Dagstuhl Seminar
"Transformation techniques in software engineering"
(TrafoDagstuhl 2005)
— Executive summary —

James R. Cordy[1], Ralf Lämmel[2], Andreas Winter[3]

[1] Queen's University, School of Computing
Kingston, ON, K7L 3N6, Canada
cordy@cs.queensu.ca
[2] Microsoft Corp., WebData/XML
One Microsoft Way, Redmond, WA, 98052, U.S.A.
ralfla@microsoft.com
[3] University of Koblenz-Landau, Institute for Software Technology
D-56016, Koblenz, Germany
winter@uni-koblenz.de

Abstract. TrafoDagstuhl brought together representatives of the research communities in re-engineering, XML processing, model-driven architecture and other areas of software engineering that involve grammar- or schema-driven transformations. These various existing fields and application contexts involve widely varying transformation techniques — the tradeoffs of which are worth analysing. This seminar initiated a process of understanding each other's transformation techniques — their use cases, corresponding methods, tool support, best practises, and open problems. This process makes it possible to exchange knowledge and experience between these various communities. This effort should also help in transposing transformation concepts from established application fields to new fields.

This executive summary reports on the conception of the seminar, the program, outcomes and future work. Most of the material from the seminar (including abstracts of all talks) as well as additional papers can be found on the dedicated web site: http://www.dagstuhl.de/05161/

1 Transformations everywhere

The idea for this seminar began with the observation of a discrepancy:

> While *software transformation* is a crosscutting theme in software engineering, the various fields in which it is used are only passingly aware of each other.

It would therefore make sense to bring together leading representatives from the different fields so that they can share problems and solutions related to their use of transformations and begin a dialogue on understanding transformation itself as a whole. Without claiming completeness, the following (somewhat overlapping) communities can be identified:

1. Program calculation
2. Language implementation
3. Model-driven development
4. Grammar(ware) engineering
5. Modelling and meta-modelling
6. Generative software development
7. Code restructuring and refactoring
8. Database reverse and re-engineering
9. Co-evolving designs and implementations
10. Data integration incl. semi-structured data
11. Design recovery and architectural recovery
12. Intentional and aspect-oriented programming

Most of these communities know of more than one kind of transformations. Also, transformation techniques are not always tied to a specific community. So it makes sense to abstract a little from the communities, and to identify some of the dimensions of variation for transformation techniques.

- The kind of grammars or schemas involved.
- The degree of automation of transformations.
- The degree of interactive transformations.
- The degree of formalisation of transformations.
- The degree of programming language support.
- The computational framework for transformations.
- The nature of transformation properties.
- The kinds of artifact: programs, data, and schemas.
- and so on.

During a week of intensive discussion, 47 participants from 12 countries attended the seminar, contributed presentations and participated in and/or organised discussion groups and panels. (International statistics for participants: Germany (14), Canada (12), U.S.A. (5), Belgium (4), the Netherlands (3), France (2), United Kingdom (2), Hungary (1), Ireland (1), Italy (1), Japan (1) and Switzerland (1).)

Special thanks go to the Dagstuhl staff for their excellent and unintrusive support in hosting this seminar. Without their help TrafoDagstuhl 2005 would not have been such a success. The Dagstuhl venue is best suited for the kind of ambitious and interdisciplinary event that TrafoDagstuhl 2005 was meant to be.

2   Handling diversity

How does one make sure that participants successfully share problems and solutions? It is clearly challenging to bring together a group with so much diversity — there is a chance that attendees simply lack enough shared common concepts to understand each other, that speakers can become confused and defensive when the established axioms in their fields are not observed by the mixed audience, and that different interpretations of technical vocabulary can lead to severe misunderstandings.

This risk was anticipated, and it was addressed in two ways.

1. The focus was moved away from "religious" issues.
   Instead, speakers were encouraged to focus as follows:

   – The presentation of technical content.
   – The discussion of accepted, best practises.
   – The presentation of foundations such as principled formalisms.
   – The selection of material that is *clearly* relevant across the community.

2. Particular technical concerns of interest were identified up-front:

   – Language-parametric transformation.
   – Reuse of transformation components.
   – Interdisciplinary transformation scenarios.
   – Coupled transformations of separate artifacts.
   – Language support for software transformation.
   – Properties of (semi-) automatic transformations.
   – Design patterns for transformational programming.
   – Validation of transformations by testing and others.
   – Best practises for the underlying grammars and schemas.

As a means to communicate the interdisciplinary character of the seminar well before the actual meeting, all participants were asked to prepare a very short introduction complete with an indication of their expectations from the seminar. By mining this information from the participants, it was possible to identify a number of themes with broad support:

– Modularity and composition of transformations.
– Correctness of transformations (in several respects).
– Evaluation and comparison of transformation setups.
– Adoption of transformation techniques ("ready for prime time").
– Transformations crossing boundaries (e.g., coupled transformations).
– Language independence and genericity in transformation techniques.

3   Scientific program

The seminar week started with 6 introductory seminar presentations. These invited presentations were meant to introduce basic concepts and vocabulary so that some common ground was established. The remainder of the week consisted of regular presentations by participants, a small set of focused discussion groups built around questions that emerged early, and a panel session based on a suggestion that emerged at mid-week.

3.1   Introductory presentations

The actual selection of introductory areas was necessarily a compromise: it was not possible to have a dedicated overview talk for every one of the 12 identified communities. Titles and abstracts of the presentations follow in the order given.

Jean-Marie Favre (LSR – IMAG)
Etymology of Model Driven Engineering and Model Transformation

This presentation just attempts to put emphasis on the importance of defining words and concepts before the start. What is a model? What is a transformation? What is a model transformation? To be honest, I don't know. While everybody could agree with her- or himself, agreeing with others is usually more challenging, especially when going into the details. We should admit that we don't have precise and unique answers to these questions and that further work is required. This presentation just aims at raising the terminological problem considering not only the MDA approach but also other technical spaces.

In November 2000, the OMG announced its new strategy, namely the Model Driven Architecture (MDA). The first word, "Model", revealed to be meaningful because this approach is based on the systematic use of explicit models at all stages in the software development process. While the term transformation does not appear explicitly in MDA, it has been found however that model transformation is actually another cornerstone of the approach. By contrast, the use of the word "architecture" in MDA is purely accidental. This choice has been largely criticised because it just adds confusion: While this architecture usually means software architectures in the software architecture, this is not the case in the MDA. In fact the etymology of MDA takes its roots in a transformation from OMA to MDA. The semantics of this transformation is the following: In the 90's the strategy of the OMG was based on "objects" (OMA = Object Management Architecture). In this new millennium, the strategy of the OMG will be based on "models" (MDA = Model Driven Architecture). At least this etymology clearly shows the shift of paradigm from Object to Models.

In fact, the term architecture is sometimes used in OMG documents to refer to the "our-layers architecture". On top of this pyramid (hence the term "architecture" ...), the MOF god plays the role of unique meta-meta-model. That it impose the language in which all metamodels should be written.

All this vision however should not be considered in isolation. MDA is just a technical space defined by the OMG, but they are plenty of other technical spaces including for instance Grammarware, Dataware, Ontologyware, XMLware, etc. Each technical space can also be arranged as a pyramid with similar structure, though the terminology used is different. For instance the pair model/metamodel in the Modelware pyramid is called sentence/grammar in Grammarware terminology, data/schema in Dataware, etc. Within each technical space the notion of transformation can be defined and transformation tools are available. For instance the standard transformation language in the OMG pyramid is QVT, but one can use also ATL, MTL, etc. They are plenty of transformation languages in the Grammarware technical space including for instance TXL and ASF/SDF; XSLT and XQuery are transformation languages in the XMLware technical space, SQL in Dataware, etc. Clearly reasoning at this level of abstraction leads to many terminological issues. This is however necessary since many transformation scenarios form industry imply to cross various technical spaces, changing from one mode of representation to another depending on the transformation to be expressed and executed. Etymology can help in explaining terms and concepts such as model and transformation, either independently from any particular technical space when needed, or to establish the mapping to a jargon used within a technical space.

Gabriele Taentzer (TU Berlin)

Specifying and Analysing Model Transformations based on Graph Transformation

Nowadays the usage of model transformations in software engineering has become widespread. Considering current trends in software development such as model driven development (MDD), there is an emerging need to develop model manipulations such as model evolution and optimisation, semantics definition, etc. If a model transformation is described in a precise way, it can be analysed later on. Models, especially visual models, can be described best by graphs, due to their multi-dimensional extension. Graphs can be manipulated by graph transformation in a rule-based manner. Thus, we specify model transformation by graph transformation. This approach offers visual and formal techniques in such a way that model transformations can be subjects to analysis. Various results on graph transformation can be used to prove important properties of model transformations such as its functional behaviour, a basic property for computations. Moreover, certain kinds of syntactical and semantical consistency properties can be shown on this formal basis.

Don Batory (University of Texas at Austin)

Understanding Aspect Composition Using Program Transformations

Program transformations can play a pivotal role in future models of software design, software architectures, and generative programming. This talk will examine AspectJ from this perspective.

AspectJ is among the more popular "new modularisation" technologies that are being explored in Software Engineering. AspectJ has clear advantages over traditional modularisation technologies (e.g., classes, packages), but has equally clear drawbacks as well. We argue that the drawbacks stem from the way in which aspects are composed, and show that step-wise development (and aspect reuse) is difficult with AspectJ. We propose an alternative model of composition, again based on program transformations, that supports step-wise development, retains the power of AspectJ, aligns AspectJ with existing results in Component-Based Software Engineering (CBSE), and simplifies program reasoning using aspects.

Eelco Visser (University of Utrecht)

Strategies for Rule-Based Program Transformation

In this talk I will sketch the development of rule-based approaches to program transformation using the program transformation language Stratego as a vehicle. The talk will cover techniques from pure rewriting, through strategies for the control of rewrite rules, to techniques for context-sensitive transformations. The techniques will be illustrated by means of examples of program transformation that prompted their development.

Andy Schürr (TU Darmstadt)

Model-Based Software Development - A Survey

Model-Based Software Development (MBSD) and OMG's recently invented three letter word MDA (Model-Driven Architecture) are the new hypes which tend to replace former terms like "object-oriented programming", "component-based software development", and the like. In this talk we will discuss the relationships between MDA and MBSD in general as well as their relationships to model and code transformation techniques. We will see that MBSD requires the combination of quite a number of todays existing transformation techniques and tools. Furthermore, we will present a long list of problems which have to be solved in order to make MBSD and transformation-based software development a "real" success story in practise. Finally, we will make some propaganda for so-called triple grammars, a special kind of declarative, bidirectional model transformation approach.

Alberto Pettorossi (CNR - Rome)

Formal Aspects of Program Transformation: Rules and Strategies

We present some program transformation techniques for: (i) deriving efficient programs from less efficient ones, (ii) synthesising programs from specifications, and (iii) proving program properties. In our examples we consider various programs taken from the area of discrete mathematics, optimisation, combinatorial mathematics, searching and sorting problems, and protocols for concurrent programming.

The techniques we present are based on the syntax of the language (in particular, we consider functional and logic programs) and on theorem proving ideas such as those of proof rules and proof strategies.

3.2   Regular presentations

Several presenters adapted their prepared talks to be broadly accessible using the concepts and the vocabulary from the introductory talks as a touchstone. These efforts worked out very well: they generated many interesting cross-community discussions, which were well in line with the original goal of the seminar. Titles and abstracts of the presentations follow in the order given.

Günter Kniesel (University of Bonn)

Towards a Unification of Refactorings, Aspects and Generative Programming

Refactorings, aspects and generative programming are examples of software engineering best practises with largely complementary strengths. Refactorings express behaviour preserving software transformations. Generative programming expresses heterogeneous behaviour changes that are applied at specifically marked locations of target modules. Aspects express homogeneous changes for a declaratively specified set of target modules, without depending on specific hooks in the target modules. The latter property is usually called 'obliviousness'. It is an essential enabler for unanticipated software evolution.

The aim of my work and of related activities of the ROOTS group is an integration of these complementary skills into an environment that can equally well express behaviour preserving and behaviour changing, homogeneous and heterogeneous software transformations, while preserving obliviousness.

The talk presents different aspects of this work. It starts top-down, introducing first the unification of aspects and generative programming in the generic aspect language LogicAJ, which provides a uniform environment for oblivious, behaviour changing, homogeneous and heterogeneous software transformations at a high level of abstraction.

The logic based formal foundation that enables such a unification is presented next. It is the language-parametric concept of conditional program / model transformations (CTs). The benefits of CTs as a unifying framework are illustrated by showing how they enable automated interaction detection and resolution between jointly deployed CTs and composition of CTs. These operations at the level of CTs enable definition of corresponding operations at the level of refactorings, aspects and generative programs. Thus CTs recommend themselves as a uniform framework for unifying important transformation based software engineering techniques and for studying and solving their open problems.

Shin-Cheng Mu (University of Tokyo)

Bidirectional updating by maintaining injective constraints

In many occasions, one encounters the task of maintaining the consistency of two pieces of structured data that are related by some transform. In some XML editors, for example, a source XML document is transformed to a user-friendly, editable view through a transform defined by the document designer. The editing performed by the user on the view needs to be reflected back to the source document. Similar techniques can also be used to synchronise several bookmarks stored in formats of different browsers, to maintain invariance among widgets in an user interface, or to maintain the consistency of data and view in databases.

This talk proposes a formal model of such tasks, basing on a programming language allowing injective functions only. The programmer designs the transformation as if she is writing a functional program, while the synchronisation behaviour is automatically derived by algebraic reasoning. The main advantage is being able to deal with duplication and structural changes. Non-injective programs, on the other hand, can be embedded into the injective language.

We now have a prototype editor based on the technique. In this talk we will give an overview of the framework, as well as current problems waiting to be solved.

David Wile (Teknowledge Corporation)

The utility of intertwining pattern matching and metaprograms

Normally, one works to separate concerns whenever possible. The natural place to do this in program transformation would appear to be to separate the concerns of program transformation and strategies for applying transformations. In the past, I have found two very useful situations in which they can be intertwined in a rather synergistic way, namely in iterated activities and in translation between languages. I will describe these and discuss the relative advantages and disadvantages of disobeying the separation of concerns maxim in this case.

Tom Mens (Université de Mons)

Formal Support for Software Evolution

In this presentation, I will present some research I have carried out in the context of the "Research Center on Software Restructuring", a Belgian inter-university research project that started on January 1, 2005. More precisely, I will explain how various formalisms can provide help to provide better support for model transformation, and model evolution in particular.

For the purpose of formalising model refactorings, I have explored the formalism of graph transformation. As it turns out, many useful properties of this formalism can be exploited to reason about dependencies between model refactorings. More in particular, together with Gabi Taentzer from the Technical University of Berlin I explored how to use the idea of critical pair analysis in order to detect conflicts between refactorings. The ideas were implemented in AGG, a state-of-the-art graph transformation tool.

Another formalism that I investigated, in collaboration with Ragnhild Van Der Straeten from the Vrije Universiteit Brussel, is description logics. More specifically, we explored how this formalism can be used to detect internal model inconsistencies as well as inconsistencies between a model and a newer version that has been obtained through refinement or refactoring. Tool support is implemented by means of a plug-in in Poseidon and an implementation in the Racer description logics engine.

Albert Zündorf (Universität Kassel)

Fujaba as a QVT language for MDA

The Model Driven Architecture approach is based on the availability of a powerful query, view, and transformation language. While there is no such language proposed by the OMG so far, this talk will show how graph transformations as e.g. provided by Fujaba may do the job.

Jean Bézivin (Université de Nantes)

Modelling in the Large and Modelling in the Small

As part of the AMMA project (ATLAS Model Management Architecture), we are currently building several model management tools to support the tasks of modelling in the large and of modelling in the small.

The basic idea is to define an experimental framework based on the principle of models as first class entities. This allows us to investigate issues of conceptual and practical interest in the field of model management applied to data-intensive applications. By modelling in the small, we mean dealing with model and meta-model elements and the relations between them. In this sense, ATL (ATLAS Transformation Language) allows expressing automatic model transformations.

We also motivate the need for the "ModelWeaver" which handles fine-grained relationships between elements of different metamodels with a different purpose than automatic model transformation. By modelling in the large, we mean globally dealing with models, metamodels and their properties and relations.

We use the notion of a "MegaModel" to describe a registry for models and metamodels. This talk proposes a lightweight architectural style for a model-engineering platform as well as a first prototype implementation demonstrating its feasibility.

Bernhard Rumpe (TU Braunschweig)

Transformation-Oriented Architecture Evolution

Currently there is much hype about how to apply transformational approaches to software engineering. However, transformations are used in SE already since the beginning of assemblers and compilers. What's new is that the writing of transformations shall be simplified thus allowing to more easily change the source language as well as the target domain/techniques and the purpose of transformation. In our talk, we give a critical overview of the state of the art as currently

perceived. It will turn out that lot's of efforts have been made, but also that we are still in the early stages to make transformational approaches such as MDD, MDA etc. a broad success. On the other hand, if model based SE shall become a success, transformational use of models is a key prerequisite for it.

Brian Malloy (Clemson University), James Power (National University of Ireland)

Reverse Engineering C++ Applications

In this talk , we describe g4re, our tool chain that exploits GENERIC, an intermediate format incorporated into the gcc C++ compiler, to facilitate analysis of real C++ applications. The gcc GENERIC representation is available through a file generated for each translation unit (TU), and g4re reads each TU file and constructs a corresponding Abstract Semantic Graph (ASG). Since TU files can be prohibitively large, ranging from 11 megabytes for a "hello world" program, to 18 gigabytes for a version of Mozilla Thunderbird, we describe our approach for reducing the size of the generated ASG.

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/424

Krzysztof Czarnecki (University of Waterloo)

Generative Software Development

Product-line engineering seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way. In product-line engineering, new system variants can be rapidly created based on a set of reusable assets (such as a common architecture, components, models, etc.). Generative software development aims at modelling and implementing product lines in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages (DSLs).

In the first part of my talk, I will give a brief overview of the basic concepts of generative software development. In the second part, I will present my latest work on feature-based configuration and mapping features to models, such as behavioural and data specifications.

Martin Gogolla (Universität Bremen)

The TrafoDag metamodel zoo

This paper explains how four model transformations between database models work: (1) An ER (Entity-Relationship) database schema is transformed into a collection of ER database states, (2) a RE (Relational) database schema into a collection of RE database states, (3) an ER database schema into a RE database schema, and (4) a collection of ER database states into a collection of RE database states. These four separate transformations may be viewed as a single transformation between the ER datamodel and the RE datamodel.

The schemas are regarded as determining the syntax of the datamodels, and the set of associated states is regarded as being the semantics of the datamodels, because the states associate meaning with the schemas. When one usually considers database models, one formally only treats syntactical aspects, i.e., schemas, and handles the semantics merely informally. Our approach allows to formally handle syntax and semantics of database models and their transformation within a single and uniform framework. The approach thus allows to precisely describe properties of the datamodels and properties of the transformation.

Recently, efforts similar to the above one have been mentioned in connection with a so-called metamodel zoo: Our approach knows metamodels for the syntax and semantics of two languages (ER and RE), i.e., the approach starts with four metamodels. Additionally, syntactical and semantical aspects common to both languages have been factored out into another two separate metamodels and the transformation may also be regarded as a metamodel. All these various beasts co-exist peacefully and may be inspected carefully with appropriate tools. This encourages us to think of them as constituting a metamodel zoo.

The method behind our approach is to divide a language into a syntax and semantics part and to describe a transformation between two languages as a direction-neutral affair. Formal properties of the languages to be transformed and formal properties of the transformation are described uniformly. Transformation properties can be properties regarding syntax and semantics. The method can be applied not only to database languages but to transformations between common computer science languages.

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/425

Jurgen Vinju (CWI - Amsterdam)

Generic language technology

We presented the ASF+SDF Meta-Environment as a composition of reusable generic language technology for implementing transformations. Both infrastructural components, and higher level domain specific languages have been highlighted. As an example of our goals with the entire system, we started with a demonstration of Cobol to Cobol, and Cobol to Control Flow Graph transformations. We then moved on to the decomposition into components. We have introduced typical properties of all individual components, detailing some of the engineering details. The goal of this talk was to provide insight into the technology behind ASF+SDF, which we release as separate LGPL components.

Thomas R. Dean (Queens University)

Security Testing using Software Transformation

Application protocols have become sophisticated enough that they have become languages in their own right. At the best of times, these protocols are difficult to implement correctly. Traditional conformance testing of these implementations does not reveal many security vulnerabilities.

In this talk we describe ongoing research where software transformation and program comprehension techniques are used to assist in the security testing of network applications. We capture a live, valid, protocol data unit, generate modified mutants and inject the mutants back into the network to see if the network application survives. Language Comprehension techniques are used to analyse the network protocol syntax and identify features of the network protocol that are most likely to be implemented incorrectly. Source transformation techniques adapted from the program comprehension community are used to use the analysis to generate the mutant packets.

Rudolf Ferenc (University of Szeged)

Transformations Among the Three Levels of Schemas for Reverse Engineering

The bases of most of the transformations in reverse engineering are schemas (also known as metamodels). The instances of these schemas (which are usually graphs) represent the subject software system on a higher level of abstraction.

In a previous Dagstuhl seminar in 2001 (Interoperability of Reverse Engineering Tools) we agreed upon that three schema-levels are needed in reverse engineering and that information exchange should be performed using GXL (Graph eXchange Language). The three levels of schemas were: 1. low-level (syntax); 2. middle-level (component) and 3. high-level (architecture). We were responsible for proposing a low-level schema for C++.

Since that seminar we designed the Columbus Schema for the C++ programming language which can be represented (among others) in GXL. The schema is a low-level schema and it models the source code of a software system extended with semantic information. We also implemented a transformation algorithm to obtain DMM (Dagstuhl Middle Metamodel) schema instances from the Columbus Schema instances using our Schema-API. Unfortunately, no reference schema was accepted for high-level schemas yet, but we are able to transform our models to obtain schema instances of the ART Software Architecting Environment.

Michael Godfrey (University of Waterloo)

Lossy Program Analysis, or Lies My Extractor Told Me

Many program transformation systems operate on full source code ASTs, making them slow and unwieldy to play with. A different, and complementary, approach is to use "lossy" program analysis tools: such a tool produces a program model according to a programming language specific schema (meta-model). It is complete relative to the level of detail of the schema, but leaves out (is lossy) relative to the full source code. The advantage of using such a tool is that it is typically much simpler and faster to create customised queries, although it is no longer possible to perform source-to-source transformations.

In this talk, I will discuss a lossy Java analysis tool I developed that is now the basis for a J2EE architecture analysis system in development inside a large software company.

Roel Wuyts (Université Libre de Bruxelles)

Supporting co-evolution of design and implementation with declarative meta programming

When developing software systems, the relation between design and implementation is typically left unspecified. As a result design or implementation can be modified independently of each other, and a modification of either one does not leave any trace in the other. The practical result of this is a number of well-known problems such as drift and erosion, documentation maintenance problems or round-trip engineering trouble. To solve these problems we proposed (a while back) to make the relation between design and implementation explicit by expressing design as a logic meta-program over implementation. This is the cornerstone for building a complete synchronisation framework that allows one to synchronise changes to design and implementation. We have implemented such synchronisation framework, and applied it successfully on two case studies.

Simon J. Thompson (University of Kent)

Designing and implementing a refactoring tool for an existing functional programming language: Haskell

This presentation will describe the design and implementation of the HaRe tool for refactoring Haskell programs. Among the constraints on the design are

- ensuring layout and comments are preserved by refactorings
- ensuring integration with existing Haskell tools: emacs and vim
- ensuring treatment of de jure (Haskell 98) and de facto (Glasgow Haskell Compiler) standards.

Markus Pizka (TU München)

Program generation in the large

While the idea to increase software development productivity through program generators is all but new existing transformation tools besides classical compilers have failed to live up to this expectation. This talk argues that the two main reasons for this are the inflexibility of generators produced in the context of domain specific languages and the -predominant "generate in one step" approach to program generation.

We propose to regard program generation as a staged process where the output of one generation stage is combined with user and additional input and fed into the next stage. By this, the power of the generation process is greatly increased without sacrificing the quality of the output such as performance. In addition to being multi-staged the proposed generative concept aims at building the generators bottom-up through continuous tool-support evolution. The evolution of the generators itself is based on program transformation whereas both, the generators and the existing input words for these generators, are consistently transformed during evolution.

Pieter Van Gorp (University of Antwerp)

CAViT: a Consistency Maintenance Framework

Design by contract is a software correctness methodology for procedural and object-oriented software. It relies on logical assertions to detect implementation mistakes at run-time or to proof the absence thereof at compile-time. Design by contract has found a new application in model driven engineering, a methodology that aims to manage the complexity of frameworks by relying on models and transformations.

A "transformation contract" is a pair of constraints that together describe the effect of a transformation rule on the set of models contained in its transformation definition: the postcondition describes the model consistency state that the rule can establish provided that its precondition is satisfied. A transformation contract of a rule can be maintained automatically by calling the rule (1) as soon as the invariant corresponding to its postcondition is violated and (2) provided that its precondition is satisfied.

Domain specific visual languages can facilitate the implementation of the actual transformation rules since they hide the complexity of graph transformation algorithms and standards for tool interoperability.

In this talk, we describe CAViT: a framework that integrates a visual model transformation tool with a design by contract tool by relying on OMG standards such as UML, OCL and MOF.

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/429

Shriram Krishnamurthi (Brown Univ. - Providence)

Controlling the Flow: Control-Oriented Domain-Specific Languages

Transformational techniques are used extensively to implement domain-specific languages. While some languages predominantly represent interesting data, others are control-oriented: they have operational behaviour that is significantly different from that of the language to which they are being mapped by a deep embedding. A classic example would be the mapping of a data flow language to an interpreter in a call-by-value language. In such cases, the mapping greatly complicates the process of giving feedback to the programmer using the DSL: error messages, profiling information, and so on are perverted by the deep embedding. I discuss this problem through examples, and outline a solution for the DrScheme programming environment.

Robert Hirschfeld (DoCoMo Euro-Labs - München)

Dynamic Service Adaptation

With increasing deployment and use of continuously running systems in combination with the shortening of product and service cycles, additional effort need to be made on allowing such software systems to be evolved without taking them offline or even shutting them down. Predicting and providing as many variation

points as possible upfront typically lead to premature abstractions that increase complexity of both runtime behaviour and system comprehension.

In the domain of telecommunications, for example, next-generation communication platforms are to provide for high service availability, best service quality, and novel situation-aware services to respective customers. Support for online updates, third-party service integration, situation-awareness, or advanced service and platform personalisation are only a few of the requirements to be satisfied. We thus need to account for changes that we cannot anticipate or prepare for, post deployment at runtime.

Dynamic Service Adaptation (DSA) is our approach to address these issues by offering appropriate mechanisms to modify service logic while services are executing. Based on a dynamic and fully reflective programming language and execution environment, DSA provides the means to introspect and navigate basic computational structures and to adjust them accordingly.

Kim Mens (Université catholique de Louvain)

Automated Derivation of Translators From Annotated Grammars

We propose a technique to automate the process of building translators between operations languages, a family of DSLs used to program satellite operations procedures. We exploit the similarities between those languages to semiautomatically build a transformation schema between them, through the use of annotated grammars. To improve the overall translation process even more, reducing its complexity, we also propose an intermediate representation common to all operations languages. We validate our approach by semi-automatically deriving translators between some operations languages, using a prototype tool which we implemented for that purpose.

Mohammad El-Ramly (University of Leicester)

Experiences in Teaching Program Transformation for Software Reengineering

Little attention is given to teaching the theory and practise of software evolution and change in software engineering curricula. Program transformation is no exception.

The presentation covers the author's experience in teaching program transformation as a unit in a postgraduate module on software systems reengineering. There is also an accompanying paper on the TrafoDagstuhl website. Both describe the teaching context of this unit and two different offerings of it, one using Turing eXtender Language (TXL) and the other using Legacy Computer Aided Reengineering Environment (Legacy-CARE or L-CARE) from ATX Software. From this experience, it was found that selecting the suitable material (that balances theory and practise) and the right tool(s) for the level of students and depth of coverage required is a non-trivial task. It was also found that teaching using toy exercises and assignments does not convey well the practical aspects of the subject, while teaching with real, even small size exercises and assignments, is almost non-feasible. Finding the right balance is very important but not easy.

It was also found that students understanding and appreciation of the topic of program transformation increases when they are presented with real industrial case studies.

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2006/423

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/423

Ric Holt (University of Waterloo)

Software Graph Transformations in the SWAG Kit Pipeline

The SwagKit set of tools extracts a model of source code and transforms that model in various ways, in order to simplify, abstract, validate and visualise the model. The model is represented in the TA fact exchange language and is successively transformed algebraically using the Grok relational language. This process will be discussed in terms of stages in the SwagKit pipeline.

James R. Cordy (Queens University)

Bridging the Gap: A Source Markup Alternative

Source manipulation tools are good for working with source. Model manipulation tools are good for working with models. Insights from the one can only be truly useful if they can be reflected to the other, and vice versa. In this presentation I discuss how, using implicit links via unique naming, it is possible to retain source as source and graph as graph while implicitly maintaining consistency.

Andrew Malton (University of Waterloo)

Models and Conformance

Box-and-arrow diagrams seem inevitable for presentation and transformation of software architecture, but the various styles of presenting them are hard to compare, and transformation between styles is difficult to specify. In this talk I show how to define models and "conformance to schemas" for a simple box-and-arrow language, TA, based purely on the algebraic structure of graphs and graph morphisms. This can form a basis for the semantics of box- and-arrow languages and the translation between technical spaces.

3.3   Discussion groups

The work of the discussion groups was carried out in two break-out sessions over two days, with a final plenary session to present the results to all participants. Discussion groups varied widely in size and amount of results, but all were based on a crosscutting theme that transcended the usual boundaries between areas, and all generated much fruitful discussion. In at least two cases, the groups generated new collaborations that are still active.

Discussion groups were formed around the following questions, agreed (with some initial hot debate) at a plenary session early in the week. In each case, an advocate for the question led the group in discussion and reported results.

When to use graph transformation, and when to use tree transformation?

Advocated by E. Visser

The session discussed the use of trees vs. graphs in transformation systems. The group worked on a matrix characterising and comparing a number of transformation systems in terms of common criteria. This is work in progress. To give an idea, the list of the discussed transformation systems includes the following: Stratego, ASF, TXL, XSLT, Progres, Fujaba, MoTMoT, AGG, and grok. Here are some of the criteria that were considered:

- Internal representation (tree, dag, typed graph, ...)
- The different grammar formalisms for source input
- Exchange language (Aterm, GXL, XMI, ...)
- Layout preservation (yes, no, optional, partial)
- Comment preservation (yes, no, optional, partial)
- Destructive update (yes, no, ...)
- Termination (guaranteed, subject to proof, ...)
- Confluence (subject to proof, partial, ...)
- Type correctness (static, runtime-checked, ...)

The notes of the work by this discussion group are obtainable from the TrafoDagstuhl website. A snapshot of the aforementioned comparison matrix is available at this site, too.

How to bridge between transformation and analysis technology?

Advocated by J. Vinju

This group discussed the intricacies of engineering practical connections between software analysis and software transformation tools. The discussion was focused on the general perspective of such connections. The discussion honoured the fact that transformation of source code and analysis of source code are two different domains. For each domain, specific tooling has been constructed with success. Each domain is specialised on different kinds of source code representations. On the one hand, transformation tools work on representations of source code that closely follow the structure of the original source code. On the other hand, analysis tools deal with more abstract representations, that are frequently referred to as facts about source code. Transformation and analysis tools perform computations in these two domains, and communication is needed between the domains. To be able to communicate, both computations need to employ a concept of identity of source code artifacts. The overall question is: "Given one arbitrary transformation tool, and one arbitrary analysis tool, how to construct a high quality bridge between the two?" Two different designs for bridges were discussed. The monolithic design links the two computations together in one process and uses co-routines to alternate. The data is marshalled using an API that translates source code artifacts on-the-fly to either representation. A heterogeneous design picks a batch approach, staging the two communications by

filtering serialised representations. It uses import/export filters to marshal the serialised representations.

The work of this discussion group has been summarised in a paper that is obtainable from the TrafoDagstuhl website.

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2006/426

How to deal with unanticipated composition of transformations?

Advocated by G. Kniesel

The discussion started from the problem of composition: when transformations are composed together from simpler building blocks one needs to define how the correctness of the combined transformation follows from the correctness of its building blocks. Eventually, the topic of the group evolved to "Correctness and Verification of Transformations for Open Systems".

The discussion clarified that any decision about interference or correctness of a transformation needs an explicit statement of the constraints that are violated or met by the transformation. These constraints could relate to the transformation output only (e.g., syntactic well-formedness) but mostly relate input and output. In the case of refactorings they state preservation of certain properties. In the general case (model transformations, aspects, etc.) they state desired changes of properties. The relevant constraints can be defined at language level (syntax, type rules, scoping rules, operational semantics) or domain level (preconditions, postconditions, invariants, behaviour protocols, or any heavyweight specification technique). In the case of domain level constraints, one can try to fall back to the simpler case of language-level constraints by using a domain specific language (DSL), thus making constraint checking part of the language. Alternatively, one can use a multiple representation approach, advocated by Don Batory. In this case, domain level constraints are captured in additional representations (performance model, security model) and checking is delegated to an additional tool.

A significant part of the discussion revolved around the open systems scenario, which can give rise to the joint application of independently developed transformations. In the case of unanticipated composition the question may arise whether the order of composition matters. Follow-up questions are then whether order dependence is acceptable and whether such a dependence is easily discoverable and understandable by the user of a transformation system. This problem also showed up in the introductory presentation by Don Batory: the conglomeration of several (AspectJ) aspects may expose interference problems that may suggest some form of explicit functional composition. Tom Mens presented a scenario where different refactorings executed in parallel on the same source also lead to the problem of determining automatically whether they would interfere or not. The discussion group analysed tradeoffs of known approaches. For instance, static analyses may be performed on the operands of the composition in an attempt to attest order independence, and to reject the composition otherwise, or to require explicit sequential ordering instead. The Condor system presented by Günter Kniesel additionally allows to automatically determine an

order that guarantees interference free execution. In the case of cyclic dependencies it determines whether an iterative execution of the transformation cycle will lead to a fix-point and reports an interference otherwise.

Grammar engineering:
How to transpose ideas between technical spaces?

Advocated by R. Lämmel

Grammar engineering covers grammar design, testing, implementation and recovery. Highly organised, well-understood semi-automated processes are available for various scenarios. This group attempted the discussion of grammar techniques across technological spaces (or even subspaces) such as the following:

– Use cases for concrete syntax
– Use cases for abstract syntax
– Use of grammar-like program types
– UML models as grammars or schemas
– Use cases of XML schemas
– Generic exchange formats
– Domain-specific exchange formats

On the one hand, the discussion group aimed to understand the transposition of concepts like grammar testing and grammar transformation to meta-modelling or to the XML space. On the other hand, the group attempted the transposition of meta-modelling and mega-modelling ideas to the grammar-based spaces. The group discussed concepts like the following:

– Grammar transformation
– Grammar translation
– Model/meta-model annotation
– Diffing for schemas
– Analysis for grammars
– Schema integration
– Schema visualisation
– Grammar modularisation
– Grammar-driven test-data generation

In order to exercise transposition, the group used a mixed scenario that involved aspects of software modelling and meta-modelling as well as XML processing:

– We start from a complex UML model understood by tool A.
– We export the model using XMI (incl. idiosyncrasies of A).
– We attempt to import the XMI data with a different tool B.
– We face the problem that tool B "throws".

Now: What to do? (How to leverage grammar engineering?)
The group discussed dimensions like the following:

– Perform a diff operation on the schemas of A and B, if they are available.
– Recover schemas A and B, using some sort of grammar recovery.
– Attempt recovery of non-schematic constraints that may cause failure.
– Employ test-data generation to measure incompatibility of A and B.

What are the applications of transformation, and how do we get them adopted?

Advocated by T. Dean

Two issues were discussed. One was the tools vs techniques issue. Do we want tools or techniques adopted. But discussion rapidly shifted to post secondary education. It was noted that Unix made many in roads due to the fact that it was made freely available to higher education. While the techniques are sometimes available in undergraduate and graduate courses, they are typically taught only in the Software Transformation and Soft Engineering Courses. But the techniques have been applied in other areas of research such as handwriting recognition, security, and data mining. The techniques are of general interest, so we should be introducing our colleagues to the techniques so that they can be generally applied in the undergraduate and graduate curriculum in much the way that programming languages are used.

The graduating students will (hopefully!) then carry the techniques into industry. The issues are what to teach and when. Some universities have some info (i.e., Trento). Other possible courses include:

– Programming Paradigms Course
– Formal Languages Course
– Design Course
– Reengineering, Software Evolution
– Software Quality/Quality Assurance
– Compiler
– Formal Methods

Topics that are already showing up in the undergraduate curriculum are:

– HTML / XSLT / GXL / SVG
– Generation of Code
– Visualisation

So the undergraduate curriculum appears to be ready for more general software techniques and tools. It was felt that there is a need for simple grounded problems and examples that can be used in an undergraduate curriculum. This may help provide opportunities to introduce the transformation techniques. One possibility is to connect an IDE such as Eclipse with transformations to make it easier to run.

Sources for these examples may come from people already using the techniques. We need examples for lectures and for assignments. The best people to create these are the experts. It would be helpful if the tool experts would provide simple lecture examples and simple undergraduate assignments for the tools and how they could be used in more general areas.

There was also some discussion of a repository for the resources. However that needs a caretaker and it would also require some rules on common formats to that the resources would be generally useful for course instructors.

3.4 A panel

The scientific program also featured a panel that was suggested by the eventual moderator, H. Müller. We include a very brief description of the panel.

Engineering Aspects of Software Transformation

Moderated by H. Müller (University of Victoria)

Panelists and topics:

- D. Batory — component-based software design
- K. Czarnecki — generative software development
- R. Lämmel — grammar engineering
- A. Schürr — graph transformation

The panel discussed application areas, challenges, advantages, and disadvantages of the various transformation approaches and technologies. A major goal was to identify the classes of problems best suited to be solved by the various approaches. Transformation-based interoperability and tool integration was discussed in the context of model-based software development and required transformations. It was observed that most of the work (ca. 90 %) is spent on synchronizing different tool APIs, whereas transformation only requires a small amount of time (ca. 10 %). Thus, standardised and problem-suited APIs and transformation engines working on federated data sources are required to enable the benefits of transformation in model-driven development environments. One participant challenged the panelists with the following question: "Which single result or success story comes to your mind as a means to support your transformation setup?" Fortunately, all panelists had such data points at hand.

4   Further plans

In the closing session of the seminar, potential spin-offs were discussed. Some of these ideas are summarised below. The organisers apologise for any omission of topics or advocates. The following list is definitely incomplete.

Transformation system implementation workshop

Advocated by Jim Cordy, Andy Schürr, Eelco Visser, et al.

It was observed that although many different implementation tricks and techniques were known to various participants, there was a lack of accessible knowledge on implementing and optimising transformation systems. The goal of such a workshop would be to provide a forum for implementation issues in transformation systems similar to compiler technology workshops.

Qualitative comparison of transformation systems

Advocated by Andrew Malton, Andy Schürr, Eelco Visser, et al.

One result of the discussion group on when to use graph transformations vs. tree transformations was a comparison table that hopefully forms the beginnings of a meaningful qualitative comparison of transformation systems based on a range of qualities. This work continues as a new collaboration between some of the attendees of the discussion group. In a similar vein, Andy Schürr proposed a workshop on the comparison of model transformation tools.

Source transformation benchmark project

Advocated by Jim Cordy, Eelco Visser, et al.

During the week, plans matured for a set of small, accessible benchmarks designed to demonstrate the similarities and differences between source transformation tools and systems. A new example language, Tiny Imperative Language, has been designed and a small set of example benchmark problems based on it has since been proposed. Solutions for different tools can be posted online.

A textbook on transformation techniques

Advocated by Jim Cordy, Ralf Lämmel, Hausi Müller, et al.

A logical continuation of the seminar would be an effort on a textbook that integrates the body of knowledge on transformation techniques. One may expect sections on model transformation and meta-modelling, graph transformation, generative programming, source transformation, program transformation and refinement, and so on. It became immediately clear that quite different books would be desirable: graduate-level vs. undergraduate-level; also a short-term compilation from existing body of knowledge vs. a mid-term compilation that incorporates still-to-be delivered results on qualitative comparison and best practises. Extra effort would be needed to get started with the textbook project. For instance, another Dagstuhl meeting could be of use for this purpose.

Some of these plans for spin-offs are actively worked on, as of writing this executive summary. For instance, a first initiative in the direction of comparing transformation systems was a meeting in Bonn at the end of May 2005 hosted by Günter Kniesel. The meeting was attended by Eelco Visser, Gabi Taentzer, and Tom Mens, as well as a number of students from the various institutes. There we combined issues from the "Trees vs Graphs' discussion and from the "Correctness of transformations" discussion. A follow-up meeting is planned.

5   Closing remarks

The main outcome of the seminar is simply the fact that researchers from different but related fields were introduced to each other. Many participants acknowledged the value of this concept and the gained insights into neighbouring fields. It was also realised that the integrative effort of the TrafoDagstuhl should not end with the actual meeting. Spin-off workshops were proposed and it was agreed that the overall integrated view on transformation techniques should eventually influence existing conferences and working groups, e.g., the GPCE conference or IFIP WG 2.1.

The seminar on 'Transformation techniques in software engineering" was a successful first step to initiate scientific discussions on transformation techniques between different communities while crossing the "religious borders" between these communities. Thereby, the seminar makes a contribution to an integrated notion of software transformation.

A   The TrafoDagstuhl song

As reported in the section on discussion groups, an enduring topic at the seminar was about the best representation of models to be transformed. Typically, one group favours tree-like structures, whereas another group supports graph and graph-grammar-based approaches. Both approaches have their advantages and disadvantages in certain fields, but a general understanding of the tradeoffs and best practises is missing.

Fruitful discussions led to a (slightly biased) song, performed by Jim Cordy in the final session:

> **Software, software graph**
> (to the tune of "Vincent (Starry Starry Night)", words by Jim Cordy)
>
> Software, software graph
> Ten thousand nodes that are confusing me
> What ever happened to the lovely tree
> That simply represents my
>
> Highly structured source
> The source I've worked with for a year or two
> It's easy to find things I am telling you
> Without edges in the way
>
> Why is it they don't see
> Just how much it is you mean to me?
> All this graph stuff just confuses me
> My lovely little tree
>
> They don't understand you,
> Don't know your power still
> Perhaps they never will