# Towards a runtime model based on colored Petri-nets for the execution of model transformations

Thomas Reiter
Information Systems Group
Johannes Kepler University
Altenbergerstr. 69
4040 Linz, Austria
+43-732-2468-9236
reiter@ifs.uni-linz.ac.at

Manuel Wimmer
Business Informatics Group
Vienna University of Technology
Favoritenstr. 9-11
1040 Vienna, Austria
+43-1-58801-18829
wimmer@big.tuwien.ac.at

Horst Kargl
Business Informatics Group
Vienna University of Technology
Favoritenstr. 9-11
1040 Vienna, Austria
+43-1-58801-18837
kargl@big.tuwien.ac.at

## ABSTRACT

Existing model transformation languages, which range from purely imperative to fully declarative approaches, have the advantage of either explicitly providing statefulness and the ability to define control flow, or offering a raised level of abstraction through automatic rule ordering and application. Existing approaches trying to combine the strengths of both paradigms do so on the language level, only, without considering the benefits of integrating imperative and declarative paradigms in the underlying execution model. Hence, this paper proposes a transformation execution model based on colored Petri-nets, which allows to combine the statefulness of imperative approaches as well the raised level of abstraction from declarative approaches. Furthermore, we show how a Petri-net based execution model lends itself naturally to the integration of an aspect-oriented style of transformation definition, as transformation rules can be triggered not only upon the input model, but on the state of the transformation execution itself.

## 1. INTRODUCTION

As model transformations play a key role in model driven development, several dedicated languages have emerged that allow to define and execute transformations between source and target metamodels. Compared to transformations implemented in a general purpose programming language or XSL transformations which operate on a models serialization, model transformation languages provide a layer of abstraction by allowing to manipulate models in terms of their abstract syntax given by its metamodel. Apart from this basic commonality, different kinds of model transformation languages exist. These approaches range from purely imperative styles allowing to define *how* an transformation is carried out, to fully declarative transformation definition styles focusing on *what* a transformation's output should be like, according to a certain input.

Declarative approaches (i.e. graph transformations) are typically based on defining rules that are later on interpreted by an execution engine to produce the desired result. Hence, the actual transformation execution as well as the order of rule application generally need not be handled by the user, although approaches based on graph transformations like AGG, or VMTS [7] allow to specify precedence of certain rules. Declarative rules typically consist of a semantically corresponding source and target patterns, whereby for each match of the source pattern in the input model, a target pattern is instantiated in the output model. Additionally, Triple Graph Grammars (TGG) [5] maintain the state of a transformation by traces that link matched source and instantiated target model elements.

Imperative approaches are similar in usage to traditional programming languages and allow the developer to explicitly manipulate transformation execution state and control flow. Although approaches such as the EOL [4], MTL or Kermeta [6] offer great flexibility and ease of use, the programming model does not support the intuitive alignment of concepts that is prevalent in metamodel or schema integration tasks, and one often needs to implement manually what a more succinct declarative description would achieve. However, what a declarative approach gains in abstraction, it loses in flexibility. Naturally, declarative specifications are convenient language constructs for recurring transformation tasks, but for “tricky” problems, a rule-based paradigm can become unwieldy.

To alleviate these limitations, hybrid approaches like ATL [3] or Xtend [8][9] combine imperative and declarative styles of transformation definition. (We regard Xtend as hybrid due to its functional style and rule-like “create” extensions.) Thus, the imperative part of a hybrid language is available to accomplish tasks that cannot be adequately solved declaratively. However, allowing to intermix imperative and declarative statements requires a developer to be aware of how exactly the engine orchestrates transformation execution. For instance, when writing imperative program parts in ATL, one has to be aware that their execution is subject to the engine’s scheduling, and one may not assume that certain declarative rules have yet been dealt with, or that a certain internal state is reached. Hence, the imperative part is often necessary simply to work around the confines of the engine’s execution procedure, as opposed to enable algorithmic computations. As an example, a common work-around is to explicitly maintain and observe custom state information in global variables, for instance to be able to manually trigger rules at certain points during a transformation's execution, in case the state information (i.e. trace between source and target model) that is automatically maintained by the execution engine does not suffice.

In general, existing declarative and hybrid approaches, are governed by an underlying execution procedure implemented in

This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806, and the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002

the respective transformation engine. In our opinion, this rigidness is the main cause for trouble when attempting to solve tricky problems with declarative approaches, or when integrating them with imperative styles. As the actual transformation definitions can be seen as merely parameterizing an intrinsically rigid, pre-defined procedure, we view declarative approaches as *data-oriented*, in the sense that they specify how input data is mapped onto output data. This is reflected in the rationale, that models are seen as graphs, and therefore graph transformations are used to describe and implement model transformations.

As opposed to declarative approaches, imperative approaches express transformations on a very fine-grained level, which is flexible but incurs explicit handling of control flow without support for the alignment of concepts as it is prevalent in schema integration tasks, for instance. Instead of specifying what input data is mapped onto what output data, imperative approaches follow a *procedure-oriented* paradigm and allow to algorithmically define a function that computes the output model from the input model.

We propose to rethink the notion of models as input and output data which is subject to a transformation that is seen either as an explicit or implicit procedure, but understand a transformation as a *process*. In a process-oriented view, a transformation execution is carried out by interacting entities that control streams of information from source to target models. The flowing information stems from the models themselves, and the actual transformation logic is made up by the behavior of individual entities and their interaction which each other.

Consequently, we propose the *transformation net* formalism, which is based on conditional, colored Petri-nets, to represent transformation processes. Such an execution model provides the explicit statefulness of imperative approaches through markings contained in the net's places. The abstraction of control flow from declarative approaches is achieved as transitions can fire autonomously depending on their environment. To describe specific firing rules for transitions, we resort to pre/post rules known from graph transformations.

The following section gives an overview of the transformation net formalism and describes how models and metamodels can be mapped onto transformation nets. The example in section three will describe how higher-level languages can be built on-top of transformation nets and how a process-oriented view favors the incorporation of aspect-oriented rules. Section four concludes with an outlook on future work.

## 2. TRANSFORMATION NETS

What sets transformation nets apart form existing approaches is their ability of making the transformation process explicit, as opposed to assuming a certain predefined execution rigor. Of course, the Petri-net based formalism needs an execution engine, too. But the Petri-net execution engine is generic and not tailored to a specific task unlike declarative model transformation engines. This makes the transformation net formalism a flexible execution environment to be targeted by generators of higher-level transformation languages, such that specific transformation and integration operators can be defined using the semantics offered by transformation nets.

As symbolically displayed in Figure 1, the “compilation” step produces a transformation net in its initial state (i.e. ready for execution) that uniformly represents models, metamodels and transformation specifications. The static parts of a transformation net that correspond to the transformation process’ inputs and outputs, are generated from models and metamodels, whereas the part that corresponds to the process’ execution logic is created from the integration specification by a custom generator for a certain higher-level language.

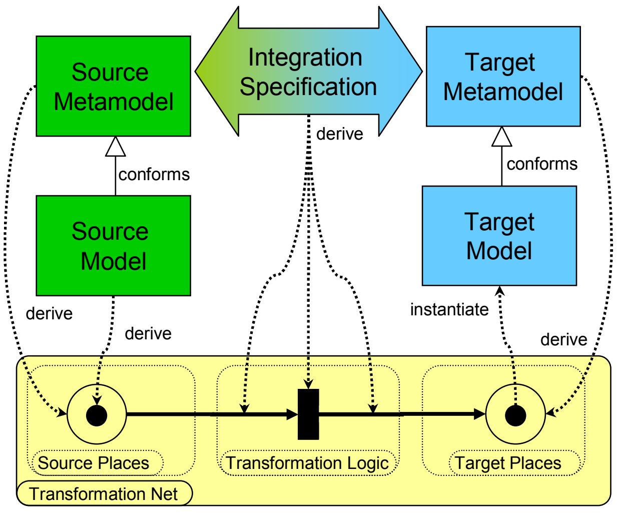

**Figure 1. Overall transformation procedure.**

The gap between the modeling and the transformation net technical space is bridged by the mapping described in the following. For reasons of brevity, we give a mapping only for the three main elements of metamodels, that are classes, references and attributes, and leave other constructs (e.g.: enumerations) aside.

**Classes, references and attributes** of metamodels are mapped to *places* of a transformation net.

**Objects**, as instances of classes are mapped to *one-colored* tokens within a place that corresponds to the object’s class. The token’s color represents an object’s unique ID.

**Links** between objects conforming to a certain reference are mapped onto *two-colored tokens* within a place that corresponds to the link’s reference. The two colors represent a link’s source (ring color) and target (center color) and stand for the ID of the linked objects.

**Values** of attributes are mapped onto *two-colored* tokens within a place that corresponds to the values’ attribute. The two colors represent an object’s unique ID and the denoted value.

To complete the transformation net and to provide the actual process logic, a system of transitions and places has to be established that is capable of streaming tokens from the places corresponding to the input metamodels to places corresponding to the output metamodel. Thereby, the transitions represent interacting entities that control the token streams by firing and removing tokens from their input places and adding tokens to their output places accordingly. During execution, state information is explicitly provided by the markings of places, which makes it possible, to trigger transitions according to a certain runtime state, as opposed to only act upon data comprising the input model. The notion of triggering transitions according to runtime events or states is similar to the notion of point-cuts determining the execution of advice in aspect-oriented programming. Hence, transformation nets naturally cater for the use of aspect-oriented techniques on the runtime level. How to incorporate a weaving mechanism on the language level will be discussed as part of next section's example which introduces a high-level integration language and demonstrates transformation net generation and execution.

## 3. EXAMPLE

The example in this chapter deals with the specification of a transformation between two metamodels, which is compiled into a net that finally executes the transformation process. Figure 2 shows the source and target metamodels, as well as the input model and the desired output model. As shown, a transformation between these two metamodels has to transform *array* input models into *linked-list* output models.

The transformation specification in-between the metamodels is given in an example language, which comprises several operators whose exact transformation net semantics will be given in the following section when describing the runtime level. On the language level, every operator stands for a certain processing entity, which has inputs and outputs by which individual operators can be assembled in a component-based way. For instance, the C2C (Class2Class) component takes objects from the "Element" class as input, and outputs them into the "Node" class. Analogously the R2R (Reference2Reference) component streams links from "contains" to "head". The C2C component offers another output port "history", of which all this components yet handled tokens can be accessed. The 2-Buf component connected to C2C's "history" sequentially fills an internal buffer of size two, which is again provided as output port. A Linker component takes the two objects in the buffer, and produces a link between them which is streamed into the "next" place. A back-link is produced by the Inverter component that produces back-links from the "next" place and streams them into the "prev" place.

Additionally to these "manually" assembled components, certain operators can cross-cut a transformation specification: Because the target metamodel classes do not have ID attributes, these should be stored within an annotation for eventual round-tripping. This can be accomplished by the Att2Annot component, which henceforth crosscuts the transformation of every object and is therefore woven with every C2C component. The transformation specification is itself a model, and due to the component-like assembly, existing model weavers can be used to merge the aspect operator into the base transformation specification. The top of Figure 2 shows the aspect's definition in a notation inspired from XWeave [2]. The query in the aspect selects all C2Cs, with three additional sub-queries "in.id", "history" and "out", relative to the current C2C operator. The results of "in.id" and "history" are bound to the "values" and "objects" ports of the Att2Annot operator, which for every transformed object instantiates a new Annotation object ("class" port) which is linked up ("ref" port) with the according Node object and sets its text attribute ("att" port) to the value of the source objects "id" attribute. Additionally, the "Annotation" class is woven into the target metamodel, as indicated through the dotted lines in Figure 2. Thereby, the result of the "out" query determines the classes to which an "annot" reference will be added.

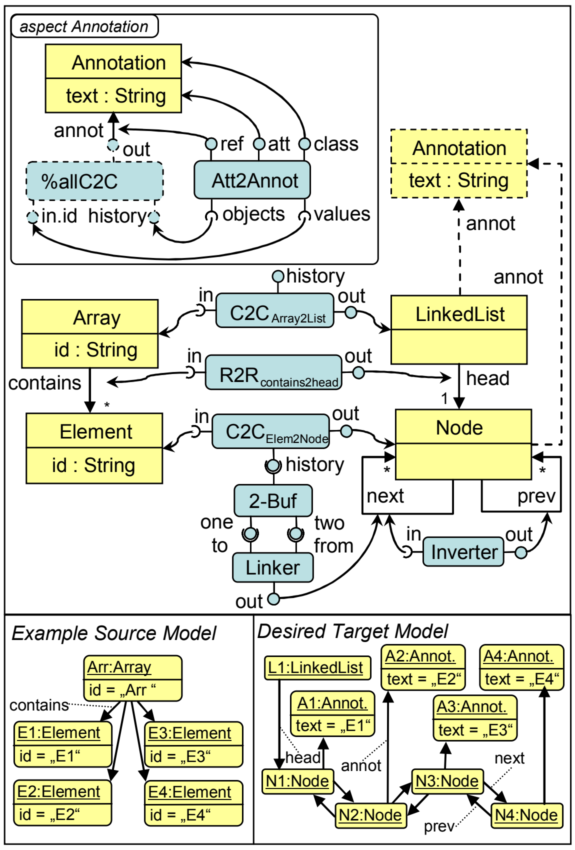

**Figure 2. Integration specification between metamodels with example models.**

After the weaving process is carried out on the language level, generation takes place to produce a transformation net out of an integration specification. Thereby Petri-net patterns are instantiated according to the transformation net semantics of the operators and assembled according to the overall integration specification. Every such pattern declares input and output arcs which represent the component ports of the respective language operators. The top of Figure 3 shows a transformation net resulting from the above integration specification. The transitions' firing rules are defined with a visual notation that uses pattern-filled tokens that can match for certain input tokens and produce output tokens whose color is either different, the same, or a combination (two-colored tokens) of the matched input colors. Places marked as "ordered" index contained tokens and provide them in a sorted fashion. For instance, the R2R component's transition matches "ArrE1" – the "first" input token. Furthermore, according to the multiplicity of a reference, a place (e.g. "head") can have a capacity, which constrains the amount of tokens a place can hold. Places holding two-colored tokens (references and attributes) have a double-lined border for easier differentiation. For simplicity reasons, the example assumes only a single array object, and since there is only a single ordered reference, the Element place is compiled into an ordered place as well, as not to unnecessarily complicate the example.

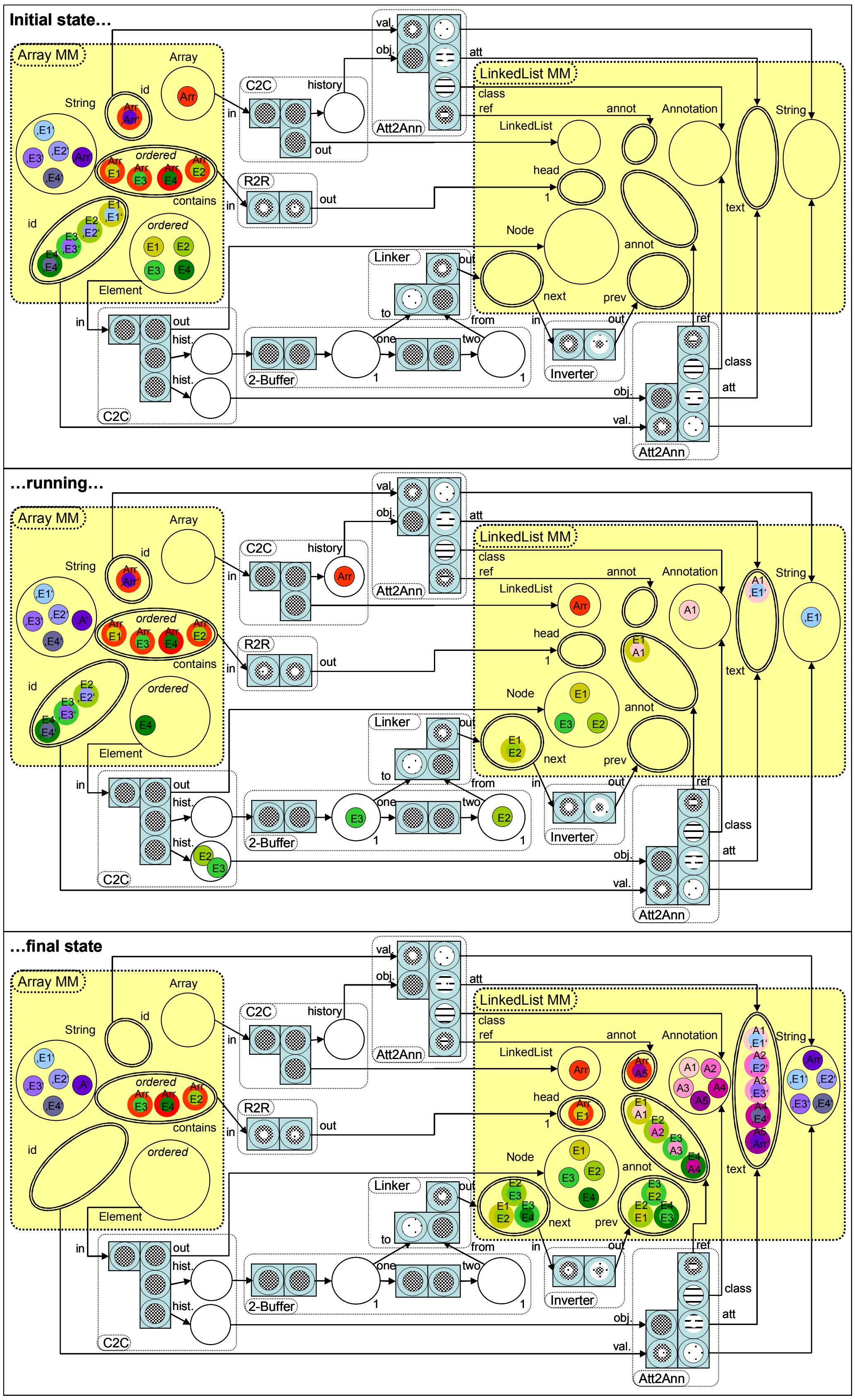

Figure 3. Transformation net execution.

The middle and the bottom of Figure 3 show the transformation net during execution and in its finished configuration. For instance, one can see how the tokens streamed through the C2C component are stored in its "history" place. (The history place is duplicated in the lower C2C, as both the Att2Annot and the 2-Buf components are bound to it.) The 2-Buf component takes in these tokens and fills its two-place buffer. Once the buffer is full (both places have a capacity of just one token), the Linker component's transition can fire and empty the buffer, producing a two-colored token which is streamed into the "next" place. Thereby it is to note, that the creation of two-colored tokens for the "next" link is based on a certain state of the execution, rather than on the input model alone.

Furthermore, one can see how the previously weaved operators form Petri-net patterns that become active after an Array or Element token was streamed. As an example, in the "running" net, the lower Att2Annot pattern has already created an annotation with the according value for the "E1" object, and is currently enabled to do the same for "E2" and "E3", as both have already been handled by a C2C component. Analogously, the rest of the patterns stream tokens from source to target places, possibly depending on other patterns in turn. The actual firing order, however, is handled by the underlying Petri-net engine. Once the transformation process has finished, the final net configuration is used to instantiate a model that conforms to the target metamodel, as shown in the bottom-right corner of Figure 2.

## 4. CONCLUSION AND FUTURE WORK

In this paper we have presented a new execution model for model transformations based on colored Petri-nets. Such a process-oriented execution model embodies the strengths of imperative and declarative paradigms and is able to explicitly represent a transformation's execution state, which furthermore allows for the natural integration of aspect-oriented transformation rules. Furthermore, although transformation nets are intended as a low-level execution model, transformation tasks like establishing the correct links in the above linked-list example can be expressed elegantly and encapsulated in reusable components.

Currently we have developed the TROPIC prototype (TRansformations on Petri-nets In Color) which can transform integration specifications established with the CARMEN mapping framework [10] into colored Petri-nets that can be executed using the ExSpecT [1] tool. After execution, the resulting Petri-net is transformed into the actual target model. The CARMEN framework builds upon an integration language that provides operators for bridging schematic heterogeneities between metamodels and ontologies. Future work will deal with extending the existing set of integration operators and generators. Due to the fact, that the transformation net approach is very generic, we will furthermore investigate in how well the approach is applicable to other model management tasks, such as model merging or incremental transformations.

Another advantage of a process-oriented view is that a transformation net represents a single artifact which embodies metamodels, models and execution logic altogether. Therefore, we deem a Petri-net based execution model beneficial for debugging purposes and visualization of a transformation's state. Consequently, besides developing generators for further integration languages (e.g.: model merging) or existing model transformation languages, our next steps will focus on developing dedicated tool support in the form of editors and debuggers for the transformation net formalism.

## 5. REFERENCES

[1] ExSpecT – Executable Specification Tool. http://www.exspect.com

[2] I. Groher and M. Völter. XWeave: models and aspects in concert. *Proceedings of the 10th international workshop on Aspect-oriented modeling, (AOSD 2007)*,Canada, Vancouver: 35-40.

[3] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica. 2005.

[4] D. S. Kolovos, R. F. Paige, and F. A.C. Polack. The Epsilon Object Language (EOL). In *Proc. of European Conference in Model Driven Architecture (EC-MDA)* Bilbao, Spain:128-142, 2006.

[5] A. Königs. Model Transformation with Triple Graph Grammars. *Model Transformations in Practice*, Satellite Workshop of MODELS 2005, Montego Bay, Jamaica, 2005.

[6] P. –A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, October 2005.

[7] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovsky, U. Prange, D. Varro, and S. Varro-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Proc. Workshop Model Transformation in Practice*, Montego Bay, Jamaica, October 2005.

[8] M. Völter, B. Kolb, . Efftinge and A. Haase. Introduction to openArchitectureare 4.1.x. *MDD Tool Implementers Forum*, TOOLS Europe, 2007.

[9] M. Völter, B. Kolb, . Efftinge and A. Haase. From Front End To Code - MDSD in Practice. *Eclipse Corner Article,* June 2006. http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html

[10] M. Wimmer, H. Kargl, M. Seidl, M. Strommer and T. Reiter. Integrating Ontologies with CAR-Mappings. *First International Workshop on Semantic Technology Adoption in Business (STAB'07)*, Vienna, Austria, May 2007.