Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 52 (2004) 1–8
Science of Computer Programming
www.elsevier.com/locate/scico

Editorial

Transformations everywhere

Abstract

This special issue is devoted to "program transformation" in the sense of *tool-supported adaptation of software systems*. Software engineering and software re-engineering rely on such transformations, which are automated in, for example, tools for refactoring, migration, program specialisation, compiler optimisation, database re-engineering, software configuration, business-rule extraction, aspect weaving, aspect mining, architectural modifications, and model-driven approaches.

This special issue bundles ten state-of-the-art contributions, while covering the broad area of program transformation in a complementary, almost survey-like manner. Three papers relate to refactoring—to the composition problem, to reasoning about correctness, and to the details of challenging refactoring samples. Two papers survey successful transformation systems, namely the Tempo system for program specialisation, and the FermaT system for software migration. One paper develops concepts for run-time system transformations. Finally, four papers communicate idioms or concepts for transformation systems: higher-order and dynamic traversals, the use of flow analysis for driving transformations, validated compiler transformations, and the cause–effect patterns in partial evaluation.

This introduction to the special issue briefly describes the articles included, and connects them to general concerns in research on program transformation. In addition, a list of research challenges is compiled, which will perhaps be useful in the further exploration of the area of program transformation.

1. Where to begin?

This special issue focuses on program transformations as they are used in software engineering and re-engineering. The use of the term "program transformation" deserves some clarification since it is highly overloaded in computer science:

- It is used in the sense of *transformational program development*, where presumably efficient programs are formally derived from high-level specifications (or less efficient programs) in a semantics-preserving manner; cf. the famous CIP project and the textbook "Specification and Transformation of Programs" by Helmuth A. Partsch, Springer-Verlag, 1990.
- Another kind of program transformations concerns *tool-supported transformations* of software systems that are performed more or less *silently and generically*—just like

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.03.001

the optimisations in a compiler. This category also includes fully automated program specialisation, aspect weaving, and conversion.
- Yet another kind of program transformations concerns *tool-supported adaptations* that are specifically *initiated by the software engineer* subject to a more or less detailed description of the relevant adaptation. This category includes automated refactoring, project-specific software modification in re-engineering, product-line specialisation, and evolutionary transformations. These adaptations may or may not be semantics-preserving. Such adaptations are performed by means of source-code transformation, template instantiation, compile-time or run-time reflective programming, and other methods.

Program transformations in the sense of transformational program development are perhaps beyond the scope of this special issue, which focuses on (applied) software engineering. The other two categories (and mixtures thereof), which deal with different kinds of tool-supported transformations, are covered by the special issue to quite some extent. This is motivated by the fact that such program transformations (or translations, or adaptations) are omnipresent in contemporary software development methodologies. The prime concerns in this research context are the following:

- Notation and idioms for writing meta-programs for program transformation.
- Means of reasoning about programs and meta-programs.
- The important application domains for such program transformations.
- Technology and frameworks for implementing and deploying transformations.

2. Articles included in the special issue

- *"Static composition of refactorings" by Günter Kniesel and Helge Koch*. This work relates to Roberts' well-known work on composition of refactorings; cf. "Practical Analysis for Refactoring" by Donald Bradley Roberts, Ph.D. Thesis, University of Illinois, 1999. The present work contributes a formal approach to the static composition of conditional transformations including refactorings. The approach facilitates the static derivation of a single joint pre-condition from the pre-conditions of the transformations that are composed. The key idea is to propagate pre-conditions backwards through previous transformations. The authors demonstrate the effectiveness of the approach by reference to Java-oriented tool support for conditional transformation and refactoring.
- *"Algebraic reasoning for object-oriented programming" by Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio*. This work relates to the established algebraic style of reasoning about program properties; cf. "Laws of Programming" by C.A.R. Hoare et al., Communications of the ACM, 1987. The work presents algebraic laws for (a subset of) Java. These laws and corresponding transformation primitives are readily useful for the derivation of provably correct refactorings, which is demonstrated by the authors. The laws are shown to be sound, and they are also shown to be complete in the sense that the laws suffice for arriving at a certain normal form. This status makes them very valuable for the design of future refactoring frameworks.
- *"Monadification of functional programs" by Martin Erwig and Deling Ren*. This work relates to the "Essence of functional programming" according to Philip Wadler

(cf. conference record of POPL'92); to the fundamental "Notions of Computation and Monads" according to Eugenio Moggi's ground-breaking work (cf. Information and Computation, 1991). The authors describe the transformation of a functional program to use monadic style for effects such as states, exceptions, and I/O. The authors provide the first detailed, technical treatment of monadification (or "monad introduction" as coined by this guest editor elsewhere). Monadification involves two aspects: the refactoring aspect of switching to monadic style and the extension aspect of inserting monadic actions for extended behaviour.

- *"Supporting incremental and experimental software evolution by run-time method transformations" by Uwe Zdun*. This work relates to current trends for unanticipated software evolution (cf. http://joint.org/use/) and dynamic weaving as a special theme within aspect-oriented programming (cf. http://aosd.net/). The present work focuses on the conceptual support of run-time method transformations, while addressing incremental and experimental scenarios in re-engineering and evolution. To this end, a non-intrusive model for method transformations is developed, and a set of transformation primitives is identified. Also, a pattern language is provided that can be used to implement dynamic method abstractions in a range of programming languages.
- *"The transient combinator, higher-order strategies, and the distributed data problem" by Victor L. Winter and Mahadevan Subramaniam*. This work relates to strategic programming, which is about designing and implementing term traversals that are inherent to program transformations and analyses. The authors contribute a higher-order and dynamic notion of traversals. These new idioms concern the manipulation of distributed data, i.e., semantically related data that is not stored contiguously in a term. The new idioms allow for the manipulation of the distributed data to be expressed directly in terms of strategies. This is in contrast to existing approaches in which the distribution of data is based on the use of explicitly extracted parameters or dynamically created rule bases. The approach is illustrated for several transformation scenarios.
- *"Pigs from sausages? Re-engineering from assembler to C via FermaT transformations" by Martin P. Ward*. This work provides a comprehensive presentation of the FermaT transformation system, which is used by Software Migrations Ltd. in industrial projects for a while now. The foundational presentation of FermaT is complemented by the nuts and bolts of a success story: a re-engineering project in the telecommunications domain that required migrating 544,000 lines of assembler code to high-level, structured, maintainable C code. The FermaT system uses formally proved program transformations that preserve or refine the semantics of a program while changing its form. FermaT provides a layered language MetaWSL for writing transformations.
- *"Automatic validation of code-improving transformations on low-level program representations" by Robert van Engelen, David Whalley, and Xin Yuan*. This work makes important progress in the field of validated compilations. The authors provide means of ensuring the correctness of compiler optimisations and hand-specified modifications at the level of machine instructions. The method is based on the derivation of semantic effects from machine instructions, where the effects are supposed to be unchanged for a semantics-preserving transformation. The authors demonstrate the

effectiveness with a validating compilation system, which is able to validate traditional compiler transformations, but also transformations that modify the branch structure of a program.

- *"Type-safe method inlining" by Neal Glew and Jens Palsberg*. This work tackles the intriguing problem of "type rot", that is, when extra effort is needed to preserve typeability of (object-oriented) programs in the course of program transformations. The authors discuss these circumstances for method inlining, which is an important transformation performed by optimising compilers, but, in fact, it is a refactoring, too. The conservative approach to the recovery of typeability employs type casts, which may affect performance. By contrast, the authors describe an approach that is based on the transformation of type annotations, which never hurts performance. Technically, the authors demonstrate the utility of flow analysis, i.e., an approximation of expression evaluation, which is used to drive transformations of types and expressions. The authors pay attention to the peculiarities of Java's type system.
- *"Transformation by interpreter specialisation" by Neil D. Jones*. This paper communicates know-how in partial evaluation written down by one of most influential researchers in this 20-year-old field. The paper describes cause–effect patterns for partial evaluation, with an emphasis on interpreter specialisation. That is, a program is transformed by specialising an interpreter for the language in which it is written. The key observation is here that the form of transformation can be controlled via the style of the interpreter (CPS, tail-recursive, etc.). The cause–effect patterns also explain how to write an interpreter so that specialisation terminates and produces efficient transformed programs. The key observation is here that the efficiency of the transformed program is determined by the efficiency of the interpreter's dynamic operations.
- *"A tour of Tempo: a program specializer for the C language" by Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur*. This work is a survey on Tempo—a success story on partial evaluation. Tempo is a powerful and mature specialiser for the C language. Tempo offers specialisation at both compile time and run time, and both program and data specialisation. To control the specialisation process, Tempo provides the program developer with a declarative language for describing specialisation opportunities for a given program. The design of Tempo has been driven by the needs of practical applications in areas such as operating systems and networking. In fact, these application domains triggered dedicated specialisation techniques. The paper gives the ultimate overview of the design of Tempo and its applications.

3. Where to go from here?

In the following, a list of research challenges in program transformation is compiled. This list attempts to align needs for a more automated, more agile software engineering with intriguing, general research topics.

- *Transformations for evolution*. Research on program transformation has historically focused on semantics-preserving transformations and program refinements—be it in the context of transformational program development or program optimisation. Software evolution provides various challenges. For instance, the following concerns

are not yet precisely understood: properties that complement semantics preservation, operator suites that describe program evolution, abstraction techniques that allow for quantification of preserved versus modified program properties.

- *Trustable weavers*. The implementation of cross-cutting concerns in the sense of aspect-oriented software development or invasive software composition requires trustable weavers—if the benefits of these methodologies should be leveraged for dependable and safety-critical systems. There is a need for scalable methods for modular reasoning, safe system transformation, rigorous and scalable validation and verification. The FOAL workshop series addresses some of these concerns. (Cf. http://www.cs.iastate.edu/~leavens/FOAL/—Foundations of Aspect-Oriented Languages.)
- *Unweaving transformations*. A very promising option for the improvement of existing software assets is the deployment of sophisticated transformations for recovering modular system structure, including means for mining aspects (i.e., cross-cutting concerns). This area is not so much challenged by the mere transformations, but rather by the required system analyses including heuristics that are needed to steer transformations. Research in this field is carried out in new aspect mining projects at some places, e.g., at the Software Evolution Research Laboratory, TU Delft; cf. http://swerl.tudelft.nl/.
- *Language-parametric transformations*. While research on the semantics of programming languages has revealed quite some reusable language concepts (cf. action semantics, abstract state machines, monadic denotational semantics, modular attribute grammars, and others), the comprehensive identification of general, language-parametric concepts for program transformation is still to be completed. There is ongoing research in this field, e.g., a project on language-parametric program restructuring at the CWI and the Free University in Amsterdam; cf. http://www.cs.vu.nl/lppr/.
- *Multi-lingual transformations*. Software applications tend to involve several programming languages, embedded languages, domain-specific notations, APIs, or schemas. The transformation of multi-lingual applications requires a firm understanding and operationalisation of mappings between the languages involved. There are fields such as software migration, meta-modelling and grammarware engineering that already contribute to an emerging discipline of multi-lingual transformations, but the discipline lacks foundations and engineering methods.
- *Co-transformations*. The following definition is presumed: A *co-transformation* transforms mutually dependent software artifacts of different kinds simultaneously, while the transformation is centred around a grammar (or schema, API, or a similar structure) that is shared among the artifacts. A specific kind of co-transformation is found in the work of Jean-Luc Hainaut et al. work on database schema transformation coupled with database instance migration. Research is needed to provide a general conceptual framework for co-transformations.
- *Co-evolution*. Another form of joint transformations describes different abstraction layers such as design versus implementation. (Cf. Jean–Marie Favre's inspirational view "Meta-model and Model Co-Evolution in the 3D Software Space", presented at ELISA 2003; cf. work done in the Progr. Tech. Lab., Vrije Universiteit Brussel, e.g., the

Ph.D. Thesis by Roel Wuyts, 1999.) Such forms of transformations have the potential of leading to a form of software development that largely abstracts from implementation vehicles. This field is in its early infancy.

- *Run-time transformations*. Application domains such as telecommunication, mobile computing, management information, and e-business services show an increasing demand of both system availability ("no shutdown–transform–start-up cycles") *and* system adaptability. This implies a need for run-time system adaptation. Zdun's paper in this special issue contributes to this field of research. In general, run-time transformations must become (more) reliable, predictable, reversible, traceable, comprehensible, and scalable. The ultimate vision is to largely eliminate the distinction of source-code adaptation versus run-time adaptation.
- *Disciplined meta-programming*. For the most part, real-life program transformations are encoded in rewrite rules, visitors, XSLT, or some other free-wheeling notation that it is not tightly aligned with the type system and the semantics of the object language at hand. Examples of more disciplined approaches include Kniesel and Koch's work as presented in this special issue, Erwig's work on an update calculus as well as hygienic and type-aware macro facilities as in Template Haskell. Further research on operator suites and frameworks is needed to capture disciplined modes of meta-programming without affecting simplicity, generality, and automation in an undue manner.
- *Transformation systems*. There will undoubtedly be more work on deploying methods and techniques for program transformation by means of transformation environments that are readily useful for software engineers. Current systems such as ASF + SDF, DMS, Progress, RainCode, RECODER, Stratego, Strafunski and TXL exhibit different trade-offs with regard to external notation, internal representations, complexity, typing, abstraction mechanisms, learning curve, and other factors. Such systems face new application domains: refactoring, aspect weaving and mining, XML processing, MDA-like transformations, and architectural modifications of deployed software.

4. Special issue—statistics

This special issue received 25 submissions. The ten selected papers represent the contributions that are best aligned with the focus of this special issue, while they also adhere to the required, high standards for an archival publication. All ten selected papers were revised properly after the first round of reviewing. For five of the ten selected papers, initial acceptance was conditional, which implied a second round of formal reviewing. There were 63 referees involved in the two rounds of reviewing. The original deadline for submission was April 1, 2003, but deadline extensions were granted generously. All final versions of accepted papers were available by February 29, 2004.

5. A companion special issue

The present special issue is oriented towards the application of program transformation methodology in software development. There is a companion special issue of the *Fundamenta Informaticae Journal* that is edited by Alberto Pettorossi and Maurizio

Proietti, which is oriented towards theoretical foundations and basic techniques of program transformation. (Cf. http://www.iasi.rm.cnr.it/~adp/fi_pt.html.) We hope that the two special issues together may provide a good coverage of the research area of program transformation, and a good starting point for future work on program transformation.

Acknowledgements

My gratitude is due to all submitting authors, even though many submissions could not be included for reasons of scope or time. I very much appreciated the willingness of authors to make an effort to meet all requests for revisions, in two or even three rounds. The quality of the review process was ensured by an exceptionally strong review committee; the referees were selected on a per-paper basis. The names and affiliations of most referees are listed below. If I have learnt one lesson from this project, then this is the importance of patience: grant deadline extensions to promising authors; wait for key reviews; initiate another time-consuming round of revision where needed. Finally, my gratitude is due to Jan Bergstra, the Editor-in-Chief of the SCP Journal, for proposing this project, and for appointing me as a guest editor, which I consider an honour. I am also very grateful to Bas van Vlijmen—the Editorial Assistant of the SCP Journal.

Appendix. List of referees for the special issue

(Some names were omitted.)

Faisal Akkawif (Northwestern University, USA), Kenichi Asai (Ochanomizu University, Japan), Jason Baker (Purdue University, USA), Ira D. Baxter (Semantic Designs, Inc., USA), Keith H. Bennett (University of Durham, UK), Juan C. Bicarregui (Imperial College, UK), Robert Biddle (Victoria University of Wellington, New Zealand), Christian Bunse (Fraunhofer IESE, Germany), Doris L. Carver (Louisiana State University, USA), Shigeru Chiba (Tokyo Institute of Technology, Japan), James R. Cordy (Queen's University, Canada), Mikhail Dmitriev (Sun Microsystems, Inc., USA), Merijn de Jonge (Utrecht University, The Netherlands), Andrea De Lucia (Università degli Studi di Salerno, Italy), Serge Demeyer (University of Antwerp, Belgium), Jin Song Dong (National University of Singapore), Eric Dubuis (Berner Fachhochschule, Biel, Schweiz), Steven Eker (SRI International, USA), Jeff Foster (University of Maryland, USA), Pascal Fradet (INRIA, Grenoble, France), Stephen Freund (Williams College, USA), Jeremy Gibbons (Oxford University, UK), Walter Guttmann (Universität Ulm, Germany), Reiko Heckel (Universität Paderborn), Jan Heering (CWI, The Netherlands), Holger Hermanns (Universität des Saarlandes, Germany), Stephan Herrmann (Technische Universität Berlin, Germany), Dirk Heuzeroth (Universität Karlsruhe, Germany), Ralf Hinze (Universität Bonn, Germany), Robert Hirschfeld (DoCoMo EuroLabs, Germany), Steven Klusener (Software Improvement Group, The Netherlands), Barbara Staudt Lerner (Williams College, USA), Sheldon X. Liang (Naval Postgraduate School, USA), Y. Annie Liu (State University of New York at Stony Brook, USA), Wolfgang Lohmann (Universität Rostock, Germany), Simon Marlow (Microsoft Research, Oxford, UK), Hidehiko Masuhara (University of Tokyo, Japan), Peter M. Maurer (Baylor University, USA),

Leon Moonen (Delft University of Technology, The Netherlands), Pierre-Etienne Moreau (INRIA Lorraine & LORIA, France), Zhenjiang Hu (University of Tokyo, Japan), James Noble (Victoria University of Wellington, New Zealand), Kasper Østerbye (IT University Copenhagen, Denmark), Klaus Ostermann (Technische Universität Darmstadt, Germany), Girish Palshikar (Tata Research Development and Design Centre, India), Elke Pulvermüller (Universität Karlsruhe, Germany), Claus Reinke (University of Canterbury, UK), Don Roberts (University of Illinois at Urbana-Champaign, USA), Ulrik P. Schultz (University of Aarhus, Denmark), Helmut Seidl (Universität Trier, Germany), Christian Stenzel (Technische Universität Kaiserslautern, Germany), Susan Stepney (University of York, UK), Mario Südholt (Ecole des Mines de Nantes/INRIA, France), Walid Taha (Rice University, USA), Peter Thiemann (Universität Freiburg, Germany), Niels Veerman (Free University Amsterdam, The Netherlands), Jurgen Vinju (CWI, The Netherlands), Eelco Visser (Utrecht University, The Netherlands), Joost Visser (Universidade do Minho, Portugal), Janis Voigtländer (Technische Universität Dresden, Germany), Philip Wadler (University of Edinburgh, UK).

Dr. Ralf Lämmel

Department of Information Management and Software Engineering,
Faculty of Sciences,
Free University,
De Boelelaan 1081a,
NL-1081 HV Amsterdam, The Netherlands

Department of Software Engineering,
Centrum voor Wiskunde en Informatica, Kruislaan 413,
NL-1098 SJ Amsterdam, The Netherlands

E-mail address: ralf@cs.vu.nl
URL: http://www.cs.vu.nl/~ralf

Available online 2 June 2004