate program representations, e.g., PDG or SSA for control o w or data o w dependencies. When re-engineering transformations are expressed at the level of intermediate formats, then these transformations still need to be mapped back to the concrete source code. Yet other forms of coupling deal with the preservation of preprocessing directives, and other low-level source-code properties.

6. Final remark

We have identi ed the notion of coupled software transformations. We have collected and integrated some basic material on the subject, while we have refrained from a discussion of technical details as they arise in speci c coupled transformation scenarios and speci c conceptual frameworks for coupled transformations.

It is very rewarding to understand that software transformations can exhibit more structure than being organised in terms of type-preserving or type-changing functions. We can have transformations on transformations on ...

Acknowledgement

The author gratefully acknowledges inspiring discussions with James R. Cordy and Andreas Winter in the context of designing the Dagstuhl seminar 05161 "Transformation techniques in software engineering".

References

[1] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional

programming. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 114–123. IEEE Computer Society Press, 1998.

[2] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[3] F. Burton and R. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[4] G. Butler and L. Xu. Cascaded refactoring for framework. In *Proc. Symposium on Software Reusability*, pages 51–57. ACM Press, 2001.

[5] A. van Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[6] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of Object-Oriented Software Design and Implementation. In *Proc. International Symposium on Software Architectures and Component Technology 2000*, 2000.

[7] C. Ellis, S. Gibbs, and G. Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[8] J.-M. Favre. Meta-models and Models Co-Evolution in the 3D Software Space. In *Proc. International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA'03)*, 2003.

[9] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.

[10] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report MS-CIS-03-08, University of Pennsylvania, 2003. Revised April 2004.

[11] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Schema Transformation Techniques for Database Reverse Engineering. In *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, 1993. E/R Institute.

[12] J. Henrad, J.-M. Hick, P. Thiran, and J.-L. Hainaut. Strategies for Data Reengineering. In *Proc. Working Conference on Reverse Engineering (WCRE'02)*, pages 211–220. IEEE Computer Society Press, 2002.

[13] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proc. ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM Press, 2004.

[14] Z. Huzar, L. Kuzniarz, G. Reggio, J. Sourrouille, and M. Staron. Consistency Problems in UML-based Software Development II, 2003. Workshop proceedings; Research Report 2003:06.

[15] M. d. Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. v. d. Brand and D. Parigot, editors, *Proc. Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier Science, 2001.

[16] G. Kiczales. The Fun has Just Begun. AOSD'03 Keynote Address, available from `http://www.cs.ubc.ca/~gregor`, 2003.

[17] J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proc. International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, Amsterdam, 2003. IEEE Computer Society Press.

[18] L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar. Consistency Problems in UML-based Software Development, 2002. Workshop proceedings; Research Report 2002:06.

[19] R. Lämmel. Evolution of Rule-Based Programs. *Journal of Logic and Algebraic Programming*, 60–61C:141–193, 2004. Special Issue on Structural Operational Semantics.

[20] R. Lämmel. Evolution scenarios for rule-based implementations of language-based functionality. In L. Aceto, W. Fokkink, and I. Ulidowski, editors, *Proc. Workshop on Structured Operational Semantics (SOS'04)*, ENTCS. Elsevier, 2004. 20 pages. To appear.

[21] R. Lämmel and W. Lohmann. Format Evolution. In J. Kouloumdjian, H. Mayr, and A. Erkollar, editors, *Proc. Re-Technologies for Information Systems (RETIS'01)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[22] A. Malton, K. Schneider, J. Cordy, T. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *Proc. International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society Press, May 2001.

[23] G. S. Novak Jr. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.

[24] OMG. Model Driven Architecture, 2001–2004. web portal `http://www.omg.org/mda/`.

[25] K. Rui and G. Butler. Refactoring use case models: the metamodel. In *Proc. Twenty-sixth Australasian computer science conference on Conference in research and practice in information technology*, pages 301–308. Australian Computer Society, Inc., 2003.

[26] A. Schürr. Speci cation of Graph Translators with Triple Graph Grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, 16–18 June 1994. Springer-Verlag.

[27] C. Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., Sept. 1995. Available from `http://citeseer.nj.nec.com/simonyi95death.html`.

[28] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, 1998.

[29] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *LNCS*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[30] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. Principles Of Programming Languages (POPL'87)*, pages 307–313. ACM Press, 1987.

[31] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.