# Web-Scheme Transformers By-Example

**DISSERTATION**
zur Erlangung des akademischen Grades
eines Doktors der Sozial- und Wirtschaftswissenschaften

Eingereicht an der Johannes Kepler Universität Linz
Institut für Wirtschaftsinformatik –
Data & Knowledge Engineering

Begutachtung:
o.Univ.-Prof. Dipl.-Ing. Dr. Michael Schrefl
a.Univ.-Prof. Mag. Dr. Werner Retschitzegger

Verfasst von: **Mag. Stephan Lechner**

Linz, im Mai 2004

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Dissertation selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe. Diese Dissertation habe ich bisher weder im Inland noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt.

Linz, am 28. Mai 2004

# Danksagung

Es bereitet mir große Freude, mich an dieser Stelle bei allen bedanken zu können, die mich beim Verfassen dieser Arbeit unterstützt, gefördert und inspiriert haben. Besonderer Dank gebührt meinem unermüdlichen Betreuer o.Univ.-Prof. Dipl.-Ing. Dr. Michael Schrefl, der mir zu jeder Tages- (und Nacht)zeit für Diskussionen zur Verfügung gestanden ist und mit wertvollen Ratschlägen weitergeholfen hat. Zudem bedanke ich mich bei ihm dafür, dass er mir Gelegenheit gegeben hat, die dieser Arbeit zugrunde liegenden Beiträge auf Konferenzen in Japan, Tschechien und Zypern zu präsentieren. Herzlicher Dank gebührt auch a.Univ.-Prof. Mag. Dr. Werner Retschitzegger, der die Zweitbegutachtung übernommen hat.

Für Rat und Hilfe sowie für das angenehme Arbeitsklima an der Abteilung danke ich weiters all meinen Kollegen, namentlich Andreas Bögl, Margit Brandl, Christian Eichinger, Mathias Goller, Ingeborg Liedlbauer und Roman Schendl, allen voran jedoch Günter Preuner, dessen stets prompte und kompetente Unterstützung insbesondere in der "Diss-Endphase" wesentlich zum Gelingen dieser Arbeit beigetragen hat. Weiters sei meinen Diplomanden Michael Karlinger und Andreas Wabro dafür gedankt, dass sie mit ihren Ideen und ihren Diplomarbeiten einen Beitrag zu dieser Arbeit geleistet haben.

Bedanken möchte ich mich auch bei meiner Mutter und meinen Großeltern, die meinen universitären Werdegang in jeder Hinsicht unterstützt haben, sowie bei meinen Schwiegereltern in spe, die wesentlich für mein leibliches und seelisches Wohl gesorgt haben.

Meiner Freundin Carolin danke ich für all das Verständnis, das sie für meine Arbeit aufgebracht hat, für all die Geduld, die sie mit mir gehabt hat, und für all die liebevolle Unterstützung, die ich durch sie erfahren habe.

Stephan Lechner.

# Kurzfassung

Komplexe Webapplikationen werden zunehmend konzeptuell mit Hilfe eines Schemas entworfen, das schrittweise erweitert und verfeinert wird. Dabei kristallisieren sich Muster von Entwurfsschritten heraus, nach denen Schemata unabhängig vom Anwendungskontext immer wieder in sehr ähnlicher Weise erweitert und verfeinert werden. So wird etwa nach dem Definieren eines Entitätstyps im Datenschema häufig eine entsprechende Seitenklasse im Hypertextschema eingefügt, die festlegt, wie die Instanzen dieses Entitätstyps auf einer Webseite darzustellen sind. Es wäre daher von Vorteil, wenn Anwender wiederkehrende Entwurfschritte automatisieren könnten, anstatt sie immer wieder manuell durchzuführen. Dieses Automatisieren von Entwurfsschritten wird durch "Transformer" ausgedrückt, die, wenn man sie auf ein Schema anwendet, dieses entsprechend erweitern und verfeinern. Der Einsatz solcher Transformer beschleunigt den Entwurfsprozess, unterstützt prototyping-orientierte Entwicklung von Webapplikationen und verbessert deren Qualität. Zu diesem Zweck ist es allerdings notwending, dass Transformer sowohl einfach zu definieren als auch einfach zu verwenden sind. Dazu kommt, dass Muster von Entwurfschritten zwar immer wieder ähnlich aber doch leicht unterschiedlich durchgeführt werden. Deswegen sollten Transformer entsprechend adaptierbar sein.

Diese Arbeit stellt die Sprache TBE (Transformer-by-example) für das Definieren und Anwenden von Transformern vor. Ein Spezifikum von TBE ist, dass ein Transformer durch zwei generische Beispielschemata (d.h. "by-example") beschrieben wird, die jeweils das Schema vor und nach der Transformation widerspiegeln. Dadurch wird ein Transformer im Wesentlichen in der gleichen (grafischen) Notation definiert, in der auch Schemata definiert werden. Aus diesen Beispielen leitet TBE ab, wie die Transformation durchzuführen ist, wenn der Transformer auf ein Schema angewandt wird. Eine weitere Besonderheit von TBE ist, dass das Verhalten vordefinierter Transformer individuell angepasst werden kann. So kann etwa die Konvention für das Benennen der neu zu generierenden Elemente geändert werden.

Weiters kann der Transformer individuell eingeschränkt werden, sodass er nur Teilbereiche eines Schemas erfasst. Diese Flexibilität ist speziell für den Einsatz im Entwurfsprozess notwendig.

Die Konzepte von TBE sind unabhängig von einer konkreten Modellierungssprache definiert, sodass verschiedene Sprachen für den konzeptuellen Entwurf von Webapplikationen gleichermaßen um das Definieren und Anwenden von Transformern erweitert werden können. In der vorliegenden Arbeit wird die Modellierungssprache WebML als Beispiel verwendet und eine prototypische Implementierung von TBE für WebML-Schemata beschrieben. Neben dem Webbereich lassen sich die Konzepte von TBE aber auch auf andere Bereiche übertragen, etwa auf Modellierungssprachen wie UML oder auf beliebige Diagramme, wie sie beispielsweise in Microsoft Visio erstellt werden können.

# Abstract

When defining a scheme of a web application, modelers repeatedly perform modelling tasks like “after having defined an entity type, add a page class for displaying the entity type’s content”. Thereby, a scheme is extended again and again in a similar manner. For such recurrent tasks, it would be convenient for modelers to have web scheme transformers (short transformers) that, when applied to a scheme, automatically perform such tasks. Extensive reuse of well-tested transformers facilitates the modelling process, contributes to rapid prototyping, and improves a web application’s quality. For this purpose, it should be easy for modelers to define and use transformers. Further, since the same modelling task is performed repeatedly in a similar manner yet with slight variations, transformers should be capable of considering particularities of individual applications.

This thesis introduces *TBE* (Transformers-By-Example), which is a language for defining and applying web scheme transformers. In order to facilitate definition of transformers, *TBE* follows a by-example approach, where a modeler defines a transformer by giving a generic example of a web scheme before and after transformation instead of specifying operations that perform the transformation. These operations are derived by *TBE* based on the provided generic examples. Transformers are defined graphically and in a notation that is similar to one with which modelers are familiar. In order to facilitate their flexible use, the behavior of a transformer can be individually adapted for each particular application. For both, expressing transformer definitions and (individualized) applications, *TBE* provides language constructs that abstract from a scheme’s internal representation and that are particularly useful for supporting modelling tasks as performed by modelers.

The concept of defining transformers by-example can be introduced in various languages and tools for modelling web applications. This is demonstrated by WebML, which is a popular web modelling language, and by WebRatio, which is a commercial tool for editing WebML schemes. A prototype of *TBE* is described, too. Further, besides enhancing web application modelling, the concepts of *TBE* can be applied to other conceptual modelling languages as well, e.g. UML. Further, *TBE* could serve as language for precisely defining design patterns, or it could facilitate the definition of diagrams within general-purpose diagram editors like, for example, Microsoft Visio.

# Contents

# Chapter 1

# Introduction

## Contents

## 1.1 Motivation for web scheme transformers

Web applications are information systems that are accessible via the web basically through navigating between pages, entering data in web forms, and triggering operations by traversing links [37]. Such web applications have, since the early stage of Web development, continuously grown in size and complexity. Today it is not uncommon that, for example, an e-commerce application provides tens of thousands of pages usually generated out of an underlying data source.

Designing such web applications is a complex task that requires to integrate several aspects like, for example, defining the data source that provides the web application's content and defining the hypertext layer that presents this content through navigable web pages. Development gets even more complex if the application shall support different front end devices or shall be customized, i.e. provide content specifically tailored to different users or user groups.

A promising approach to manage this complexity is to develop web applications using conceptual web modelling languages like, for example, [32, 37, 59, 70, 98, 123, 134]. Such languages usually integrate multiple models that cover the different aspects of web application design. For example, most languages provide at least a content model and a hypertext model that cover the aspects of defining the data source and the hypertext layer, respectively. Note that these models are not independent as, for example, the hypertext model depends on the content model. Thus, a web application is defined through a web application scheme which integrates particular sub-schemes such as a content sub-scheme and a hypertext sub-scheme. The web application scheme will subsequently be shortly referred to as *scheme*.

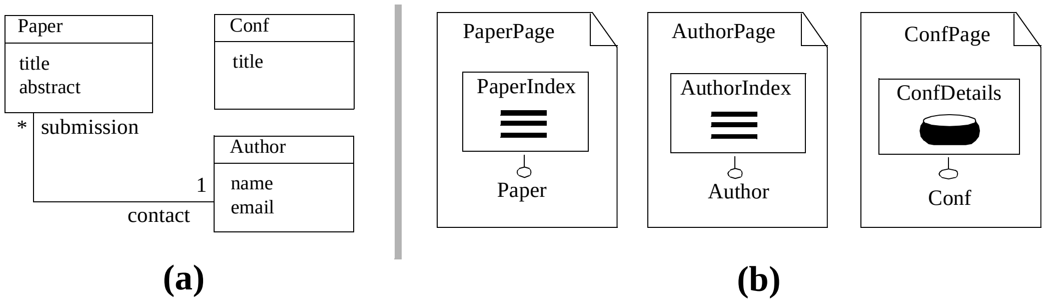


Figure 1.1: WebML scheme of conference web site: content sub-scheme (left), hypertext sub-scheme (right)

**Example 1** *Fig. 1.1 depicts the scheme of a conference web site defined with the conceptual web modelling language WebML [32]. The left part illustrates the content sub-scheme that defines the web site's content in terms of an entity-relationship diagram. Entity type* `Conf` *describes information about the conference itself. This entity type is intended to be a singleton, i.e. to contain only one member. Entity types* `Author` *and* `Paper` *describe information about authors and submitted papers, respectively.*

*The right part of Fig. 1.1 illustrates the hypertext scheme that defines how the content is to be organized in web pages. This organization is expressed by page classes that contain content units referring to entity types. Page class* `AuthorPage` *contains an index unit* `AuthorIndex` *and defines that a list of authors is to be presented, i.e. members of entity type* `Author`*. Page class* `PaperPage` *is structured analogously. However, for page class* `ConfPage`*, an index unit for presenting a list of conferences is not appropriate because entity type* `Conf` *contains exactly one member. Thus, page class* `ConfPage` *contains a data unit* `ConfDetails`*, which defines that this sole member is to be presented in detail on a single page.*

When defining a scheme of a web application, developers (modelers) successively extend or refine this scheme in order to cover the requirements that have been specified for the web application. It is the task of a modeler to specify proper configurations of scheme elements like entity types, attributes of entity types, page classes, links, etc. such that requirements are met.

We use the abstract term *modelling task* for a process of extending or refining an input scheme in order to achieve an output scheme that then meets particular requirements[1]. Each such modelling task is characterized by a configuration of scheme elements as before performing the task, i.e. an *input configuration*, and an extended or refined configuration of scheme elements as thereafter, i.e. an *output configuration*. For the purpose of this introduction, we mention here just the following two examples of modelling tasks, but we will provide more examples in the course of this thesis:

1. *Modelling task "IndexPCForEnt":* The members of entity types are often to be listed on hypertext pages. Consequently, after having defined an entity type, modelers often define a corresponding page class with an index unit. However, this task only makes sense if the entity type comprises at least one attribute[2]. Otherwise, there would be nothing to present on a page. Thus, the input configuration of modelling task "IndexPCForEnt" is an entity type that comprises at least one attribute. The output configuration is a page class with an index unit that has the entity type as content source.

[1]In the context of software development, a similar term "programming task" is used for the process of extending or refining a program in order to meet a particular design pattern [72].

[2]There may be entity types without attributes as well. For example, WebML does not offer a model primitive for expressing ternary relationships. Instead, such a situation is expressed through a "central" entity type that is connected through binary relationships to each of the participants of the ternary relationship. The central entity type usually does not comprise attributes but is only for connecting entities.

2. *Modelling task "AugmentPCsWithUserData":* In web applications that are accessed by registered users, pages often display data of the user currently logged in. Performing this task requires (i) to have an entity type that defines user data and (ii) to have a page class that does not yet display user data. Then, this page class is augmented by a unit that retrieves the data of the user currently logged in from the user-data entity type. Thus, the input configuration is a pair of an entity type, which represents user data, and a page class, which shall be augmented. The output configuration is an extended page class that has been augmented by a unit as described above.

Many modelling tasks are, though applied in different contexts, performed again and again in a similar manner. It would therefore be convenient for modelers to have web scheme transformers (short: *transformers*) that, when applied to a scheme, perform extensions or refinements as required. For example:

1. Concerning modelling task "IndexPCForEnt", it would be convenient to have a transformer `IndexPCForEnt` that, when applied to an entity type, generates a corresponding page class with an index unit according to some naming policy. This transformer could then be repeatedly applied to various entity types. The transformer might also be applied to a set of entity types such that corresponding page classes are generated all at once. Further, it may also be desired that the transformer selects all the entity types to which it can be applied, i.e. which comprise at least one attribute.

2. Concerning modelling task "AugmentPCWithUserData", it would be useful to have a transformer `AugementPCWithUserData` that takes an entity type representing user data and a page class as input and that augments this page class by a unit for presenting data of the current user. This transformer could then be repeatedly applied in order to augment a particular page class or a particular set of page classes. The transformer could also be applied such that is selects all the page classes of a scheme and extends them as desired. Thereby, a single application of this transformer could augment a possibly large number of page classes all at once.

From a declarative point of view, a transformer is a generalized description of a recurrent modelling task and is therefore characterized by the following two parts: (1) a generic description of the input configuration, i.e. a configuration of scheme elements that the modelling task aims to extend or refine, and (2) a generic description of how the extended or refined output configuration looks like and how it depends on the input configuration.

From the viewpoint of assisting modelers in performing modelling tasks, a transformer is executable in the sense that, when it is applied to an input configuration, it performs extensions and refinements and achieves an output configuration as intended.

The advantages of such transformers are manyfold: (1) Modelers are freed from performing similar tasks again and again in the same manner. This can be uneconomic, annoying, and cumbersome. With transformers, modelers can concentrate on the main goals, i.e. to chose a proper design for fulfilling the web site's requirements. (2) Transformers facilitate design reuse because modelers can have well-proven configurations of scheme elements generated rather easily. Consequently, the quality of web schemes is improved. (3) Transformers provide for fast development of prototypes and contribute to shortening evaluation cycles. This is particularly appropriate for the development of web applications where usability is a key success factor [32, 71, 105].

## 1.2 Requirements for web scheme transformers

In the following, we elaborate requirements that web scheme transformers (transformers) should fulfill in order to assist modelers in performing modelling tasks. In the course of this requirement analysis, we first assume that we have a library of predefined transformers available. Then, we discuss if and how modelers should be enabled to define transformers on their own.

**Understandability.** As for any predefined procedure that is to be reused, each transformer must provide a *comprehensible* and *comprehensive* description of its behavior. This description shall enable modelers to quickly grasp for what the transformer is good for, i.e. in which situations it can be employed and which result it achieves. Understandability is basically a matter of disciplined documentation and usually requires a description in textual form as well as in form of typical use-cases.

**Flexibility.** As already motivated in the examples of modelling tasks "IndexPCForEnt" and "AugmentPCWithUserData" above, the following two scenarios of how a transformer is applied can be distinguished:

1. The modeler explicitly declares to which scheme element configuration (or to which set of scheme element configurations) the transformer is applied. For example: "apply transformer `IndexPCForEnt` to entity types `Paper` and `Author`".

2. The transformer parses the scheme to which it is applied and selects *all* scheme element configurations that match the transformer's input configuration. For example: "apply transformer `IndexPCForEnt` such that simply all entity types that comprise attributes are considered, i.e. such that the modeler needs not to enumerate them explicitly".

We argue that each transformer should support both scenarios. The modelling task that a transformer supports is the same, regardless of whether the modeler explicitly declares to which scheme element configuration (or to which set of such configurations) the transformer is to be applied or whether the transformer by itself selects proper scheme element configurations. Thus, it would be unreasonable if one had to define different transformers for different scenarios of their application.

Moreover, it may even be required that a transformer supports a mix of these scenarios. Reconsider, for example, modelling task `AugmentPCsWithUserData`, which requires as input configuration (1) an entity type that represents user data and (2) a page class that is to be augmented. If all page classes of a scheme shall be augmented, it would be convenient to have them selected by the transformer. However, within a number of entity types of a scheme, usually only one entity type represents user data. Thus, it makes sense that the modeler explicitly declares which particular entity type shall serve for that purpose.

**Adaptability.** Though a modelling task is performed repeatedly in a similar manner, the way it is performed in particular may vary from application to application as follows: (1) The naming policies for the new scheme elements often vary; (2) In some cases, the modelling task requires to embed new scheme elements into existing ones, whereas in other cases the embedding scheme element it to be generated, too. (3) In some cases, it is required to embed a set of new scheme elements in one newly generated scheme element, whereas in other cases, a separate embedding scheme element is to be generated for each of these elements.

**Example 2** *Transformer* `IndexPCForEnt` *could be adapted as follows: (1) The naming policy that defines how the newly generated page classes and index units are to be named could be overridden; (2) The index unit is to be generated within an already existing page class instead of generating this page class as well. (3) The transformer could generate two index units for entity types* `Paper` *and* `Autor`*, respectively, but could embed them in one page class, which is generated as well.*

Of course, for all above mentioned variants, one could define a separate transformer. However, in order to avoid a number of transformer definitions that support the same modelling task but in slight variation, it should be possible to adapt the way a transformer generates new elements individually for each application.

**Usability.** In order to be extensively reused, it should be easy for modelers to apply transformers. For a scenario where the transformer selects proper scheme element configurations and where no adaptations are to be defined, usability is not problematic; the transformer only has to be activated, e.g. by clicking on a corresponding icon. However, if the transformer is to be applied to particular scheme element configurations only or if its way of generating scheme elements is to be adapted, the transformer should provide an proper way for specifying these individualizations.

Up to now, we have assumed that transformers are already defined and only have to be used by modelers. It remains to discuss who shall be in charge of identifying typical modelling tasks and, consequently, who shall define transformers. Depending on this target group, the language with which transformers are to be expressed is chosen.

To some extent, typical modelling tasks can be predicted by the developers of the respective conceptual web modelling language and/or the vendors of according CASE tools. For this target group, i.e. the developers of CASE tools, it is legitimate to define the language based on operators that access schemes in their internal representation, e.g. a representation in XML as maintained by a CASE-tool.

However, many modelling tasks cannot be predicted because a large variety of business requirements exist, and for each of them, various ways of defining a scheme such that these business requirements are met may be found. Thus, it would be beneficial if the users of a CASE-tool, i.e. modelers who develop a scheme, could define transformers on their own. Then, a large variety of modelling tasks repeatedly performed could be supported by transformers.

**Adequateness.** Thus, the language with which transformers are defined should target modelers who are familiar with defining schemes in a graphical representation but who are usually not familiar with internal representations of these schemes. To be adequate, the constructs provided by that language shall fulfill the following requirements: (1) They shall widely *abstract from a scheme's internal representation.* (2) Despite the necessity of abstraction, the language shall allow to *express typical modelling tasks in a practical manner.*

Of course, the criteria "typical modelling tasks" and "practical manner" cannot be formalized. Moreover, both criteria may depend on the particular modelling language for which transformers are to be defined. Thus, identifying typical modelling tasks requires to analyze modelling processes. Then it becomes possible to evaluate whether the language's constructs are practicable for supporting these modelling processes.

**Generality.** Orthogonal to the requirements understandability, usability, flexibility, adaptability, and adequateness, we should keep in mind that there is already a large number of different conceptual web modelling languages. It would therefore be beneficial if the way of defining transformers were not tailored to one particular web modelling language but could be employed for various web modelling languages.

## 1.3 Existing approaches for web scheme transformers

Currently, only a few approaches for web scheme transformation exist, which are usually provided in the context of CASE tools that support model-driven development of web applications. The following two ways of supporting transformations can be distinguished: (1) A tool may provide a set of predefined transformers that are to be used off-the-shelf, and (2) a tool may provide, besides a library of predefined transformers, a language for defining additional transformations. In the following, we briefly explain these approaches and discuss to which extent they meet the requirements introduced in the previous section.

The web modelling language WebML [32] offers a single predefined transformation that generates a default hypertext scheme based on a given content scheme. This transformation adheres to a fixed set of rules and aims to quickly generate a first draft of the hypertext scheme and, consequently, a first prototype of the web application. Examples of such rules are "For each entity type, generate a data unit" or "Data units over entity types are put in different page classes". However, the generated hypertext scheme is very coarse and requires extensive adaptations. The predefined rules cannot be parameterized, and defining new transformation rules or adapting existing ones is not supported. Thus, besides offering an easy way of generating a first draft, this predefined transformation cannot further support the development process.

The web modelling language Araneus [98] also offers a set of predefined *transformation primitives*. One such transformation primitive is, for example, `PS-FROM-NE` [3] that generates page classes from entity types. The name of a page class is obtained by concatenating the name of the entity type with the string "`Page`". However, like in WebML, these primitives are fixed and aim at generating a first prototype with a coarse structure but not at further facilitating the modelling process.

The web modelling language OO-H [70] provides a language for defining *transformation rules* [71] that drive web scheme transformations. Such a transformation rule is a sequence of schema modification operations over the internal representation of schemes of the conceptual web modelling method OO-H. These scheme modification operations are expressed in OCL-like syntax [137]. For example, creating a new page class $p$ named "head" in an OO-H abstract presentation diagram[4] (APD) named *DList* is expressed by `DList->addAPDPage (p); p.name="head"`. Adding links to all page classes named "head" such that each such page is connected to a page named "homepage" is expressed by `home=DList->select(name="homepage"); Dlist->select(name="head")->AddLink(home)`.

However, defining transformations using OCL-like scheme modification operations has the following disadvantages: (1) Transformers are specified by operations over an internal representation of schemes, e.g. a representation in XML. However, in a conceptual web modelling language, modelers usually edit schemes graphically and do not specify operations at the level of an internal representation. Further, defining transformers that comprise many scheme modification operations can be difficult and error prone. Thus, modelers might hesitate to define their own transformers. (2) As the semantics of

[3]In Araneus, "PS" stands for a page scheme and corresponds to a page class. "NE" stands for navigational entity and corresponds to an entity type.

[4]An OO-H abstract presentation diagram corresponds to a hypertext scheme enhanced with presentational features.

transformers is defined in terms of textual operations over an internal representation of a scheme, extra documentation is required for each transformer to describe how it behaves and how it is to be applied. Otherwise, modelers might hesitate to use existing transformers. (3) Transformation rules, once defined, can be applied only as they are; they do not provide an interface that allows modelers to individually adapt the rule's behavior.

A straightforward approach to define transformers would be to use a macro language for specifying sequences of scheme modification operations. Many productivity tools like, for example, Microsoft Word or Excel, already provide a macro language for manipulating their respective documents, and such an approach could also be employed for web development CASE tools. Such macros usually are parameterized by a "selection", i.e. a set of scheme elements a user has selected in the editor at the time when the macro is called. Of course, one can define macros with additional input parameters, but the way for gathering these parameters at the time when the macro is applied must be encoded in the macro.

However, the flexibility and adaptability of transformers defined in macro-languages is limited. If the scenario where the transformer collects proper scheme element configurations shall be supported, then this collection procedure must be hard-coded in the macro and can therefore not be adapted at the time when the macro is applied. If the way the transformer generates new scheme elements shall be adaptable, the macro must provide additional input parameters that are then considered in conditional expressions like IF-statements within the macro. Further, macro languages also require to specify operations based on an internal representation of schemes. Thus, apart from supporting parametrization, defining transformers with macro languages has the same disadvantages as the scheme modification operations of OO-H.

Transformers could also be expressed through programs or scripts specified in a general purpose programming or query language like Java [130], XSLT [36], or XQuery [14]. However, these programs or scripts again require to access schemes in their internal representation such that defining transformers that way compares to defining them with a macro language.

Fig. 1.2 summarizes how existing approaches for web scheme transformation fulfill the requirements that have been identified in the previous section. All approaches require to specify operations over the internal representations of schemes such that requirement understandability is only moderately fulfilled. None of these approaches except that of using macro languages allows for

| Approaches / Requirements | WebML's Built-In Trans-formation | Araneus' Built-In Trans-formations | OO-H Trans-formation Rules | Macro-Languages, Programs, Scripts |
|---|---|---|---|---|
| Understandability | ~ | ~ | ~ | ~ |
| Flexibility | – | – | – | ~ |
| Adaptability | – | – | – | ~ |
| Usability | + | + | + | + |
| Adequate for user-defined Transformers | – | – | ~ | ~ |
| Generality | – | – | – | – |

Figure 1.2: How existing approaches for web scheme transformation fulfill identified requirements.

parameterizing predefined transformers such that requirements flexibility and adaptability are not fulfilled. Transformers specified in a macro-language can be parameterized, yet requirements flexibility and adaptability are still fulfilled only moderately.

Of course, in absence of parametrization, transformers are easy to use as they simply have to be activated. Transformers specified in a macro-language are easy to use if they are parameterized by a "selection". If additional input parameters are required, usability depends on the way these input parameters are to be provided. WebML and Araneus do not support user-defined transformers. OO-H and macro languages allow for specifying user-defined transformers, but they require knowledge of the internal representations of schemes. Thus, requirement adequateness is fulfilled only moderately. For the same reason, requirement generality is not fulfilled, because different modelling languages use different internal representations of schemes.

## 1.4 By-example web scheme transformers

In this thesis, we present *TBE* (transformers-by-example) as a concept for defining web scheme transformers. We will subsequently refer to web scheme transformers that are defined in *TBE* as *by-example transformers* or shortly as *transformers*. *TBE* adopts a visual by-example approach, where the transformer's semantics is substantially defined in a declarative manner. As a novelty, we adopt the graphical notation used for defining schemes now for defining transformers. This has already been motivated in previous work which this thesis builds on [90, 89]. The rest of this section is organized as follows: We start with explaining *TBE*'s basic principles. We then show how the identified requirements understandability, usability, flexibility, adaptability, adequateness, and generality are addressed by *TBE*.

**Defining by-example transformers.** In the definition of a transformer, we explicitly distinguish the following two parts: (1) One part defines the input configuration. It is specified through constraints that a particular configuration of scheme elements must fulfill in order to get extended or refined by the transformer. Based on this declarative specification, *TBE* derives a query that retrieves from a scheme all scheme element configurations that match the input configuration. Thus, we refer to this part as the transformer's *query part*. (2) The other part generically defines the output configuration based on the input configuration. For each scheme element configuration matching the input configuration, the output configuration defines how to generate new scheme elements and how to modify existing ones. We refer to this part as the transformer's *generative part*.

As we adopt a visual by-example approach, a transformer's query part and generative part are both expressed by giving an "example" of what is desired instead of specifying operations for achieving the result. These "examples" are expressed basically in the same graphical notation as used for defining schemes. However, the "examples" are generic specifications from which the *TBE*-system derives an executable transformer. Thus, we will refer to these examples as *templates*, and for expressing a transformer's query part and generative part, we consequently distinguish between a *query template* and a *generative template*, respectively. A pair of a query template and a generative template makes up a transformer definition.

**Example 3** *Fig. 1.3 depicts a by-example definition of transformer* `IndexPCForEnt` *in terms of the web modelling language WebML. Thus, the transformer defines a transformation of WebML schemes and is, apart from the symbols "$\surd$" and "$\oplus$" and the string concatenation expressions, graphically notated like a WebML scheme.*

*Without having explained the semantics of these symbols, one should quickly grasp the transformer's purpose: The query template shown in*

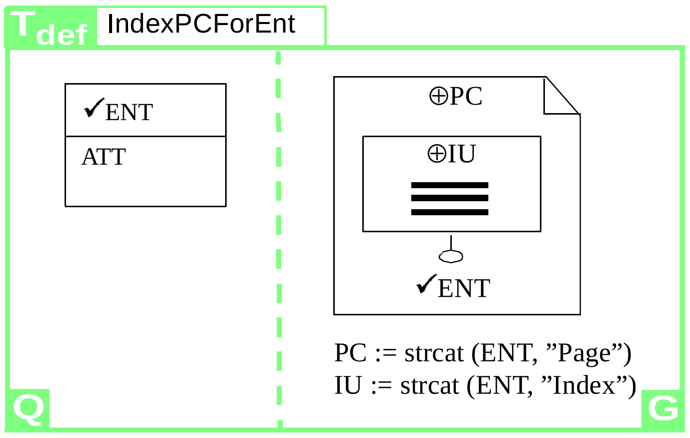


Figure 1.3: Transformer `IndexPCForEnt`: by-example definition in WebML notation

*the left part marked with "*`Q`*" looks like an entity type comprising an attribute; it defines that entity types that comprise at least one attribute are to be selected. Similarly, the generative template shown in the right part marked with "*`G`*" looks like a page class comprising an index unit; it defines that a page class comprising an index unit is to be generated.*

Each template is defined by using the same scheme elements and the same notation as used for specifying schemes. However, there are the following two differences: (1) Instead of concrete values, the entries in scheme elements are *variables*. In order to express this characteristic, we will subsequently refer to scheme elements comprising variables as *scheme element skeletons*. (2) Additionally, a template contains symbols, comparison constraints, and expressions. A fixed set of symbols is used for tagging variables in order to distinguish, for example, variables representing parameters from variables representing scheme elements to be generated. Comparison constraints are used to constrain variable bindings. Expressions define how to derive new property values, e.g. by means of string concatenation.

**Example 4** *Reconsider the transformer shown in Fig. 1.3. The query template has the following meaning: Variables* `ENT` *and* `ATT` *represent entity types and attributes, respectively, because they are placed in the name section and in the attribute section of a graphical shape representing an entity type. In the context of a template, this shape is referred to as an entity type skeleton.*

*When applied to a scheme, the query looks for valid bindings of variables* `ENT` *and* `ATT` *out of the domain of entity types and attributes, respectively. Since variable* `ATT` *is placed in the same entity type skeleton as variable* `ENT`*, only those bindings are valid where the attribute represented by* `ATT` *is defined with the entity type represented by* `ENT`*. Variable* `ENT` *is a result-variable as it is preceded by symbol* $\surd$*, whereas variable* `ATT` *is a non-result variable. Thus, the query's result comprises only the entity types but not the attributes.*

*The generative template has the following meaning: Variables* `PC`*,* `IU`*, and* `ENT` *represent a page class, an index unit, and an entity type as expressed by the graphical placement of these variables. Variables* `PC` *and* `IU` *are new-element variables as expressed by symbol "*$\oplus$*"; they represent a new page class and a new index unit to be generated. Variable* `ENT` *is a parameter variable as expressed by symbol* $\surd$*; it represents an entity type that is to be provided as parameter each time the generative template is instantiated. This parameter is usually provided through a corresponding result variable of the query template.*

*Due to the graphical arrangement of variables* `IU` *and* `PC`*, the new index unit is embedded within the new page class as represented by these variables, respectively. Variables* `IU` *and* `ENT` *are placed in the same index unit skeleton, where the former represents the index unit's name and the latter represents the content source. Hence, the entity type provided as parameter becomes the content source of the new index unit. The names for the new page class and the index unit are defined by attaching literals* `"Page"` *and* `"Index"` *to the entity type's name, respectively, as defined by the respective* `strcat`*-expressions denoting string concatenation.*

**Applying by-example transformers.** A transformer, once defined, can be applied to various schemes, each time extending or refining the scheme as defined by the transformer's templates. We first explain how a transformer behaves if it is applied off-the-shelf and then show how to adapt a transformer's behavior with each individual application.

An off-the-shelf application of a transformer is processed as follows: First, the query template is evaluated in the context of the scheme to which the transformer has been applied. It achieves a relation whose tuples represent configurations of scheme elements that match the transformer's input configuration. In a second step, the generative template is iteratively instantiated for each such tuple $t$, each time having the generative template's parameter variables bound to the corresponding scheme elements in $t$.

**Example 5** *Suppose that by-example transformer* `IndexPCForEnt` *depicted in Fig. 1.3 is applied to the content scheme illustrated in the left part of Fig. 1.1. Then, entity types* `Paper`*,* `Author`*, and* `Conf` *are selected because they all match the pattern "Entity type comprising an attribute". For each of these entity types, the generative template is instantiated separately. For example, for entity type* `Paper`*, a page class* `PaperPage` *comprising an index unit* `PaperIndex` *referring to entity type* `Paper` *is generated. Similarly, page classes with index units are generated also for entity types* `Author` *and* `Conf`*.*

Besides applying transformers off-the-shelf as explained above, *TBE* offers the following alternatives for adapting a transformer's behavior individually for each application: (1) Modelers may individually constrain query template variables in order to control which parts of the scheme shall be considered. Such an individual constraint will be referred to as an *application-specific constraint* and is comparable to specifying additional WHERE-clauses when applying a predefined SQL statement. (2) Modelers may specify expressions that override those defined in the generative template in order to adapt the transformer's outcome. Such an individual expression will be referred to as *application-specific expression*.

In order to facilitate the specification of application-specific constraints and expressions, a transformer is applied graphically through a graphical shape representing the transformer. Thereby, commonly used types of constraints and expressions can be expressed graphically in a convenient manner.

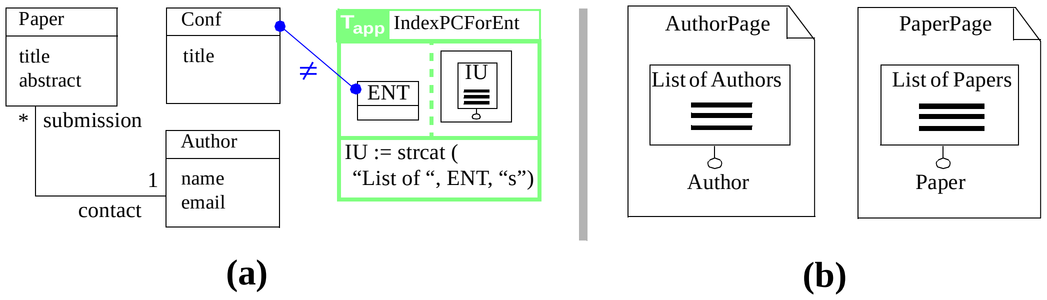


Figure 1.4: (a) Individualized application of transformer `IndexPCForEnt`: page classes for all entity types except `Conf`; (b) Result of (a)

**Example 6** *Suppose that the application of transformer* `IndexPCForEnt` *shall be individualized such that (1) entity type* `Conf` *is not considered,* *and (2) the names of index units follow a naming policy of the form* `"List of " + ENT + "s"`*, e.g. as in "List of Papers". Fig. 1.4(a) illustrates this individualized application.*

*Transformer* `IndexPCForEnt` *is represented by a graphical shape, which repeats the scheme element skeletons of the transformer definition. For reasons of conciseness, however, many details from the transformer definition have not been taken over, e.g. the expressions and the variables* `ATT` *and* `PG`*. Note that abstracting from details does not influence the transformer's behavior.*

*The application is individualized (1) by application-specific constraint* `ENT` $\neq$ `"Conf"`*, which is notated graphically by the line from entity type* `Conf` *to the entity type skeleton represented by variable* `ENT`*, and (2) by application-specific expression* `IU := strcat ("List of ", ENT, "s")`*, which overrides the expression defined by the transformer, i.e.* `IU := strcat (ENT, "Index")`*. The result of this application is depicted in Fig. 1.4(b).*

**Understandability.** By-example transformers facilitates understandability because the transformer definition by itself is a *comprehensive* and *comprehensible* description of the transformer's behavior.

Comprehensibility is achieved in two ways: (1) The language for defining transformers extensively reuses the graphical notation of schemes. Thus, modelers are familiar with this notation. (2) Further, the query template and the generative template represent generic examples of a scheme as it looks like before and after performing a modelling task, respectively. Due to this separation, the transformer describes a typical use-case, i.e. a typical situation where it can be applied and a typical result that it then achieves. Thus, we argue that a by-example transformer enables modelers to quickly grasp the transformer's purpose.

Comprehensiveness is achieved in that the templates expose all the transformer's semantics and there is no "hidden code". Of course, for analyzing the detailed meaning of by-example transformers, one has to understand how query templates and generative templates work together and how the symbols are to be read. However, *TBE* requires only a few such symbols, far less than the large number of operations usually provided by macro languages. This is possible since *TBE* 's "language constructs" are for the most part scheme element skeletons, which are directly taken over from the modelling language at hand.

**Flexibility** By-example transformers can be flexibly applied because they support off-the-shelf applications as well as individualized applications. By default, a transformer selects all proper scheme element configurations by itself. However, the same transformer can be applied with application-specific constraints that individually constrain the transformer to consider only a particular scheme element configuration or a particular set of scheme element configurations.

**Adaptability.** By-example transformers achieve adaptability through the following constructs that can be specified with each individual application of the transformer: (1) Each generative template expression that derives a value for a new scheme element can be overridden by an application-specific expression. (2) Each new-element variable can be turned into a parameter variable such that a scheme element that would be generated is replaced by an existing scheme element that is provided at transformer application. (3) Each new-element variable of the generative template can be pinned and is instantiated only once per transformer application rather than iteratively for each tuple of the query template's result.

**Usability.** By-example transformers are easy to use because their graphical application is specifically tailored to specify the individualizations mentioned in the paragraphs *Flexibility* and *Adaptability*. Of course, a simple off-the-shelf application can be expressed easily, too.

**Adequateness.** As mentioned in the requirements analysis, the langauge with which transformers are defined should (1) abstract from a scheme's internal representation and (2) allow to express transformers for typical modelling tasks in a practical manner. These issues are discussed in the following.

*TBE* abstracts from a scheme's internal representation because it substantially adopts scheme elements as skeletons but keeps the number of additional language constructs like symbols, comparison constraints, and expressions rather small. Thus, we argue that modelers who are familiar with defining schemes and with using existing transformers will quickly be able to define their own transformers. Actually, for many modelling tasks, defining a transformer is just like performing the task once in a sample scheme. However, instead of entering concrete values, one chooses proper variable names. Further, one annotates symbols for distinguishing different kinds of variables and, if necessary, specifies comparison constraints and expressions that are not yet expressed through the graphical arrangement of the scheme element skeletons.

Whether *TBE* can assist average modelers in the modelling processes depends on *TBE*'s practicability. We have already argued that our visual by-example paradigm, which substantially builds on the notation of scheme elements as provided by a modelling language at hand, targets modelers who are familiar with exactly that notation. However, practicability has to be analyzed from the viewpoint of which modelling tasks are typically performed and, subsequently, which language constructs are appropriate for defining scheme transformers supporting such tasks.

In order to figure out typical modelling tasks, we have taken the following approaches:

1. *Analysis of own projects.* Many modelling tasks have been drawn from our experience in web application modelling; whenever we have extended or refined a scheme repeatedly in a similar manner, we have tried to describe these extensions and refinements as a general modelling task. We suppose this as the primary way of how modelers will define modelling tasks and, subsequently, scheme transformers.

2. *Analysis of third-party projects.* We have analyzed web schemes of third party development projects, although only a few such web schemes are documented or publicly available, e.g. the ACME furniture example [139], the Web Forum example [140], and the conference management use case [5, 25, 33, 125].

3. *Analysis of web modelling languages.* Web modelling languages already suggest many modelling tasks in form of "design patterns" or "common use cases", e.g. as in [29, 71].

4. *Analysis of hypertext patterns.* A great pool of well-documented hypertext patterns [12, 115, 114, 116, 64, 124] already exists, each describing a recurrent hypertext design problem and an *abstract solution*, i.e. independent of modelling languages. Of course, these patterns are not yet transformers because they are neither "executable" nor are they expressed *concretely*, i.e. as well-proven configurations of scheme elements in terms of a particular modelling language. However, such patterns motivate the definition of such configurations and, consequently, the definition of transformers that generate such configurations.

Most of the identified modelling tasks are of the form "if a configuration of scheme elements matches a particular input configuration, then extend this configuration by a set of new scheme elements", where most input configurations are rather simple. They are usually of the form "there must be a set of scheme elements that are arranged in one particular manner". Most extensions and refinements are simple, too. They basically require to generate exactly one set of new scheme elements in one particular arrangement.

**Example 7** *Examples of simple modelling tasks are: (a) For each entity type comprising attributes, generate a page class with an index unit; (b) For each pair of a page class with a data unit* du *and a page class with an index unit* iu*, create a link from* iu *to* du*. (c) For each page class with an index unit* iu*, extend this page class by an index of categories such that the items displayed in* iu *can be constrained to those of a selected category.*

However, some modelling tasks are more complex for the following reasons: (1) The input configuration may require to express alternatives, negation, or universal quantification. Further, if the input configuration comprises a large number of scheme elements, modularization may be required. (2) Some of the extensions and refinements may be conditional, i.e. to be performed only if some precondition holds. Further, if a large set of different scheme elements are to be generated, modularization should be provided. (3) The modelling task may require to generate scheme elements in nested loops. (4) A modelling task may be composed of smaller, subsequent ones such that the result of the first task is input for a second task.

**Example 8** *(1) Examples of complex input configurations are: (a) entity types that comprise* no *attributes; (b) page classes where* no *incoming link exists; (c) page classes that are reachable from every other page class; (d) page classes that contain an index unit or a data unit or both.*

*(2) Examples for complex extensions and refinements are: (a) depending on a relationship role's cardinality, generate either a data unit or an index unit; (b) generate a pair of connected page classes and, as a detail, embed index units in this page class.*

*(3) Examples for generating scheme elements in nested loops are: (a) For each entity type comprising a set of attributes, generate a page class with an entry form, and for each of the respective entity type's attributes, embed an entry field in this entry form. (b) for each entity type comprising a set of attributes, generate an index unit and, for*

*each attribute of the respective entity type, generate an entry in the index units "list of shown attributes".*

*(4) Examples for subsequent modelling tasks are: (a) First, for each entity type, generate a page class with an index unit (cf. example 7(a)). Second, for each pair of a page class with a data unit* du *and a page class with an index unit* iu*, create a link from* iu *to* du *(cf. example 7(b)). Note that finding pairs of data units and index units in the second step requires to query the scheme after having performed the first step.*

According to this classification of modelling tasks, *TBE* distinguishes between basic concepts and advanced concepts: The basic concepts are those that have already been outlined in the example of transformer `IndexPCForEnt` above. They cover simple query templates and simple generative templates. A simple transformer is a simple combination of a query template and a generative template in the sense that the generative template is instantiated once for each tuple of the query template's result relation. As mentioned above, most modelling tasks can be expressed with these basic concepts.

Just for giving the reader an impression of the expressive power of these basic concepts, we draw the following analogies: Simple query templates are comparable to SQL-queries of the form SELECT ... FROM ... WHERE. Simple generative templates are comparable to a set of parameterized SQL INSERT-statements. The combination of a query template and a generative template can then be seen as if the set of INSERT-statements were processed iteratively for each tuple $t$ of the SQL-queries result, each time parameterized by the values provided by $t$.

For supporting more complex modelling tasks, *TBE* offers advanced concepts that increase *TBE*'s expressive power. These advanced concepts are assigned to categories complex query templates, complex generative templates, and complex transformers as follows:

- *Complex query templates* support disjunctive queries, (possibly negated) sub-queries, and universal quantification. In order to facilitate modularization and reuse, query templates can be predefined and reused later on.
- *Complex generative templates* support conditional scheme element generation and delegation of details to generative sub-templates. Again, generative templates can be predefined for later reuse.

- *Complex transformers* concern the matters of (1) scheme element generation in nested loops and of (2) defining a sequence of modelling tasks where the scheme as extended or refined by the first task is input for a subsequent task. Item (1) is supported through nested relations, where the complex query template achieves a hierarchy of nested relations and the complex generative template processes this hierarchy correspondingly in preorder traversal. Item (2) is supported through cascading transformers, i.e. transformers that initiate the application of subsequent transformers.

**Generality.** The concept of defining transformers by-example can be applied to different modelling languages the same way. This is achieved in that *TBE* is composed of (1) a set of scheme element skeletons as provided by a modelling language, and (2) a set of symbols, comparison constraints, and expressions for specifying transformer's semantics. Of course, the set of scheme element skeletons varies between different modelling languages. However, the set of symbols, constraints, and expressions is fixed, such that several web modelling languages can be extended towards defining by-example transformers for their respective schemes the same way.

Further, the semantics of *TBE* 's language constructs is precisely defined in terms of a calculus over a relational representation of schemes. *TBE* abstracts thereby from different storage structures and internal representations of schemes as used by the various CASE tools. The TBE-calculus, which is introduced in this work for that purpose, extends the relational query language domain relational calculus [87, 93] (short DRC) by simple operations for manipulating relations. DRC has been chosen as basis for TBE-calculus because DRC has a strong formal foundation and has proven successful as basis for other by-example query languages.

Note, however, that TBE-calculus is used only for defining *TBE* 's formal semantics but not for providing an efficient implementation. Efficient implementation would require operations and optimizations that consider the internal representation of schemes. Yet this would contradict our goal of defining semantics precisely and independent of a particular modelling language.

## 1.5 Outline of the thesis

Chapter 2 surveys related work that does not actually concern web scheme transformation but that has either motivation or realization in common with

*TBE*. Approaches for web scheme transformation have already been discussed in Section 1.3.

Chapter 3 demonstrates how web scheme transformers can facilitate the process of conceptual web application modelling. We illustrate the process of web application development in general and the phase of conceptual modelling in particular, and we describe abstractly how transformers support this process. We then introduce a sample web scheme, which serves as running example throughout this thesis.

Chapter 4 introduces the basic concepts of by-example transformers. The chapter starts with an overview over *TBE*'s basic architecture. We define TBE-calculus, which is used for precisely defining *TBE* 's semantics, and then introduce query templates and generative templates, both informally based on examples as well as precisely based on TBE-calculus. The chapter concludes with defining transformers as a combination of query templates and generative templates.

Chapter 5 covers graphical applications of transformers, and both off-the-shelf applications as well as individualized applications are addressed. The chapter particularly addresses the concepts supporting individualization, i.e. application-specific constraints, application-specific expressions, turning new-element variables to parameters, and pinning new-element variables.

In Chapter 6, we develop advanced constructs that increase *TBE*'s expressive power, namely complex query templates, complex generative templates, and complex transformers as categorized at the end of the previous section.

Chapter 7 describes how *TBE* can be realized. In particular, we illustrate (1) techniques for specifying transformer definitions and transformer applications for a given conceptual modelling language and CASE-tool, and (2) an implementation of TBE-calculus.

Chapter 8 demonstrates the use of by-example transformers by means of an extensive use-case, i.e. by demonstrating the use of transformers in the development of a real web site.

Chapter 9 concludes the thesis and gives an outlook to future activities.

# Chapter 2

# Related work

## Contents

In Section 1.3, we have discussed work that particularly deals with web scheme transformation and that is therefore directly related to our approach. In this chapter, we describe related work that does not actually deal with web scheme transformation but that has either motivation or realization in common with *TBE*. There are several fields of related work:

- In *TBE*, transformers are defined *by-example* such that our approach compares to by-example manipulation languages.
- For defining transformers, *TBE* uses the graphical notation of modelling languages such that it is related to visual languages.
- Web scheme transformers express recurrent modelling tasks and compare therefore to design patterns.
- *TBE* is about transforming web schemes, and scheme transformation has a long tradition in the database field.
- Due to the by-example approach of *TBE*, the process of defining a transformer is similar to that of defining a scheme. This resembles programming-by-demonstration approaches.

For each of these fields, we summarize the basic principle, usually by picking out one representative approach for explanation, and then discuss the differences to *TBE*.

Further, *TBE* deals with transforming schemes of web modelling languages such that approaches for conceptual web application modelling form the context of this work. A brief overview over such approaches is given at the end of this chapter.

## 2.1 By-example approaches

In a by-example approach, users express queries or data manipulations by giving a generic examples of the desired results instead of specifying operations for computing the results. These operations are derived from the provided example.

The by-example approach has first been introduced by Zloof who developed QBE (Query-by-example) [143, 144], which is both the name of a relational data-manipulation language and a database system including this language. Today, QBE or variants thereof are implemented in several database systems, e.g. the database system Microsoft Access. Further, many other approaches have either been influenced by QBE or share basic ideas of QBE. We start with explaining the pioneering approach of QBE and then summarize other approaches for querying and/or constructing data in a by-example manner.

## 2.1.1 Query-by-example: the pioneer

Query-by-example (QBE) is a language for querying and manipulating relational data, i.e. data stored in relations. These relations are represented as *tables* with ordered *columns*. Each column has a domain and represents an attribute of the relation. QBE-statements are expressed using *skeleton tables*, each representing a particular table. A skeleton table consists of one or more example rows comprising constants, constraints, and variables. The domain of a variable is defined through the particular column in which the variable is placed. The provided constants and constraints constrain valid bindings of variables.

Commands define how to proceed with the tuples that match the example rows. For example, command `D` placed to the left of an example row stands for "delete" and denotes deletion of tuples; command `P` placed in a column stands for "projection" and means to select values of this column; similar commands are defined for expressing order, prevention of duplicate elimination, and the insertion or modification of tuples.

| Person | persno | name | position |
|---|---|---|---|
| | _PERSNO | P. _NAME | |

| Department | dno | name | head |
|---|---|---|---|
| | | dke | _PERSNO |

Figure 2.1: QBE-query selecting the name of the head of department `"dke"`.

**Example 9** *Consider a database scheme comprising two relations* Person($persno, name, position$) *and* Department($dno, name, head$)*, where $\pi_{head}(Department) \subseteq \pi_{persno}(Person)$. Then a query "select the name of the person heading the department named* `'dke'`*" is expressed in QBE as illustrated in Fig. 2.1.*

*The query comprises two skeleton tables* Person *and* Department*, which contain one example row each. Entries* `_PERSNO` *and* `_NAME` *are variables as denoted by the preceding underscore, whereas entry* `dke` *is a constant. Each value bound to variable* `_NAME` *is part of the query's result as denoted by command* `P`*. Skeleton table* Department *retrieves tuples of relation* Department*, where only those tuples that have* `dke` *as the value for the department's name are selected. For each such tuple, variable* `_PERSNO` *represents the value of attribute* `head`*, i.e. the person number of the department's head. Skeleton table* Person *retrieves tuples of relation* Person*, and variables* `_PERSNO` *and* `_NAME` *represent the values of attributes* `persno` *and* `name`*, respectively. Placing both skeleton tables in the same query expresses a natural join where the relations are connected via equally named variable* `_PERSNO`*.*

*TBE*, which manipulates schemes, is quite similar to QBE, which manipulates relational data. Analogously to skeleton tables that look like relations, *TBE* uses skeletons of scheme elements that look like scheme elements of the particular modelling language in use. For example, to define a query that selects all entity types, an entity type skeleton is used as follows: a domain variable `ENT` is placed at an entity type's name and defines thereby that this variable ranges over all entity type names. Tagging the variable with a $\surd$-symbol expresses that these names are to be selected. Thus, the $\surd$-symbol of *TBE* has the same meaning as command `P` of QBE.

However, besides *TBE* 's graphical notation, there are the following differences between *TBE* and QBE: First, *TBE* is based on the graphical notation of web modelling languages where most relations between scheme elements are expressed implicitly by graphical arrangement. In *TBE*, such relations are represented implicitly by the graphical arrangement of variables, too. This is in contrast to QBE, which is built solely on skeleton tables, and all relations need to be addressed explicitly.

**Example 10** *Fig. 2.2 illustrates the difference between QBE and TBE regarding implicit relations. Fig. 2.2(a) repeats the TBE query*

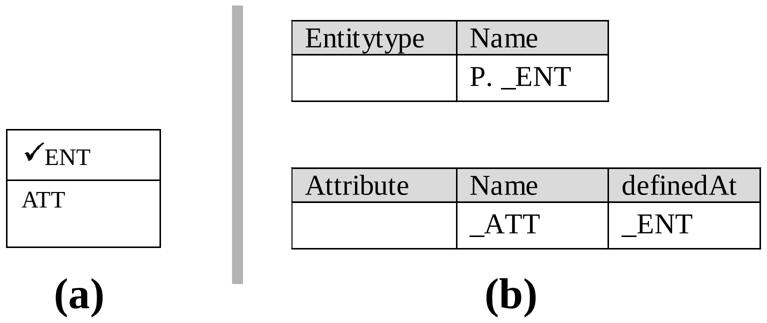


Figure 2.2: Selecting entity types comprising attributes:
(a) QBE-query; (b) *TBE* query template.

*template of the introduction, which selects entity types comprising attributes. Fig. 2.2(b) depicts a corresponding query in QBE, which assumes a database scheme comprising the relations* Entitytype (name) *and* Attribute (name, definedAt), *with* $\pi_{definedAt}(Attribute) \subseteq \pi_{name}(Entitytype)$.

*In contrast to the TBE query template, where a relation "definedAt" constraining variables* `ATT` *and* `ENT` *is implicitly expressed by placing variable* `ATT` *in the attribute's section of entity type skeleton* `ENT`*, QBE requires to address relation "definedAt" explicitly, i.e. by means of foreign-key attribute* definedAt *of relation* Attribute.

Further, *TBE* offers many complex constructs like sub-queries, cascading applications of transformers, etc. Such constructs are not included in the early work of Zloof in 1977. But also extensions of QBE do, to the best of our knowledge, not provide such constructs.

### 2.1.2 Other by-example query languages

The ideas of QBE have been widely adopted for query systems for other data models. Aggregation-by-example [83] (Abe) and Summary-table-by-example [107] are query-by-example (QBE-) languages that extend QBE by aggregated values and hierarchical aggregations, respectively; Time-By-Example [135] is a QBE-language for historical relational databases, i.e. over a relational data model extended by attribute time stamping; QBEN [92] and ESCHER [142] both are QBE-languages defined for NF2 data models (e.g.

[117]); CQL [79] is a QBE-language for relations containing hierarchically classified data.

Each of the above mentioned approaches is form-based and is defined for a single data model. Another by-example language called XCX [106], in contrast, generalizes the basic concepts common to different QBE-language. XCX provides thereby a general way for defining by-example query languages for various data models including object oriented and visual models. However, the differences between *TBE* and QBE, which have been listed above in section 2.1.1, exist between *TBE* and XCX analogously.

### 2.1.3 Other by-example construction languages

The idea of constructing new elements from skeletons that may comprise variables and expressions in addition to concrete values has also been applied in other fields. For example, programming languages provide generic classes [99, 49], and in the field of web site development, HTML templates are widely used [131, 51, 105]. Both, generic classes and HTML templates, are instantiated by binding concrete values to variables. This can be either done explicitly through assignment or implicitly through, for example, a database query providing values for the variables. The different approaches are rather similar such that we do not explain all of them. Instead, we pick out constructive templates [105] for explanation.

Constructive templates [105] define how to construct web pages based on values provided by a query over a semi-structured data source. The web pages to be generated are defined through a template that looks like such a web page but has some of the concrete values replaced by variables and expressions. One of these expressions is for defining IDs for the web documents to be generated and is comparable to a constructor known from object-oriented programming languages. Such an ID-generating expression looks like, for example, `UID = &("Hub-",&HubName)`. It defines that a new web page with an ID concatenated of literal `"Hub"` and the value of a variable `&HubName` is to be generated, such that one web page is generated for each distinct value bound to variable `&HubName`. As an ID-value identifies a web page, links to web pages are expressed by means of such ID-values. Further, for each constructive template, a *generation trigger* can be specified, which reacts to changes in the data source and automatically triggers the instantiation of the constructive template.

Constructive templates (and other approaches for web site development) generate web pages at the instance level, but the concept could be adopted also for generating scheme elements at the scheme level. However, the following differences arise: First, constructive templates would generate scheme element independently from each other, and relations between them would have to be established by explicitly referring to their IDs, i.e. as in links referring to web pages. *TBE*, in contrast, generates configurations of related scheme elements, whereby these relations are simply expressed by the graphical arrangement of scheme element skeletons. Second, as a transformer retrieves the values required for template instantiation from the scheme to which it is applied, constructive templates would require to define a textual query over the internal representation of schemes. In contrast, *TBE* uses by-example query templates for this purpose.

## 2.2 Visual languages

For defining transformers, *TBE* uses the graphical notation of web modelling languages for which transformers are to be defined. Therefore, *TBE* is related to visual programming languages. A subset of these languages are visual query systems, which are closer to our approach than other visual programming languages are. We start with explaining visual query systems.

### 2.2.1 Visual query systems

Visual query systems (VQS, surveyed in [27]) are graphical interfaces to databases. For most of these languages, the graphical elements are one-to-one visualizations of the elements of the underlying data model. Only a few visual languages additionally make use of semantic relationships like visual containment or spatial relationship.

GQL [109, 108] is a graphical query language for the functional data model [127], which represents data by nodes and labelled edges. Consequently, the visual elements of GQL reflect nodes and labelled edges, too. XML-GL [30, 31] is a graphical query and manipulation language for XML data, and XML-GL's visual elements represent therefore XML elements and attributes. Doodle [43, 39, 41] and its successor Delaunay [44, 42] are graphical query languages for the object model of F-logic [82], and the visual elements of Doodle and Delaunay reflect the formalism of F-logic, i.e. they represent terms of first-order logic. In addition to predefined visual elements, Delaunay supports user-defined iconic representations of F-logic terms. Query-by-diagram (QBD∗) [3, 4] is a graphical query language for the relational model, where relations are visually represented through the model elements of the entity-relationship model, i.e. the elements of an ER-diagram.

Obviously, these languages cannot be used for specifying a by-example query, for example, for WebML schemes. Despite their graphical notation, formulating a query in either of these languages would require to address WebML schemes in their internal representation rather than using the visual elements of WebML for query formulation. Consequently, any visual correspondence between the query and WebML schemes is lost. In *TBE*, in contrast, queries and constructions are expressed by using the graphical scheme elements of the modelling language at hand, e.g. WebML. The following example illustrates the difference:

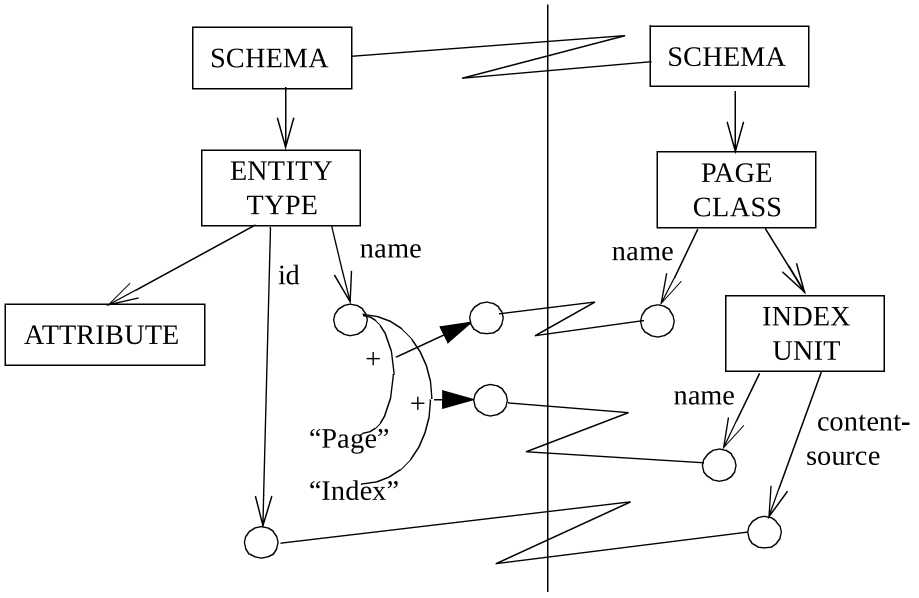


Figure 2.3: Transformer `IndexPCForEnt` expressed in XML-GL

**Example 11** *Suppose that we have a WebML scheme represented as an XML document and that we utilize XML-GL to manipulate this XML-document. Fig. 2.3 depicts a definition of transformer* `IndexPCForEnt`*, which has already been explained in the introduction, now expressed in XML-GL.*

*The left part of Fig. 2.3 depicts an XML-GL query that selects XML-elements of sort* `entity type` *together with their XML-attributes* `ID` *and* `name`*. Further, two string-concatenation expressions are defined, each expressed by an arc that is labelled "+", where the ends of the arc represent the expression's arguments. Thus, each of these two string-concatenation expressions derives a new value based on the value of attribute* `name`*.*

*The right part of Fig. 2.3 depicts an XML-GL construction part. This construction part, which is instantiated for every entity type selected by the query part, defines how to generate a new XML-element* `PageClass` *with an XML-attribute* `name`*. The value of this attribute is defined by one of the string-concatenation expressions. An XML-element* `index unit` *is generated as sub-element of XML-element* `page class`*. It is again named after the entity type and additionally embeds an XML-attribute* `contentsource` *that refers to the ID of this entity type.*

When comparing the XML-GL query of the example above to the *TBE* transformer depicted in Fig. 1.3, the following two differences appear: (1) the XML-GL query looks quite different from the graphical representation of WebML schemes themselves. (2) as in QBE, XML-GL requires to explicitly refer to relations between scheme elements even if in WebML these relations are expressed implicitly, e.g. by visual containment.

VISUAL [7, 6] is a graphical query language for an object-oriented data model. In contrast to above mentioned visual languages, queries in VISUAL make use of both (1) user-defined representations of objects and (2) semantic relationships like visual containment or spatial arrangement. Thus, a query is expressed as an arrangement of object-representations that looks similar to an arrangement of objects that matches the query. However, the semantic relationships that can be used are built-in and are intrinsic parts of the definition of VISUAL. In contrast, *TBE* is capable of utilizing (1) the visual representation of scheme elements *and* (2) the visual representation of relationships between them, both as defined by the modelling language at hand.

### 2.2.2 Visual programming languages

Apart from visual query systems, there exists a wide range of visual programming systems (VP-systems). An overview can be found in [120]. Most of these systems have little in common with *TBE* as they do not strive for declarative transformer specifications but rather visualize imperative paradigms like control flows. Worth mentioning, however, are approaches that follow a constraint or rule based paradigm.

### Constraint-based visual programming languages

In a constraint based approach, a program comprises (1) concrete values and variables and (2) constraints relating variables to variables or variables to concrete values. These constraints are considered to hold at any point in time. If the value of a variable is changed, a constraint satisfier tries to manipulate the values of the other variables such that all constraints hold in the end.

For example, in ThingLab [20], a user can graphically specify the coherence between degrees Fahrenheit and degrees Celsius (i.e. $f = c * 1,8 + 32$). Variables $f$ and $c$ represent Fahrenheit and Celsius values, respectively. With this formula, a user can bind a value to either of these variables, whereas the constraint satisfier determines a valid binding of the respective other variable. This allows for flexible user interaction.

Similar to constraint satisfaction approaches, query templates in *TBE* express domain variables and constraints over them. As soon as a query template is applied to a scheme, the *TBE* system derives valid bindings for all domain variables. However, for deriving these bindings, *TBE* does not use a constraint satisfier for arbitrary formulas. *TBE* translates constraints to queries over a relational data source and yields thereby a simpler and a more efficient implementation.

### Rule-based visual programming languages

In rule based visual programming (VP-) systems, a program consists of several transformation rules, each describing the transformation of a set of graphical shapes to another set of graphical shapes within some environment. Each rule comprises a head and a body. The head defines a pattern of shapes and denotes a situation before transformation. The rule's body denotes the transformation's result. If a graphical shape in the environment matches a rule's head, this shape is replaced by the graphical shape of the rule's body. The program ends when no head of a transformation rule is matched any more.

Examples for rule based VP-systems are *LEGOsheets* [78] for shapes representing physical artifacts like vehicles, and *ChemTrains* [10] for logical electronic units like `AND`-gates. Also graph rewriting systems [13], which generically deal with domain models represented by graphs, fall into this category. For example, the VP-system *PROGRES* [121] has been employed as a visual database query language as well as a system for detecting well-formed control-flow diagrams.

A rule's head and a rule's body reflect parts of an environment as before and after a transformation, respectively. This is much in the sense of *TBE*'s distinction of query templates and generative templates. However, *TBE* differs from rule based VP-systems in the following ways: First, transformers in *TBE* do not focus on being activated automatically (i.e. by pattern matching) but rather on being explicitly applied to parts of a scheme by the modeler. For this purpose, *TBE* provides a much more flexible way of individualized applications, each time individually controlling which part of a scheme shall be affected. Second, with query templates *TBE* provides a powerful query language which goes far beyond pattern matching expressions of rule based VP-systems.

## 2.3 Design Patterns

A design pattern [24, 63] describes a recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. In the context of conceptual web application modelling, hypertext design patterns and design patterns for web application development have emerged. *TBE* can help to define patterns in a way such that they can be used as components that drive pattern instantiation. Similar effort has already been done for design patterns arising in the context of object-oriented software development. Although object-oriented software design patterns per se have little in common with *TBE*, the motivation for putting patterns to work in development environments is comparable to the motivation of *TBE*.

### 2.3.1 Hypertext design patterns

Hypertext patterns describe recurrent problems in hypertext design and provide solutions for them. Although hypertext patterns seem as to have a motivation similar to transformers, a main difference exists: Patterns describe recurrent hypertext design problems and their solutions abstractly, i.e. independently of a particular web modelling language. Transformers, in contrast, provide solutions in terms of a *particular* modelling language, e.g. WebML. Therefore, transformers do not formalize hypertext patterns per se but may help instantiating them once defined as modelling tasks within a particular modelling language.

There has been much effort in formulating hypertext patterns [12, 115, 114, 116, 64, 124], and there is therefore a great potential in extending web modelling languages by high-level modelling constructs. Suppose, for example, a location pattern, which can be described as follows: "on every page of the hypertext, inform users about the navigation path which has been followed to this page". For this purpose, one could define a transformer that (i) augments every outgoing link by parameters for updating the path information and (ii) augments every page class such that the provided path information is displayed adequately.

Of course, web modelling languages already provide model primitives for many hypertext patterns. For example, WebML provides a primitive for marking a page as a "Landmark" and defines thereby implicitly that this page is reachable from every other page in the hypertext scheme. The same is true for set-based navigation patterns supported in WebML by index and scroller units.

However, the drawback of model primitives is that they usually can be used only out of the box. For example, WebML provides no way for adapting the primitive "Landmark" such that some pages of an area do not "see" a landmark page. This would, for example, be useful when gathering user inputs over a sequence of pages where no "intermediate exit" other than an explicit commit or abort shall be possible. In contrast, transformers can be defined individually and specifically tailored to a particular problem domain.

### 2.3.2 Web-application design patterns

Web Design Frameworks [124] are meta-level descriptions of a family of similar web applications, e.g. a family of electronic shop applications. For a particular family, a meta-scheme defines which concepts must be instantiated in a web application scheme in order to belong to that family. Thereby, a web design framework specifies both the common aspects of applications of a particular family and the hot-spots where the specificities of a particular application are accommodated. For example, a meta-scheme could claim that an electronic shop must instantiate a concept “shopping basket”, enabling users to put selected products into shopping carts. Hot-spots are, for example, the particular entity type representing products or different properties which are used to categorize products.

Web Design Frameworks focus on meta-modelling such that, after having defined a web scheme, one can check whether this scheme accords to a particular family of web applications as defined through a meta-scheme. However, such constraint verifications are in general not performed automatically by the system. Further, Web Design Frameworks do not yet provide a mechanism for having a scheme or parts thereof generated, e.g. by parameterizing generic classes of the framework.

*TBE* is not considered as a meta-modelling language and does therefore not incorporate predefined primitives for checking whether a scheme adheres to a meta-scheme, e.g. if a particular generic entity type has been instantiated. However, as *TBE* is a language for querying and manipulating schemes, one can employ query templates for expressing constraints that a configuration of scheme elements shall fulfill. Further, transformers can be used to have configurations of scheme elements instantiated that make up a concept like “shopping basket”.

### 2.3.3 Object-oriented software design patterns

Object-oriented design patterns [63] are solutions to design problems repeatedly occurring in object-oriented software development. The work of Gamma et. al. has initiated extensive research concerning the identification of design patterns and the development of languages for describing design patterns (confer, for example, Buschmann et.al. [24], the series of pattern languages of program design [54], work concerning the formalization of design patterns [48, 101, 133], and the hillside group [68]).

Some software development environments are capable of either checking whether a piece of design/code accords to a predefined design pattern or of assisting developers in defining such pieces of design/code [19, 112, 61]. We pick out Fred [72, 73, 61] (Framework Editor) for explanation because we feel this approach to be representative.

Fred (and its follow-up JavaFrames) is a research prototype of a development environment that supports definition and use of design patterns. It is for developing Java projects and is embedded in the eclipse platform [75]. Each design pattern is defined abstractly in terms of roles that are to be instantiated. For that purpose, the environment (1) interactively guides the developer of an application through a list of tasks that must be performed in order to achieve a valid instantiation of a design pattern and (2) assists the developer in performing such tasks. This is also known as *interactive, task-based programming* [72]. Moreover, once code fragments are assigned to a pattern's roles, the environment supervises changes that are applied to the code later on. The system alerts the developer whenever the code does not accord to the pattern any more.

**Example 12** *Suppose that a class shall accord to the JavaBeans-specification [132]. Among other things, this specification requires that for each property of a class, a method for accessing the value of this property must be provided. Such a method is referred to as a* getter-method*. Thus, whenever a developer defines a property* `xyz`*, then a task "Define or select a method that plays the role of a getter-method for property* `xyz`*" is to be performed. The developer can also decide to have the task performed by the environment, i.e. to have a getter-method together with a default implementation generated. The developer may change the generated code afterwards. However, if the getter-method for property* `xyz` *were removed, the system would alert an error.*

*TBE*, which supports task-based modelling, is comparable to a development environment like Fred, which supports task-based programming. There are, however, the following differences: In Fred, a pattern is described as a list of simple tasks that basically consist of assigning role-players to roles. The criteria that a role-player must fulfill are defined as check-routines in terms of scripts. Fred does not by itself select role-players but rather uses these check-routines to validate manual role-player assignments. In *TBE*, in contrast, such criteria are defined declaratively in a visual by-example language, and they are used for both (1) validating role-player assignments performed manually and (2) selecting all role-players that fulfil the criteria. Thereby, *TBE* allows to have a task performed iteratively for a set of role-players. For example, in *TBE*, a single transformer application could generate corresponding getter-methods for all the properties of a class.

## 2.4 Database scheme transformation

Transformations of schemes for databases (short: DB-scheme transformations) have attracted much interest in the database community (e.g. [2, 9, 50, 26, 110]). Work has concentrated on (1) identifying and classifying transformation primitives that can be applied to a DB-scheme, and on (2) identifying patterns of transformations applied in the process of DB-scheme design and/or evolution.

The primitives that can be applied usually depend on the scheme that is to be transformed. For example, Batini [9] describes primitives to be applied to entity-relationship diagrams, Banerjee [8] provides a taxonomy of transformation primitives for an object-oriented model, Castelli [26] additionally allows to transform specifications of constraints and operations, and Proper [110] further considers internal (logical) representations of conceptual database schemes to be subject to transformation.

Based on these primitives, patterns of complex DB-scheme transformations have been identified that usually intend to improve the quality of a DB-scheme, e.g to achieve minimality and normalization [9], or to optimize the DB-scheme with respect to a particular target platform to which the scheme is to be mapped [110]. Further, transformations that extend, refine, or even reduce a DB-scheme have been defined [9].

However, defining executable transformers that generate an improved, extended, refined, or reduced output DB-scheme each time they are applied to an input DB-scheme has not been in the focus of this work.

*TBE* complements work concerning DB-scheme transformation in that executable transformers can be defined. Note that *TBE* can be employed for various models including models for defining DB-schemes. However, *TBE* itself is not concerned with the question of whether a particular transformer yields an improvement, extension, refinement, or a reduction of a scheme because this depends on the semantics of the particular model in which *TBE* is employed. As mentioned above, in the context of particular models, a large amount of research has already been done for answering such questions.

## 2.5 Programming by demonstration

In Programming by demonstration (PBD) [45, 102], a program is generated based on tasks users perform, for example, in a text editor. Two metaphors for such approaches are distinguished, the “do-what-I-did” metaphor and the “do-what-I-mean” metaphor [103, 94, 120].

### 2.5.1 Do-what-I-mean metaphor

In the “do-what-I-mean”-metaphor, a user interacts with a system and demonstrates typical scenarios of the way input data is transformed into result data. Based on predefined heuristics, the PBD-system tries to generalize these scenarios and derives a program that cannot only be applied to situations that are equal to the demonstrated ones but also to situations that are just similar.

This resembles query-by-example approaches explained above. However, there is a main difference: QBE-examples are templates that are to be matched by data, and the meaning of these templates is precisely defined. In contrast, scenarios of user interaction are widely ambiguous because it is hard to detect the *reasons* for why a user performs a particular action [102, 104, 120]. Thus, as demonstrated by the following example, it is hard to detect conditions or loops and the code produced by the PBD system has therefore to be reworked.

**Example 13** *Suppose that transformer* `IndexPCforEnt` *is defined using a programming-by-demonstration approach as follows: The developer selects an entity type, e.g. entity type* `Paper`*, and then defines a new page class named* `PaperPage`*. From this interaction, the system could derive that a new page class shall be named after an existing entity type. However, the system cannot detect that a developer does not select entity types that do not comprise attributes. Consequently, the system cannot detect that page classes for all entity types that comprise attributes are to be generated. The loop and the condition “must comprise attributes” has rather to be inserted manually into the code fragment generated by the PBD system.*

In *TBE*, in contrast, conditions are expressed precisely in a visual and declarative manner. Consequently, sets of items that fulfil particular conditions can be defined such that loops can be expressed precisely, too.

### 2.5.2 Do-what-I-did metaphor

In the "do-what-I-did"-metaphor, a user turns on a macro-recorder and then interacts with the system as usual, i.e. performs some commands via the user interface. These commands operate normally but are additionally recorded. The recorded macro can then be re-executed later in different contexts. Such macro-recorders have a long history, especially in text editors (e.g. Emacs [113] and Microsoft Word) and spreadsheets (e.g. Microsoft Excel).

Macro recorders can also be employed in tools for conceptual web modelling. Then, after having turned on the recorder, the tool records the commands modelers perform via the graphical interface and generates a script of schema modification operations. However, for getting a useful transformer, the generated scripts have to be reworked extensively because the macro-recorder does not detect conditions or loops. Furthermore, a macro-recorder does not detect dependencies like "name the page class after the entity type". A further disadvantage of Macro recorders is that a transformer's semantics is hidden in scripts. Extra documentation is required to describe how the transformer behaves and how it is to be applied.

In *TBE*, in contrast, a by-example transformer is precisely defined as a visual representation of the modelling task to be performed. Thus, the transformer by itself is a comprehensible documentation in form of a typical example of the transformation supported.

## 2.6 Approaches for Conceptual Web Application Modelling

Approaches for conceptual modelling of web applications provide the context in which *TBE* is applied. In this section, we give a brief overview over those approaches which we feel to be representative but we do not claim that this overview is complete. For an extensive comparison, the reader may refer to dedicated survey articles, e.g. [55, 56, 40, 86, 84].

HDM [67] (Hypertext Design Model) introduces model-based conceptual design of hypermedia applications, i.e. design at a schema level independent of a particular implementation. A schema of a hypermedia application comprises a (hyper)base layer and an access layer. The base layer describes information objects in terms of entity types whose entities can be sub-structured into a tree of components. Semantic relationships between information objects are expressed in terms of relationship types. The access layer defines collections [65] of information objects and navigable links between information objects and/or collections. A CASE tool for HDM is provided [15, 66].

RMM [76] (Relationship Management Methodology) extends HDM in two directions: (1) RMM provides a methodology of hypermedia application design by proposing a seven-step design process together with design guidelines. A CASE tool [47] supporting this design process is provided, too. (2) RMM's data model RMDM is based on the entity-relationship model [35] but provides further model primitives for hypertext design. For example, among others, model primitives *index* and *guided tour* are introduced. An index acts as a table of contents to a list of entity instances and provides direct access to each listed item. A guided tour denotes a linear path through a collection of items, allowing to move forward and backward in the path.

OOHDM [123, 122] (Object-Oriented Hypermedia Design Model) is based on HDM but is inspired by object-oriented modelling. OOHDM is a hypermedia design *methodology* which reduces the RMM life-cycle to the four steps *conceptual design*, *navigational design*, *abstract interface design*, and *implementation*. Further, OOHDM incorporates model primitives reflecting hypertext design patterns, i.e. solutions for repeatedly occurring problems in hypertext design [115, 116, 114]). For example, a pattern called *Navigational Context* [115] deals with the problem that a particular object may be observed differently depending on the context in which it shall be perceived. Hence, OOHDM provides a model primitive *Context* for expressing such situations.

The pioneering principles developed in HDM, RMM, and OOHDM have been extended and refined by several methodological approaches, e.g. HDM-lite [58], Strudel [51, 52, 53], OO-H [70, 71, 69], Araneus [96, 98], and WebML [32, 29]. We pick out Araneus and WebML for explanation because these two together cover most of the concepts that are also provided by the other approaches.

Araneus [96, 98] focuses on the definition and implementation of data-intensive web sites. Besides the distinction of content, navigation, and presentation modelling, Araneus stresses the separation of conceptual design and logical design. Content is conceptually modelled using the entity-relationship model, the logical design is expressed with the relational model. Navigation is conceptually modelled in a notation called the Navigation Conceptual Model (NCM), whereas logical design is expressed using the Araneus

Data Model (ADM). Presentation is specified orthogonally to content and navigation modelling using an HTML template approach. The Navigation Conceptual Model and the Araneus Data Model are both subsumed by an extended nested relational model [117], which is the backbone of a CASE tool called *Homer* [97].

WebML (web modelling language) is a visual language for specifying the content structure of a Web application and the organization and presentation of such content in a hypertext [32, 29, 95]. Besides the visual representation, WebML also provides a normative textual notation in XML syntax which is used by tools implementing WebML, e.g. by the commercial CASE-tool WebRatio [141].

WebML offers two basic models, the structural model for describing a web site's content and the hypertext model for describing the hypertext's topology, i.e. the pages that compose it (composition model) and the links between pages (navigation model). Further, WebML offers several other features useful for modelling web applications like, for example, modelling operations for manipulating content [17, 29], user modelling and personalization [32, 29], derivation of redundant content [34] using OQL [28], modelling of web services and their composition [22, 21], modelling workflow-driven web applications [23], modelling collaboration [95], and modelling of log-analysis [57].

Several other commercial products that provide model-driven design are already available. Hyperwave [80] is an environment that allows users to browse, annotate, and maintain documents distributed over the web. Hyperwave adopts a simplified hypermedia model and allows for defining hierarchically organized document collections at a high level of abstraction. However, Hyperwave defines the information base as a set of flat documents annotated with meta data but lacks a proper structural model. Oracle designer [46], in contrast, generates Web applications from entity-relationship diagrams augmented with some navigation and presentation flavor. Thus, compared to WebML or Araneus, which both provide separate models for defining content, navigation, and presentation, Hyperwave and Oracle designer are navigation centric and database centric, respectively.

# Chapter 3

# Conceptual web application modelling and transformers

## Contents

This chapter deals with the phase of conceptual modelling of web applications within the process of web application development and illustrates the role that transformers can play in this phase. Section 3.1 illustrates the overall process of web application development, whereas Section 3.2 concentrates on the phase of conceptual modelling. In Section 3.3, we describe abstractly how transformers can be utilized. In Section 3.4, we introduce a sample web scheme defined in the web modelling language WebML. Both WebML and the sample scheme serve as basis for explaining by-example transformers throughout the paper. Finally, in Section 3.5, we identify recurrent modelling tasks in the running example, i.e. recurrent patterns of extension or refinement that could be supported by transformers.

## 3.1 The process of web application development

Web applications are possibly large software systems which are developed in different phases. As first described by Boehm's spiral model [16] and later by other methods for software development [18, 77, 38, 85, 95], phases are repeated and the application gets iteratively extended or refined until results meet the requirements. The application undergoes several cycles, each producing a prototype which gets evaluated. Such evaluation cycles are particularly appropriate for the development of web applications where usability is a key success factor [32, 71, 105].

We start with introducing the development process of web applications and then show how this process is typically supported by CASE tools.

### 3.1.1 Development cycle.

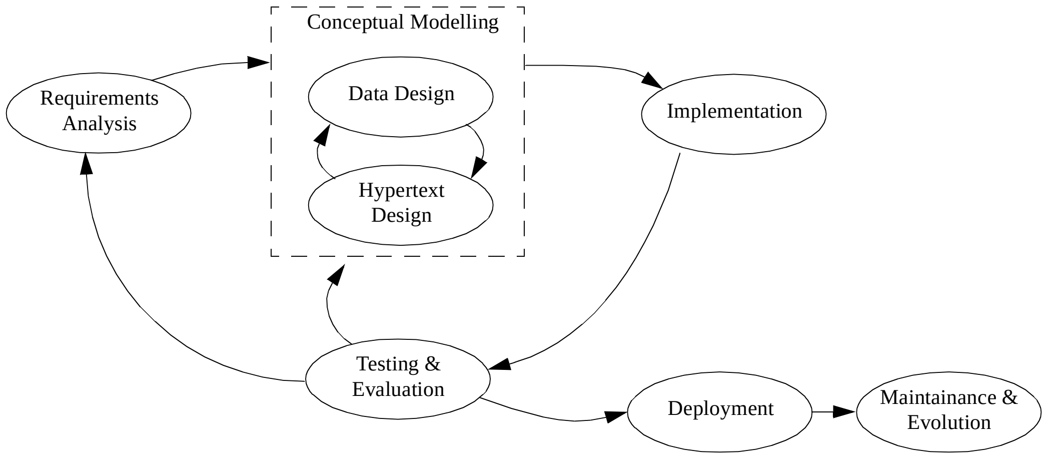


Figure 3.1: Phases in the process of web application development [95]

Fig. 3.1 illustrates the development cycle of web applications. This process has been defined in the context of the WebML method, yet it is in line with other methods for web application development (confer, for example, [38, 85]).

As a starting point, the business requirements are analyzed. This yields a specification of the functional and non-functional requirements of the web site. Based on this analysis, a conceptual web scheme is defined that specifies (a) the content of the web site and (b) the topology of pages, navigable links, and operations that allows to view and manipulate this content via the Web. This process is iterative because, for example, adding a new entity type typically requires to add a page class for presenting the content of this entity type.

When the web site's core functionality is defined, the web scheme gets implemented as a prototype, e.g. by creating a database scheme, JSP and HTML/XML pages, code for accessing the underlying database, stylesheets for rendering the page's content, etc. This prototype is evaluated and tested, whereby it usually turns out that either (a) existing requirements are not yet fulfilled or (b) that new requirements have to be specified. In both cases, the site's conceptual scheme is adapted and the next cycle starts.

At some point, when the prototype's functionality is satisfactory, the web application is deployed, i.e. packaged and then installed in a production environment. At this point, the phase of maintenance and evolution starts, and modifications effected after this point are, in contrast to the other phases of development, applied to an existing system.

### 3.1.2 CASE tool support.

Although web modelling languages could be used without any tool support as well, web application development greatly benefits from CASE tools that provide graphical scheme editors and that can assist or automate the implementation and deployment phase. Further, if a CASE tool is in place, then the maintenance & evolution phase benefits from the existence of a conceptual scheme. Requests for changes of the application are turned into changes at the design level and then propagated to the implementation rather than applied solely at the implementation level.

Most web modelling languages provide a CASE tool that allows to edit web schemes graphically and to generate running web applications therefrom. A brief overview over these languages and tools has been given in Section 2.6.

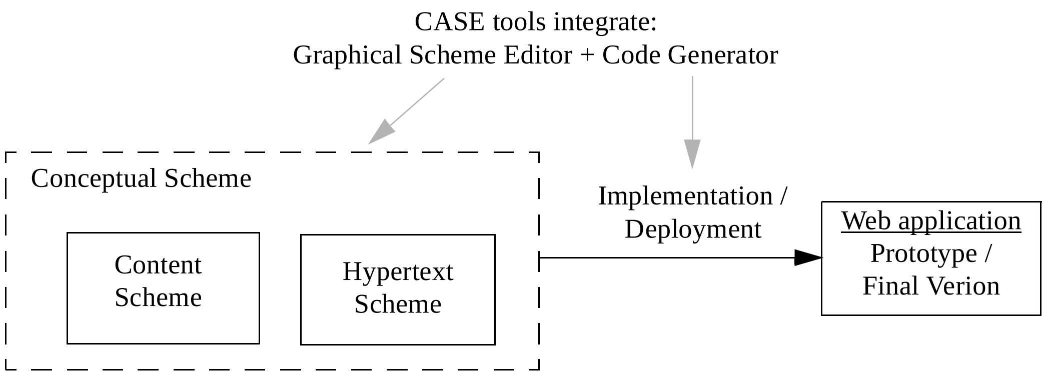


Figure 3.2: Web application development using a CASE tool.

Fig. 3.2 illustrates how existing CASE tools for web application modelling typically support the modelling process. These tools usually provide (1) a graphical editor for defining a content scheme and a hypertext scheme as well as (2) a code generator that creates an effective web application. The development phases introduced in the previous subsection are supported to the following extent:

Requirements analysis itself is loosely structured and requires informal methods and heuristics [74]. Thus, CASE tools usually do not support the analysis process per se but provide at most for documenting the results, e.g. in terms of use-case diagrams.

Conceptual modelling, which is split into the phases data design and hypertext design, is usually supported by providing a graphical editor that uses the notation of the respective web modelling language. This allows to conveniently define and maintain the conceptual scheme of the web application. Further, these editors usually have some built-in consistency checking.

Implementation and deployment is supported by the code generator, which creates from the scheme a web application in a particular target architecture like, for example, Java servlets combined with JDBC. The code generator creates a database scheme, JSP-pages and servlets, code for accessing the underlying database, XSL/CSS stylesheets for rendering the pages, etc. The generated code usually can be reworked and optimized.

Testing & evaluation is usually supported by the code generator, too. For some CASE tools, the code generator creates, besides a database scheme, also demo data that can then be accessed and manipulated via the web application [95, 141]. Further, the CASE tool could instrument the generated code by log points that can then be used for tracing user interactions or for analyzing performance.

Maintenance & evolution is also supported by the graphical editor together with the code generator. Requested changes of the web application are applied to the scheme rather than directly to the implementation, and the code generator then propagates these changes to the existing web application.

To summarize, the benefit of CASE tools is that web applications are developed and maintained basically at the level of a conceptual scheme, whereas the implementation and deployment phase is basically automated. Of course, the phases of requirements analysis and testing & evaluation are important as well. However, as mentioned above, these activities are usually decoupled from the CASE tool. Thus, from the viewpoint of a CASE tool, the main activity when developing and maintaining a web application is the definition and maintenance of the conceptual scheme.

## 3.2 Conceptual modelling of web applications

This thesis concentrates on conceptual modelling of web applications, where an initial web scheme is iteratively extended and refined. In the early stages of development, core entity types are defined and are complemented by page classes for displaying their content. Then, these page classes are linked and advanced access structures like, for example, search forms are added. Further, where content management is necessary, page classes that allow to insert new entities into the database or to change or delete existing ones are defined. Later on, entity types that provide supplementary information or that facilitate access to the entities of other entity types are added, again complemented by corresponding page classes.

Fig. 3.3 illustrates the iterative and incremental process of conceptual web application modelling. The schemes with the numbers attached represent snapshots of the scheme as it evolves in this process, and the numbers indicate the sequence of extension and refinement. As development proceeds, the content sub-scheme as well as the hypertext sub-scheme gets iteratively extended and refined as described above.

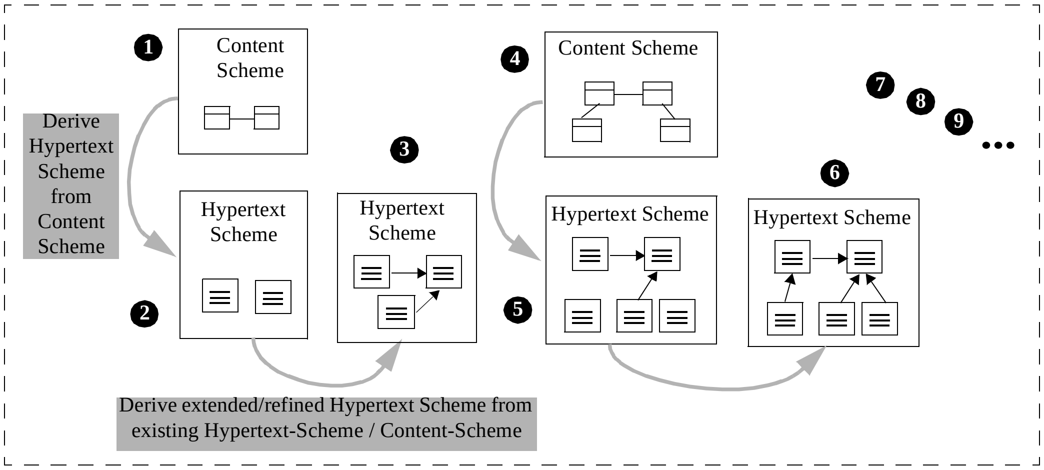

Content Scheme
Hypertext Scheme
Derive Hypertext Scheme from Content Scheme
Derive extended/refined Hypertext Scheme from existing Hypertext-Scheme / Content-Scheme
1
2
3
4
5
6
7
8
9
...

Figure 3.3: Supporting the phase of conceptual modelling by transformers.

The number of iterations may vary, depending on when the modeler decides to have a prototype generated for evaluation or to deploy the web application. Of course, as described in the overall development cycle of web applications, the phase of conceptual modelling may be re-entered later on, and the scheme is again extended and refined.

## 3.3 Supporting conceptual modelling of web applications by transformers

As already motivated in the introduction, during the incremental and iterative adaptation of web schemes in the conceptual modelling phase, many patterns of extending or refining a scheme repeatedly occur. For such recurrent patterns, it would be convenient to have transformers that, when applied to a scheme, perform extensions and refinements as required.

Generally, we have identified the following two cases where such transformers can support the process of conceptual web application modelling:

1. A transformer can derive a hypertext scheme (or a portion thereof) from a content scheme (or a portion thereof) as illustrated in Fig. 3.3 by the gray arcs from content sub-schemes to hypertext sub-schemes. For example, for entity types, often page classes are generated that display the members of this entity type.

2. A transformer can derive an extended hypertext scheme based on an existing hypertext scheme (or a portion thereof) as illustrated in Fig. 3.3 by the gray arcs between hypertext sub-schemes. For example, often an existing page class is extended such that it presents, besides some core entities, also supplementary information for these entities.

Note that from a technical point of view, a transformer could also be used for (1) deriving an extended content scheme based on an existing one, and (2) deriving an extended content scheme based on a hypertext scheme. However, from a practical point of view, it turned out that these cases occur rather seldom: Concerning case (1), an extended content scheme is seldom derived from an existing one; the additional scheme elements are rather defined from scratch. Concerning case (2), a content scheme is seldom derived from an existing hypertext scheme. In the modelling process, usually the hypertext scheme is built after the content scheme but not vice versa. We concentrate on the typical directions of deriving an (extended) hypertext scheme from a content scheme or an hypertext scheme.

Despite the usefulness of web scheme transformers, they are not or only marginally supported by CASE tools. If provided at all, these transformers are defined by means of operations over an internal data structure as used by the CASE tool to store the scheme. Further, these transformers are usually defined for generating an initial hypertext scheme from scratch. Yet they cannot be used for extending an existing hypertext scheme based on entity types that have been added to the content scheme later on. As argued in the introduction, such transformers do not meet the requirements of understandability, usability, flexibility, adaptability, and adequateness.

In this thesis, we introduce the concept of by-example web scheme transformers, which can be used to extend existing modelling languages and CASE tools towards defining and applying transformers. Notably, we do not introduce another web modelling language nor another CASE tool but facilitate conceptual modelling of web applications in the context of existing modelling languages and CASE tools.

## 3.4 Conceptual modelling with WebML: a running example

In this section, we describe WebML, which is a language for conceptual modelling of web applications, by means of a sample web scheme that models a simple web site for a conference. Note that we do not describe WebML completely but introduce a subset of its constructs that is sufficient for our running example. For a complete description of WebML, the reader may refer to dedicated articles and books [32, 29, 95].

WebML basically comprises a content model and a hypertext model that is divided into a composition model, a navigation model, and an operation model. The content model and the hypertext model are introduced in Section 3.4.1 and Section 3.4.2, respectively.

### 3.4.1 Content model

The WebML content model adopts model primitives of the Entity-Relationship model for representing the organization of the web application data. Its fundamental elements are therefore entity types that represent collections of data items and relationship types that represent semantic relationships between data items. Entity types have named properties called attributes. Binary relationship types are composed of pairs of named relationship roles which are attached to entity types. Relationships can be restricted by means of cardinality constraints attached to relationship roles.

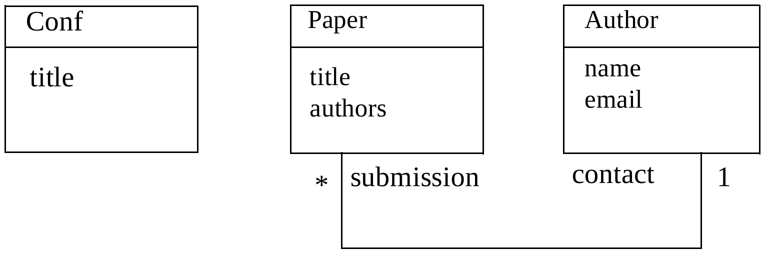
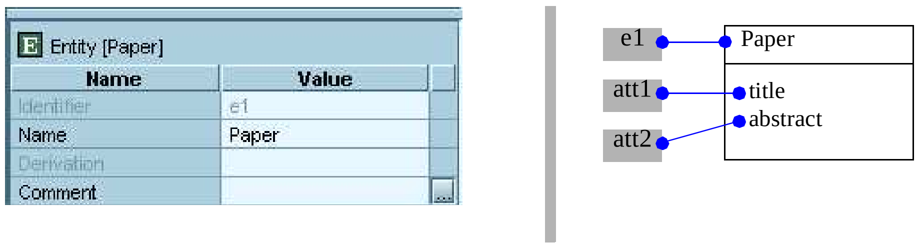


Figure 3.4: WebML content scheme of conference example

**Example 14** *The content scheme depicted in Fig. 3.4 models the content of a conference's web site. Entity type* `Conf` *describes information about the conference itself, i.e. the conference's title represented by attribute* `title`*. This entity type is intended to be a singleton, i.e. to contain only one member. Entity type* `Author` *with attributes* `name` *and* `email` *describes information about authors, while entity type* `Paper` *with attributes* `title` *and* `abstract` *describes information about submitted papers. Relationship roles* `submission` *and* `contact` *express that an author may have submitted many papers and that a paper has exactly one contact, respectively.*

Besides the values like the names of entity types, attributes, and relationship roles, which are shown also in the graphical notation, some properties of scheme elements are usually shown only in separate forms. For example, each of the scheme elements mentioned above has an internal ID that is usually not visualized in the graphical notation. Nevertheless, these IDs are essential parts of web schemes, and for reasons of conciseness, we visualize these IDs directly in the graphical notation whenever required. These IDs are attached to the graphical shape representing the respective scheme element.



Entity [Paper]

| Name | Value |
|---|---|
| Identifier | e1 |
| Name | Paper |
| Derivation | |
| Comment | |

**(a)** **(b)**

Figure 3.5: (a) Property form of entity type `Paper`; (b) Entity type `Paper` having IDs visualized.

**Example 15** *Fig. 3.5(a) depicts the property form of entity type* `Paper`*, which pops up when clicking on this entity type. Besides the entity type's name and an (empty) comment field, the property form exposes the internal ID of entity type* `Paper`*, i.e. ID-value* `e1` *generated by WebML. Attributes* `title` *and* `abstract` *have IDs as well, and they are also shown in a property form that pops up upon clicking on the respective attribute.*

*Fig. 3.5(b) depicts entity type* `Paper` *in the graphical notation as used in this thesis, where the IDs may be visualized. Value* `e1` *represents the ID of entity type* `Paper` *as it is attached to the graphical shape of this entity type. Analogously, values* `att1` *and* `att2` *represent the IDs of attributes* `title` *and* `abstract`*, respectively.*

Besides predefined properties like names and IDs of scheme elements, users may individually define properties of their own and attach them to any scheme element. Thereby, scheme elements can be characterized by properties that could not be expressed otherwise. Note, however, that these properties have no meaning for WebML but for the modelers who have defined them.

A user-defined property is an arbitrary name/value-pair and has the form `propertyname` = `value`, where both `propertyname` and `value` are individually defined by the modeler. As a particularity, omitting the value denotes a boolean property and has the same meaning as `propertyname` = `true`. Further, a property of the form `propertyname` = `false` expresses that the scheme element to which it is attached *does not have* this property set.

In WebRatio, user-defined properties are shown in separate forms. However, for reasons of conciseness, we include these properties in the graphical notation using the style of UML [118]. Thus, a user-defined property is enclosed in curly brackets and is either attached to the scheme element graphically by a line or it is notated together with the name of the respective scheme element to which it is to be attached.

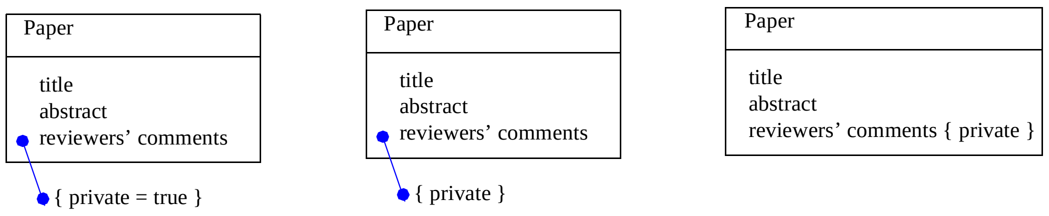


Figure 3.6: Three alternative ways of attaching user-defined property `private` to attribute `reviewers' comments`.

**Example 16** *Fig. 3.6 depicts three alternative ways of attaching a user-defined boolean property named* `private` *to attribute* `reviewers' comments`*. This property shall express that attribute* `reviewers' comments` *is not to be presented to the public but only on pages for internal use. Note, however, that property* `private` *has no meaning for WebML. It is rather subject to the modeler to interpret this property and use the respective attribute accordingly.*

### 3.4.2 Hypertext model

WebML's hypertext model comprises three sub-models, namely the composition model, the navigation model, and the operation model. The composition model describes how content is composed to pages (composition in the small) and how these pages are organized in areas and site-views (composition in the large). The navigation model defines how the pages are to be linked. The operation model specifies operations that can be performed on the underlying data source. Since these operations take parameters from the user interface and are triggered when users traverse particular links, WebML considers the operation model as part of the hypertext model. All these models are explained next.

#### Composition model: composition in the small

From the viewpoint of composition in the small, the composition model defines pages, represented by *page classes*, and their internal organization in terms of elementary pieces of publishable content, represented by *content units*. Most content units publish data that is extracted dynamically from the entities of the content scheme. This is expressed by associating each content unit with an entity type serving as content source. However, entry units, which are a special sort of content units, represent user input at runtime.

WebML provides different kinds of content units, each kind offering an alternative way of arranging content. We make use of data units, index units, and entry units. Data units are used to publish a single object at a time (e.g. a single author). In contrast, index units are used to publish a list of objects at a time (e.g. a list of authors) and enables users choosing one of the presented objects at runtime. Both units specify which attributes of an object shall be displayed. Entry units represent forms and comprise therefore named fields that correspond to the input fields found in the form constructs of markup languages. Entry units are described later in this section.

**Example 17** *Fig. 3.7 depicts a WebML hypertext scheme which is based on the content scheme introduced above. Please ignore the arrows for the moment. They will be explained with the navigation model.*

*Page class* `ConfPage` *defines the conference's main page. It contains a data unit* `ConfDetails` *that has entity type* `Conf` *as content source. Thus,* `ConfPage` *presents the sole member of entity type* `Conf`*. Page*

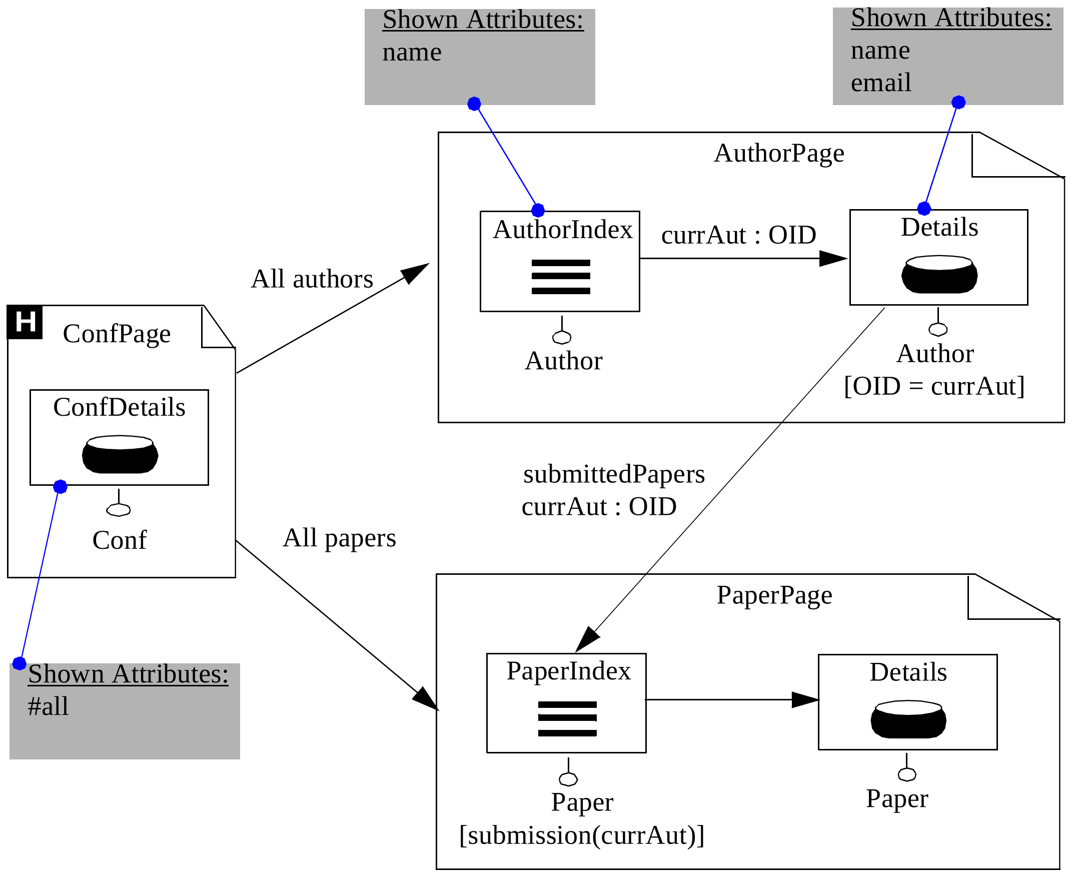


Figure 3.7: WebML hypertext scheme of conference example

*class* `AuthorPage` *defines pages which present a list of authors together with the details of an author that a user chooses therefrom at runtime. Index unit* `AuthorIndex` *represents the list of authors, whereas data unit* `Detail` *is for presenting a single author in a detailed manner. Both units have entity type* `Author` *as content source. Page class* `PaperPage` *is composed analogously.*

*The shaded areas that are attached to some data units and index units and that are labelled "*`Shown Attributes`*" reflect the set of attributes that shall be displayed by the respective unit. In index unit* `Author Index`*, only the names of the authors are displayed, whereas in data unit* `Details` *an author's email address is shown, too. For data unit* `ConfDetails`*, special-value* `#all` *indicates that every attribute of the* *underlying entity type is to be displayed. For index unit* `PaperIndex` *and the corresponding data unit* `Details`*, the list of "shown attributes" is defined but not given in the figure in order not to clutter the scheme with too much details.*

Page classes are further characterized by some distinguishing properties which highlight their "importance" in the Web site. One page class is declared as the site's home page, which is presented whenever a user enters the Web site via its default address. Further, page classes can be designated as landmarks, which are then reachable from every part of the hypertext [1]. In the graphical notation, home pages and landmarks are marked by symbols "`H`" and "`L`", respectively. For example, page class `ConfPage` depicted in Fig. 3.7 is declared as the Web site's home page as indicated by symbol "`H`".

Note that page classes, index units, data units, and links also have IDs that are presented in separate forms. Further, the list of "Shown attributes" of index units and data units are usually presented only in these forms, too. However, for reasons of conciseness, in the graphical notation as used in this thesis, they may be included in the graphical notation.

## Composition model: composition in the large

From the viewpoint of composition in the large, a Web site design can be organized hierarchically by using modularization constructs such as site views and areas.

A site view defines a stand-alone hypertext to be delivered to a particular group of users. A particular site view is entered whenever a user logs into the Web site. Usually a default user group (and a corresponding site view) is provided, which represents visitors of the site that do not log in. For the sake of simplicity, we factor out the definition of user groups and login processes.

Areas are part of a site view and are for grouping page classes that deal with some related topic, such as, for example "maintain paper submissions". Note, however, that a site view may also comprise page classes outside any area. A site view may comprise several areas and, similar to page classes, distinguished areas may be declared as landmarks.

[1]Additionally, page classes can be marked as default pages, yet we do not make use of this property.

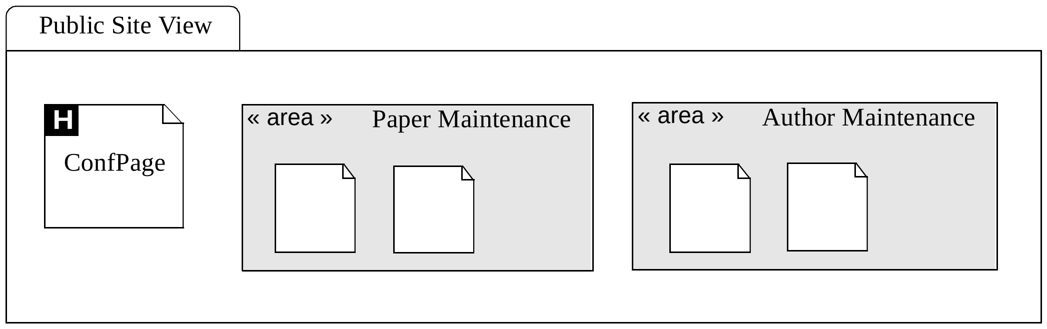


Figure 3.8: Site view `Public Site View` comprising areas `Paper Maintenance` and `Author Maintenance`.

**Example 18** *Fig. 3.8 depicts how the conference web site could be organized to a site view named* `Public Site View` *comprising two areas* `Paper Maintenance` *and* `Author Maintenance`*. These areas comprise some page classes, which are not further discussed.*

*Note that this organization is only for illustrating the concepts "site view" and "area". Obviously, since our running example is necessarily small, it does not make sense to organize it "in the large". Henceforth, we assume that all page classes are contained in a site view, and we defer the use of areas to subsequent chapters.*

### Navigation model

The navigation model defines how pages and content units are linked to form a hypertext. Navigation is expressed at schema level through directed links that either connect a content unit to a content unit or a page class to a page class. A link may transport information in form of link parameters. Links transporting parameters are called *contextual*, in contrast to links without parameters, which are called *non-contextual*.

Links between pages are non-contextual, whereas links between content units are contextual. If the content of a destination page is to be derived independently of the content of the source page (e.g. a link to the site's home page), then the page classes are connected by a non-contextual link. In contrast, if the content of the destination page shall depend on that of the source page, then a contextual link is used, i.e. a link from a content unit of the source page to a content unit of the target page (e.g. a link from a particular author's details to the index unit listing her submitted papers). Thereby, content of the source unit is passed as parameter to the target unit, which then derives its content based on the provided parameter.

Let $L$ be a link from a content unit $U_{src}$ to another content unit $U_{tgt}$. Then a link parameter is notated explicitly with link $L$ as a name/value-pair of the form `paramname:value`. `paramname` stands for the name of the parameter, whereas `value` is either a constant or an attribute of the entity type serving as $U_{src}$'s content source. If $U_{src}$ is a data unit, a reference to an attribute addresses the entity presented in this data unit at the time when the link is traversed. If $U_{src}$ is an index unit, a reference to an attribute addresses an entity selected by the user from the list.

**Example 19** *Reconsider the hypertext scheme of Fig. 3.7. Link* `All authors` *leading from page class* `ConfPage` *to page class* `AuthorPage` *is non-contextual, i.e. no parameters are passed. When traversing link* `All authors`*, a default author page is presented that contains a list of all the authors stored in the database. Link* `All papers` *is defined analogously.*

*The link from index unit* `AuthorIndex` *to data unit* `Details` *is contextual because a parameter* `currAut : OID` *is passed;* `currAut` *is the name of the parameter, whereas* `OID` *represents the author entity that a user selects from the list of authors displayed by index unit* `AuthorIndex`*. Link* `submittedPapers` *is contextual, too.* `OID` *represents the author entity displayed in data unit* `Details` *at the time when the user traverses this link.*

A content unit may have selection conditions attached, which constrain the items presented in the unit to those that fulfill the respective conditions. If the content unit is targeted by a link, then this selection condition may be parametric, i.e. it may be based on a link parameter. A parametric selection condition has one of the following two forms:

1. A comparison condition has the form `[attrib` *op* `paramname]`, where *op* is a comparison operator, `attrib` refers to an attribute of the target unit's content source, and `paramname` refers to a link parameter. Thereby, only those entities of the target unit's content source are presented where this comparison condition is true for the actual value of the link parameter.

2. A relationship condition has the form `[relrole (paramname)]`, where `relrole` refers to a relationship role attached to the target unit's content source, and `paramname` refers to the value of a link parameter, which must be an OID. Thereby, only those entities of a target unit's content source are presented that are connected by relationship `relrole` to the object represented by parameter `paramname`.

**Example 20** *Reconsider the hypertext scheme of Fig. 3.7. Data unit* `Details` *in page class* `AuthorPage` *has a comparison predicate* `OID = currAtt` *attached. It defines that the entity to be presented must have the same OID as that provided through the incoming parameter* `currAut`*. In other words, data unit* `Details` *presents the author selected by the user in index unit* `AuthorIndex` *at runtime.*

*Index unit* `PaperIndex` *in page class* `PaperPage` *has a relationship condition* `submission(currAut)` *attached. It defines that only those papers are presented that are submissions of the author object provided through parameter* `currAut`*.*

For reasons of conciseness, WebML defines default parameters and default selection conditions. Unless stated otherwise, a link from an index unit or a data unit always passes a parameter representing the selected or currently presented entity, respectively. Analogously, unless stated otherwise, a target data unit presents the entity provided as parameter. Further, if a unit has only one incoming parameter, a relationship condition needs only to refer to the relationship role and is implicitly parameterized by the respective parameter.

**Example 21** *Reconsider the hypertext scheme of Fig. 3.7. The link from index unit* `PaperIndex` *to data unit* `Details` *specifies no parameter explicitly. Thus, a parameter of the form* `curr:OID` *is implicitly defined. Data unit* `Details` *does not specify a selection condition, although it is the target of a link. Thus, a selection condition of the form* `OID = curr` *is implicitly defined, such that the link from index unit* `PaperIndex` *to data unit* `Details` *in page class* `PaperPage` *behaves analogously to the link from index unit* `AuthorIndex` *to data unit* `Details` *in page class* `AuthorPage`*. Further, index unit* `PaperIndex` *has only one incoming link. Thus, it would be sufficient to write* `[submission]` *instead of* `[submission(currAut)]`*.*

Each parametric selection condition can be tagged "`implied`" to denote that the condition is optional. In this case, the absence of a value for the parameter used in the condition denotes that this condition is not checked at all. Optional selection conditions are typically necessary where the same content unit is target of several contextual links, each providing a different link parameter and each requiring a different selection condition. Consequently, depending on which link is traversed, selection conditions that refer to a parameter that is *not* provided are ignored.

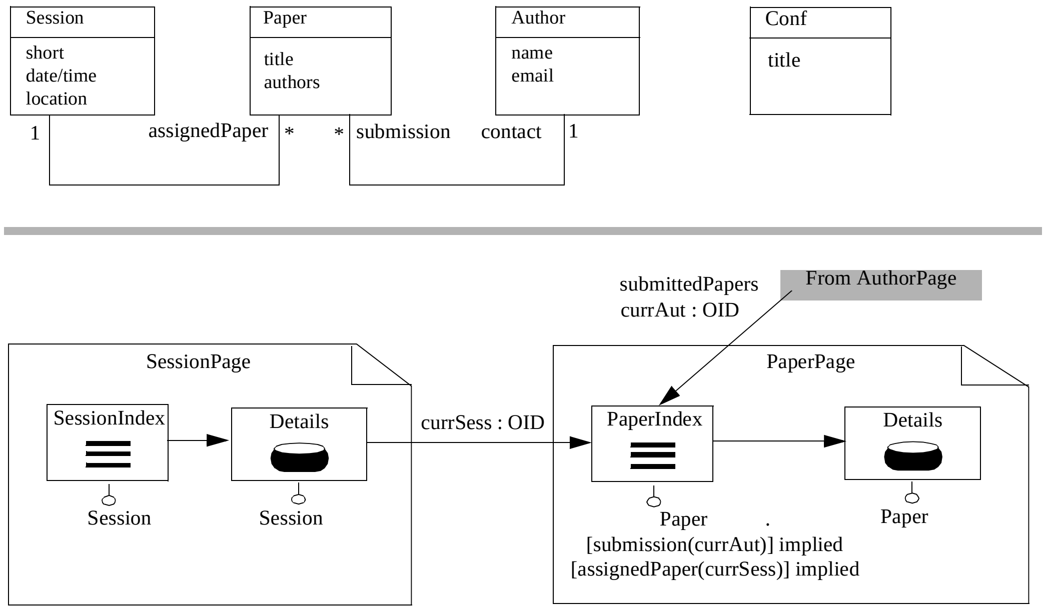


Figure 3.9: WebML scheme extended by an entity type `Session` and a corresponding page class.

**Example 22** *Suppose that the content scheme of our running example is extended as illustrated in the upper part of Fig. 3.9. Entity type* `Session` *together with a relationship to existing entity type* `Paper` *have been added. Each paper is assigned to exactly one session, and each session may have several papers assigned. Suppose further that a page class* `SessionPage` *together with an index unit* `SessionIndex` *and a data unit* `Details` *have been added to the hypertext scheme as illustrated in the lower part of Fig. 3.9.*

*Note that index unit* `PaperIndex` *has been extended by a relationship condition* `[assignedPaper(currSess)] implied`*. Link* `assignedPapers` *from the data unit representing a session entity to index unit* `PaperIndex` *works as follows: When traversed, the session object currently presented in the data unit is passed along the link via the parameter named* `currSess`*. In target index unit* `PaperIndex`*, only relationship condition* `[assignedPaper(currSess)] implied` *is checked such that the papers assigned to the respective session is displayed. The other relationship condition* `[submission(currAut)] implied` *is not checked in this case because link* `assignedPapers` *does not provide a parameter* `currAut`*. Analogously, if link* `submittedPapers` *is traversed, then relationship condition* `[assignedPaper(currSess)] implied` *is not checked.*

### Operation model

The operation model defines operations which allow to manipulate the data source via the web interface. Such operations are expressed by *operation units* which, like content units, can be navigated, take parameters via incoming links, and which pass parameters via outgoing links. However, in contrast to content units, operation units do not display content but rather perform actions based on the parameters provided. Therefore, these units are placed outside page classes. As a further difference, each operation unit has exactly two outgoing links, namely a KO-link which is traversed automatically when the operation fails, and an OK-link which is traversed automatically when the operation succeeds.

WebML provides the operation units for creating, deleting, and modifying data items as well as for establishing or removing relations between data items. We explain here *create units*. Each create unit performs the creation of a new data item within an associated entity type. For that purpose, the create unit assigns each attribute of the entity type a value which is either a constant or a parameter value coming from an input link. These parameters typically origin from entry units but can be attributes of other content units as well. If creation succeeds, the newly generated object is passed along the outgoing OK-link. If creation fails, the KO-link is traversed yet without passing any parameter.

**Example 23** *Fig. 3.10 depicts a WebML hypertext scheme which comprises, among other elements, a page class* `AuthorCreation` *with an entry unit* `AuthorForm`*. This entry unit provides two entry fields labelled* `nameEntry` *and* `emailEntry` *and enables users to enter values for an*

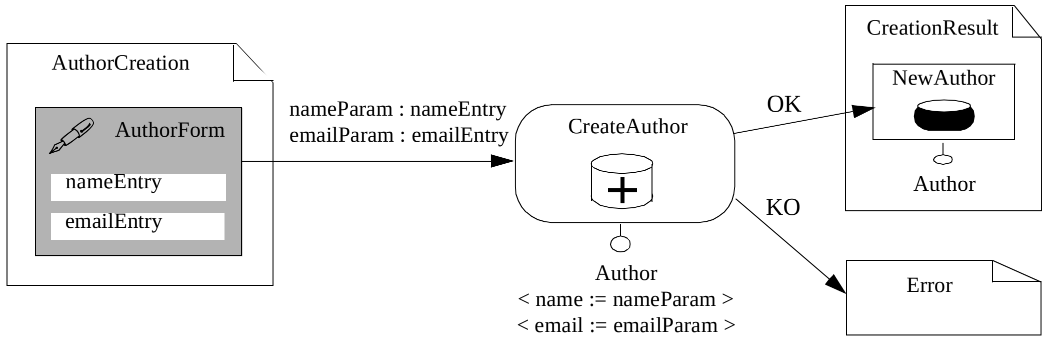


Figure 3.10: WebML hypertext scheme with operation unit "Create Author"

*author's name and her email address, respectively. These values are used for creating a new author as explained next.*

*If a user submits her input, the values entered in these fields are passed to create unit* `CreateAuthor` *via parameters labelled* `nameParam` *and* `emailParam`*. The create unit is associated with entity type* `Author` *and assigns the incoming parameter values to the entity type's attributes. Hence, a new author is created with attribute values as entered by the user. If creation succeeds, the OK-link is traversed and the newly generated author is presented in data unit* `NewAuthor` *defined in page class* `CreationResult`*. If creation fails, the KO-link is traversed and an error message is displayed.*

## 3.5 Use cases for transformers in the running example

When analyzing the scheme of the running example introduced in this section, one can identify a number of patterns of extending and refining the scheme that have been applied. In the remainder of this section, we identify and describe these recurrent modelling tasks and motivate thereby the use of transformers supporting them.

**Generate page class with index and data unit.** In the running example, a page class comprising an index unit and a data unit has been generated repeatedly, for entity type `Paper`, for entity type `Author`, and entity type `Session`. However, this modelling task has not been performed for singleton entity type `Conf`.

Suppose that we have a transformer `IuDuPCforENT` that generates a page class with an index and a data unit for each entity type of a scheme. Then, this transformer could be applied to the content scheme and generate corresponding page classes all at once. Note that entity type `Session` has been defined later on and that entity type `Conf` is not to be considered. Thus, it must be possible to apply the transformer twice, i.e. the first time such that it considers entity types `Paper` and `Author` but not `Conf`, and a second time for entity type `Session` only.

**Connect data unit to index unit.** In the running example, data units have been connected to index units several times according to the following rule: (1) Suppose a data unit *du* and an index unit *iu*, two entity types $ent_{du}$ and $ent_{iu}$ as their respective content sources, and a relation from $ent_{du}$ to $ent_{iu}$ via a to-many relationship role *relrole*. (2) Then, a link from data unit *du* to index unit *iu* is generated, which has a link parameter of the form "*curr* = *OID*" attached. Further, index unit *iu* is extended by a relationship condition of the form "`[`*relrole*(*curr*)`] implied`". In the running example, this modelling task has been performed twice, namely for the two links that target index unit `PaperIndex`.

Suppose that we have a transformer `ConnectDuToIu` that generates a link according to the rule described above. This transformer could be applied for the data unit representing an author's details as well as for the data unit representing a session's details. Since the session details have been defined later on, there are two alternatives of how `ConnectDuToIu` can be applied, namely (1) separately for each of these data units, and (2) only once such that both data units are considered by a single application. Both alternatives achieve the same result.

**Provide entry unit with create unit.** In the running example, an entry unit with entry fields has been generated after an entity type with attributes. Further, a create unit has been defined that takes the values of these entry fields as parameters and, once triggered, creates a new entity type according to these values. Though this task has been performed only once, i.e. for entity type `Author`, it is likely that entry units with subsequent create units should be provided for other entity types as well.

Suppose that we have a transformer `ProvideCreateOp` that generates an entry unit and a create unit for an entity type as described above. Then, this transformer could simply be applied to entity type `Author`, achieving an extended hypertext scheme with a new entry unit and a new create unit as depicted in Fig. 3.10.

# Chapter 4

# By-example transformers: basics

## Contents

In this chapter, we define the basic principle of how to define transformers declaratively through graphical templates that adopt the graphical notation used for defining schemes. We distinguish two kinds of templates, namely query templates and generative templates. A query template defines retrieval of scheme elements from schemes. A generative templates defines how to extend or refine scheme elements that are provided as parameter. A transformer combines a generative template and a query template such that the scheme elements retrieved by the query template are passed as parameters to the generative template.

In Section 4.1, we start with an overview of the basic architecture of *TBE*. In Section 4.2, we define the data model to which graphical schemes are mapped and the data manipulation language TBE-QML, which queries and manipulates schemes represented in terms of that data model. In Section 4.3, we introduce query templates as follows: We first informally explain the concept of graphical query templates. Then, we define a model of query templates using a formal notation, which expresses also those parts of a query template explicitly that are expressed implicitly in the graphical notation, e.g. by semantic relationships such as visual containment. Based on this formal notation, we define the semantics of query templates in terms of TBE-QML. Finally, we describe how the graphical notation of query templates maps to the formal one. Analogously, Section 4.4 introduces generative templates. Section 4.5 defines transformers, which are composed from query templates and generative templates.

## 4.1 Overview

A transformer comprises the following two parts: (1) One part defines the input configuration. It is specified through constraints that a particular configuration of scheme elements must fulfill in order to get extended or refined by the transformer. Based on this declarative specification, *TBE* derives a query that retrieves from a scheme all scheme element configurations that match the input configuration. Thus, this part is referred to as the transformer's *query part*. (2) The other part generically defines the output configuration based on the input configuration. For each scheme element configuration matching the input configuration, the output configuration defines where to generate new scheme elements and where to modify existing ones. This part is referred to as the transformer's *generative part*.

A transformer's query part and generative part are both expressed by graphical templates, i.e. by a *query template* and a *generative template*, respectively. Each template is defined by using the same scheme elements and the same notation as used for specifying schemes. However, there are the following two differences:

1. In place of concrete values of scheme elements like names, cardinalities, scheme element IDs, etc., a template comprises typed variables that represent such values. The graphical shapes of the scheme elements as used in templates are the same as those used in schemes but are referred to as *scheme element skeletons*. The type of a variable is determined by the scheme element skeleton in which the variable is placed.

2. Additionally, a template contains *symbols*, *comparison constraints*, and *expressions*. Symbols are used for tagging variables in order to distinguish, for example, variables representing scheme elements to be provided as parameter from variables representing scheme elements to be generated. Comparison constraints are used to constrain variable bindings. Expressions define how to derive new values, e.g. by means of string concatenation. We will subsequently refer to these symbols, comparison constraints, and expressions as *TBE-directives*.

The transformer's graphical specification, which is given by a query template and a generative template, is mapped to operations in terms of TBE-QML (TBE query and manipulation language) introduced in this work. TBE-QML is based on domain relational calculus [87, 93] (short DRC) such that the semantics of transformers notated graphically is precisely defined in terms of a concise language with strong formal foundation.

The left part of Fig. 4.1 illustrates the process of defining a by-example transformer. We have taken WebML as an example for a modelling language. The parts outside of the gray underlay are those presented to the modeler. The parts in gray underlays are processes performed internally by the *TBE* -system. The by-example transformer is given by a query template and a generative template, both specified using WebML scheme elements in addition with *TBE* -directives. From these templates, process "TBE-QML generator" generates TBE-QML statements that represent the executable transformer.

Once defined, a transformer can be applied many times, each time turning an input scheme into an output scheme as desired. However, since TBE-QML

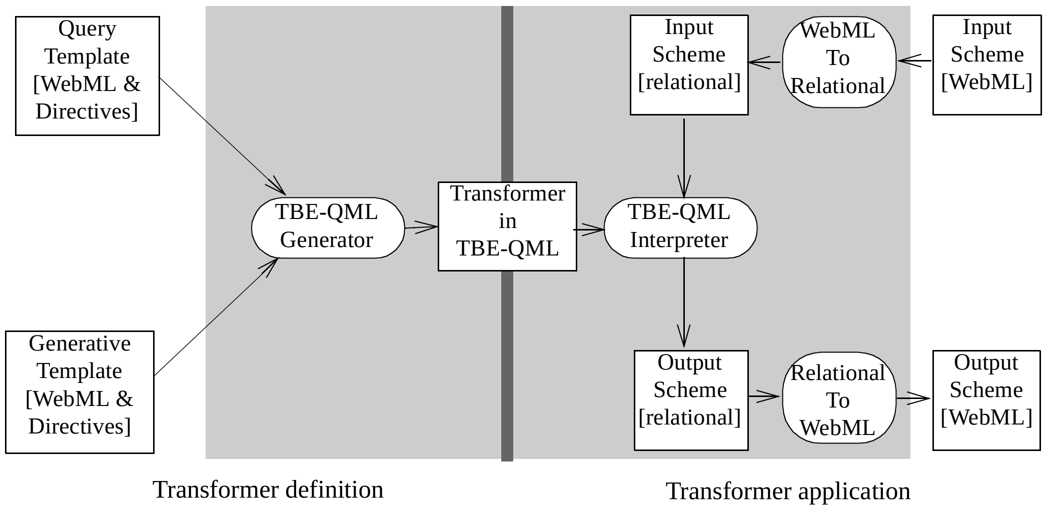


Figure 4.1: Left: Process of defining by-example transformers within WebML; Right: Process of applying by-example transformers to schemes

is a relational data manipulation language, the input scheme as represented by a particular modelling language $L$ has to be mapped to a relational representation. Similarly, the result of processing the TBE-QML statements is an output scheme in relational representation and has to be mapped back to the representation of schemes as used by modelling language $L$. The process performing these mappings must be defined once for each modelling language $L$ that shall support *TBE*, and this process must support two directions, i.e. $L$_To_Relational and Relational_To_$L$.

The right part of Fig. 4.1 illustrates the application of a given transformer to an input scheme by the example of the web modelling language WebML. Again, the elements in gray underlays denote processes performed internally be the *TBE*-system, whereas elements outside this area are presented to the modeler. This application is processed as follows: First, process “WebML_To_Relational“ maps the input scheme defined in WebML to a relational representation. Process “TBE-QML Interpreter” then performs the transformation as defined by the transformer. The resulting output scheme, still in relational representation, is then mapped back to the actual result, i.e. an output scheme in WebML representation.

## 4.2 Data model and data manipulation language

In this section, a relational representation of schemes is defined and explained by the example of WebML. This representation is the basis for defining transformer's semantics in terms of the data manipulation language TBE-QML, which is defined in this section, too.

### 4.2.1 Relational representation of schemes

#### Model

The basic model for representing schemes is given by a set of universes $\widehat{U}$ and a set of relations $\widehat{R}$ (sets of universes or relations are marked by a $\widehat{}$-symbol). Each universe $U \in \widehat{U}$ represents a sort $U$ of scheme elements. Each relation $R \in U_1 \times \ldots \times U_n$, with $R \in \widehat{R}$ and, for $i = 1 \ldots n, U_i \in \widehat{U}$, represents a set of relationships between scheme elements of sorts $U_i$. Relations may be overloaded, i.e. several relations may share the same name but have different universes as arguments. For convenience, we write $R \in (U_1 \cup U_2) \times U3$ as short for $R \in U_1 \times U_3$ and $R \in U_2 \times U_3$. Further, for each relation, one or more key constraints may be defined, each notated by underlining the universes forming the key.

Note that some relations actually represent single-valued functions. However, for defining a data manipulation language, it is more convenient to have a model that is solely based on relations. This is not a restriction because any single-valued function, which has the general form $f : U_{d1} \times \ldots \times U_{dn} \rightarrow U_{r1} \times \ldots \times U_{rm}$, can be represented through a relation of the form $R \in \underline{U_{d1} \times \ldots \times U_{dn}} \times U_{r1} \times \ldots \times U_{rm}$.

**Example 24** *WebML's content model is defined by the following universes and relations: Entity types, their attributes, and relationship roles are represented by universes $E$, $A$, and $Role$, respectively. The names of these elements are drawn from universe $N$ and are defined by relation* name $\in (\underline{E \cup A \cup Role}) \times N$*. Each attribute is defined at an entity type as expressed by relation* definedAt $\in \underline{A} \times E$*. Each WebML binary relationship type connecting two entity types is composed of two relationship roles that are attached to the entity types. This is expressed by relation* attachedTo $\in \underline{Role} \times E$, *whereas the binary relationship type is represented by relation* binrel $\in \underline{Role} \times \underline{Role}$. *Further, a relationship role's multiplicity is drawn from literal universe* $Mul = \{\texttt{0..1}, \texttt{1..1}, \texttt{0..N}, \texttt{1..N}\}$ *and is defined by relation* mul $\in \underline{Role} \times Mul$.

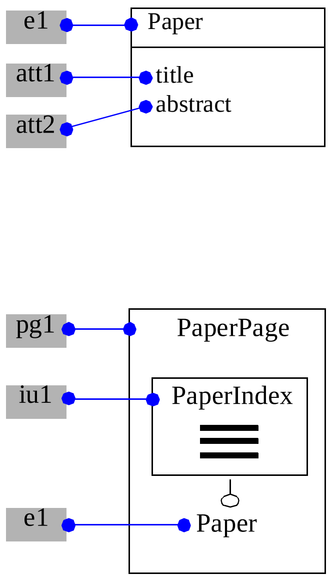
*WebML's hypertext model consists of the following universes and relations: Page classes, data units, index units, and links are represented by universes* $P$, $DU$, $IU$, *and* $Link$, *respectively. For convenience, we define a universe of content units* $CU = DU \cup IU$ *that generalizes data and index units. All above mentioned elements have names as expressed by relation* name $\in (\underline{P \cup CU \cup Link}) \times N$. *A link is directed and connects either a page class to another page class (non-contextual link) or a content unit to another content unit (contextual link). This is captured by relations* nctxLinkFromTo $\in \underline{Link} \times P \times P$ *and* ctxLinkFromTo $\in \underline{Link} \times CU \times CU$, *where the first argument denotes the link while the second and the third argument denote its source and target, respectively. Each unit is contained in a page class as expressed by relation* containedIn $\in \underline{CU} \times P$. *Further, each unit refers to an entity type that serves as content source as captured by relation* contentSource $\in \underline{CU} \times E$.

Note that formalizing WebML schemes is not the scope of this work but is required for defining transformer's semantics. Thus, we have presented here universes and relations only for a subset of WebML scheme elements. The complete set of universes and relations is given in [81].

### Extension

Each scheme $S$ is represented by a set of universes $\widehat{U}$ and a set of relations $\widehat{R}$. Each scheme element $e$ is member of exactly one universe $U \in \widehat{U}$ and is notated as $e : U$. A tuple of a relation $R \in U_1 \times \ldots \times U_n$ is denoted as $R\langle e_1, \ldots, e_n\rangle$, with $R \in \widehat{R}$ and, for $i = 1 \ldots n, e_i \in U_i$. Note that we use the same notation as for *signatures* of universes and relations also for *extensions* thereof. To be accurate, when referring to a scheme $S$, we would have to write $\widehat{U}_S$ and $\widehat{R}_S$, etc. However, for reasons of readability, we omit the index $S$ as it is clear from the context whether we refer to a signature or to an extension.

**Example 25** *The upper part of Fig. 4.2(a) depicts the graphical representation of entity type* `Paper`. *The values in gray underlays represent*



**(a)**

| Universe Members | Relation Members |
|---|---|
| e1 : E<br>att1 : A<br>att2 : A<br>Paper : N<br>title : N<br>abstract : N | name ⟨e1, Paper ⟩<br>name ⟨att1, title ⟩<br>name ⟨att2, abstract⟩<br>definedAt ⟨att1, e1⟩<br>definedAt ⟨att2, e1⟩ |

| Universe Members | Relation Members |
|---|---|
| pg1 : P<br>PaperPage : N<br>iu1 : IU<br>PaperIndex : N<br>e1 : E<br>Paper : N | name ⟨pg1, PaperPage⟩<br>name ⟨iu1, PaperIndex⟩<br>containedIn ⟨iu1, pg1⟩<br>contentSource ⟨iu1, e1⟩<br>name ⟨e1, Paper ⟩ |

**(b)**

Figure 4.2: (a) Entity type "Paper" and page class "PaperPage" in graphical representation; (b) corresponding relational representations

*scheme element IDs. The upper part of Fig. 4.2(b) illustrates the corresponding relational representation of this entity type* `Paper`*. Members of universes are notated as $m : U$, where $m$ is the universe member and $U \in \widehat{U}$ is the universe to which $m$ belongs.*

*Scheme elements* `e1` $: E$ *and* `Paper` $: N$ *represent the entity type's ID* `e1` *and its name* `"Paper"`*, respectively. Tuple* $\mathrm{name}\langle$`e1`$,$`Paper`$\rangle$ *attaches the name property to the entity type. Scheme elements* `att1` $: A$ *and* `title` $: N$ *represent the ID and the name of attribute* `title`*, whereas scheme elements* `att2` $: A$ *and* `abstract` $: N$ *represent the ID and the name of attribute* `abstract`*. Relations* $\mathrm{name}\langle$`att1`$,$`title`$\rangle$ *and* $\mathrm{name}\langle$`att2`$,$`abstract`$\rangle$ *both attach an attribute's name property to the respective ID. Relations* $\mathrm{definedAt}\langle$`att1`$,$`e1`$\rangle$ *and* $\mathrm{definedAt}\langle$`att2`$,$`e1`$\rangle$ *specify that attributes* `title` *and* `abstract` *are defined at entity type* `Paper`*, respectively. Note that relation* definedAt *is defined over IDs of attributes and entity types.*

*Similarly, the lower part of Fig. 4.2(a) depicts the graphical representation of page class* `PaperPage`*, whereas the corresponding relational* *representation is shown to the right in the lower part of Fig. 4.2(b). The mapping should be self-explanatory.*

## 4.2.2 Relational manipulation language TBE-QML

TBE-QML is a relational data manipulation language for manipulating web schemes in a relational representation. It comprises a query part for retrieving scheme elements and a modification part for manipulating these scheme elements. Note that TBE-QML treats schemes as "data" to be manipulated, and the "database schema" of this "data" is the relational representation of web schemes as defined previously in Section 4.2.1.

The manipulation part of TBE-QML is rather simple. It consists of expressions for deriving new values and of operations for inserting tuples into relations. This will be explained with generative templates later in this chapter. TBE-QML's query part builds on domain relational calculus (DRC) which is a query language for the relational model. In order to be self-contained, we shortly review DRC here. For a more detailed explanation, confer to the definition of DRC [87, 93] or to a standard database book, e.g. Ramakrishnan et.al. [111].

An expression in DRC is of the form

$$\{\langle x_1, \ldots, x_n\rangle \in U_1 \times \ldots \times U_n \mid P(x_1, \ldots, x_n)\}$$

where, for $i = 1 \ldots n$, each $x_i$ is a domain variable ranging over the corresponding universe $U_i$ and $P$ is a formula (a predicate) with $x_1, \ldots, x_n$ as free variables. The value of such an expression is the set of labelled tuples $\langle x_1 : a_1, \ldots, x_n : a_n\rangle$ such that $P(a_1, \ldots, a_n)$ evaluates to true. For each $i$, $a_i$ is an element of the universe associated with $x_i$, and the label of $x_i$ serves as label for the particular attribute of the result relation.

Formulae are composed of atoms, each having one of the following forms:

- $\langle x_1, \ldots, x_n\rangle \in R$, where $R \in \widehat{R}$ is a relation with $n$ attributes, and for $i = 1 \ldots n$, each $x_i$ is a domain variable. This will be referred to as *relationship constraint.*

- $x\ \Theta\ y$, where $x$ is a domain variable, $y$ is either a domain variable or a domain constant, and $\Theta$ is a comparison operator $(<, \leq, =, \neq, >, \geq)$. The domains of $x$ and $y$ must be comparable by operator $\Theta$. This will be referred to as *comparison constraint.*

Formulae of DRC are defined using the following rules:

- An atom is a formula. Such a formula will be referred to as *simple constraint.*
- If $P$ is a formula, then so are $\neg P$ and $(P)$. Further, if $P_1$ and $P_2$ are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \rightarrow P_2$. These formulae will be referred to as *complex constraints.*
- If $x$ is a domain variable, then $\forall x$ and $\exists x$ are quantifiers containing $x$. If $P(x)$ is a formula that contains $x$ but no quantifier containing $x$, then $(\forall x P(x))$ and $(\exists x P(x))$ are formulae. These formulae will be referred to as *quantified constraints.*

For reasons of readability, we directly associate a variable $x$ with the universe $U$ that serves as domain for $x$ and write, for example, $\exists x \in U : P(x)$. Further, in order not to clutter formulae with parentheses, we write $\exists x_1 \in U_1, \exists x_2 \in U_2 : P(x_1, x_2)$ as a shorthand for $\exists x_1 \in U_1 : (\exists x_2 \in U_2 : P(x_1, x_2))$.

$$
\begin{aligned}
\underline{\text{EntWithAttribs}} ::= \{ \langle\, &\text{ENT\_ID, ENT} \rangle \in \text{E} \times \text{N} \mid \\
&\exists\ \text{ATT\_ID} \in \text{A}, \\
&\exists\ \text{ATT} \in \text{N} : ( \\
&\langle \text{ENT\_ID, ENT} \rangle \in \text{name} \wedge \\
&\langle \text{ATT\_ID, ATT} \rangle \in \text{name} \wedge \\
&\langle \text{ATT\_ID, ENT\_ID} \rangle \in \text{definedAt} \wedge \\
&\text{ATT} = \text{“title”} ) \\
\}\ &
\end{aligned}
$$

Figure 4.3: DRC statement `EntWithAttribs` retrieving IDs and names of entity types with an attribute named `"title"`.

**Example 26** *Fig. 4.3 illustrates a DRC expression that retrieves the IDs and the names of entity types that comprise at least one attribute named* *"title". The result relation* $Rel_Q(\texttt{ENT\_ID}, \texttt{ENT})$ *is defined by the tuple of domain variables* $\langle \texttt{ENT\_ID}, \texttt{ENT} \rangle \in E \times N$. *Domain variables* `ATT_ID` *and* `ATT` *with domains* $A$ *and* $N$*, respectively, are existentially quantified and do not occur in the query's result.*

*Relationship constraint* $\langle \texttt{ENT\_ID}, \texttt{ENT} \rangle \in name$ *states that only those pairs of entity type IDs and names are to be selected, where the name is actually associated to the ID via relation "name". The IDs and names of attributes are related to each other analogously by relationship constraint* $\langle \texttt{ATT\_ID}, \texttt{ATT} \rangle \in name$. *Further, only entity types that comprise an attribute named "title" are considered. This is expressed by the following two constraints: (1) Relationship constraint* $\langle \texttt{ATT\_ID}, \texttt{ENT\_ID} \rangle \in definedAt$ *in the context of existential quantified variable* `ATT_ID` *states that there must exist some attribute* `ATT_ID` *defined at entity type* `ENT_ID`*. (2) Comparison constraint* $\texttt{ATT} = \text{“}\texttt{title}\text{”}$ *states that only attributes named "title" are considered.*

*The result of applying this DRC-expression to the content scheme of our running example (depicted in Fig. 3.4) is a relation* $Rel_Q(\texttt{ENT\_ID}, \texttt{ENT})$ *with tuples* $\langle \texttt{e1}, \texttt{Paper} \rangle$ *and* $\langle \texttt{e3}, \texttt{Conf} \rangle$.

## 4.3 Query templates

### 4.3.1 Informal description

A query template expresses a query that, when applied to a scheme, retrieves scheme elements therefrom. In particular, the query template defines that combinations of scheme elements are to be retrieved that adhere to a set of constraints. This is expressed declaratively by a set of *variables*, each representing a scheme element of a particular domain, and a set of *constraints*, each restraining valid bindings of one or more variables. The domains of variables as well as most of the constraints are implicitly expressed by the placement of variables within scheme element skeletons. Further, some of the query template's variables are declared as *result variables* such that their bindings appear in the query's result. The other *non-result variables* are for expressing constraints only.

**Example 27** *Fig. 4.4 (a) depicts query template* `EntWithAttribs` *that selects entity types comprising at least one attribute named "title". Variables* `ENT` *and* `ATT` *represent entity types and attributes, respectively, as*

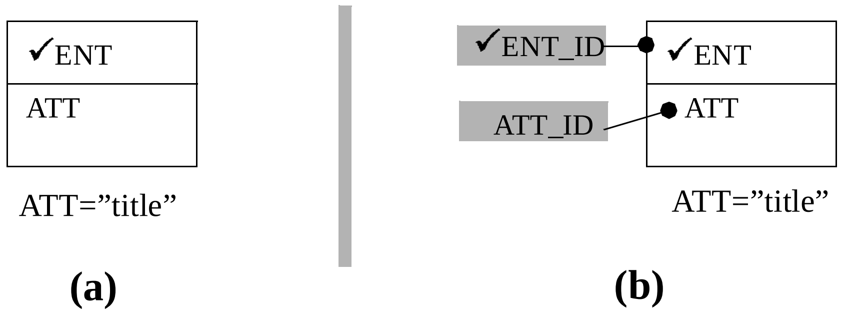


Figure 4.4: Query template "EntWithAttribs": (a) ID-variables hidden; (b) ID-variables visualized.

*expressed by the graphical arrangement of these variables. Variable* `ENT` *is a result-variable as it is preceded by symbol* $\surd$*, whereas variable* `ATT` *is a non-result variable. Thus, the query's result comprises only entity types but no attributes. However, since variable* `ATT` *is graphically contained in the entity type represented by variable* `ENT`*, only those entity types are selected that comprise at least one attribute. Moreover, such an attribute must be named "title" as expressed by constraint* `ATT` = "`title`".

In the example above, we have abstracted from scheme element IDs. This abstraction is much in the sense of tools for editing schemes, where IDs are maintained in the background. However, despite being maintained in the background, such IDs are essential parts of schemes, and query templates consequently have to deal with such IDs, too. For example, it might be required to specify a query that selects IDs of scheme elements. Further, for specifying constraints that check whether two scheme elements are identical, one must compare the IDs of the respective scheme elements.

Therefore, we introduce *ID-variables*, i.e. variables that are in place of scheme element IDs and that represent therefore such IDs. These ID-variables can be visualized in query templates the same way as IDs can be visualized in schemes and can then be used like any other variable. For example, if a query shall select IDs of scheme elements, the ID-variable representing the respective IDs is declared to be a result variable. Similarly, ID-variables can be used in comparison constraints. The following two examples illustrate this:

**Example 28** *Fig. 4.4 (b) depicts query template* `EntWithAttribs` *that has ID-variables visualized. Variable* `ENT` *represents names of entity types, whereas variable* `ENT_ID` *represents entity types themselves, i.e. their IDs. Since both variables are result variables, the query template will selects pairs of entity type IDs and names. Obviously, due to the graphical arrangement of variables* `ENT_ID` *and* `ENT`*, only those pairs are selected where the name actually is that of the scheme element represented by the respective ID. Analogously, variable* `ATT` *represents names of attributes, whereas variable* `ATT_ID` *represents their respective IDs.*

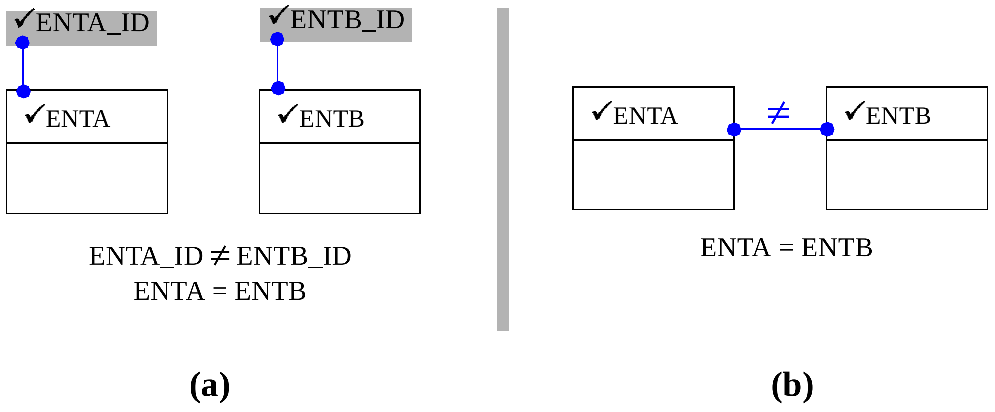


Figure 4.5: Query template "DuplicateEntityTypes": (a) comparison via visualized ID-variables and (b) via graphical shapes representing ID-variables

**Example 29** *Fig. 4.5(a) illustrates query template* `DuplicateEntityTypes` *selecting pairs of entity types that are different but share the same name. The names of such two entity types are represented by variables* `ENTA` *and* `ENTB`*, whereas the respective IDs are represented by ID-variables* `ENTA_ID` *and* `ENTB_ID`*. Thus, constraint* `ENTA_ID` $\neq$ `ENTB_ID` *checks whether two entity types are actually different, whereas constraint* `ENTA` $=$ `ENTB` *checks whether they are named equally.*

The above mentioned technique, which addresses ID-variables by visualizing them, has the following drawback: When editing schemes, modelers usually are not concerned with IDs of scheme elements such that they usually need not to be visualized. In contrast, when defining query templates, IDs of scheme elements often are to be selected or compared such that it would be the standard case that ID-variables are visualized. Consequently, query templates, which make extensive use of ID-variables, would look different from schemes, which usually hide IDs of scheme elements.

Thus, we provide additional techniques that allow for the most common use cases to address ID-variables implicitly and, as we feel, in an intuitive manner. We provide a way (1) for comparing two ID-variables and (2) for expressing whether an ID-variable becomes a result variable or not, each time without the need of visualizing the respective ID-variables.

Concerning point (1), graphical shapes of scheme elements are declared to implicitly represents the ID of this scheme element. Consequently, the graphical shape of a scheme element skeleton represents the corresponding ID-variable, which is defined implicitly. Thus, ID-variables can be addressed by pointing to graphical shapes, and comparing two ID-variables can be expressed by connecting these shapes by a line labelled with the respective comparison operator.

**Example 30** *Fig. 4.5(b) depicts query template* `DuplicateEntityTypes`*, where the ID-variables are addressed via the graphical shapes of the entity type skeletons. The line labelled "*$\neq$*" connecting these two entity type skeletons has the same meaning as constraint* `ENTA_ID` $\neq$ `ENTB_ID` *as expressed in Fig. 4.5(a) by referring to the names of ID-variables, which have to be visualized for that purpose.*

Concerning point (2), a default coupling between the name of a scheme element and the ID of this scheme element is declared such that when the former is to be selected, then the latter is selected, too. For example, when the name of an entity type is selected, then the ID of this entity type is implicitly selected, too. A similar default coupling is defined also for the names of attributes, relationship roles, page classes, index units, data units, etc.

Consequently, whenever the variable representing the name of a scheme element is a result variable, then the corresponding ID-variable becomes a result variable, too. This default behavior has turned out to be appropriate in most cases. For those cases where it is not appropriate, one still can visualize the ID-variable and tag this variable individually, thereby overriding the default coupling.

**Example 31** *Reconsider query template* `EntWithAttribs` *as shown in Fig. 4.4(a). Variable* `ENT` *represents the name of an entity type. Because variable* `ENT` *is a result variable, the ID-variable representing the ID of this entity type becomes a result variable, too. Thus, it is not required to visualize this ID-variable in order to tag it manually. Variable* `ATT` *represents the name of an attribute. Since this variable is a non-result variable, the corresponding ID-variable is a non-result variable, too. Thus, the query template shown in Fig. 4.4(a), which hides ID-variables, has exactly the same meaning as that depicted in Fig. 4.4(b), which visualizes ID-variables.*

These two ways allow, for the very most cases, to abstract from ID-variables. However, there are the following (seldom) cases where we still have to visualize them. One such case is if we want to select only IDs of scheme elements but not their names or vice versa. This must be expressed by visualizing ID-variables such that each variable can be individually tagged as desired. Other cases are complex constraints comprising ID-variables, e.g. a constraint of the from (`ENTA_ID` = `ENTB_ID`) $\vee$ (`ENTA_ID` = `ENTC_ID`). Specifying such constraints by means of lines would be impractical. Thus, the ID-variables are better addressed explicitly.

## 4.3.2 Formal notation and semantics

A query template is formally notated as $Q(V_r, V_{nr}, C, dom)$ and comprises the following parts: $V_r$ is a set of result variables, whereas $V_{nr}$ is a set of non-result variables. These two sets together denote the set $V = V_r \uplus V_{nr}$ of variables. $C$ is a set of constraints over variables. Total function $dom : V \rightarrow \widehat{U}$ associates to each variable $v \in V$ a universe $U \in \widehat{U}$ serving as domain for $v$. For notational convenience, an association $dom(v) = U$ with $v \in V$ and $U \in \widehat{U}$ is shortly expressed together with the variable as $v : U$.

Generally, each constraint $c \in C$ is a formula of DRC as defined in Section 4.2.2. Note, however, that simple query templates introduced in this chapter are limited as they cannot capture every formula that could be expressed in DRC. A constraint $c \in C$ is rather limited to (a) a single relationship constraint or comparison constraint, (b) a complex constraint formed solely of comparison constraints, or (c) complex constraints formed of other complex constraints of type (b). Complex query templates, which are introduced in Chapter 6.1.2, overcome this limitation.

| Vr : dom(Vr) | Vnr : dom(Vnr) | C |
|---|---|---|
| ENT_ID : E<br>ENT : N | ATT_ID : A<br>ATT : N | ⟨ENT_ID, ENT⟩ ∈ name<br>⟨ATT_ID, ATT⟩ ∈ name<br>⟨ATT_ID, ENT_ID⟩ ∈ definedAt<br>ATT = “title” |

Figure 4.6: Formal representation of query template “EntWithAttribs”

**Example 32** *Fig. 4.6 depicts query template* `EntWithAttribs` *notated formally. Variables* `ENT_ID` *and* `ENT` *depicted in column* $V_r$ *are result variables of domains* $E$ *and* $N$*, respectively. Variables* `ATT_ID` *and* `ATT` *depicted in column* $V_{nr}$ *are non-result variables of domains* $A$ *and* $N$*, respectively. The constraints depicted in column* $C$ *correspond to those explained in example 26 with domain relational calculus.*

*Note that this formal notation captures also those variables and constraints explicitly that are expressed implicitly in the graphical notation of query template* `EntWithAttribs`*. For example, ID-variables that are implicitly represented by graphical shapes are now shown explicitly. Further, a relationship constraint like* $\langle \texttt{ATT}, \texttt{ENT}\rangle \in \textit{definedAt}$*, which is graphically expressed implicitly by the graphical arrangement of these variables, is expressed explicitly, too.*

The meaning of query templates is defined in terms of domain relational calculus. A query template $Q(V_r, V_{nr}, C, dom)$ is interpreted as a DRC-expression of the form

$$\begin{aligned}
\{\ & \langle v_{r1}, \ldots, v_{rx}\rangle \in dom(v_{r1}) \times \ldots \times dom(v_{rx}) \mid \\
& \quad \exists v_{nr1} \in dom(v_{nr1}), \ldots, \exists v_{nry} \in dom(v_{nry}) : \\
& \quad (c_1 \wedge \ldots \wedge c_z) \\
\}
\end{aligned}$$

where $v_{r1}, \ldots, v_{rx}$ and $v_{nr1}, \ldots, v_{nry}$ represent the sets of result and non-result variables $V_r$ and $V_{nr}$, respectively, and $c_1 \ldots c_z$ represent the set $C$ of constraints. We continue to notate query templates formally as $Q(V_r, V_{nr}, C, dom)$; this notation is shorter than the corresponding DRC-expression, which is used only for defining semantics.

**Example 33** *Reconsider query template* `EntWithAttribs` *notated formally as shown in Fig. 4.6. This query template maps to the DRC-expression that is illustrated in Fig. 4.3 and that has been explained in example 26.*

### 4.3.3 Graphical notation and mapping to the formal one

The graphical notation of query templates consists of scheme element skeletons, which contain variables, and of *TBE* -directives. The placement of a variable in a scheme element skeleton implicitly expresses the variable's domain, whereas the graphical arrangement of variables implicitly expresses relationship constraints, which restrain bindings of these variables. The following *TBE* -directives can be used in query templates: (1) A "$\surd$"-symbol preceding a variable declares that this variable is a result variable, while all other variables are non-result variables. (2) Comparison constraints, simple and complex, are explicitly notated in textual form. Further, a comparison constraint relating two ID-variables can also be notated graphically by a gray line between the graphical shapes representing these ID-variables. This line is labelled with the respective comparison operator.

The mapping from the graphical notation of query templates to the formal notation is straightforward, particularly as far as tagged variables and comparison constraints are concerned. Worth mentioning are relationship constraints, which are identified as follows: In a scheme, the graphical arrangement of scheme elements imposes relations between these scheme elements. Analogously, in a query template, the graphical arrangement of scheme element skeletons imposes "relations" between these scheme element skeletons. However, as the entries in scheme element skeletons are variables, these "relations" do not relate concrete scheme elements but variables representing scheme elements. Each such "relation" is then interpreted as a relationship constraint. The following example illustrates this mapping:

**Example 34** *Fig. 4.4(b) depicts query template* `EntWithAttribs` *in graphical notation, which is mapped to the corresponding formal notation shown in Fig. 4.6 as follows: Variables* `ENT_ID` *and* `ENT` *become result variables as they are preceded by symbol $\surd$. In contrast, variables* `ATT_ID` *and* `ATT` *become non-result variables.*

*The graphical arrangement of variables* `ENT_ID` *and* `ENT` *defines a relationship constraint* $\langle \texttt{ENT\_ID}, \texttt{ENT} \rangle \in$ *name. This is analogous* *to schemes where this arrangement would have denoted a tuple* name$\langle$`ENT_ID`, `ENT`$\rangle$. *From the arrangement of variables* `ATT_ID` *and* `ATT`, *relationship constraint* $\langle$`ATT_ID`, `ATT`$\rangle \in$ *name is derived in a similar manner. Further, since the attribute represented by variable* `ATT_ID` *is defined at the entity type represented by variable* `ENT_ID`, *a relationship constraint* $\langle$`ENT_ID`, `ATT_ID`$\rangle \in$ *definedAt is derived. Comparison constraint* `ATT` = "`title`" *is simply taken over.*

## 4.4 Generative templates

### 4.4.1 Informal description

A generative template generically defines how to generate new scheme elements in the context of existing scheme elements that are provided as parameter. All scheme elements are represented by the template's variables, which are consequently distinguished into *parameter variables* and *new-element variables*. The parameter variables represent scheme elements of the scheme to which the template is applied. The new-element variables represent the scheme elements to be generated. All relations that have to be established between new and/or existing scheme elements are expressed implicitly by the graphical arrangement of the respective variables. Thereby, most of a generative template's semantics is expressed implicitly. Only initial values of properties of new scheme elements have to be defined explicitly by expressions attached to new-element variables.

**Example 35** *Fig. 4.7(a) depicts generative template* `PCWithIndex` *that generates a page class with an index unit for a given entity type. The page class, the index unit, and the entity type are represented by variables* `PC`, `IU`, *and* `ENT`, *respectively, as expressed by the graphical placement of these variables. Variable* `ENT` *is a parameter variable (denoted by symbol* $\surd$*), while variables* `PC` *and* `IU` *are new-element variables (denoted by symbol* $\oplus$*).*

*The containment relation, which is to be established between the new index unit and the new page class, is expressed implicitly by the graphical arrangement of the corresponding variables. Further, the content source of the new index unit is the entity type that is provided as parameter. This is again expressed implicitly by the placement of parameter variable* `ENT`. *The names for the new page class and the index unit are*

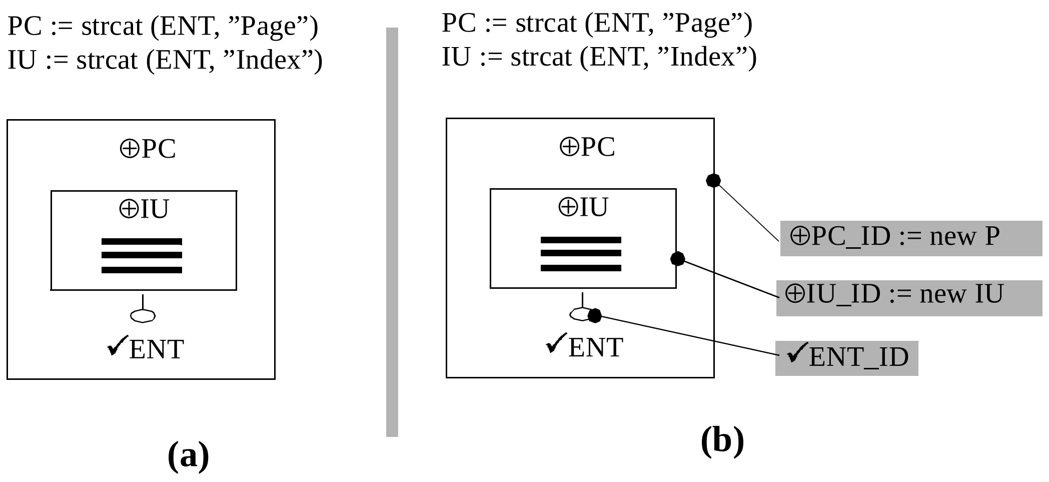

PC := strcat (ENT, ”Page”)
IU := strcat (ENT, ”Index”)
⊕PC
⊕IU
✓ENT
(a)
PC := strcat (ENT, ”Page”)
IU := strcat (ENT, ”Index”)
⊕PC
⊕IU
✓ENT
⊕PC_ID := new P
⊕IU_ID := new IU
✓ENT_ID
(b)

Figure 4.7: Generative template `PCWithIndex`: (a) ID variables defined implicitly; (b) ID-variables visualized.

*defined by attaching literals* `"Page"` *and* `"Index"` *to the entity type's name, respectively, as expressed by the respective* `strcat`*-expressions denoting string concatenation.*

*Suppose that this generative template is applied having entity type* `Paper` *bound to parameter variable* `ENT`*. Then, a new page class* `PaperPage` *comprising an index unit* `PaperIndex` *that has entity type* `Paper` *as content source is generated.*

In the above explanation of generative template `PCWithIndex` we have abstracted from IDs of scheme elements. However, generative templates, which generate new scheme elements, have to provide for generating IDs for these elements, too. Further, generative templates may require IDs of scheme elements to be provided as parameter. For example, to define the content source of a new index unit, the ID of the respective entity type is required.

Thus, ID-variables are introduced, which are again defined implicitly. If such an ID-variable is declared as new-element variable, then it must have a construction expression attached that derives a new ID. If an ID-variable is declared as parameter variable, then the ID of an existing scheme element must be provided as parameter.

**Example 36** *Fig. 4.7(b) depicts generative template* `PCWithIndex` *having the implicitly defined ID-variables visualized. ID-variables* `PC_ID` *and*

`IU_ID` *are new-element variables, which represent the IDs of a new page class and a new index unit, respectively. The attached construction expressions define that such IDs are to be derived. ID-variable* `ENT_ID` *is a parameter variable. It represents the ID of an existing entity type that shall serve as content source for the new index unit.*

In order to abstract from ID-variables, we again define a default coupling between variables representing names of a scheme elements and ID-variables representing the respective scheme element IDs. This is analogous to query templates. Thus, if a variable representing the name of a scheme element is declared as new-element variable, then the corresponding ID-variable becomes a new-element variable, too. Further, the ID-variable automatically gets a construction expression attached that derives a new ID value. However, if a variable representing the name of a scheme element is declared as parameter variable, then the corresponding ID-variable becomes a parameter variable, too.

**Example 37** *Reconsider generative template* `PCWithIndex` *depicted in Fig. 4.7(a). New-element variable* `PC` *represents the name of a page class, whereas the page class itself is represented by an ID-variable that is defined implicitly. Since variable* `PC` *is a new-element variable, the ID-variable becomes a new-element variable, too. Similarly, new-element variable* `IU`*, which represents the name of an index unit to be generated, defines that the corresponding ID-variable becomes a new-element variable, too. However, variable* `ENT`*, which represents the name of the content source of index unit* `IU`*, is a parameter variable. Consequently, the corresponding ID-variable becomes a parameter variable, too. Thus, the generative template shown in Fig. 4.7(a), which hides ID-variables, has exactly the same meaning as that depicted in Fig. 4.7(b), where ID-variables are visualized.*

### 4.4.2 Formal notation and semantics

A generative template is formally notated as $G(V_p, V_g, R, dom)$, where $V_p$ is a set of parameter variables, $V_g$ is a set of new-element variables, $R$ is a set of relation constructors defining to construct relations between members of $V_p$ and $V_g$, and $dom$ is a total function $dom : V \rightarrow \widehat{U}$ associating to each variable $v \in V$ a universe $U \in \widehat{U}$ serving as domain for $v$. Again, for notational convenience, an association $dom(v) = U$ with $v \in (V_p \cup V_g)$ and $U \in \widehat{U}$ is shortly expressed together with the variable as $v : U$. These parts form the generative part of TBE-QML and are defined as follows:

1. Each parameter variable $v_p \in V_p$ represents a given scheme element of a universe $U \in \widehat{U}$ and is notated as $v_p : U$.

2. Each new-element variable $v_g \in V_g$ represents a scheme element (of a universe $U \in \widehat{U}$) to be generated and has, for that purpose, a construction expression *exp* attached. This is notated as $v_g : U := exp$. A construction expression has one of the following forms:

   - ":= *new* $U$" assigns a new ID out of universe $U$
   - ":= *lit*" assigns a literal value
   - ":= $v_p$" assigns the value of a parameter variable $v_p \in V_p$
   - ":= `func`$(x_1, \ldots, x_n)$" assigns the result of a function call with, for $i = 1 \ldots n$, $x_i \in (V_p \cup V_g \cup lit)$. One such function provided is, for example, $strcat(N, N) : N$ for concatenating two names

3. Each relation constructor $r \in R$ specifies the creation of a new relation between members of universes. It is denoted as $+R\langle x_1, \ldots, x_n\rangle$ with $R \in \widehat{R}$ and, for $i = 1 \ldots n$, $x_i \in (V_p \cup V_g)$.

| Vp | Vg := exp | R |
|---|---|---|
| ENT_ID : E<br>ENT : N | PC_ID : P = new P<br>IU_ID : IU = new IU<br>PC : N = strcat (ENT, "Page")<br>IU : N = strcat (ENT, "Index") | +name ⟨PC_ID, PC⟩<br>+name ⟨IU_ID, IU⟩<br>+containedIn ⟨IU_ID, PC_ID⟩<br>+contentSource ⟨IU_ID, ENT_ID⟩ |

Figure 4.8: Generative template `PCWithIndex` as TBE-QML manipulation statement

**Example 38** *Fig. 4.8 illustrates generative template* `PCWithIndex` *notated formally as a manipulation statement in TBE-QML. Parameters, new-element variables with construction expressions, and relation constructors are depicted in separate columns. Parameters and new-element variables have already been explained.*

*Relation constructor* $+\text{name}\langle\texttt{PC\_ID},\texttt{PC}\rangle$ *associates the newly generated name, which is represented by variable* `PC`*, to the newly generated page class ID, which is represented by variable* `PC_ID`*. Similarly, relation constructor* $+\text{name}\langle\texttt{IU\_ID},\texttt{IU}\rangle$ *associates the index unit's name to its ID. Relation constructor* $+\text{containedIn}\langle\texttt{IU\_ID},\texttt{PC\_ID}\rangle$ *establishes the containment relation between the new index unit and the new page class via their respective IDs. Relation constructor* $+\text{contentSource}\langle\texttt{IU\_ID},\texttt{ENT\_ID}\rangle$ *defines that the entity type provided as parameter shall serve as content source of the new index unit.*

A generative template $G$ is instantiated within a scheme $S$ by binding each parameter variable of $G$ to an equally sorted scheme element of $S$. Such an instantiation is denoted as $G[B]$, where $B$ is a set of parameter variable bindings. A binding of a parameter variable $v_p$ to a scheme element $e$ of $S$ is denoted as "$v_p = e$".

When an instantiation $G[B]$ is processed in the context of a scheme $S$, new scheme elements and new relations are generated as follows: (1) for each new-element variable $v_g : U$, its attached construction expression *exp* is evaluated and the result becomes a member of universe $U$; (2) for each relation constructor $+R\langle x_1,\ldots,x_n\rangle$, a relation of sort $R$ is established between the scheme elements represented by parameter variables/new-element variables $x_1,\ldots,x_n$.

**Example 39** *Let $G$ be generative template* `PCWithIndex` *and let $S$ be the content scheme of our running example. Then, an instantiation* $G[\{\texttt{ENT\_ID} = \texttt{e1}, \texttt{ENT} = \texttt{"Paper"}\}]$ *generates a page class* `PaperPage` *with an embedded index unit* `PaperIndex` *as shown graphically in the lower part of Fig. 4.2(a).*

*The lower part of Fig. 4.2(b) illustrates the corresponding relational representation of the resulting scheme: The results of evaluating the construction expressions (e.g.* $\texttt{pg1} : P$ *generated from* $\texttt{PG_ID} = $ *new $P$, and* $\texttt{PaperPage} : N$ *generated from* `PG := strcat(ENT, "Page")`*) have become members of the corresponding universes. Also, tuples generated by the relation constructors have been inserted into the respective relations, e.g. tuple* $\text{containedIn}\langle\texttt{iu1},\texttt{pg1}\rangle$ *generated from* $+\text{containedIn}\langle\texttt{IU\_ID},\texttt{PG\_ID}\rangle$ *and tuple* $\text{contentSource}\langle\texttt{iu1},\texttt{e1}\rangle$ *generated from* $+\text{contentSource}\langle\texttt{IU\_ID},\texttt{ENT\_ID}\rangle$*.*

### 4.4.3 Graphical notation and mapping to the formal one

Similar to query templates, the graphical notation of generative templates consists of scheme element skeletons, which contain variables, and of *TBE*-directives. The placement of a variable in a scheme element skeleton implicitly expresses the variable's domain, whereas the graphical arrangement of variables implicitly expresses relation constructors, which define to generate relations between the elements represented by these variables. The following *TBE*-directives can be used in generative templates: (1) A tick "$\surd$" preceding a variable declares a parameter variable, whereas symbol "$\oplus$" declares a new-element variable. (2) Construction expressions are annotated anywhere in textual form.

The mapping to the formal notation is mostly straightforward. Depending on whether a variable is preceded by symbol "$\surd$" or by symbol "$\oplus$", the variable becomes a parameter variable or a new-element variable, respectively. Construction expressions are simply taken over. A "relation" of variables, which is derived from the graphical arrangement of scheme element skeletons and the variables contained therein, becomes a relation constructor.

**Example 40** *Fig. 4.7(b) depicts generative template* `PCWithIndex` *in a graphical representation, which is mapped to the corresponding formal notation shown in Fig. 4.8. Variables* `PG_ID`,`PG`, `IU_ID`, *and* `IU` *become new-element variables as they are preceded by symbol "$\oplus$". In contrast, variables* `ENT_ID` *and* `ENT`, *which are preceded by symbol "$\surd$", become parameter variables. The construction expressions are taken over.*

*The graphical arrangement of variables* `PG_ID` *and* `PG` *defines a relation constructor* $+\text{name}\langle \texttt{PG\_ID}, \texttt{PG}\rangle$, *whereas the arrangement of variables* `IU_ID` *and* `IU` *denotes a relation constructor* $+\text{name}\langle \texttt{IU\_ID}, \texttt{IU}\rangle$. *Further, since the index unit represented by variable* `IU_ID` *is contained in the page class represented by variable* `PG_ID`, *a relation constructor* $+\text{containedIn}\langle \texttt{IU\_ID}, \texttt{PG\_ID}\rangle$ *is derived.*

## 4.5 Combining query templates and generative templates to transformers

A transformer $T(Q, G)$ is a combination of a query template $Q$ and a generative template $G$, where the result variables of $Q$ serve as parameters for

$G$. A combination of $Q$ and $G$ is proper, if (1) for each result variable of $Q$ there exists an equally named and equally sorted parameter variable of $G$, and (2) all parameters of $G$ are provided by $Q$.

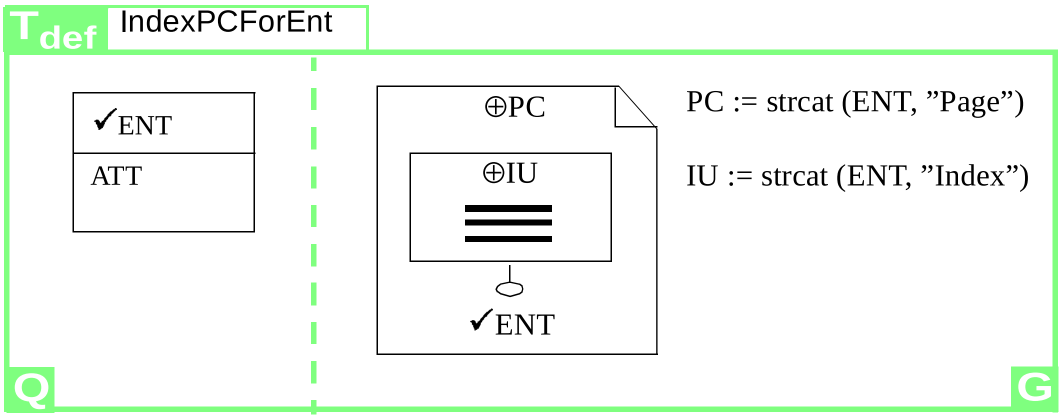


Figure 4.9: Definition of transformer “IndexPCforEnt”

A transformer is notated as a rectangular box that is topped with label “$T_{def}$” followed by the transformer's name. The rectangular box comprises two sections, one for the query template and one for the generative template. The two sections are labelled “$Q$” and “$G$” according to their respective purpose and are separated by a dashed line. The form or the arrangement of these sections is irrelevant.

**Example 41** *Fig. 4.9 illustrates the definition of transformer* `IndexPCforEnt` *with a query template (Q) in the left part and a generative template (G) in the right part. The templates themselves have already been explained in Sections 4.3 and 4.4, respectively.*

*The transformer combines Q and G properly because result variable* `ENT` *of Q matches parameter variable* `ENT` *of G and both variables represent entity types. Note, however, that query template Q implicitly defines a result ID-variable* `ENT_ID`*, and that generative template G implicitly defines a parameter ID-variable* `ENT_ID`*. Obviously, these ID-variables match properly, too.*

Combining a query template and a generative template to a transformer expresses that the query template provides the parameters for instantiating the generative template. Since the query template achieves a *result relation* that may contain several tuples, the following three ways of how the generative template is instantiated can be distinguished:

1. The whole generative template is instantiated once for each tuple of the query template's result, i.e. for each such tuple, all construction expressions attached to new-element variables are evaluated.
2. The whole generative template is instantiated at most once per transformer application, i.e. once if the query template's result relation contains at least one tuple, or not at all if the result relation is empty.
3. As a mixture of ways 1 and 2, some of the generative template's new-element variables are instantiated once per tuple, whereas others are instantiated at most once in the context of a transformer application.

A transformer that accords to case 1 is referred to as *not having an application context*, whereas one according to cases 2 or 3 is referred to as *having an application context*. Case 1 is the standard case and is described first. Case 2 is a special form of case 3 and is rather seldom. It is therefore explained together with case 3.

## 4.5.1 Transformers without application context

In a transformer without application context, the generative template is instantiated separately for each tuple of the query template's result. Thus, when applying a transformer $T(Q, G)$ to a scheme, query template $Q$ is evaluated first. It achieves a result relation $Rel_Q(v_{r1}, \dots, v_{rn})$, where $v_{r1}, \dots, v_{rn}$ is the set of $Q$'s result variables. Then, generative template $G$ is iteratively instantiated for each tuple $t \in Rel_Q$, each time having parameters of $G$ bound to the corresponding attribute values of current tuple $t$.

**Example 42** *Suppose that transformer* `IndexPCforEnt` *is applied to the content scheme of our running example. Then, query template $Q$ achieves a relation $Rel_Q(ENT, ENT\_ID)$ containing tuples $\langle e1, Paper\rangle$, $\langle e2, Author\rangle$, and $\langle e3, Conf\rangle$. For each of these three tuples, generative template $G$ is instantiated separately. Thus, the result of the transformer application is a hypertext scheme with page classes* `PaperPage`, `AuthorPage`, *and* `ConfPage`, *each of them comprising a corresponding index unit. The resulting scheme is shown graphically in Fig. 4.10.*

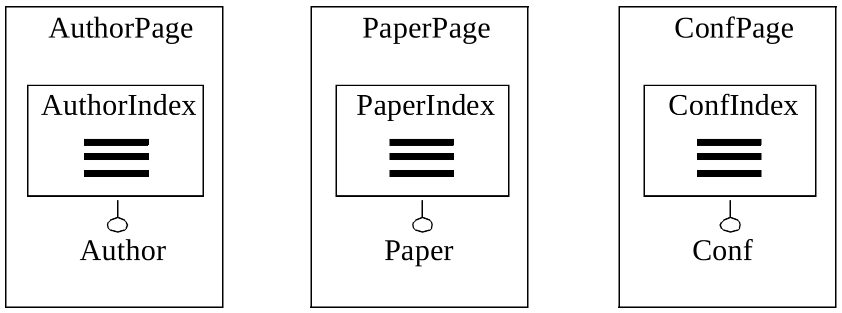


Figure 4.10: Result of applying transformer `IndexPCForEnt` to the content scheme of the running example.

### 4.5.2 Transformers with application context

Many scheme elements are containers in the sense that they may embed several other scheme elements. Hence, it may be required to generate a container scheme element only once whereas to generate the scheme elements embedded therein iteratively. The following two examples illustrate such cases:

1. Page classes may comprise several index units. Suppose that index units shall be generated for each entity type of a scheme and that these index units are to be embedded in the same page class, which is to be generated, too. This situation requires to generate index units iteratively, whereas the page class is to be generated only once per transformer application.

2. Page classes are often grouped within an area. Thus, if a list of new page classes shall be embedded in the same new area, it is required to generate the page classes iteratively, whereas the area is to be generated only once per transformer application.

In this subsection, we introduce transformers with application context as a straightforward concept that is sufficient for expressing above mentioned situations. This concept is rather simple but has some limitations which are overcome in Section 6.3 by providing the more complex concept of nested relations. We start with introducing transformers with application context and then explain the limitations.

## Concept

In a transformer with application context, some of the generative template's new-element variables are *pinned*, i.e. marked such that they are instantiated only once per transformer application rather than separately for each tuple of the query template's result. The value of such a new-element variable is then constant in the context of the whole application.

A pinned new-element variable is notated graphically by a "$"-symbol preceding the variable's name. Again, a default coupling between names and IDs is defined such that when a variable representing a scheme element's name is pinned, the corresponding ID-variable is by default pinned, too. Additionally, to further emphasize that an implicitly defined ID-variable is pinned, a pin-symbol is placed at the graphical shape representing this ID-variable.

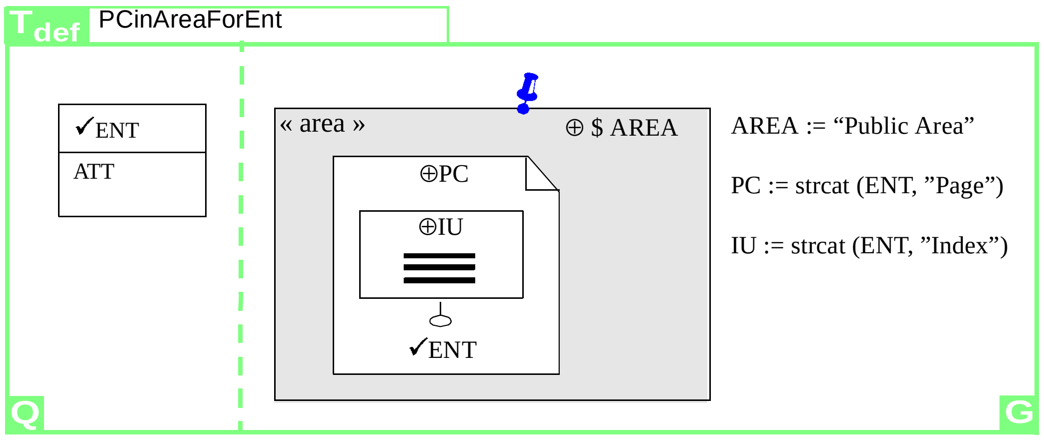


Figure 4.11: Transformer `PCinAreaForEnt`, generating page classes iteratively within an area generated once.

**Example 43** *Fig. 4.11 depicts the definition of transformer* `PCinAreaForEnt`*, which generates page classes within an area that is generated, too. The page classes are generated iteratively, i.e. one for each entity type retrieved by the query template. The area scheme element is generated only once for each application of this transformer. This is expressed by pinning the new-element variables representing the area, i.e. variable* `AREA` *and ID-variable* `AREA_ID`*, which is implicitly represented by the area's graphical shape. That these variables are pinned is expressed by the $-symbol preceding variable*

`AREA` *and by the pin-symbol placed at the graphical shape of the area scheme element.*

**Formal notation and semantics**

Formally, a transformer with application context is notated as $T(Q, G, \mathit{Pinned})$, where $\mathit{Pinned} \subseteq V_g$ denotes a non-empty set of new-element variables that are to be pinned. Note that from a technical point of view, it is admissible to pin every new-element variable of the generative template. In this special case, the whole generative template would be instantiated at most once. But as already mentioned at the beginning of this section, this case is rather seldom.

The meaning of a transformer $T(Q, G, \mathit{Pinned})$ with application context is defined as follows: When applied to a scheme $S$, query template $Q$ is evaluated first and achieves a result relation $\mathit{Rel}_Q$. If this result relation is not empty, for each pinned new-element variable $v_{pin} \in \mathit{Pinned}$, its attached construction expression is evaluated. Such a construction expression must unambiguously derive a constant value; otherwise the application is invalid and an error is risen. Afterwards, the generative template $G$ is iteratively instantiated for each tuple $t \in \mathit{Rel}_Q$ as usual. However, the expressions attached to the pinned new-element variables are not re-evaluated; these variables rather retain their value already assigned.

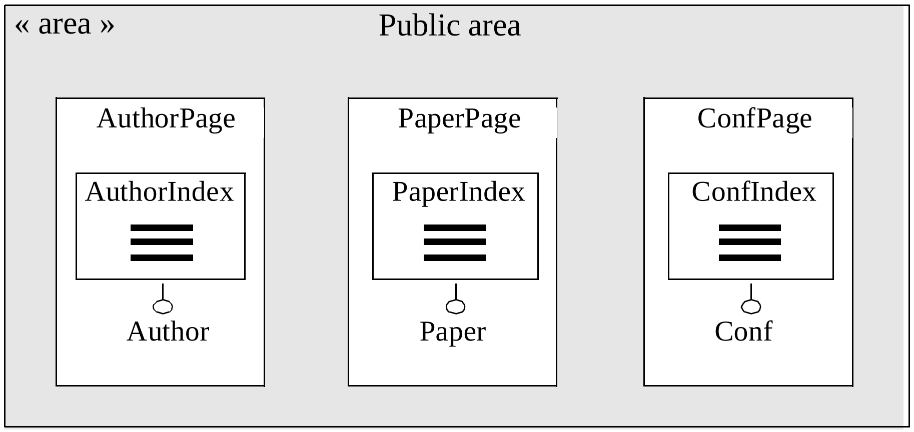


Figure 4.12: Result of applying transformer `PCinAreaForEnt` to the content scheme of the running example.

**Example 44** *Reconsider transformer* `PCInAreaForEnt` *as illustrated in Fig. 4.11. This transformer is formally notated as* `PCInAreaForEnt` $(Q, G, \{\texttt{AREA}, \texttt{AREA\_ID}\})$*. When applied to the content scheme of our running example, query template* $Q$ *achieves a relation* $Rel_Q(ENT, ENT\_ID)$ *containing tuples* $\langle e1, Paper\rangle$, $\langle e2, Author\rangle$, *and* $\langle e3, Conf\rangle$*. Since* $Rel_Q$ *is not empty, the generative template is instantiated as follows:*

*The construction expressions attached to pinned new-element variables are evaluated, i.e.* `AREA_ID := new Area` *and* `AREA := "Public area"`*. This achieves a new area scheme element with ID* `area1` *and a new name* `"Public area"`*, which are assigned to variables* `AREA_ID` *and* `AREA`*, respectively. Then, for each tuple of* $Rel_Q$*, the generative template* $G$ *is instantiated separately, yet the values of the pinned variables remain constant.*

*Thus, the result of this application is a single area* `Public Area` *that has three page classes with index units embedded, i.e.* `PaperPage`*,* `AuthorPage`*, and* `ConfPage`*, each of them comprising a corresponding index unit. The resulting scheme is shown graphically in Fig. 4.12. Note that if the query template had not retrieved any tuple, then no page classes and no area scheme element would have been generated.*

Each construction expression attached to a pinned new-element variable must unambiguously derive a constant value. This may be problematic if such a construction expression depends on a query template variable because a query template may return more than one tuple. Consequently, a reference to a query template variable is ambiguous if its value is not the same over all these tuples. Such an application is then considered as invalid.

**Example 45** *Suppose that transformer* `PCinAreaForEnt` *attaches a construction expression of the form* `AREA := strcat (ENT, "Area")` *to pinned new-element variable* `AREA`*. Then an application of this transformer to the content scheme of our running example would be invalid. The query template returns three tuples with three different values for result variable* `ENT`*, i.e.* `"Author"`*,* `"Paper"`*, and* `"Conf"`*. Hence, it is not determined which value shall be considered for evaluating expression* `AREA := strcat (ENT, "Area")`*. However, if the query template would achieve a relation where the value of* `ENT` *were the same for all the tuples of this relation, then the application would be valid.*

### Limitations

The advantage of the concept of transformers with application context is that it is fairly simple. Yet its practicability is limited to those cases where some of the new-element variables are to be instantiated only *once* per application. Thus, one can express that an entry unit is generated once whereas its entry fields are generated iteratively.

Suppose, however, that (1) a separate entry unit is to be generated for each entity type of a scheme, and that (2) for each attribute of such an entity type, an entry field is to be generated in the corresponding entry unit. This situation cannot be expressed with transformers as explained so far: If the new-element variable representing the entry unit were pinned, then a single entry unit would be generated that embeds entry fields for each attribute contained in any entity type. If the new-element variable representing the entry unit were *not* pinned, then a separate entry unit would be generated for each attribute of an entity type.

To overcome this limitation, it must be possible to generate scheme elements in nested loops. For example, in the outer loop, an entry unit for each entity type is generated, whereas in an inner loop, an entry field is generated for each of the attributes of the entity type currently processed by the outer loop. For this purpose, Section 6.3 introduces the concept of nested relations.

# Chapter 5

# Applying transformers

## Contents

A transformer definition is distinguished from applications of this transformer within schemes. Whereas a transformer definition is an entity of its own, a transformer application is specified within the scheme to which the transformer shall be applied. Processing the transformer application leads to an extended or refined scheme as defined by the transformer's query template and generative template.

There are two general ways of how a transformer can be applied: (1) We may take the transformer exactly as it is, i.e. without extending or changing the set of constraints and the set of construction expressions as specified in the transformer definition. This will be referred to as an *off-the-shelf application*. (2) We may individualize the transformer by adapting its behavior specifically for a particular application, i.e. without the need of changing the transformer definition per se or the need of defining a new transformer. This will be referred to as an *individualized application*, and the following forms of adaptations are provided:

1. Application-specific constraints can be added to those constraints defined already in the transformer's query template. This allows to constrain the transformer to consider only a particular portion of a scheme. Further, application-specific expressions may override expressions as defined by the transformer's generative template. This allows to individually adapt the transformer's outcome.

2. Each new-element variable of the transformer's generative template can be individually pinned. This variable is then instantiated only once for the application rather than once for each tuple of the query template's result.

3. Each new-element variable of the transformer's generative template can be adapted such that it does not generate a new scheme element but requires an existing scheme element to be assigned instead. Consequently, the transformer extends or refines an existing scheme element instead of one that otherwise would be generated, too.

In Section 5.1, the syntax of transformer applications is introduced. Section 5.2 covers off-the-shelf applications, whereas Section 5.3 presents the various forms of individualized transformer applications.

## 5.1 Specifying transformer applications

Transformer applications are specified within the scheme to which the transformer is to be applied. Specifying an off-the-shelf application is fairly simple because one has to declare only which transformer shall be applied. For individualized transformer applications, however, application specific adaptations must be expressed as well.

Each transformer application can be specified textually. In an off-the-shelf application, it is sufficient to refer to the transformer's name. For individualized applications, the respective adaptations are specified textually, too.

The concrete syntax will be introduced together with the various forms of individualization in Section 5.3.

Additionally, each transformer application can be expressed graphically by means of an application symbol which represents the transformer and which basically looks like the transformer definition but is much smaller in size. The purpose of a graphical application is twofold: (1) the symbol schematically reflects the transformer definition and gives an impression of the transformer's purpose; (2) the (small) scheme element skeletons represent ID-variables, which can therefore be addressed through a graphical shape rather than through referring to the name of the ID-variable.

A graphical application of a transformer is notated as a rectangular box that is topped with label "$T_{app}$" followed by the name of the transformer that it represents. The transformer definition or parts thereof is repeated in this box in a schematic manner, and many details may be not shown for reasons of conciseness. Note, however, that the necessity of neglecting details does not influence the transformer's semantics: If parts of the transformer definition are not presented in the application symbol, then the only effect is that one has to refer to these parts textually.

**Example 46** *The right part of Fig. 5.1 depicts a graphical application of transformer* `IndexPCforEnt` *as denoted by label "*$\mathtt{T}_{\mathtt{app}}$*" preceding the transformer's name. The transformer definition is repeated schematically by means of a small entity type skeleton, a small page class skeleton, and a small index unit skeleton. Transformer variables* `ATT` *and* `IU` *are not presented. Nevertheless, one could refer to these variables textually as they are part of the transformer definition.*

## 5.2 Applying transformers off-the-shelf

The simplest way of applying a transformer to a scheme is to take the transformer as it is, i.e. without specifying any application-specific adaptations. The meaning of such an off-the-shelf application has been explained in Section 4.5.

Formally, an off-the-shelf application of a transformer $T$ to a scheme $S$ is notated as $S.\mathtt{apply}(T)$, where $T$ stands for the name of the transformer. Graphically, such an application is notated simply by placing a graphical application-symbol representing $T$ within the respective scheme $S$.

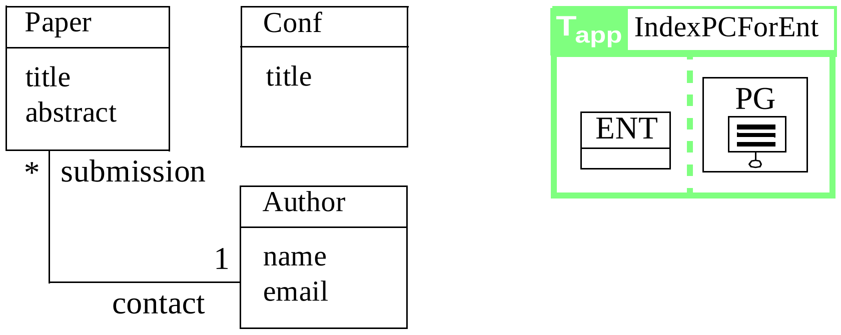


Figure 5.1: Off-the-shelf application of transformer `IndexPCForEnt`

**Example 47** *Fig. 5.1 depicts an off-the-shelf application of transformer* `IndexPCforEnt` *to the content scheme of our running example by means of a graphical application-symbol. Formally, this application would be notated as* $S$.`apply`(`IndexPCforEnt`)*, where* $S$ *is assumed to represent the content scheme. The result of this application is depicted in Fig. 4.10 on page 89.*

*Note that a page class with an index unit is also generated for entity type* `Conf`*. However, an index unit, which is for displaying a list of entities, is here not appropriate since entity type* `Conf` *is meant to contain only one member. Hence, it is impractical to apply transformer* `IndexPCForEnt` *off-the-shelf because page class* `ConfPage` *would have to be reworked afterwards. To overcome this problem, we provide a notion for adapting a transformer's behavior for individual applications.*

## 5.3 Individualized transformer applications

In this section, we define various ways of how the behavior of existing transformers can be adapted individually for particular applications and without the need of changing the transformer's definition. The purpose of individualization is to avoid a large number of transformer definitions that would support the same modelling task yet in slight variations. Thus, individualization contributes to the flexibility and adaptability of transformers, and this section introduces concepts that are particularly useful for that purpose.

Section 5.3.1 introduces application-specific constraints and construction expressions. Section 5.3.2 explains how new-element variables can be pinned individually for particular transformer applications. Section 5.3.3 introduces the concept of turning new-element variables into parameter variables. Section 5.3.4 gives a summary of the various forms of individualization.

## 5.3.1 Application-specific constraints and expressions

Applications of transformers within schemes may be individualized by application-specific constraints and expressions. With each individual application, we may further constrain query template variables and override expressions attached to new-element variables in the generative template. These constraints and expressions are then treated as if they had already been defined in the query template and the generative template of the transformer to apply, respectively.

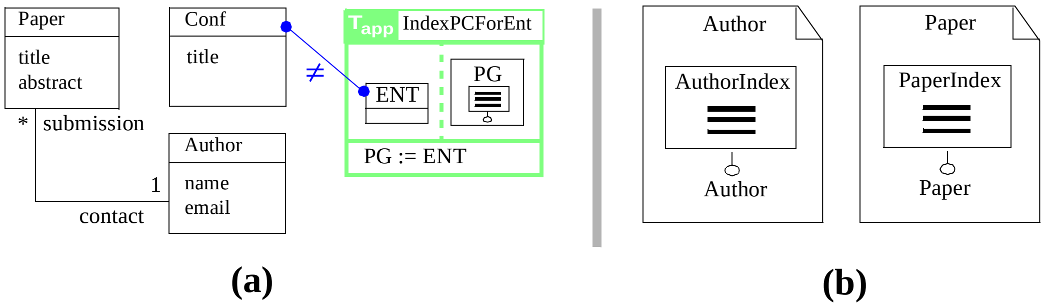


Figure 5.2: (a) Individualized application of transformer `IndexPCForEnt`; (b) result of (a).

**Example 48** *Fig. 5.2(a) depicts an individualized application of transformer* `IndexPCForEnt`*. The line labelled "≠" denotes an application-specific constraint expressing that entity type* `Conf` *shall not be considered. Further, only for this application, the naming policy defined by the transformer is overridden such that the generated page classes are now named directly after the corresponding entity types. The new naming policy is defined by application-specific expression* `PG` := `ENT`*. The result of this application is shown in Fig. 5.2(b).*

Formally, an individualized application of a transformer $T(Q, G)$ to a scheme $S$ is denoted as $S.\texttt{apply}(T, ASC, ASE)$, where $ASC$ is a set of application specific comparison constraints and $ASE$ is a set of application specific construction expressions. The meaning of members of $ASC$ and $ASE$ is similar to that of comparison constraints and expressions, respectively, as they are known from the definition of transformers.

**Example 49** *Let $T$ be transformer* `IndexPCForEnt` *and let $S$ be the content scheme of our running example. Then, the individualized application explained in example 48 is notated as* $S.\texttt{apply}(T, \{\texttt{ENT\_ID} \neq \texttt{e3}\}, \{\texttt{PG} := \texttt{ENT}\})$. *Query template variable* `ENT_ID` *represents IDs of entity types to be selected and ID* `e3` *is assumed to be the ID of entity type* `Conf`. *Application-specific expression* `PG := ENT` *overrides expression* `PG := strcat(ENT, "Page")` *as defined in the generative template of transformer $T$.*

An individualized application of a transformer is notated graphically through extending the application-symbol by an *individualization section.* In this section, application-specific constraints and expressions are specified textually, where the notation is the same as used for specifying comparison constraints and expressions within template definitions. Consequently, the mapping from the textual notation to the formal one is the same as defined for templates. Further, application-specific constraints that compare an ID-variable with the ID of a scheme element can also be notated graphically by a line that is labelled with the respective comparison operator. This notation is analogous to that for comparing ID-variables in query template definitions.

**Example 50** *Fig. 5.2(a) depicts the graphical representation of the transformer application* $T[S, \{\texttt{ENT\_ID} \neq \texttt{e3}\}, \{\texttt{PG} := \texttt{ENT}\}]$ *explained in example 49. Application-specific expression* `PG := ENT` *is notated textually. Application-specific constraint* $\texttt{ENT\_ID} \neq \texttt{e3}$ *is expressed graphically by the line labelled "$\neq$" connecting entity type* `Conf` *to entity type skeleton* `ENT`. *The handles attached to these elements actually refer to the implicitly defined id-value* `e3` *and the implicitly defined id-variable* `ENT_ID`, *respectively.*

We further provide a graphical notation for complex constraints of the form "$(\texttt{V} = \texttt{e}_1)$ OR ... OR $(\texttt{V} = \texttt{e}_\texttt{n})$", where `V` is an ID-variable and $e_1, \ldots, e_n$ are IDs of scheme elements. Such a constraint is very common since a transformer application often shall be individualized such that a fixed set of scheme elements $e_1, \ldots, e_n$ is considered while other scheme elements are not. This complex constraint can be expressed more conveniently as a membership constraint of the form $\texttt{V} \in \{e_1, \ldots, e_n\}$. Graphically, the set of scheme elements to be considered is notated by a shape with symbol $\cup$, where each set member $e_1, \ldots, e_n$ is connected to this shape by a line. The membership constraint is expressed by attaching the shape to a variable by a line labelled "$\in$". The following example demonstrates the advantage of the notation using a membership constraint over that using a complex constraint.

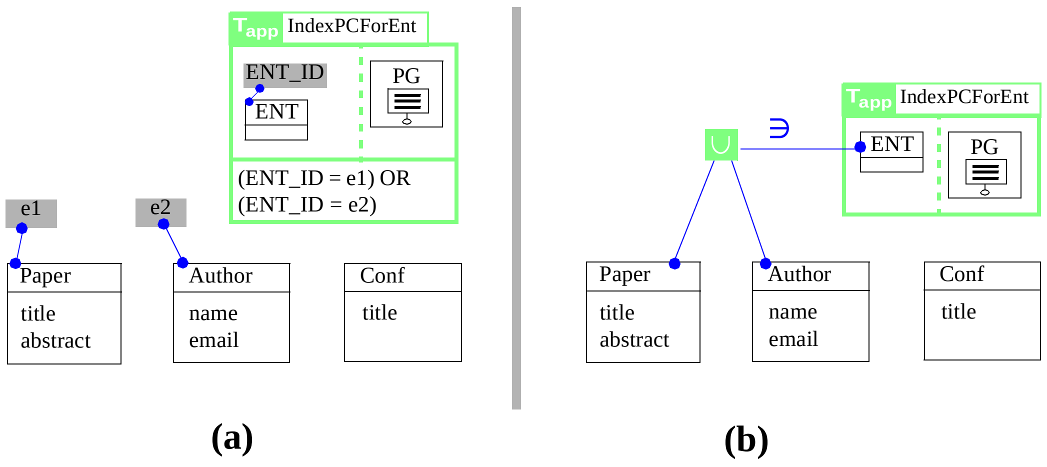


Figure 5.3: Individualized transformer application using a complex constraint (a) and a membership constraint (b)

**Example 51** *Fig. 5.3 illustrates two variants of an individualized application of transformer* `IndexPCForEnt`*, where an application-specific constraint expresses that only entity types* `Paper` *and* `Author` *are considered. Variant (a) makes use of a complex constraint of the form "*$\texttt{(ENT\_ID} = \texttt{e1)}$ `OR` $\texttt{(ENT\_ID} = \texttt{e2)}$*" that refers to the query template's ID-variable* `ENT_ID` *and to the IDs of entity types* `Paper` *and* `Author`*, i.e.* `e1` *and* `e2`*, respectively. The ID-variable and the IDs are visualized for that purpose.*

*Variant (b) makes use of a membership constraint that has the form "*$\texttt{ENT\_ID} \in \{\texttt{e1}, \texttt{e2}\}$*" and is notated graphically as follows: The shape with symbol $\cup$ together with the two lines to entity types* `Author` *and* `Paper` *denotes the set* $\{\texttt{e1}, \texttt{e2}\}$ *of entity types. The membership constraint itself is expressed by the line labelled "$\in$" that connects the $\cup$-shape to the entity type skeleton of the query template. Obviously, both variants express the same application-specific constraint. However, we* *feel that variant (b) is more intuitive because it abstracts from IDs and ID-variables.*

### 5.3.2 Pinning new-element variables individually

Pinning a new-element variable expresses that the construction expression attached to this variable is evaluated at most once for each transformer application. This concept has been introduced in Section 4.5.2 at the level of a transformer definition, i.e. in a static manner.

However, it may be the case that the same transformer shall be used for situations where a particular new-element variable is to be instantiated once per application, whereas in other situations the same variable is to be instantiated once per tuple of the query template result.

**Example 52** *Suppose that transformer* `IndexPCForEnt` *shall be applied such that new index units for entity types* `Paper` *and* `Author` *shall be embedded in the same page class* `Overview`*, which shall therefore be generated only once. This could be expressed by re-defining transformer* `IndexPCForEnt` *such that new-element variable* `PC`*, which represents the page class to be generated, is pinned.*

*However, there may also be cases where each new index unit shall actually be embedded in a separate page class. This would require that new-element variable* `PC` *is* not *pinned. Consequently, one had to define two transformers that only differ in that one transformer has new-element variable* `PC` *pinned and the other one has not.*

To overcome this problem, each new-element variable of an already defined transformer can be pinned individually at transformer application. This improves the flexibility of predefined transformers and avoids the need of defining similar transformers that only differ in the set of pinned new-element variables.

Formally, a transformer application that is individualized by pinning a set of new-element variables is notated as $S.\texttt{apply}(T, \mathit{IndivPinned})$, where $S$ is the scheme to which the transformer is applied and $\mathit{IndivPinned}$ is a subset of the transformer's new-element variables, i.e. $\mathit{IndivPinned} \subseteq V_g$. Of course, besides pinning new-element variables, the application may additionally define application-specific constraints and expressions.

In a graphical application, pinning a new-element variable is simply expressed by preceding the respective variable with a \$-symbol. Further, if an ID-variable shall be pinned, a pin-symbol is attached to the graphical shape representing this variable. In order to distinguish pins and \$-symbols that are already part of the transformer definition from those that are specified at transformer application, the latter are shown inverse.

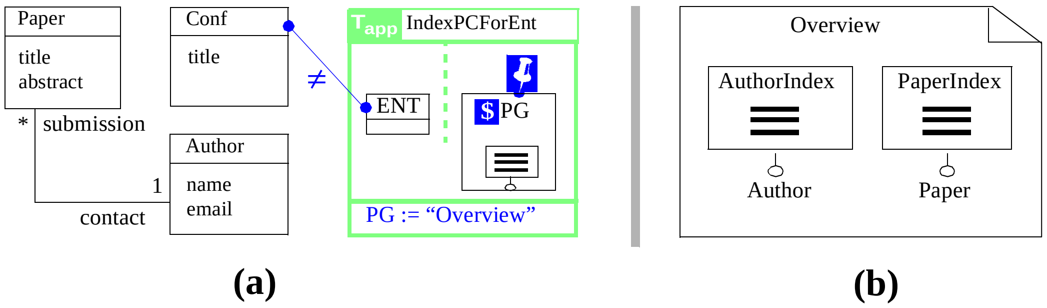


Figure 5.4: (a) Application of transformer `IndexPCForEnt`, generating the page class only once. (b) result of (a).

**Example 53** *Fig. 5.4(a) depicts an application of transformer* `IndexPCForEnt` *that is individualized according to the requirements specified in example 52 above. Application-specific constraint* `ENT_ID` $\neq$ `e3` *excludes entity type* `Conf` *from consideration. New-element variables* `PG` *and* `PG_ID` *are pinned as expressed by the "\$"-symbol preceding variable* `PG` *and by the pin that is placed at the graphical shape representing ID-variable* `PG_ID`*, respectively. Both symbols are shown inverse as they are not part of the transformer definition but have been introduced individually for this application. Further, application specific construction expression* `$PG := "Overview"` *defines the name for the single page class to be generated. Textually, this application would be notated as*

$$\begin{aligned}S.\texttt{apply}\ (\texttt{IndexPCForEnt},\ & ASC = \{\texttt{ENT\_ID} \neq \texttt{e3}\},\\ & ASE = \{\$\texttt{PG} := \text{"}\texttt{Overview}\text{"}\},\\ & IndivPinned = \{\$\texttt{PG}, \$\texttt{PG\_ID}\}\quad )\end{aligned}$$

*When processing this application, the query template returns two tuples, i.e. one for entity type* `Author` *and one for entity type* `Paper`*. Then, the* *generative part is processed as follows: First, the pinned new-element variables are instantiated and a new page class named* `Overview` *is created. Then, the remaining new-element variables are generated as usual, i.e. iteratively for each tuple of the query result. Thus, two index units* `AuthorIndex` *and* `PaperIndex` *are generated, which both are embedded in page class* `Overview`*. The result of this application is illustrated in Fig. 5.4(b).*

*Note that application-specific expression* `PG` := ”`Overview`”*, which overrides expression* `PG := strcat (ENT, "Page")`*, is necessary as otherwise the application would be invalid. The query template returns two tuples with two different values for variable* `ENT`*, i.e. values* `"Paper"` *and* `"Author"`*. Thus, if expression* `PG := strcat (ENT, "Page")` *were to be evaluated only once, the binding of variable* `ENT` *and, consequently, that of variable* `PG` *would be ambiguous.*

### 5.3.3 Turning new-element variables into parameter variables

It often is desired that an existing scheme element shall be extended or refined, although a predefined transformer is defined such that this scheme element would be generated as well. For example, transformer `IndexPCForEnt` generates a page class and an index unit for each entity type retrieved by the query template. This may be practical in many cases. In other cases, however, one might want to have index units to be generated into a page class that has already been defined previously and that shall therefore not be generated again.

Of course, one could define a new transformer where the respective new-element variables are replaced by parameter variables. Consequently, the query template would have to be extended by a corresponding result variable in order to provide a value for the parameter variable. However, this approach is cumbersome and leads to a number of separate transformers that differ from each other only in that one transformer defines a new-element variable where another transformer defines a parameter variable.

**Example 54** *Fig. 5.5 depicts a transformer* `IndexForEntWithinPC` *that is specifically defined for generating new index units into a page class that must already exist. Besides this issue, the purpose of transformer* `IndexForEntWithinPC` *is much the same as that of transformer* `IndexPCForEnt` *as used so far. However, the definition of*

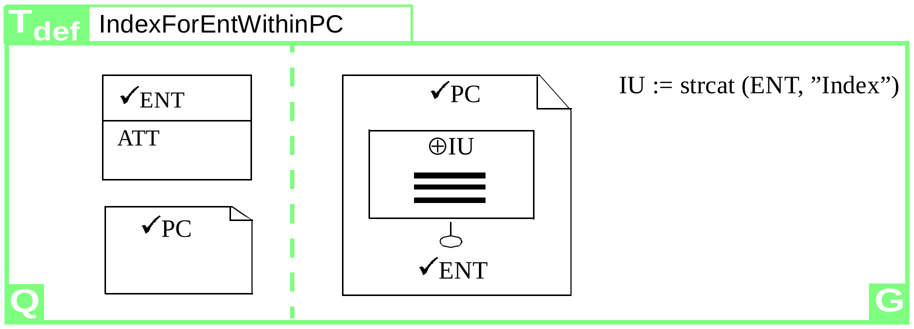


Figure 5.5: Transformer `IndexForEntWithinPC`, dedicated for generating index units into an existing page class

`IndexForEntWithinPC` *differs from that of* `IndexPCForEnt` *as follows: (1) Instead of a new-element variable, variable* `PC` *is defined as parameter variable, which is to be provided by the query template. (2) The query template is extended such that it selects pairs of entity types and page classes, which are represented by result variables* `ENT` *and* `PC`*, respectively. Thus, in order to have index units generated into an existing page class* `p`*, one has to constrain query template variable* `PC` *such that it accepts only page class* `p`*, i.e. by means of an application-specific constraint of the form* `PC = p`*.*

To avoid a large number of different transformers that have similar purpose but require different definitions, *TBE* allows to turn any new-element variable into a parameter variable individually at the time when a transformer is applied, i.e. without the need of changing the transformer definition. Turning a new-element variable into a parameter variable is achieved through the following two steps:

1. The respective new-element variable is declared as parameter variable.
2. This additional parameter variable must be bound to an existing scheme element, because the transformer's query template is not prepared for providing a value for this variable.

**Example 55** *Fig. 5.6 depicts an individualized application of transformer* `IndexPCForEnt`*, where new-element variable* `PG` *is turned into a parameter variable. This is indicated in the graphical application by the*

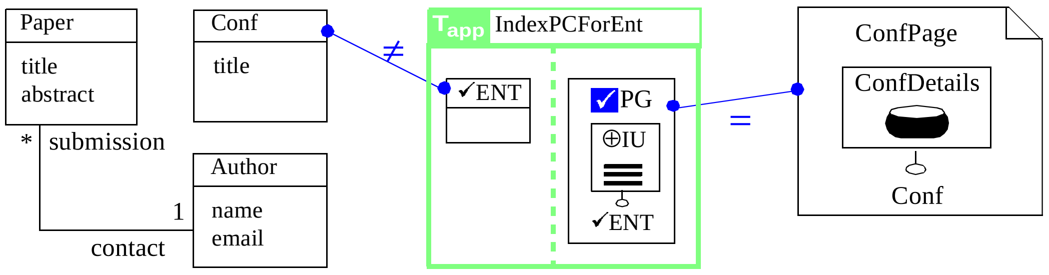


Figure 5.6: Individualized application of transformer "IndexPCForEnt": new-element variable `PG` is turned to a parameter variable.

*inverse "$\surd$"-symbol. Further, as the query template does not provide a page class for parameter variable* `PG`*, this variable is bound to page class* `ConfPage`*. Thus, instead of generating a new page class as defined in the transformer definition, existing page class* `ConfPage` *gets extended by new index units. Such index units are generated for each entity type except* `Conf`*. The result of this application is shown in Fig. 5.7*

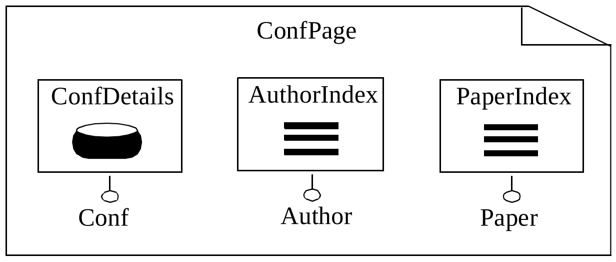


Figure 5.7: Result of transformer application depicted in Fig. 5.6.

Formally, a transformer application that is individualized by turning new-element variables into parameter variables is notated as $S.\texttt{apply}(T, \mathit{IndivParam})$, where $\mathit{IndivParam}$ is a set of new-element variables that are bound to scheme elements. Each such binding is notated as $v_g = e$, where $v_g \in V_g$ and $e$ is a scheme element of $S$. Again, besides turning new-element variables into parameter variables, the application may additionally define application-specific constraints and expressions as well as individually pinned new-element variables.

The meaning of turning a new-element variable into a parameter variable is straightforward: When the transformer application is processed, the query template is evaluated as usual, and the generative template is then instantiated separately for each tuple of the query template's result relation. For each newly declared parameter variable, the value is not provided by the query template but by binding this variable directly to an existing scheme element. Hence, the value of such a variable is the same over all instantiations of the generative template.

**Example 56** *Consider an application of transformer* `IndexPCForEnt` *that is individualized in that (1) new-element variables* `PC` *and* `PC_ID` *are turned into parameter variables and bound to scheme elements* `pg3` *and* `ConfPage`*, respectively, and in that (2) entity type* `Conf` *(*`e3`*) is not to be considered. This application is notated textually as*

$$\begin{array}{ll} S.\texttt{apply}\ ( & \texttt{IndexPCForEnt}, \\ & ASC = \{\texttt{ENT\_ID} \neq \texttt{e3}\}, \\ & \mathit{IndivParam} = \{\texttt{PG\_ID} = \texttt{pg3}, \texttt{PG} = \texttt{ConfPage}\} \quad ) \end{array}$$

*This application is processed as follows: The query template retrieves a relation $Rel_Q$ that comprises one tuple for each entity type except* `Conf`*, i.e. one tuple for entity type* `Author` *and one for entity type* `Paper`*. Thus, the generative template generates two new index units* `AuthorIndex` *and* `PaperIndex`*. Both are embedded in existing page class* `ConfPage` *with ID* `pg3`*. The result of this application is shown in Fig. 5.7.*

In a graphical application, turning a new-element variable to a parameter variable is expressed by preceding the respective variable by a "$\surd$"-symbol as it is used generally to indicate parameter variables. In order to emphasize that a new-element variable has been turned to a parameter variable, the respective "$\surd$"-symbol is shown inverse. As usual, when turning a new-element variable representing the name of a scheme element into a parameter variable, the corresponding ID-variable is by default turned into a parameter variable, too.

Alternatively to the textual notation of variable bindings, the binding of an ID-variables to a scheme element ID can be expressed by a line from the graphical shape representing the ID-variable to the graphical shape representing the respective scheme element ID. Further, as a notational shorthand, when binding an ID-variable to a scheme element $e$, then the name of $e$ is by default bound to the corresponding variable representing the scheme element's name.

**Example 57** *Reconsider the graphical application of transformer* `IndexPCForEnt` *as depicted in Fig. 5.6. Besides variable* `PG`*, which represents the name of a page class, implicitly defined ID-variable* `PG_ID` *is turned into a parameter variable, too. This ID-variable is bound to the page class with ID* `pg3` *as expressed by the line connecting the respective graphical shapes. Consequently, the name of this page class, i.e.* `ConfPage`*, is implicitly bound to variable* `PG`*. Thus, this graphical application of transformer* `IndexPCForEnt` *corresponds to the application as specified textually in example 56.*

### 5.3.4 Summary

This section has described various ways of how the behavior of existing transformers can be adapted without changing the transformer's definition. Application-specific constraints and construction expressions, individually pinned variables, and the possibility of turning new-element variables to parameter variables have been introduced. It has been shown that with these concepts of individualization, a predefined transformer can be flexibly used for various purposes.

**Example 58** *Predefined transformer* `IndexPCForEnt` *can be used for the following purposes: (1) It can be applied to one entity type, to every entity type of a scheme, or only to a particular set of entity types. (2) Its naming policy can be individually adapted. (3) It can be used to have index units generated iteratively, whereas the page class is generated only once. (4) It is also possible to have the index units generated into an already existing page class. All these different use cases can be expressed with a single predefined transformer.*

Thus, the benefit of individualization is that it reduces the number of different transformers that have to be defined; the more flexibly a particular transformer can be used, the less different transformers have to be defined.

However, as a flip side of the coin, the more one can influence the behavior of a predefined transformer, the more semantics is specified together with the transformer's application, and the less semantics gets tested and documented in the transformer's definition.

We had to define a reasonable boundary for individualization, i.e. to define up to which point a transformer shall be adaptable at application time and from which point a new transformer has to be defined. We decided to draw the boundary at the point where an adaptation would require to (a) introduce additional variables, (b) to rearrange or remove existing variables, or (c) to change existing constraints. The reason for this boundary is that we feel that it is impractical and error-prone to specify these adaptations through a graphical application-symbol.

More precisely, the reason for item (a) is that adding a new variable means to introduce a new scheme element skeleton. Yet adding skeletons is better done at the detailed level of a transformer definition rather than within a schematic graphical application-symbol. This is also true for item (b), which would mean to remove scheme element skeletons or rearranging them. Item (c) is motivated by the fact that most constraints are expressed implicitly by the graphical arrangement of scheme elements. Consequently, such constraints can only be changed by rearranging scheme element skeletons, and this is again impractical at the level of a graphical application-symbol.

We have also thought of disabling comparison constraints, which are notated explicitly in a transformer definition, individually at the time of transformer application. However, it turned out that disabling such constraints often turned a transformer into being impractical. In contrast, the cases where eliminating such a predefined constraint would make sense are comparably seldom. Consequently, we decided not to support this feature.

We feel that the concepts of individualization as presented in this section are both (1) sufficient for expressing adaptations as commonly required, and (2) practical in the sense that they can be expressed within the boundaries of a small-sized graphical application-symbol.

# Chapter 6

# Advanced concepts

## Contents

The expressive power of transformers as explained so far is limited to simple query templates, simple generative templates, and simple transformers that have the following limitations:

*Simple query templates* support only one basic form of DRC expression, which is built from only one complex predicate with a set of existentially quantified variables together with a conjunction of constraints. With this basic form, queries like "select page classes where *no* incoming link exists" or "select page classes that are reachable either via a contextual link or via a non-contextual link" cannot be expressed. Further, query templates do not support a notion of modularization, abstraction, and reuse.

*Simple generative templates* support the generation of a fixed set of scheme elements, but they do not support conditional scheme element generation. Further, they do not provide for modularization, abstraction, and reuse.

*Simple transformers* combine a query template and a generative template in that the flat relation achieved by the query template is iterated once for instantiating the generative template. This corresponds basically to generating scheme elements in a single loop pass. However, this simple combination is insufficient as soon as a modelling task requires to generate scheme elements in some kind of nested loop. Further, performing subsequent modelling tasks in a single transformer application cannot be expressed.

This chapter is dedicated to advanced concepts that overcome these limitations as follows:

- *Complex query templates* provide sub-queries, disjunctive queries, and universal quantification. Further, query template can be predefined and then reused for defining other query templates. This facilitates modularization, abstraction, and reuse.
- *Complex generative templates* provide delegation of details to separate or hierarchically organized generative sub-templates that may be conditional. Again, generative templates can be predefined for facilitating modularization, abstraction, and reuse.
- *Complex transformers* provides the following two concepts: (1) In order to support scheme element generation in nested loops, query templates and generative templates can be nested. The complex query template achieves a hierarchy of nested relations, and a corresponding hierarchy of nested generative templates processes these nested relations in pre-order traversal. (2) In order to support subsequent modelling tasks, transformer applications can be cascaded such that one transformer initiates the application of subsequent transformers.

Complex query templates, complex generative templates, and complex transformers are covered in Section 6.1, Section 6.2, and Section 6.3, respectively.

## 6.1 Complex query templates

With simple query templates, one can express only queries like "select page classes where there *exists* an incoming link". However, neither a query like "select page classes where there *does not exist* any incoming link" nor a query like "select page classes that are reachable from *every* other page class" can be expressed. Further, one cannot express queries of the form "select page classes where either a contextual or a non-contextual link leads to this page class".

In this section, we introduce complex query templates to overcome this limitation. We provide a notion of sub-query templates, i.e. simple query templates that represent a set of variables and constraints over these variables and that can be hierarchically combined using logical and relational connectives. This is much like sub-queries in SQL. Further, in order to facilitate modularization, abstraction, and reuse, query templates can be predefined.

As a prerequisite, Section 6.1.1 introduces predefined query templates, and query template applications, which provide for their flexible reuse. In Section 6.1.2, sub-queries of the form WHERE EXISTS... or WHERE NOT EXISTS... are explained, before Section 6.1.3 describes how to combine such sub-queries by logical connectives. Section 6.1.4 defines queries that are composed of sub-queries by using relational set-operators. Section 6.1.5 illustrates how queries involving universal quantification are expressed in *TBE*.

### 6.1.1 Predefined query templates

Query templates can be predefined and can then be reused many times via query template applications. Application-specific constraints known from individualized transformer applications can be employed analogously for individualizing the behavior of predefined query templates. For the sake of simplicity, we assume that a query template, once declared "predefined", is globally available. Of course, tools implementing *TBE* will provide some kind of repository.

A query template that is to be predefined is notated in a box labelled $\mathtt{Q}_{\mathtt{def}}$ followed by a name identifying the query template. Such a predefined query template is then applied via a query template application, which is notated as a box labelled $\mathtt{Q}_{\mathtt{app}}$ followed by the name of the query template to apply. Application-specific constraints are specified either textually within an individualization section or, if ID-variables are concerned, through lines connected to graphical shapes representing these ID-variables. For the purpose of attaching such lines, the query template definition or parts thereof can be schematically repeated in the graphical application.

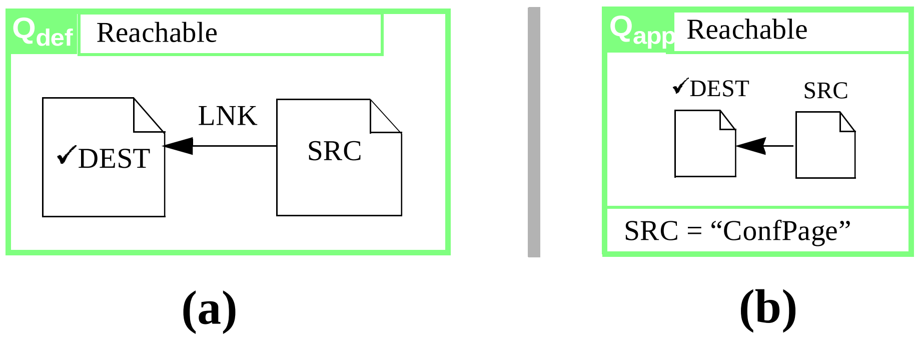


Figure 6.1: (a) Predefining query template `Reachable`; (b) Individualized application of predefined query template `Reachable`.

**Example 59** *Fig. 6.1(a) depicts the definition of query template* `Reachable`*, which selects page classes that are reachable. A page class* `DEST` *is reachable if there exists some page class* `SRC` *and some link* `LNK` *such that* `SRC` *is connected to* `DEST` *via link* `LNK`*. This query template is declared predefined as indicated by label* $\mathtt{Q}_{\mathtt{def}}$ *and is identified by its name* `Reachable`*.*

*Fig. 6.1(b) depicts an application of this predefined query template* `Reachable` *as indicated by label* $\mathtt{Q}_{\mathtt{app}}$*. The application is individualized by application-specific constraint* `SRC = "ConfPage"`*, which further constrains the query in that only outgoing links of a page class named* `"ConfPage"` *are considered.*

Formally, the application of a predefined query template $Q$ is notated as $Q[ASC]$, where $ASC$ is a set of application-specific constraints. Each explicit constraint that could have been specified in the query template definition can also be specified as application-specific constraint. The meaning of such an application is then defined by the meaning of the predefined query template extended by the application-specific constraints. This is analogous to individualized applications of transformers as defined in Section 5.3.

The following example illustrates the meaning of the application depicted in Fig. 6.1(b). For simplicity, we omit ID-variables in this example as well as in subsequent examples because the resulting DRC-expressions would be cluttered with many details. Instead, we treat variables representing names of scheme elements as if representing the scheme elements themselves. Consequently, instead of relations defined over scheme element IDs, we use derived relations that are defined over the names of these scheme elements. Further, we omitted domain function *dom* where the domain of variables is self-evident.

**Example 60** *Let relation* N_nctxLinkFromTo $\in N \times N \times N$ *be a relation where the first argument represents the name of a link whereas the second and the third argument represent the names of the link's source and target page class, respectively. The extension of this relation contains a tuple of a link name, a source page class name, and a target page class name if and only if a tuple of the respective IDs of these elements is contained in relation* nctxLinkFromTo $\in$ *Link* $\times P \times P$. *Thus, relation* N_nctxLinkFromTo *is defined as follows:*

$$\begin{aligned} N\_nctxLinkFromTo = \{ \ & \langle \mathtt{LN}, \mathtt{SN}, \mathtt{TN} \rangle \in (N \times N \times N) \mid \\ & \exists \ \mathtt{L\_ID} \in \mathit{Link}, \exists \ \mathtt{S\_ID} \in P, \exists \ \mathtt{T\_ID} \in P : \ ( \\ & \langle \mathtt{L\_ID}, \mathtt{S\_ID}, \mathtt{T\_ID} \rangle \in \mathit{nctxLinkFromTo} \ \wedge \\ & \langle \mathtt{L\_ID}, \mathtt{LN} \rangle \in \mathit{name} \ \wedge \\ & \langle \mathtt{S\_ID}, \mathtt{SN} \rangle \in \mathit{name} \ \wedge \\ & \langle \mathtt{T\_ID}, \mathtt{TN} \rangle \in \mathit{name} \ ) \\ \} & \end{aligned}$$

*Under the assumption that names of scheme elements represent the scheme elements themselves, predefined query template* `Reachable` *is given shortly as* $Q(\{\mathtt{DEST}\}, \{\mathtt{SRC}, \mathtt{LNK}\}, \{\langle \mathtt{LNK}, \mathtt{SRC}, \mathtt{DEST} \rangle \in \text{N\_nctxlinkFromTo}\}, \mathit{dom})$. *The individualized application of this query template as depicted in Fig. 6.1(b) is then notated as* $Q[\ \{\mathtt{SRC} = \text{"}\mathtt{ConfPage}\text{"}\}\ ]$. *Expressed in DRC, this application has the following meaning:*

$$\begin{aligned} \{ \ & \langle \mathtt{DEST} \rangle \in N \mid \exists \ \mathtt{SRC} \in N, \exists \ \mathtt{LNK} \in N : \\ & \quad \langle \mathtt{LNK}, \mathtt{SRC}, \mathtt{DEST} \rangle \in N\_nctxLinkFromTo \\ & \quad \wedge (\mathtt{SRC} = \text{``}\mathit{ConfPage}'') \\ \} & \end{aligned}$$

*The first two lines stem from query template Q, whereas the third line reflects application-specific constraint* `SRC = "ConfPage"`.

### 6.1.2 EXISTS/NOT-EXISTS sub-queries

Query templates may apply other query templates as EXISTS/NOT EXISTS sub-queries, whereby sub-queries can be arbitrarily nested. The meaning is straightforward: a query template which is applied within a main query template $Q_m$ as an EXISTS sub-query represents a constraint of $Q_m$. This constraint is fulfilled if and only if the sub-query returns at least one tuple. Analogously, a NOT EXISTS sub-query constraint is fulfilled if and only if the sub-query returns no tuples.

A sub-query template is a simple query template, which comprises a set of local variables and constraints over them. Additionally, the variables of the main query template are in the scope of the sub-query template and can be used for constraints that correlate the sub-query to the main query. Thus, sub-query templates are similar to correlated sub-queries in SQL. In *TBE*, however, correlation is expressed through application-specific constraints rather than through constraints that are hard-coded in the sub-query. This technique allows to reuse predefined query templates as correlated sub-queries, even though a predefined query template is generally not aware of a correlation.

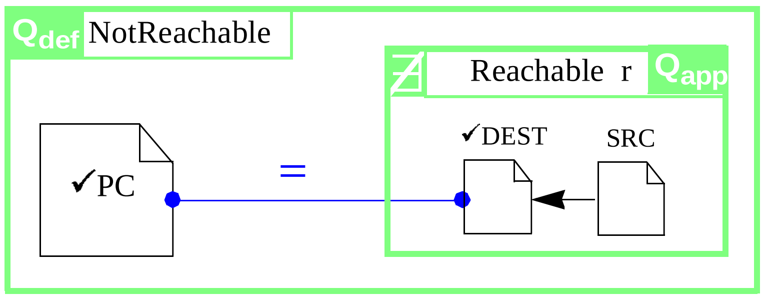


Figure 6.2: Query template `NotReachable`, expressed through applying query template `Reachable`

**Example 61** *Suppose that we want to define a query template* `NotReachable` *that selects isolated page classes, i.e. those page classes that are not reachable from any page class via a non-contextual link. This query can be expressed by applying predefined query template* `Reachable` *as a* NOT-EXISTS *sub-query as illustrated in Fig. 6.2. Query template* `NotReachable` *reads as follows: "Select those page classes represented by variable* `PC`*, where it is not the case that* `PC` *is* *reachable". Predicate "not reachable" is fulfilled for a particular page class* p *if query template* `Reachable` *returns no tuples in the context of* p*, i.e. when having variable* `DEST` *constrained to accept only page class* p*.*

Formally, the application of *EXISTS* and *NOT EXISTS* sub-queries is notated as follows: Let $Q_m(V_{rm}, V_{nrm}, C_m, dom_m)$ and $Q_s(V_{rs}, V_{nrs}, C_s, dom_s)$ be two distinct query templates, which consequently have disjoint sets of variables. Then an application of $Q_s$ as an *EXISTS* or *NOT EXISTS* sub-query within $Q_m$ as main query is notated as $\exists Q_s[ASC]$ or $\nexists Q_s[ASC]$, respectively. $ASC$ is a set of application-specific constraints as generally used in query template applications. Each of these constraints may constrain variables of $Q_s$ as described in Section 6.1.1, but also based on variables that are not defined in $Q_s$ but that are in the scope of the sub-query, i.e. variables of $Q_m$. As sub-queries can be arbitrarily nested, the variables of a (main) query are in the scope of all direct or indirect sub-queries.

Informally, one can think of a sub-query as a query that is evaluated for each tuple of the main query. Domain relational calculus, however, provides no notion of "sub-queries" but only (complex) predicates. Thus, a query template $Q_s$ applied as $\exists Q_s[ASC]$ in a main query template $Q_m$ denotes a complex constraint $c \in C_m$ of $Q_m$. Query template $Q_m$ translates to a DRC expression as usual, i.e. as described in Section 4.3.2, yet the resulting DRC-expression comprises an additional constraint *SubQ*. The following DRC-expression shows where the complex constraint *SubQ* fits in the usual translation:

$$\begin{array}{l} \{\ \langle v_{rm_1}, \ldots, v_{rm_x} \rangle \in dom_m(v_{rm_1}) \times \ldots \times dom_m(v_{rm_x}) \mid \\ \quad \exists v_{nrm_1} \in dom_m(v_{nrm_1}), \ldots, \exists v_{nrm_y} \in dom_m(v_{nrm_y}) : \\ \quad (c_1 \wedge \ldots \wedge c_z \wedge (SubQ)) \\ \} \end{array}$$

*SubQ* is a place-holder for the term representing sub-query template $\exists Q_s[ASC]$. This term is built from the definition of query template $Q_s$ and from the application-specific constraints $ASC$ as follows: (1) All result-variables of $Q_s$ are treated as non-result variables. This is because the sub-query is not to *return* the tuples that fulfill all constraints but only to check whether there *exists* at least one such tuple. (2) Then, query template $Q_s$ is mapped to a DRC expression as usual. (3) As the sub-query template may be correlated to the main query template through the set of application-specific constraints $ASC$, these constraints are directly taken over into the set of constraints derived from the definition of $Q_s$. Thus, a sub-query application of the form $\exists Q_s[ASC]$ yields the following term $SubQ$,

$$\begin{array}{ll} SubQ \equiv & (\ \exists v_{s1} \in dom_m(v_{s1}), \ldots, \exists v_{si} \in dom_m(v_{si})\ : \\ & \quad c_1 \wedge \ldots \wedge c_j \\ & \quad \wedge\ asc_1 \wedge \ldots \wedge asc_k \\ & ) \end{array}$$

where variables $v_{s1}, \ldots, v_{si}$ denote the set of variables $V_s = (V_{rs} \cup V_{nrs})$, constraints $c_1, \ldots, c_j$ denote the set of constraints $C_s$, and $asc_1, \ldots, asc_k$ denote the set of application-specific constraints $ASC$. It is easy to see that variables of $Q_m$ are actually in the scope of $Q_s$ such that applications specific constraints $asc_1, \ldots, asc_k$ may refer to them. Note that $SubQ$ is not a query that returns tuples but is a (boolean) predicate. Obviously, if $Q_s$ were applied within $Q_m$ as $\nexists Q_s[ASC]$, then term $SubQ$ would be negated, i.e. $\neg(SubQ)$.

**Example 62** *Reconsider query templates* `Reachable` *and* `NotReachable`. *Query template* `Reachable` *is given as* $Q_s(\{\texttt{DEST}\}, \{\texttt{SRC}, \texttt{LNK}\}, \{\langle \texttt{LNK}, \texttt{SRC}, \texttt{DEST}\rangle \in N\_nctxLinkFromTo\}, dom_s)$*, whereas query template* `NotReachable` *is then given as* $Q_m(\{\texttt{PC}\}, \{\}, \{\nexists\, Q_s[\{\texttt{PC} = \texttt{DEST}\}]\ \}, dom_m)$. *Query template* $Q_m$ *is mapped to the following DRC-expression:*

$$\begin{array}{l} \{\ \langle \texttt{PC} \rangle \in N\ | \\ \quad \neg\ (\ \exists\ \texttt{DEST} \in N, \exists\ \texttt{SRC} \in N, \exists\ \texttt{LNK} \in N : \\ \qquad \langle \texttt{LNK}, \texttt{SRC}, \texttt{DEST} \rangle \in N\_ncxtLinkFromTo \\ \qquad \wedge (\texttt{PC} = \texttt{DEST}) \\ \quad ) \\ \} \end{array}$$

*The first line reflects the result of* $Q_m$*, whereas the following lines represent the constraint as expressed through sub-query application* $\nexists Q_s[\{\texttt{PC} = \texttt{DEST}\}]$. *Note that* $Q_m$ *does not comprise any non-result variables nor any further constraints except the sub-query. The second line represents the variables of query template* $Q_s$*, each of them being existentially quantified. The third line represents the constraints expressed in the definition of* $Q_s$*, whereas the fourth line represents the application specific-constraint, which correlates the sub-query to the main query. This expression can be transformed to the following equivalent yet more condensed form:*

$$\{\ \langle \mathtt{PC} \rangle \in N \mid\ \neg\ (\ \exists\ \mathtt{SRC} \in N, \exists\ \mathtt{LNK} \in N : \langle \mathtt{LNK}, \mathtt{SRC}, \mathtt{PC} \rangle \in N\_nctxLinkFromTo\ )\ \}$$

Graphically, the application of a sub-query template $Q_s$ within a main query template $Q_m$ is notated like an ordinary query template application. However, symbol $\exists$ or $\nexists$ expresses whether the sub-query is applied as an EXISTS or as a NOT EXISTS sub-query. When referring to a sub-query template's variable within an application-specific constraint, this variable is qualified by the name of the sub-query template. This qualification is necessary in case of name-clashes between different variables that are in the same scope. In order to facilitate this qualification, each sub-query template may define an alias that is attached to the sub-query template's name and that can then be used in place of this name.

**Example 63** *Fig. 6.2(b) illustrates the graphical representation of query template* `NotReachable`*, which applies predefined query template* `Reachable` *as* NOT EXISTS *sub-query. The variables of the sub-query template can be addressed through alias* `r` *defined in the application. The application-specific constraint correlating sub-query variable* DEST *to main query variable* PC *is expressed by the line labelled "=" connecting these two variables. If notated textually, this application-specific constraint would be written as* $\mathtt{PC} = \mathtt{r.DEST}$.

Besides utilizing predefined query templates, a sub-query template can also be defined "inline", i.e. such that it is defined only locally. Inline definitions are practicable where the sub-query is unlikely to be reused in a different context later on. Such a sub-query template is defined like a predefined query template but is placed directly in the main query template in which is is to be applied. In order to express that the sub-query template is defined only locally, the respective box is labelled "Q" instead of "$\mathtt{Q}_{def}$". The name of the sub-query template is optional.

**Example 64** *Fig. 6.3 depicts a definition of query template* `NotReachable` *where sub-query template* `Reachable` *is defined inline as indicated by label "*`Q`*". Note that it is irrelevant whether the sub-query template comprises result-variables or not since it is applied as* NOT-EXISTS *sub-query.*

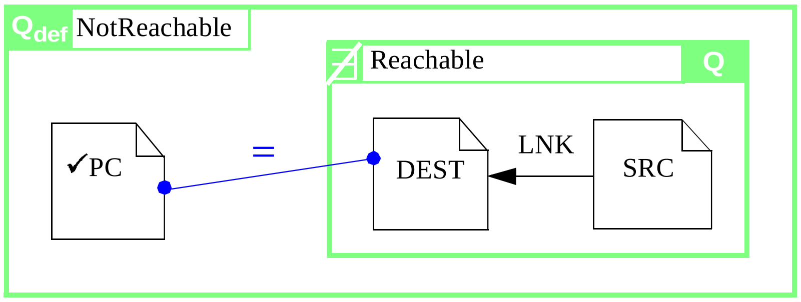


Figure 6.3: Query template `NotReachable`, sub-query template `Reachable` defined inline

## 6.1.3 Combining sub-queries by logical connectives

Several EXISTS/NOT EXISTS sub-queries can be combined using the logical connectives $\vee$, $\wedge$, and $\neg$. This combination is straightforward since each such sub-query represents a predicate that either evaluates to "true" or to "false". Thereby, it becomes possible to express formulae built from more than one complex predicate. For example, a formula of the form $(\exists v1 : P(v1)) \vee (\exists v2 : P(v2))$ can be expressed by two EXISTS sub-queries that represent these two parts, respectively, and that are connected by logical connective "`OR`". As explained in Section 4.3.3, this situation cannot be expressed with one simple query template, which comprises only one set of existentially quantified variables.

**Example 65** *Fig. 6.4 depicts query template* `HomeOrLandmark`*, which selects page classes that are either home pages or landmarks, i.e. that are marked either with symbol* `"H"` *or with symbol* `"L"`*. Result variable* `HPORLMP` *of the main query template ranges over all page classes, yet the two sub-queries check whether such a page class* `HPORLMP` *is either a home page or a landmark, respectively.*

Formally, sub-query templates combined by logical connectives are notated as complex constraints having the sub-query templates as operands, e.g. $\exists Q_{s1}[ASC_1] \vee \exists Q_{s2}[ASC_2]$. The meaning in terms of DRC is straightforward: Let $c$ be a complex constraint comprising $n$ sub-query templates of the form $\exists Q_s[ASC]$ or $\nexists Q_s[ASC]$. Then, for $i = 1 \ldots n$, each $Q_{si}[ASC_i]$ is mapped to a term $(SubQ_i)$ as described in Section 6.1.2. The meaning of

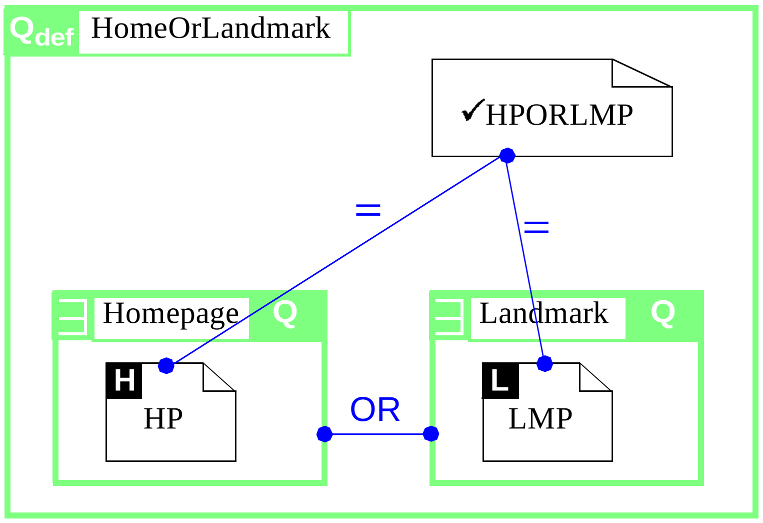


Figure 6.4: Query template `HomeOrLandmark`, combining two EXISTS sub-queries by logical connective "`OR`"

complex constraint $c$ is then defined by the meaning of each $SubQ_i$ together with that of the logical connectives.

**Example 66** *Reconsider query template* `HomeOrLandmark` *as described in example 65. Let $Q_{s1}$ and $Q_{s2}$ be query templates* `Homepage` *and* `Landmark`*, respectively, and let $Q_m$ be main query template* `HomeOrLandmark`*. They are defined as follows:*

$$\begin{aligned}
&Q_{s1}\ (\ \{\mathtt{HP}\}, \{\}, \{\mathtt{HP} \in \mathit{homepage}\})\\
&Q_{s2}\ (\ \{\mathtt{LMP}\}, \{\}, \{\mathtt{LMP} \in \mathit{landmark}\})\\
&Q_m\ (\ \{\mathtt{HPORLMP}\}, \{\}, \{\ \ (\exists Q_{s1}[\mathtt{HPORLMP} = \mathtt{HP}]) \vee\\
&\qquad\qquad\qquad\qquad\qquad (\exists Q_{s2}[\mathtt{HPORLMP} = \mathtt{LMP}])\ \}\ )
\end{aligned}$$

*Query template* `HomeOrLandmark` *is then mapped to the following DRC-expression:*

$$\begin{aligned}
\{\ \langle\mathtt{HPORLMP}\rangle \in P \mid\ &(\ \exists\ \mathtt{HP} \in P : \mathtt{HP} \in \mathit{homepage} \wedge \mathtt{HPORLMP} = \mathtt{HP}\ ) \vee\\
&(\ \exists\ \mathtt{LMP} \in P : \mathtt{LMP} \in \mathit{landmark} \wedge \mathtt{HPORLMP} = \mathtt{LMP}\ )\\
\}\qquad&
\end{aligned}$$

In the graphical notation, a complex constraint formed of sub-query template applications is notated as an expression specified textually. The constituting sub-query applications are referred to by their names or, if specified, by their aliases. Additionally, complex constraints the form $(Q_{app1} \vee, \ldots \vee, Q_{appn})$ can be notated by connecting the constituting sub-query applications $Q_{app1}, \ldots, Q_{appn}$ by lines labelled "`OR`". For complex constraints of the form $(Q_{app1} \wedge, \ldots \wedge, Q_{appn})$, no such lines are necessary since connective "`AND`" is the default. All other forms of complex constraints must be notated textually as described above.

**Example 67** *Fig. 6.4 depicts the graphical representation of constraint* $(\exists Q_{s1}[\texttt{HPORLMP} = \texttt{HP}]) \vee (\exists Q_{s2}[\texttt{HPORLMP} = \texttt{LMP}])$*, where* $Q_{s1}$ *stands for query template* `Homepage` *and* $Q_{s2}$ *stands for query template* `Landmark`*. The disjunction of these two sub-query applications is expressed by the line labelled "OR" connecting them. If specified textually, this constraint would be notated as "(Homepage* $\vee$ *Landmark)" instead of the line labelled "OR".*

### 6.1.4 Combining sub-queries by set operators

Since each query template returns a set of tuples, several query template applications can be combined by the usual set operators $\cup$, $\cap$, and $\setminus$. The use of set operators is often more intuitive than the use of logical connectives $\vee$, $\wedge$, and $\neg$ as explained previously in Section 6.1.3. However, as usual for set operators, all constituting query templates must comprise equal sets of result variables, i.e. they must comprise the same number of variables that agree in name and type. This can be problematic when reusing predefined query templates, which have the set of result variables already fixed.

**Example 68** *Fig. 6.5 depicts query template* `NotReachable`*, which is now defined through set operator "MINUS". Thereby, the set of unreachable page classes is defined by subtracting the set of reachable page classes from the set of all page classes. This is expressed by the line labelled "MINUS", which connects an application of query template* `AllPageClasses` *(the left hand side) to an application of query template* `Reachable` *(the right hand side).*

*Similarly, Fig. 6.6 illustrates query template* `HomepageOrLandmark`*, which is now defined through set operator "UNION". Note, however,*

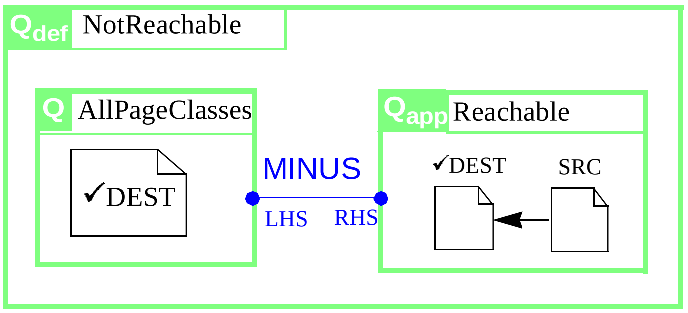


Figure 6.5: Query template `NotReachable`, defined using set operator "MINUS"

*that query templates* `Homepage` *and* `Landmark` *as defined in Fig. 6.4(b) could not be used for that purpose, because these two query templates have different result variables, i.e. variable* `HP` *in* `Homepage` *and variable* `LMP` *in* `Landmark`*. Thus, their sets of result variables are not equal as required by set operator "UNION".*

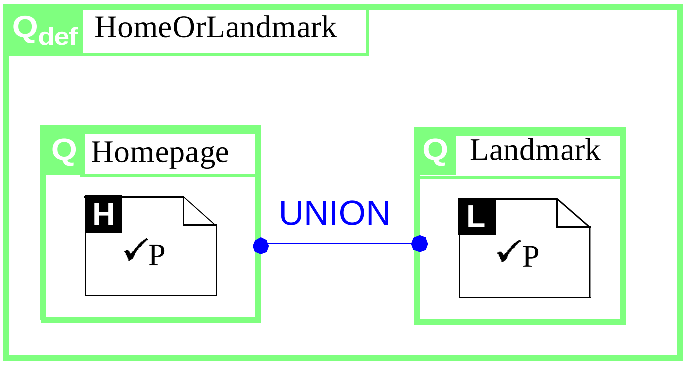


Figure 6.6: Query template `HomeOrLandmark`, defined using set operator "UNION"

Formally, a query template $Q$ combining several query template applications through set operators is notated as a complex expression having these applications as operands. This is similar to complex expressions combining sub-query templates by logical connectives. However, query template $Q$ must not comprise other elements than the query template applications. The meaning of $Q$ is derived from the constituting query template applications and the set operators combining them.

**Example 69** *Reconsider query template* `HomeOrLandmark` *as described in example 68. It combines an application of query templates* $Q_{hp}$=`Homepage` *and* $Q_{lmp}$=`Landmark` *by set operator "UNION". Thus, query template* `HomeOrLandmark` *is expressed as* $Q = (Q_{hp}[\{\}] \cup Q_{lmp}[\{\}])$*, i.e. by two applications of query templates* $Q_{hp}$ *and* $Q_{lmp}$*. In this case, these applications do not comprise any application-specific constraint, although this could be practicable in other cases.*

In the graphical notation, query templates that are to be combined by set operators are expressed as usual, i.e. either as query template applications or as query templates defined inline. The combination itself is notated as a complex expression specified textually. The query template applications serving as operands are referred to by their names or, if specified, by their aliases. Additionally, complex expressions of the form $Q_{app1} \cup, \ldots, \cup Q_{app_n}$ or $Q_{app1} \cap, \ldots, \cap Q_{app_n}$ can be notated as lines that are labelled by the respective set-operator and that connect operands $Q_{app1}, \ldots, Q_{app_n}$. For set operator "MINUS", which is not commutative, the left hand side operand and the right hand side operand are marked by symbols "`LHS`" and "`RHS`", respectively.

**Example 70** *Fig. 6.5 depicts the graphical representation of query template* `NotReachable`*, which is expressed through subtracting the set of reachable page classes from that of all page classes. This substraction is expressed by a line from the application of query template* `AllPageClasses`*, which serves as the left hand side operand, to the application of query template* `Reachable`*, which serves as the right hand side operand. Query template* `AllPageClasses` *is defined inline as denoted by symbol "*`Q`*", whereas* `Reachable` *refers to a predefined query template as denoted by symbol "*$\mathtt{Q_{app}}$*".*

### 6.1.5 Queries involving universal quantification

Queries involving universal quantification typically have the form "from a domain $D_X$, select those $X$ for that all $Y$ out of domain $D_Y$ satisfy condition $C$". Logically, this would be expressed as a formula of the form $\{(X) \in D_X \mid \forall\, Y \in D_Y : C(X, Y)\}$. For example, a query of the form "select page classes that are globally reachable" would be formulated as "select page classes `P` where all other page classes `OP` satisfy condition (`OP` reaches `P`). "

However, most declarative query languages, both textual and visual ones, do not provide universally quantified variables but express such a formula equivalently as "those $X$, where there does not exist an $Y$ such that condition $C$ does not hold, i.e. $\{(X) \mid \neg(\exists Y : \neg C(X, Y))\}$. Thus, above mentioned query is formulated as "select those page classes `P` where there does not exist another page class `OP` such that `P` is not reachable from `OP`". Examples of languages that make use of this equivalent are SQL [128], Datalog [91], XCX [106], XQuery [14], XML-GL [31], and many others.

In *TBE*, universal quantification is also expressed through existential quantification and negation, i.e. by means of nested NOT-EXISTS sub-queries. These sub-queries have already been introduced in the beginning of this section, such that expressing "universal quantification" does not require to introduce a new construct but only to use existing ones. The following example illustrates how to use two nested NOT-EXISTS sub-queries for expressing universal quantification:

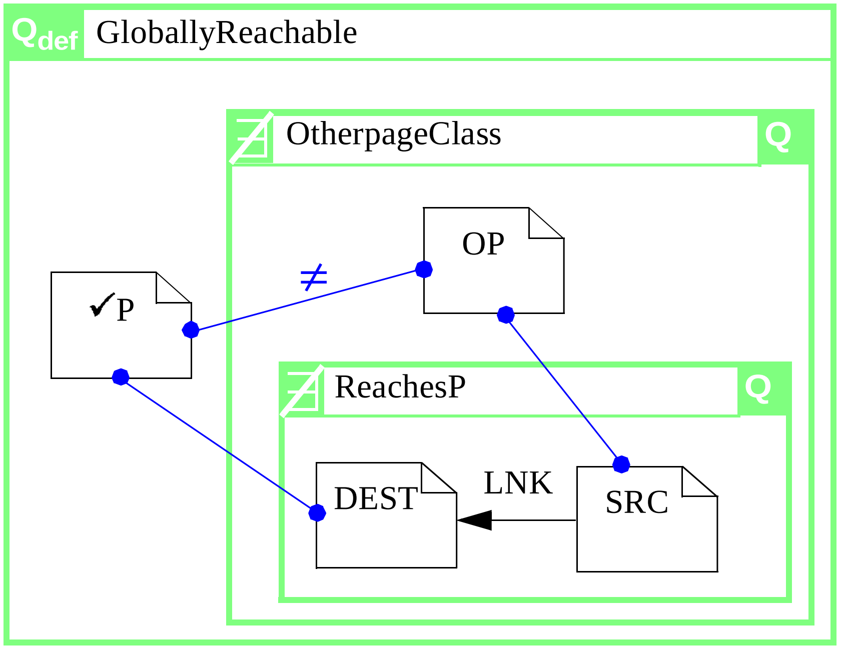


Figure 6.7: Query template `GloballyReachable`, expressed using two nested NOT-EXISTS sub-queries.

**Example 71** *Fig. 6.7 depicts query template* `GloballyReachable`*, which selects those page classes which are reachable from every other page class. The query is expressed by two nested* NOT-EXISTS *sub-queries* *and reads as follows: "Select those page classes* P *where there does not exist a page class* OP *different from* P *such that* P *is not reachable from* OP*". From this query template, the following DRC expression is derived:*

$$
\begin{array}{l}
\{\ \langle \texttt{P} \rangle \in N \mid \neg\,(\ \exists\ \texttt{OP} \in N : (\texttt{P} \neq \texttt{OP}) \wedge \\
\qquad\qquad \neg\ \ (\ \exists\ \texttt{DEST} \in N, \texttt{SRC} \in N, \texttt{LNK} \in N : \\
\qquad\qquad\qquad (\ \langle \texttt{LNK}, \texttt{SRC}, \texttt{DEST} \rangle \in N\_nctxLinkFromTo\ ) \wedge \\
\qquad\qquad\qquad (\texttt{P} = \texttt{DEST} \wedge \texttt{OP} = \texttt{SRC}) \\
\qquad\qquad\ \ ) \\
\qquad\ ) \\
\}
\end{array}
$$

*This expression becomes more condensed by replacing variable* DEST *with* P *and variable* SRC *with* OP*. This achieves the following equivalent DRC-expression, which reads as "select those* P *where there does not exist an* OP *different from* P *such that there does not exist a link* LNK *from* OP *to* P*":*

$$
\begin{array}{l}
\{\ \langle \texttt{P} \rangle \in N \mid \neg\,(\ \exists\ \texttt{OP} \in N : (\texttt{P} \neq \texttt{OP}) \wedge \\
\qquad\qquad \neg\ \ (\ \exists \texttt{LNK} \in N :\ \langle \texttt{LNK}, \texttt{OP}, \texttt{P} \rangle \in N\_nctxLinkFromTo\ ) \\
\qquad\ ) \\
\}
\end{array}
$$

## 6.2 Complex generative templates

In simple generative templates, all scheme elements to be generated are defined in a single template. This generative template may become complex if a multitude of different scheme elements is to be generated based on many different parameters. This suggests to modularize generative templates, i.e. to delegate details of scheme element generation to other generative templates. Furthermore, modelling tasks often are to be performed in slight variations such that, depending on some property of the scheme element configuration to which the transformer is applied, some scheme elements are to be generated in a different way.

This section covers several ways of splitting complex generative templates into simpler ones. As a prerequisite, Section 6.2.1 describes how generative templates can be predefined for later reuse. Section 6.2.2 introduces conditional generative templates, whereas Section 6.2.3 explains how to organize generative templates hierarchically.

### 6.2.1 Predefining generative templates

Like transformers and query templates, generative templates can be predefined and can henceforth be used just like generative templates defined inline. When applying a predefined generative template, the full palette of individually adapting its behavior as known from transformer applications can be used, i.e. defining application-specific expressions, pinning new-element variables, and turning new-element variables into parameters.

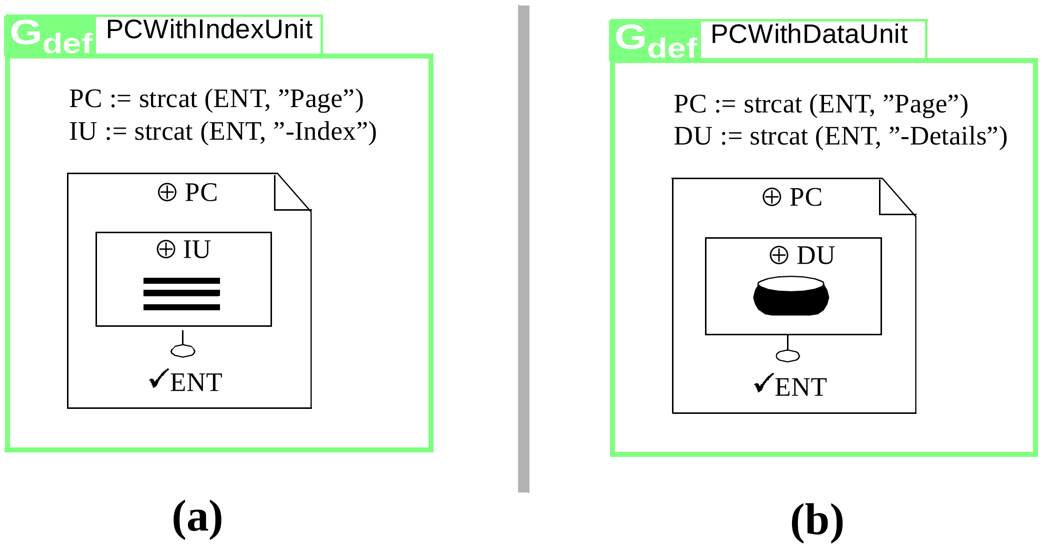

Gdef PCWithIndexUnit
PC := strcat (ENT, ”Page”)
IU := strcat (ENT, ”-Index”)
⊕ PC
⊕ IU
✓ENT
(a)
Gdef PCWithDataUnit
PC := strcat (ENT, ”Page”)
DU := strcat (ENT, ”-Details”)
⊕ PC
⊕ DU
✓ENT
(b)

Figure 6.8: (a) and (b): Definitions of generative templates `PCWithIndexUnit` and `PCWithDataUnit`.

**Example 72** *Fig. 6.8 depicts the definition of two generative templates. The one depicted in (a) generates a new page class with an index unit, whereas the one depicted in (b) generates a new page class with a data unit. These templates are reused throughout this section.*

*Fig. 6.9 illustrates the definition of transformer* `DUPageForSingleton`*, which generates a page class with a data unit for each entity type that is declared as* `singleton`*. The generative part refers to the generative template* `PCWithDataUnit` *defined previously.*

Formally, an application of a generative template $G$ is notated as $G[B]$, where $B$ is a set of parameter-variable bindings, each binding a parameter variable

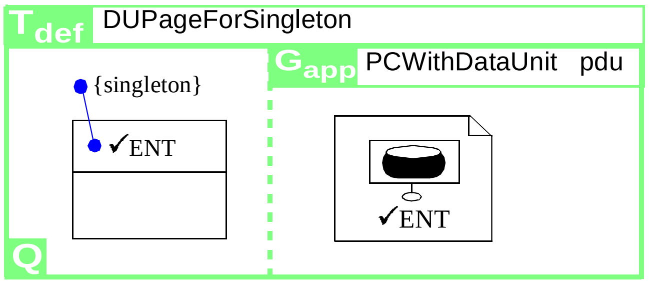


Figure 6.9: Transformer `DUPageForSingleton` applying predefined generative template `PCWithDataUnit`.

$v_p$ either to a scheme element $e$ or to a result variable $v_r$ of a query template. Such bindings are notated as "$v_p = e$" or "$v_p = v_r$", respectively. The meaning of applying a predefined generative template is the same as that of defining a generative template inline as introduced in Section 4.4.

Graphically, an application of a generative template $G$ is indicated by a box labelled "$\mathtt{G}_{\mathtt{app}}$" followed by the name of the generative template and, optionally, an alias. To avoid ambiguities, a reference to a variable of $G$ can be qualified by the generative template's name or, if specified, by its alias. If a generative template $G$ is used in conjunction with a query template $Q$, parameter variables of $G$ are by default bound to equally named result variables of $Q$. This default behavior can be overridden through explicit variable bindings expressed by connecting the respective variables by a line.

**Example 73** *Reconsider transformer* `DUPageForSingleton` *depicted in Fig. 6.9, whose generative part applies predefined generative template* `PCWithDataUnit`*. Alias* `pdu` *represents this application, which is formally notated as* $G[\mathtt{ENT} = \mathtt{pdu.ENT}]$*. Variable binding "*`ENT = pdu.ENT`*" is established implicitly based on the equal names of both variables.*

### 6.2.2 Conditional generative templates

The way new scheme elements are to be generated often varies depending on a property of the scheme element configuration to which the transformer is applied. When expressing such situations with generative templates as explained so far, which generate a set of new scheme elements always in the same manner, one had to define a separate transformer for each variant. This can be tedious and leads to a large number of similar transformer definitions. To overcome this problem, conditional generative templates are introduced. The generative part of a single transformer is split into several generative templates, each guarded by a condition over parameter variables such that it is instantiated only if the attached boolean condition is fulfilled.

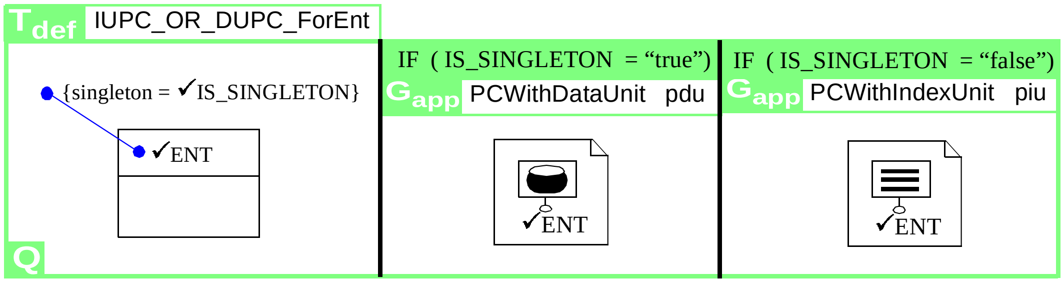


Figure 6.10: Transformer `IUPC_OR_DUPC_ForEnt`, defined using conditional generative templates.

**Example 74** *Fig. 6.10 depicts the definition of transformer* `IUPC_OR_DUPC_ForEnt`*, which generates either a page class with a data unit or a page class with an index unit, depending on whether the underlying content source is a singleton entity type or not. This is achieved as follows: The query template selects entity types together with their respective value of user-defined property* `singleton`*. Result variable* `IS_SINGLETON` *is true if the entity type has this property set, otherwise false.*

*The generative part comprises two conditional generative templates, which refer to predefined* [1] *generative templates* `PCWithDataUnit` *and* `PCWithIndexUnit`*, respectively. Only one of them is instantiated depending on the value of variable* `IS_SINGLETON`*.*

Formally, a conditional application of a generative template $G$ is notated as $G\{condition\}[B]$, where *condition* is a boolean condition that may refer to parameter variables of $G$. The condition can also be simply “true”

[1]Note that conditional generative templates can also be defined inline. We use predefined generative templates only for convenience.

such that the application is unguarded and corresponds to applications of generative templates as explained so far. When a conditional application $G\{condition\}[B]$ is processed, first condition *condition* is evaluated in the context of the particular binding of parameter variables as defined in $B$. If it evaluates to true, the generative template is instantiated, otherwise not. Note that the conditions guarding generative templates are not required to be mutual exclusive, i.e. two or more conditional generative templates may get instantiated for the same tuple of a query template's result relation.

**Example 75** *Reconsider transformer* `IUPC_OR_DUPC_ForEnt` *depicted in Fig. 6.10. The two generative templates constituting the transformer's generative part are formally notated as follows:*

$$\begin{array}{ll}
\texttt{PCWithDataUnit} & \{\texttt{pdu.IS\_SINGLETON} = \text{"}\texttt{true}\text{"}\} \\
 & [\ \texttt{ENT} = \texttt{pdu.ENT}, \texttt{IS\_SINGLETON} = \texttt{pdu.IS\_SINGLETON}\ ] \\
\texttt{PCWithIndexUnit} & \{\texttt{piu.IS\_SINGLETON} = \text{"}\texttt{false}\text{"}\} \\
 & [\ \texttt{ENT} = \texttt{pdu.ENT}, \texttt{IS\_SINGLETON} = \texttt{pdu.IS\_SINGLETON}\ ]
\end{array}$$

*If the generative part is instantiated with parameter variable* `IS_SINGLETON` *bound to value* `"true"`*, generative template* `PCWithDataUnit` *is instantiated, whereas generative* `PCWithIndexUnit` *is not. Obviously, if parameter variable* `IS_SINGLETON` *is bound to value* `"false"`*,* `PCWithIndexUnit` *is instantiated and not* `PCWithDataUnit`*.*

Graphically, a conditional generative template is expressed by placing the condition atop the box representing the generative template definition or application. Omitting the condition has the same effect as a condition of the form `IF("true")`.

### 6.2.3 Generative template hierarchies

Generative templates can be hierarchically organized as a tree, where a main generative template is distinguished from generative sub-templates contained therein. A main generative template usually provides the "big picture" of the scheme elements to be generated, whereas details tend to be shifted to generative sub-templates.

A hierarchy of generative templates is instantiated from the root to the leaves, i.e. a main generative template is instantiated before any generative sub-template contained therein is instantiated. Thus, parameter variables as well as new-element variables of a main generative template may serve as parameters for a generative sub-template, which then defines further details for the scheme elements already generated. Of course, a generative sub-template *sub* may contain further generative sub-templates *subsub*, and all variables that are in the scope of *sub* are also in the scope of *subsub*.

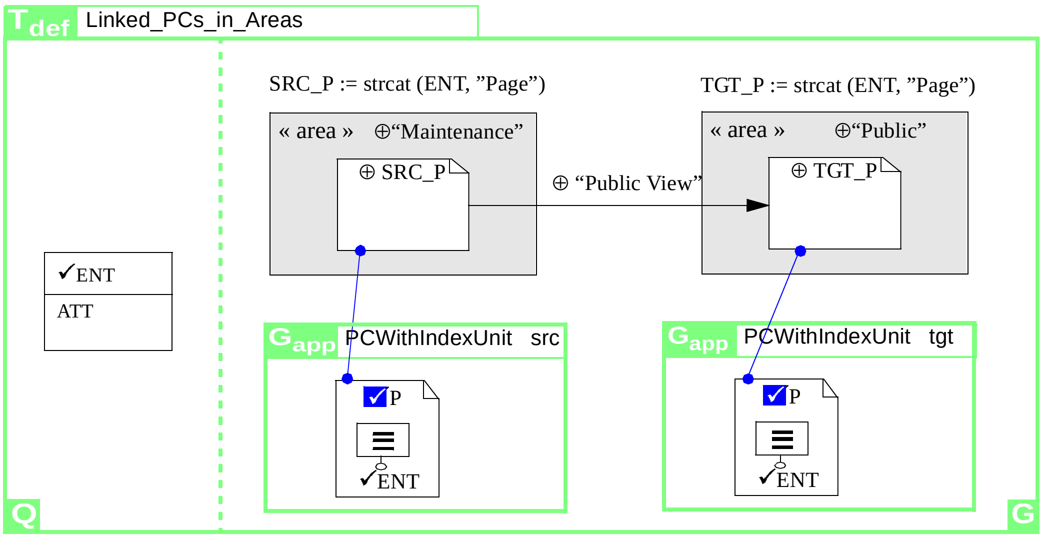


Figure 6.11: Transformer `Linked_PCs_in_Areas`, defined using a main generative template with two generative sub-templates.

**Example 76** *Suppose that we want to define a transformer* `Linked_PCs_in_Areas` *with the basic purpose of generating a pair of interconnected page classes that are placed in two different areas* `Maintainance` *and* `Public`*, which are generated, too. Additionally, these page classes shall both comprise an index unit.*

*Fig. 6.11 depicts a definition of this transformer* `Linked_PCs_in_Areas`*, which comprises a main generative template and two generative sub-templates. The main generative template covers the generation of the areas and the page classes. The generation of the index units is regarded as detail and is shifted to generative sub-templates, where the newly generated page classes serve as parameters. The details concerning the generation of the index unit are covered by the generative sub-templates.*

Formally, the application of a generative sub-template $G_{sub}$ within a main generative template $G_{main}$ is notated like any other application of a generative template, i.e. $G_{sub}[B]$. The variables of $G_{sub}$ may be bound to both parameter variables and new-element variables of $G_{main}$. Such an application expresses that, once main generative template $G_{main}$ has been instantiated, generative sub-template $G_{sub}$ is processed. Consequently, variables of $G_{sub}$ that are bound to new-element variables of $G_{main}$ actually refer to scheme elements that have just been generated.

The graphical notation of generative sub-templates is the same as that of other applications of generative templates and needs no further explanation.

**Example 77** *Reconsider transformer* `Linked_PCs_in_Areas` *depicted in Fig. 6.11. Let $G_{main}$ be the main generative template and let $G_{sub}$ be the application of generative sub-template* `PCWithIndexUnit` *with alias* `src`*. This application is formally notated as*

$$G_{sub}[\quad B = \{\texttt{ENT} = \texttt{src.ENT}\},\ \ IndivParam = \{\texttt{SRC\_P} = \texttt{src.P}\}\ ]$$

*Note that variable* `src.P`*, which is defined as new-element variable in predefined generative template* `PCWithIndexUnit`*, is turned into a parameter variable; it is bound to the page class that is generated by the main generative template through new-element variable* `SRC_P`*.*

*Graphically, $G_{sub}$ is notated as an application of predefined generative template* `PCWithIndexUnit`*. Parameter binding* `ENT=src.ENT` *is defined implicitly since parameter variable* `ENT` *of $G_{main}$ has the same name as parameter variable* `ENT` *of $G_{sub}$. Parameter binding* `SRC_P = src.P` *is notated explicitly by connecting these two variables with a line.*

## 6.3 Complex transformers

Transformers as explained so far are simple in the sense that the combination of a query template and a generative template basically expresses a single loop over the query template's flat result relation, regardless of whether the query template or the generative template are complex on their own. This is insufficient as soon as a modelling task requires to generate scheme elements in nested loops. Further, simple transformers are not capable of performing modelling tasks subsequently.

To overcome these limitations, this chapter introduces transformers utilizing nested relations, which are described in Section 6.3.1, and cascading transformer applications, which are explained in Section 6.3.2.

### 6.3.1 Transformers utilizing nested relations

Many scheme elements are containers, which embed several other scheme elements. If a set of such containers shall be generated, where each embeds a set of components that shall be generated as well, some kind of nested loop is required.

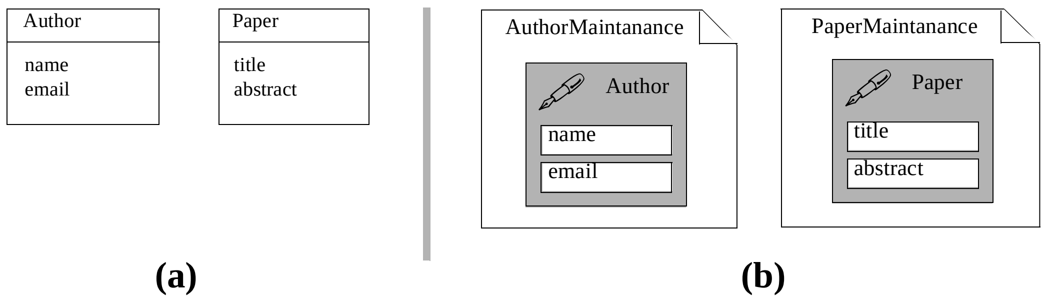


Figure 6.12: (a) Content scheme with two entity types; (b) Hypertext scheme with two page classes, each comprising an entry unit with several entry fields.

**Example 78** *Suppose that we want to define a transformer* `EntryUnitForEnt` *that generates entry units based on entity types such that (1) for each entity type, a page class with an entry unit is generated, and (2) subsequently, for each attribute of such an entity type, an equally named entry field is generated within the corresponding entry unit. Fig. 6.12 illustrates a sample of such a transformation, where the left part depicts the entity types and the right part depicts the corresponding page classes with the entry units.*

#### Concept

In order to support generation of scheme elements in nested loops, transformers that utilize nested relations are introduced. A nested loop is expressed in that (1) the query part is divided into a main query template and one or more nested query templates, and (2) the generative part is correspondingly divided into a main generative template and one or more nested generative templates. The main query template returns a nested relation whose tuples comprise a set of atomic values, one for each result variable of the main query template, and a set of relation-valued attributes, one for each nested query template. The overall generative part is instantiated iteratively for each such complex tuple $t$. The main generative template processes the atomic values of $t$, whereas a nested generative template is iteratively instantiated for each tuple of a relation-valued attribute of $t$.

For each nested query template of the query part, the generative part must provide a corresponding nested generative template. This correspondence is expressed as follows: Each relation-valued attribute is named after the nested query template, i.e. either after the nested query template's name or, if specified, after its alias. Correspondingly, the name (or, if specified, the alias) of a nested generative template determines for which relation-valued attribute it is to be processed. We start with an example and then discuss the particularities of nested query templates and nested generative templates.

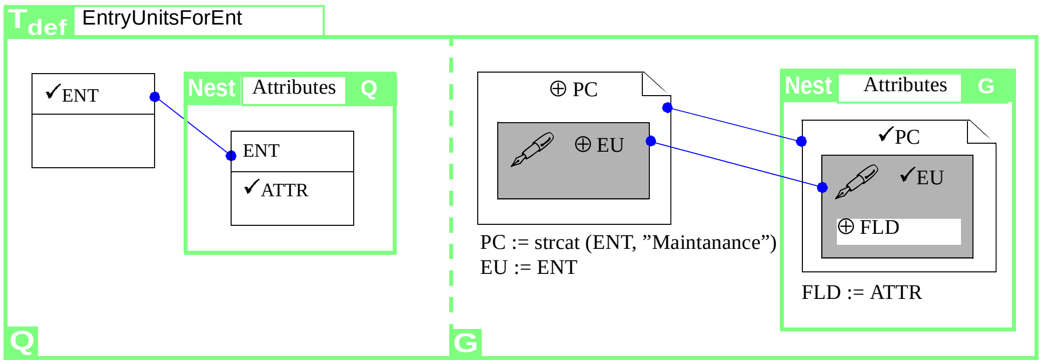


Figure 6.13: Definition of transformer `EntryUnitsForEnt` using a nested query template `Attributes`.

**Example 79** *Fig. 6.13 illustrates transformer* `EntryUnitsForEnt`*, which generates page classes and entry units comprising entry fields according to entity types comprising attributes.*

*The main query template selects entity types* `ENT` *and further applies a nested query template named* `Attributes`*. This name is user-defined and represents the name of the relation-valued attribute storing the* *nested query's result. Nested query template* `Attributes`*, which selects attributes* `ATT` *of an entity type* `ENT`*, is correlated to the main query template through application-specific constraint* `ENT` = `Attributes.ENT`*. Hence, for each entity type* `ENT` *selected by the main query template, relation-valued attribute* `Attributes` *contains the attributes of this entity type* `ENT`*.*

*For each tuple of the main query template's result, the generative template is processed as follows: (1) The main generative template generates a page class* `PC` *with an entry unit* `EU`*, both named after entity type* `ENT`*; (2) Subsequently, the nested generative template processes the relation-valued attribute* `Attributes`*. Thereby, page class* `PC` *and entry unit* `EU` *just generated serve as parameters, and for each tuple of relation-valued attribute* `Attributes`*, an entry field* `FLD` *is generated within entry unit* `EU`*.*

Syntactically, nested query templates and nested generative templates correspond to sub-query templates and generative sub-templates as explained so far, respectively. However, the concept of nested query templates and nested generative templates is orthogonal to that of sub-query templates and generative sub-templates, respectively. Their purpose and semantics differs as follows:

A nested query template is object-describing. It represents a relation-valued attribute whose extent may depend on result variables of the main query template but the nested query template does not constrain these variables. Consequently, a nested query template defining the extent of a relation-valued attribute of a tuple $t$ is evaluated after the main query template has retrieved the atomic values of $t$. A sub-query template, in contrast, is object-defining as it is used for constraining the main query template. Furthermore, the "result relation" of a sub-query template is used only for checking constraints but is not included in the main query template's result.

A nested generative template is usually instantiated several times for each single instantiation of the main generative template. Thus, for each tuple $t$ of the query template's result relation, after the main generative template has been processed for $t$'s atomic values, a nested generative template is instantiated iteratively for each tuple of a relation-valued attribute of $t$. A generative sub-template is instantiated after the main generative template, too, yet only once.

Since nested query templates are orthogonal to sub-query templates, both can be combined arbitrarily to a tree. Note, however, that sub-query templates that comprise nested query templates, though technically admissible, do not make sense since the "result relation" of a sub-query template is never included in the overall query's result. Nested generative templates and generative sub-templates are orthogonal, too, and can also be combined arbitrarily to a tree.

### Nested query templates: formal notation and semantics

Formally, the application of a query template $Q_{nest}$ as a nested query template within a main query template $Q_m$ is notated as `Nest` $Q_{nest}[ASC]$, where $ASC$ is a set of application-specific constraints known from sub-query templates. However, variables of nested query templates must not be correlated to a non-result variable $v_{nr}$ of $Q_m$. Otherwise, the extension of the nested relation would be ambiguously defined. Since many valid bindings of a non-result variable $v_{nr}$ may exist for a particular result tuple $t$, it would not be clear which of these values is to be considered for constraining the nested query template.

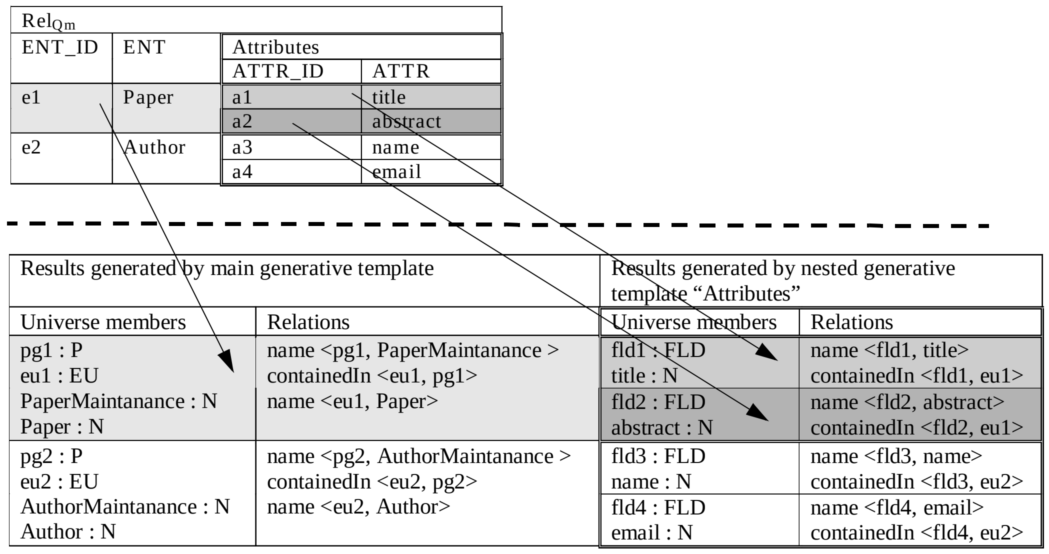
A nested query template `Nest` $Q_{nest}[ASC]$ applied in a main query template $Q_m$ represents a relation-valued attribute of $Q_m$ that is named after $Q_{nest}$. The result of $Q_m$ is a nested relation of the form $Rel_{Qm}(v_{rm1}, \ldots, v_{rmx}, Rel_{Qnest}(v_{rqn1}, \ldots, v_{rqny}))$. Variables $v_{rm1}, \ldots, v_{rmx}$, which represent the set of result variables of $Q_m$, denote the set of atomic values of $Rel_{Qm}$. $Rel_{Qnest}$ denotes a relation-valued attribute of $Rel_{Qm}$. Variables $v_{rqn1}, \ldots, v_{rqny}$ represent the set of result variables of $Q_{nest}$, which become the attributes of the nested relation.

A main query template $Q_m$ that applies a nested query template `Nest` $Q_{nest}[ASC]$ is evaluated as follows: First, query template $Q_m$ is evaluated while neglecting any nested query template. It achieves a result relation $Rel_{Qm}$, where for each tuple the atomic-valued attributes have assigned a value. Then, for each such tuple $t$, nested query template `Nest` $Q_{nest}[ASC]$ is evaluated and the result is assigned to relation-valued attribute $Rel_{Qnest}$ of $t$. Due to application-specific constraints, the extent of relation-valued attribute $Rel_{Qnest}$ depends on the atomic values of tuple $t$.

**Example 80** *Reconsider transformer* `EntryUnitsForEnt` *shown in Fig. 6.13. Main query template* $Q_m$ *selects entity types and* *applies a nested query template $Q_{nest}$ labelled* `Attributes`*, which selects attributes of entity types. Thus, the main query template achieves a nested relation* $Rel_{Qm}(\texttt{ENT\_ID}, \texttt{ENT}, \texttt{Attributes}(\texttt{ATTR\_ID}, \texttt{ATTR}))$*.*

*Suppose that query template $Q_m$ is applied to the content scheme depicted in Fig. 6.12(a). First, the atomic attributes* `ENT_ID` *and* `ENT` *of $Rel_{Qm}$ are evaluated whereas relation-valued attribute* `Attributes` *remains empty. Thus, $Rel_{Qm}$ contains tuples* $\langle \texttt{e1}, \texttt{Paper}, \{\}\rangle$ *and* $\langle \texttt{e2}, \texttt{Author}, \{\}\rangle$*. Then, for each such tuple $t$, query template* `Nest` $Q_{nest}[\texttt{ENT\_ID} = \texttt{Attributes.ENT\_ID}]$ *is evaluated and the result relation is assigned to the respective relation-valued attribute* `Attributes`*. Thus, for the tuple with* $\texttt{ENT\_ID} = \texttt{e1}$*, relation-valued attribute* `Attributes` *comprises the tuples* $\langle \texttt{a1}, \texttt{title}\rangle$ *and* $\langle \texttt{a2}, \texttt{abstract}\rangle$*; for the tuple with* $\texttt{ENT\_ID} = \texttt{e2}$*, relation-valued attribute* `Attributes` *comprises the tuples* $\langle \texttt{a3}, \texttt{name}\rangle$ *and* $\langle \texttt{a4}, \texttt{email}\rangle$*.*

*The overall result is illustrated in the upper part of Fig. 6.14. The atomic attributes of result relation $Rel_{Qm}$ are depicted in the first two columns, relation-valued attribute* `Attributes` *is depicted in the third column.*



| $Rel_{Qm}$ | | | |
|---|---|---|---|
| ENT_ID | ENT | Attributes | |
| | | ATTR_ID | ATTR |
| e1 | Paper | a1 | title |
| | | a2 | abstract |
| e2 | Author | a3 | name |
| | | a4 | email |

| Results generated by main generative template | | Results generated by nested generative template “Attributes” | |
|---|---|---|---|
| Universe members | Relations | Universe members | Relations |
| pg1 : P<br>eu1 : EU<br>PaperMaintanance : N<br>Paper : N | name <pg1, PaperMaintanance ><br>containedIn <eu1, pg1><br>name <eu1, Paper> | fld1 : FLD<br>title : N | name <fld1, title><br>containedIn <fld1, eu1> |
| | | fld2 : FLD<br>abstract : N | name <fld2, abstract><br>containedIn <fld2, eu1> |
| pg2 : P<br>eu2 : EU<br>AuthorMaintanance : N<br>Author : N | name <pg2, AuthorMaintanance ><br>containedIn <eu2, pg2><br>name <eu2, Author> | fld3 : FLD<br>name : N | name <fld3, name><br>containedIn <fld3, eu2> |
| | | fld4 : FLD<br>email : N | name <fld4, email><br>containedIn <fld4, eu2> |

Figure 6.14: Nested relation achieved by application of `EntryUnitsForEnt`

### Nested generative templates: formal notation and semantics

Formally, the application of a nested generative template $G_{nest}$ within a main generative template $G_m$ is notated as `Nest` $G_{nest}[B]$. The name of $G_{nest}$ represents the name of the relation-valued attribute to process, and $B$ is a set of parameter-variable bindings as known from applications of generative sub-templates. Each parameter binding may bind a parameter variable of $G_{nest}$ either to (a) a parameter variable of $G_m$, which is in the scope of $G_{nest}$, (b) to a new-element variables of $G_m$, which will have assigned a value at the time when the nested generative template is instantiated, or (c) to a result variable of the corresponding nested query template $Q_{nest}$.

The meaning of a main generative template $G_m$ that applies a nested generative template `Nest` $G_{nest}[B]$ is defined as follows: Let $Rel_{Qm}$ be a nested relation comprising a relation-valued attribute labelled $Rel_{Qnest}$. Then, $G_m$ is iteratively instantiated for each tuple $t_m$ of $Rel_{Qm}$ as usual, where $G_m$ refers only to the atomic values or $t_m$. Further, with each instantiation of $G_m$, $G_{nest}$ is iteratively instantiated for each tuple $t_{nest}$ of relation-valued attribute $Rel_{Qnest}$ of $t_m$.

**Example 81** *Reconsider transformer* `EntryUnitsForEnt` *depicted in Fig. 6.13. Main generative template $G_m$ generates page classes* `PC_ID` *named* `PC` *with entry units* `EU_ID` *named* `EU` *based on entity types provided by relation $Rel_{Qm}$*(`ENT_ID`, `ENT`, `Attributes`(`ATTR_ID`, `ATTR`))*. Nested generative template* `Attributes` *is applied within $G_m$ as*

$$\texttt{Nest Attributes}\ \left[\ \begin{array}{l}\texttt{EU\_ID} = \texttt{Attributes.EU\_ID},\\ \texttt{EU} = \texttt{Attributes.EU},\\ \texttt{PC\_ID} = \texttt{Attributes.PC\_ID},\\ \texttt{PC} = \texttt{Attributes.PC}\end{array}\ \right]$$

*Page class* `PC` *and the entry unit* `EU` *generated by $G_m$ are passed as parameters to generative template* `Attributes`*, which generates therein entry fields in an iterative manner, i.e. for each attribute provided by nested relation* `Attributes`(`ATTR_ID`, `ATTR`)*, a corresponding entry field* `FLD` *is generated.*

*Suppose that generative template $G_m$ is instantiated based on the first tuple $t_m = \langle$*`e1`, `Paper`, $\{\langle$`a1`, `title`$\rangle$, $\langle$`a2`, `abstract`$\rangle\}\rangle$ *of nested relation $Rel_{Qm}$ depicted in the upper part of Fig. 6.14. Then, a page class* `pg1` *named* `PaperMaintenance` *is generated, which contains a newly* *generated entry unit* `eu1` *named* `Paper`*. Subsequently, for each of the two tuples of relation-valued attribute* `Attributes`*, nested generative template* `Attributes` *is instantiated, achieving entry fields* `fld1` *named* `title` *and* `fld2` *named* `abstract`*, both being embedded within entry unit* `eu1`*.*

*The lower part of Fig. 6.14 illustrates the result of this instantiation. The arrows and the gray shaded areas show which universe members and relations have been generated from which part of the input tuple. The second input tuple* $\langle$`e2`, `Author`$\rangle$ *is processed analogously. The complete result of this application is depicted graphically in Fig. 6.12(b).*

### General and shorthand graphical notation

Two ways for notating nested templates, i.e. nested query templates and nested generative templates, are provided: (1) In the general graphical notation, which has been used in the motivating example of Fig. 6.13, nested templates are notated explicitly. (2) In order not to clutter template definitions with too many nested templates, a shorthand notation is provided, where applications of nested templates are expressed implicitly. This shorthand notation captures the typical use cases of nested templates, but also has some limitations. We start with describing the general graphical notation. Then, we show how nested templates are typically used, introduce the shorthand notation, and discuss its limitations.

**General graphical notation.** In the general graphical notation, a nested query template is notated like a sub-query template, i.e. within a separate box labelled either `Q` or $\texttt{Q}_{\texttt{app}}$, yet keyword "`Nest`" is declared in the upper left corner of the box. The name of the sub-query template is mandatory and may be complemented by an alias. If only the name is specified, then this name serves as label for the relation-valued attribute. Otherwise, this attribute is named after the alias. Analogously, a nested generative template is notated within a separate box labelled either `G` or $\texttt{G}_{\texttt{app}}$ together with keyword "`Nest`" declared. Again, the name of the nested generative template (or the alias, if specified) identifies the relation-valued attribute to be processed.

**Typical use of nested templates.** Experience has shown that a nested query template $Q_{nest}$ further describes the atomic attributes of main query template $Q_m$, i.e. $Q_{nest}$ selects *further* scheme elements that fulfill *additional* constraints. Thus, a nested query template is typically structured as follows:

1. The set of result variables of the main query template $Q_m$ represents the context in which a nested query template $Q_{nest}$ is evaluated. For this purpose, each result variable of $Q_m$ is repeated within $Q_{nest}$ as a non-result variable and correlated to the corresponding result variable of $Q_m$.

2. In this context, the nested query template defines result variables of its own, which make up the attributes of the nested relation.

**Example 82** *Reconsider the query template of transformer* `EntryUnitsForEnt` *depicted in Fig. 6.13. The main query template selects entity types as denoted by result variable* `ENT`*. The nested query template selects attributes of entity types, yet in the context an entity type* `ENT` *already selected by the main query template. This is expressed in that result variable* `ENT` *of the main query template is repeated as non-result variable in the nested query template and correlated accordingly.*

Similarly, a nested generative template $G_{nest}$ embeds new scheme elements in the scheme elements that have been generated previously by a main generative template $G_m$ and that are passed as parameters. Consequently, new-element variables of $G_m$ are repeated as parameter variables in $G_{nest}$ and are bound accordingly.

**Example 83** *Reconsider the generative template of transformer* `EntryUnitsForEnt` *depicted in Fig. 6.13. The main generative template $G_m$ generates a page class and an entry unit as denoted by result variables* `PC` *and* `EU`*, respectively. In the context of such a generated page class and entry unit, the nested generative template $G_{nest}$ generates new entry fields.*

Thus, a nested query template $Q_{nest}$ typically is a copy of the main query template $Q_m$ yet with the following differences: (1) Result-variables of $Q_m$ are non-result variables in $Q_{nest}$ but are bound to the corresponding result variables of $Q_m$. (2) $Q_{nest}$ defines result variables of its own. Similarly, a nested generative template $G_{nest}$ is a copy of the main generative template $G_m$, but (1) new-element variables of $G_m$ are parameter variables in $G_{nest}$, and (2) $G_{nest}$ defines new-element variables of its own.

**Shorthand notation.** Based on the rules identified above, a shorthand notation is introduced, which allows to express nested templates implicitly. The advantage of this notation is that it is more concise and that it visually corresponds to the schemes that match the template.

In the shorthand notation, the result-variables of a nested query template are simply notated within the main query template, but the name of the nested query template is attached in brackets in order to distinguish these variables from other variables of the main query template. Similarly, new-element variables of a nested generative template are notated in the main generative template, but have the name of the nested generative template attached in brackets. Variables that have the name of a nested template attached are subsequently referred to as *nest variables*.

This shorthand notation is interpreted by a pre-compiler that derives the actual main query template and nested query template as follows: The actual main query template is defined by simply ignoring all nest-variables, whereas a nested query template $Q_{nest}$ is derived from (1) a group of nest-variables that have the label of $Q_{nest}$ attached in brackets, and (2) a set of non-result variables, one for each result variable of the main query template, and each bound to its corresponding result variable. The derivation of a main generative template and a nested generative template works analogously.

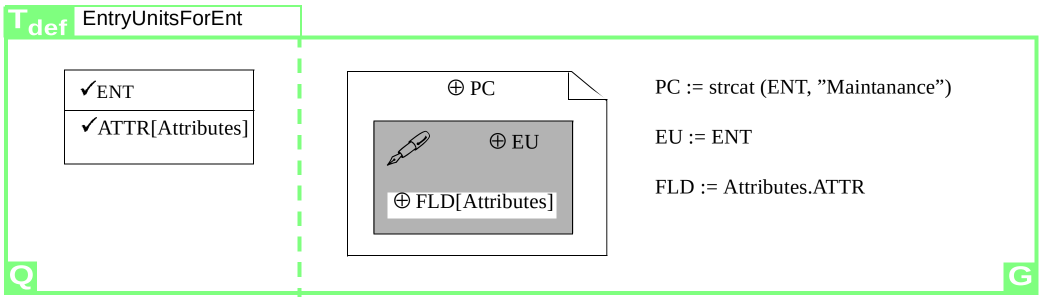


Figure 6.15: Transformer `EntryUnitsForEnt`, nested templates for nested relations defined implicitly.

**Example 84** *Fig. 6.15 depicts transformer* `EntryUnitsForEnt` *using the shorthand notation for nested templates. In the query template, a nested query template* `Attributes` *is implicitly defined through nest-variable* `ATTR`[*Attributes*]. *Similarly, the generative template implicitly* *defines a nested generative template* `Attributes` *through nest-variable* `FLD[Attributes]`.

*This definition of transformer* `EntryUnitsForEnt` *using the shorthand notation is equivalent to the definition depicted in Fig. 6.13. However, the shorthand notation visually corresponds better to the scheme of Fig. 6.12 than the other notation does.*

**Limitations of the shorthand notation.** Whereas the shorthand notation of nested generative templates is as expressive as the notation where these nested templates are expressed explicitly, the shorthand notation of nested query templates covers only those cases where (1) the non-result variables of the nested query template corresponds to the set of result variables of the main query template, and (2) where corresponding variables are strictly correlated. This is the common case. However, the following example cannot be expressed in the shorthand notation:

**Example 85** *Suppose that we want to modify the example above such that, besides the entry fields for the attributes of the corresponding entity type, each entry unit provides also entry fields for the attributes of an entity type named "*`Conf`*". In the explicit notation, this could be expressed by relaxing application-specific constraint "*`ENT = Attributes.ENT`*" to "(*`ENT = Attributes.ENT`*) OR (*`Attributes.ENT = `*"Conf")". This could not be expressed in the shorthand notation, which imposes a strict correlation between variables* `ENT` *and* `Attributes.ENT`*, i.e. a constraint of the form* `ENT = Attributes.ENT`.

### 6.3.2 Cascading transformers

Modelling tasks often suggest to perform other modelling tasks subsequently such that the result of the first task is input for the second task. For example, after having defined a set of page classes with index units in a first modelling task, these index units get linked according to relationships between the underlying entity types in a subsequent modelling task.

For performing subsequent modelling tasks, cascaded transformers are introduced, i.e. main transformers that automatically apply subsequent transformers. This technique is straightforward since subsequent modelling tasks could

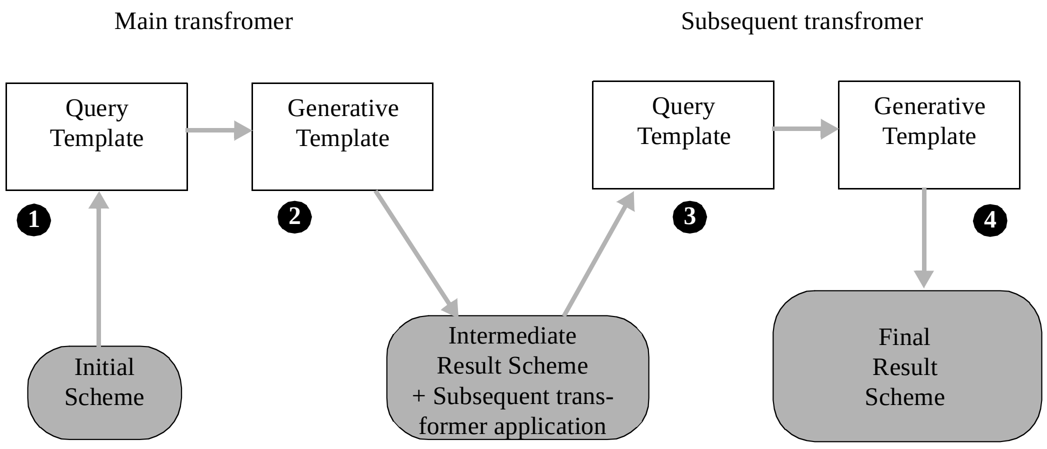


Figure 6.16: Evaluation of cascading transformers.

be performed in that the user manually applies one transformer after another. However, cascaded transformers are capable of performing subsequent modelling tasks all at once without requiring user interaction in-between.

Fig. 6.16 illustrates the architecture of cascaded transformers, i.e. the flow of results as indicated by the arrows, and the sequence of evaluation as indicated by the numbers. The transformation starts with applying the first transformer to an initial scheme. This achieves an intermediate result scheme that has been extended and refined and that is input for a subsequent transformer. This subsequent transformer application is triggered automatically and achieves the final result.

As a prerequisite, we start with explaining how subsequent modelling tasks can be supported by applying subsequent transformers manually. Then, we show how to define transformers that automatically trigger the application of a subsequent transformer.

### Applying subsequent transformers manually

Suppose a modelling task that aims at defining interconnected index units as follows: First, for each entity type of a content scheme, a page class with an index unit shall be generated. Second, the index units just generated shall be connected by contextual links in accordance to relationships between the entity types that serve as their respective content source. If an entity type $S$ is related to an entity type $T$ via a relationship role of multiplicity "*", then

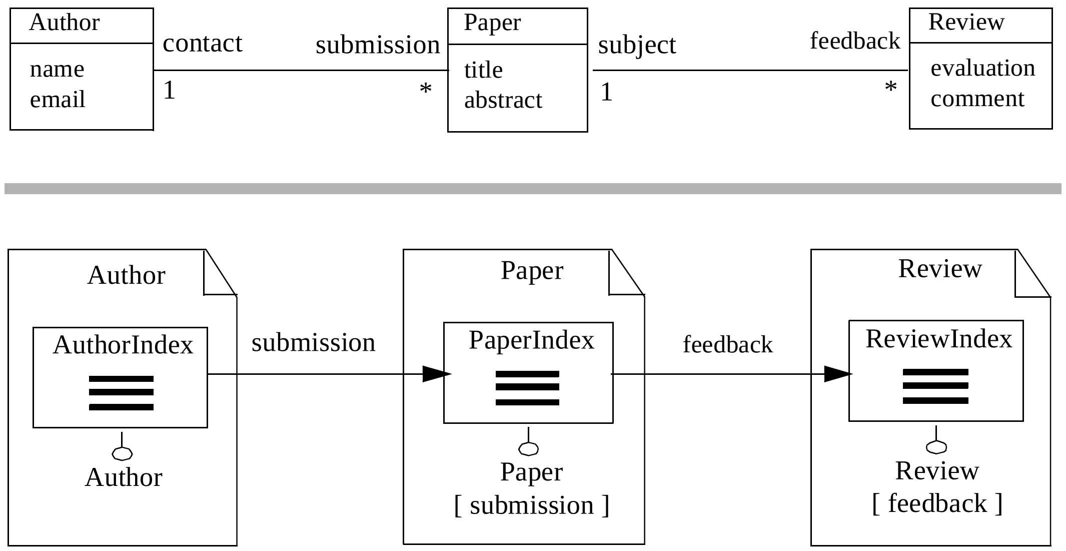


Figure 6.17: Upper part: content scheme with two 1:*-relationships; lower part: corresponding hypertext scheme with linked index units

a link shall be drawn from the new index unit referring to $S$ to the new index unit referring to $T$. The upper part of Fig. 6.17 depicts a content scheme that serves an input for this task. The lower part illustrates the hypertext scheme that results from performing this task.

This modelling task can be performed by subsequently applying two transformers, one that selects entity types and generates page classes with index units therefrom, and a subsequent one that selects pairs of index units and connects them. The first transformer corresponds to transformer `IndexPCForEnt`. The second transformer is referred to as `LinkIUs` and is shown in the upper part of Fig. 6.18. It connects those pairs of index units `SIU` and `TIU` which have their underlying entity types connected via a role `TROLE` of multiplicity "*".

The lower part of Fig. 6.18 depicts the scheme after having processed an application of transformer `IndexPCForEnt` and after having defined that transformer `LinkIUs` shall be applied subsequently. Processing this subsequent application then achieves the hypertext scheme as shown in the lower part of Fig. 6.17.

Thus, in order to define a transformer `ConnectedIUsforEnts` that performs above mentioned modelling task all at once, transformer `IndexPCForEnt`

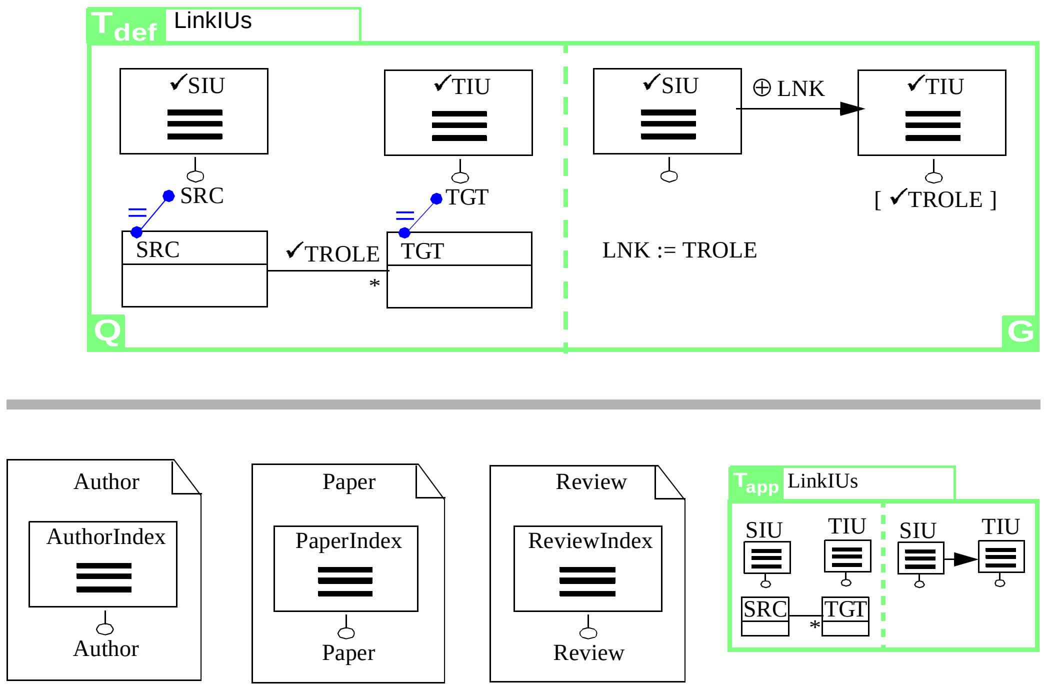


Figure 6.18: Upper part: Transformer `LinkIUs`; Lower part: scheme after application of `IndexPCForEnt` and before application of `LinkIUs`

has only to be extended such that it generates an application of transformer `LinkIUs`, which then is processed automatically, i.e. without requiring further user interaction.

### Generating applications of subsequent transformers

A subsequent application of a transformer is defined in that a graphical application-symbol, which up to now has only been used for applying transformers manually, is now generated by a transformer. The *TBE* directives for specifying transformer applications are treated as a first order scheme elements that can be generated like any other scheme element. Each transformer application generated that way is then processed as if modelers had specified them manually.

The subsequent transformer application can be off-the-shelf or can be individualized. In an off-the-shelf application, the query template of the subsequent transformer considers the entire scheme, i.e. all scheme elements that have existed in the initial scheme as well as those just generated by the main transformer. With an individualized application, the subsequent transformer can be constrained such that it considers only specified parts of the intermediate scheme, e.g. only the scheme elements just generated. Such an individualization is expressed by *generating* application-specific constraint and expressions together with the generation of the application-symbol of the subsequent transformer. We start with an example without application-specific constraints to be generated and then extend the example in this direction.

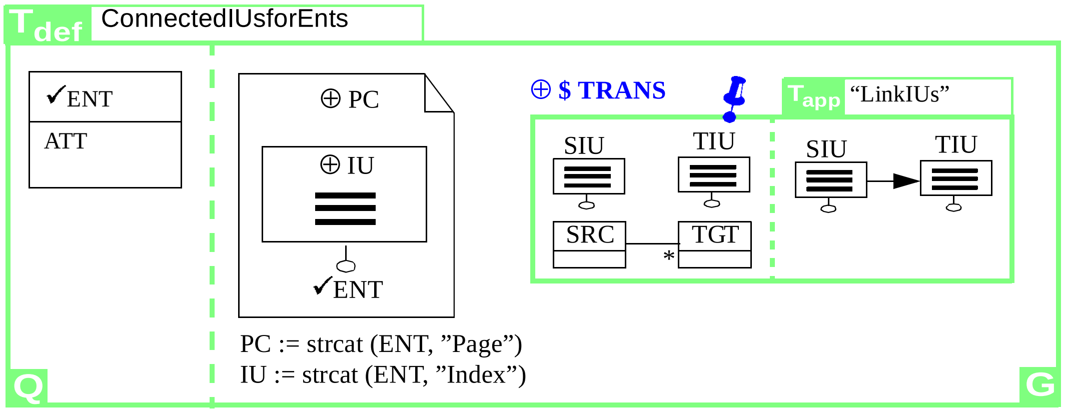


Figure 6.19: Definition of transformer `ConnectedIUsforEnts` generating a subsequent application of transformer `LinkIUs`

**Example 86** *Fig. 6.19 depicts the definition of transformer* `ConnectedIUsforEnts`*, which generates page classes and index units and additionally applies transformer* `LinkIUs`*. For that purpose, a graphical application of transformer* `LinkIUs`*, which is represented by new-element variable* `TRANS`*, is generated. Note that this new-element variable is pinned (as expressed by* $ `TRANS`*) such that only one transformer application is generated instead of generating a separate application for each page class.*

*Suppose that transformer* `ConnectedIUsforEnts` *is applied to the content scheme depicted in the upper part of Fig. 6.17. Although this application achieves the final result as illustrated in the lower part of Fig. 6.17 without requiring further user interaction, it is internally processed in the following two steps: (1) First, an intermediate result* *scheme is achieved, where additional page classes and index units together with an application of transformer* `LinkIUs` *have been generated. This intermediate scheme is depicted in the lower part of Fig. 6.18. (2) Then, the generated application of transformer* `LinkIUs` *is processed and achieves the final result.*

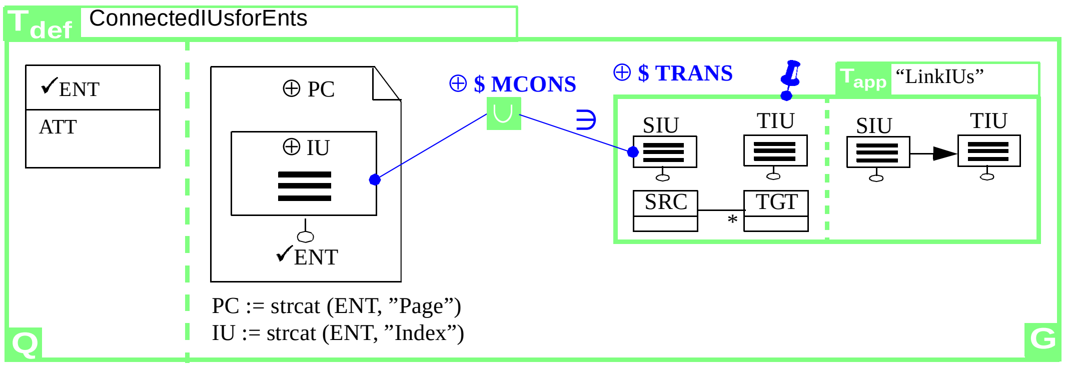


Figure 6.20: Transformer `ConnectedIUsforEnts`, generating an individualized application of transformer `LinkIUs`

In the example above, an off-the-shelf application of transformer `LinkIUs` is generated such that the the entire intermediate scheme is re-considered. Thus, if there were any index units which have already existed before applying `ConnectedIUsforEnts`, these index units would be connected, too. However, it may be desired that transformer `LinkIUs` shall consider only the newly generated index units. This can be expressed by generating application-specific constraints together with the generation of the subsequent transformer application.

**Example 87** *Fig. 6.20 depicts a variant of transformer* `ConnectedIUsforEnts`*, which generates a graphical application together with application-specific constraints. In this variant, subsequent transformer* `LinkIUs` *creates only links from index units that have just been generated by the main transformer. This is expressed through generating a membership constraint that constrains variable* `SIU` *of subsequent transformer* `LinkIUs` *to accept only such index units. This membership constraint is represented by pinned new-element variable* `MCONS` *and is generated once with the graphical application of* *subsequent transformer* `LinkIUs`. *The line connecting new-element variable* `IU` *to membership constraint* `MCONS` *expresses that each newly generated index unit* `IU` *is a member of this membership constraint.*

# Chapter 7

# Realization

## Contents

*TBE* by itself is not a web modelling language of its own but rather a set of language constructs for extending existing web modelling languages towards defining and applying transformers in a by-example manner. Thus, we do not strive for implementing a new CASE tool that supports the generation of web applications out of web schemes but to incorporate *TBE* 's language constructs into existing CASE tools for web application development.

This chapter deals with enabling existing CASE tools for defining and applying by-example transformers. This concerns two aspects, namely (1) the specification of by-example transformers and their application to a scheme, and (2) the implementation of the TBE-QML language, which has been used for defining the semantics of by-example transformers. These two aspects are covered in Sections 7.1 and 7.2, respectively.

## 7.1 Specifying transformers for existing CASE tools

In the previous chapters, the *TBE* language has been introduced. In this section, we discuss the aspect of how to incorporate *TBE*'s language constructs into existing CASE tools, i.e. of how to support the definition and application of by-example transformers therein.

In Section 7.1.1, we start with requirements analysis. Then, three ways of enabling CASE tools for supporting *TBE* are discussed, namely (1) adapting a CASE tool's source code, (2) utilizing a CASE tool's extension interface, and (3) developing an external add-on for defining and applying transformers. These ways are covered in Sections 7.1.2, 7.1.3, and 7.1.4, respectively.

### 7.1.1 Requirements

*TBE* comprises two sorts of language constructs: (1) The graphical notation of scheme elements as used by the respective CASE-tool for that transformers shall be specified, and (2) the *TBE* -directives, i.e. the symbols, comparison constraints, construction expressions, transformer applications, subtemplates, and so on. For enabling a CASE-tool for defining and applying transformers, the following two minimum requirements must be fulfilled:

1. For each scheme element, a corresponding scheme element skeleton must be provided, i.e. a scheme element that allows to specify variables in place of concrete values.

2. It must be possible to specify *TBE* -directives.

As discussed later in this section, these minimum requirements are fulfilled by each way of supporting *TBE* in the realm of an existing CASE tool. However, for fulfilling these requirements, the different ways may follow different approaches, which are more or less practicable.

### Requirements for defining transformers

Although templates making up a transformer definition look similar to schemes, templates and schemes have different purposes, and the requirements for specifying templates differs from those for specifying schemes. The following particularities of templates should be considered:

*Support for scheme element skeletons.* Although scheme element skeletons used in templates look like scheme elements used in schemes, the CASE tool should distinguish the definition of scheme element skeletons from that of scheme elements. Scheme element skeletons differ from scheme elements in that the former comprise variables, whereas the latter comprise concrete values. Thus, the CASE tool should consider the particularities of variables. For example: Each variable is of one of the sorts "result variable", "non-result variable", "parameter variable", and "new-element variable". New-element variables typically have construction expressions attached. Each variable has a name, i.e. a "string-value", even if the property it represent is of a numeric type.

*Graphical specification of directives.* Specifications of templates differ from those of schemes in that the former comprise *TBE*-directives. Although these directives can be specified in a textual syntax as well, the practicability of *TBE* greatly depends on a proper graphical representation. This requires to introduce new graphical shapes or to adapt existing graphical shapes. For example, constraints are often expressed by lines connecting scheme elements skeletons, and the graphical shapes must be extended by ports for attaching such lines.

*Overall view.* In contrast to schemes, which usually assign scheme elements to different sub-schemes defined at different places, templates shall provide an overall view, which requires to have all sorts of scheme element skeletons available at one place. Otherwise, the definition of a template would be scattered to different places and conciseness would get lost. Suppose, for example, that a query template shall be defined which selects index units that have an entity type without attributes as content source. This requires to place an index unit skeleton and an entity type skeleton in the same template, even if an index unit is part of the hypertext sub-scheme and an entity type is part of the content sub-scheme.

Thus, templates require an organization different from schemes: A scheme is a large document comprising several sub-schemes and a multitude of separate forms for specifying properties. In contrast, a template is comparably small unit where all elements should be visualized at a glance. Further more, a template may comprise other correlated templates, whereas such a concept is usually not necessary for defining schemes.

*Visualization at a glance.* A CASE-tool should be capable of permanently visualizing variables that represent detailed scheme element properties, which are usually shown in separate forms. Such a visualization-at-a-glance then exposes all the template's semantics, which otherwise would have to be explored by opening separate forms. This requirement is similar to that of providing an overall view. However, visualization at a glance concerns the possibility of permanently visualizing details whereas an overall view concerns the organization in the large.

*Checking syntax and semantics.* Of course, the correctness criteria of templates differ from that of schemes. Thus, a CASE tool supporting the specification of transformers should provide for checking the correctness of transformers as defined by *TBE*. Such a correctness criterion is, for example, that result variables of a query template must properly match a parameter variable of the generative template. For assisting modelers in defining correct transformers, a CASE tool may provide many facilities, which are not discussed in this thesis.

### Requirements for applying transformers

Besides defining transformers, a CASE tool should provide for applying transformers in a practical manner. This concerns the usability of transformer applications and requires proper techniques of user interaction.

*Graphical applications of transformers.* Although transformer applications can be specified textually as well, graphical applications are more appropriate because both transformers and schemes are defined graphically. In a graphical application, techniques for individualizing a transformer's behavior like application-specific constraints could be specified graphically as well, and the CASE tool should allow to express such constraints in a convenient manner. For example, a membership constraint could be specified simply by selecting a set of scheme elements.

*Exploring the repository.* The CASE-tool should provide a repository of transformers that allows to quickly find appropriate transformers for modelling tasks to be performed. This suggests to split the possibly large number of transformers into different groups, to provide descriptions that allow to quickly grasp a transformer's purpose, and to zoom into the transformer definition for exploring its detailed semantics, if required.

### 7.1.2 Adapting a CASE-tool's source code

When striving for a commercial product, *TBE* should be incorporated directly into the CASE-tool. Obviously, if the developer has full control over the tool's software architecture, the requirements specified in the previous section can be fulfilled, and the CASE-tool could seamlessly integrate the development of web schemes with the definition and application of web scheme transformers.

In this thesis, however, we do not strive for a commercial product but for a prototype implementation that demonstrates the usefulness of transformers. From this point of view, adapting a CASE-tool's source code has the following disadvantages: (1) It requires to have the tool's source code available, yet vendors of CASE tools usually ship only the binaries. (2) If available, the source code is usually quite extensive and complex as a CASE tool is not only concerned with editing schemes but also with generating web applications therefrom. (3) The software architecture of the CASE tool is usually not prepared for things like an overall scheme. Introducing such a concept into this architecture may require great effort, even if only a prototypical implementation shall be achieved.

Thus, we decided not to adapt a CASE-tool's source code. Instead, we took the following two approaches: (1) We utilized the built-in extension facilities of a third-party CASE tool for incorporating *TBE*, and (2) we implemented an external editor that provides for editing schemes and for defining and applying transformers, whereas the generation of web applications is left to the CASE tool. These approaches are explained next.

### 7.1.3 Utilizing a CASE-tool's extension interface

CASE tools usually provide an extension interface, which allows to add user-defined properties in form of user-defined tag/value pairs or even to define new scheme elements together with graphical shapes for them. For example, WebRatio [141] can be extended by new modelling primitives (i.e. scheme elements) defined in Java or C#.

However, a CASE-tool's extension interface is usually not prepared to change the behavior of the diagram editor or of existing scheme elements. It is therefore unlikely that the diagram editor can be adapted such that variable names can be entered in places where the numeric values are expected or where fields are not editable at all, existing scheme elements are extended by ports for attaching lines, or such that the diagram editor provides an overall-scheme view.

Nevertheless, as soon as a CASE-tool provides for annotating scheme elements by tag/value-pairs, the minimum requirements necessary for specifying transformer definitions and applications are fulfilled. We demonstrate this by the example of the commercial CASE tool WebRatio [141], which is a tool for defining WebML schemes and which we use off-the-shelf for defining transformers for WebML schemes and for applying these transformers.

As described more detailed in [81], WebRatio can be enabled for defining templates basically by interpreting property values of scheme elements as variable names and by annotating constraints, expressions, transformer applications, etc. in form of user-defined tag/value-pairs. This "extension" requires only a minimal effort and works as follows:

- If a property of a scheme element is of a string-type and is editable, then the name of a variable representing this property can be entered directly in place of a property value. Otherwise, the scheme element is tagged by a tag/value-pair of the form `prop = varname`, where `prop` stands for the property and `varname` stands for the variable representing the property.

  **Example 88** *The name of an entity type is a property of a string-type and is editable. Thus, one can directly enter a variable named, for example,* `ENT` *in place of the name of an entity type. The cardinality specified at a relationship role, however, is a numeric value. Thus, in order to specify a variable named* `C` *representing the multiplicity of a relationship role, the role is tagged by a user-defined tag/value-pair of the form* `card` = `C`*. Similarly, in WebRatio scheme element IDs are non-editable. Thus, for specifying a variable* `ATT_ID` *representing the ID of an attribute, the attribute is tagged with* `id` = `ATT_ID`*.*

- Whether a variable is a result variable, a non-result variable, a parameter variable, or a new-element variable is encoded in the variable's name as follows: Result-variables and parameter variables start with "`R_`" and "`P_`", respectively. New-element variables start with "`NEW_`". All other variables are non-result variables.

- Transformer applications are expressed through a user-defined tag named `TApp`. The name of the transformer that shall be applied as well as application-specific constraints and expressions are notated in the tag's value. Constraints and expressions, both application specific and other ones, are notated as tag/value-pairs, too.

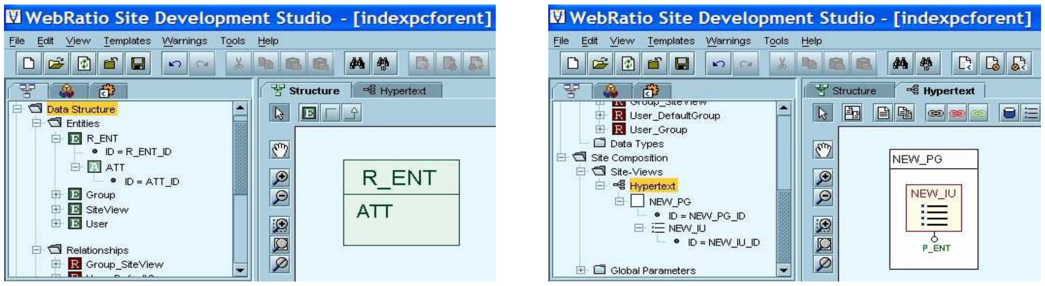


Figure 7.1: By-example transformer specified in WebRatio: Query template (left); Generative template (right)

**Example 89** *Fig. 7.1 illustrates transformer* `IndexPCForEnt` *defined in the third-party CASE-tool WebRatio. The query template is depicted in the left part. Variable* `R_ENT` *is a result variable representing an entity type name, whereas* `ATT` *is a non-result variable representing an attribute name. For both variables, corresponding ID-variables are defined in terms of a tag/value pairs, i.e.* `ID` = `R_ENT_ID` *and* `ID` = `ATT_ID`*, which are attached to the entity type and the attribute, respectively. The generative template is depicted in the right part. Variables* `NEW_PG` *and* `NEW_IU` *are new-element variables, whereas variable* `P_ENT` *is a parameter variable. Again, corresponding ID-variables are defined through attached tag/value-pairs.*

Note that WebRatio is used off-the-shelf and is therefore not aware of templates but only of schemes. Thus, the tag/value-pairs simply have no meaning to WebRatio, the "variables" are interpreted as string values, and of course, the annotated tag/value-pairs by itself do not change non-editable fields into editable ones. In a preprocessing step, however, the *TBE* -system interprets these annotations such that the values contained in non-editable fields get replaced by the variables as specified by the attached tag/value-pairs. After this preprocessing step variables are actually in place of concrete values.

| Requirement | Level of fulfillment | Comments |
|---|---|---|
| Support of scheme element skeletons | ~ | Only to a minimum |
| Graphical specification of TBE-directives | ✗ | Only textually |
| Overall-scheme view | ✗ | |
| Visualization at a glance | ✗ | |
| Checking transformer syntax and semantics | ~ | Only through an external tool |
| Graphical applications of transformers | ✗ | Only textually |
| Exploring the repository | ✗ | |

Figure 7.2: Requirements fulfilled when extending the third-party CASE-tool WebRatio using its standard extension facilities.

Fig. 7.2 illustrates that above mentioned technique of extending the third-party CASE-tool WebRatio fulfills the requirement introduced at the beginning of this section only to a minimum level. None of the requirements that would have imposed an adaptation of the graphical diagram editor can be fulfilled. Therefore, most of the semantics of a template is only annotated rather than defined in-place such that the visual correspondence between schemes and templates gets lost. Further, since a third-party CASE-tool used off-the-shelf is not aware of templates, it cannot assist modelers in defining transformers, e.g. by detecting type-mismatches or syntax errors in expressions. This has rather to be done through an external tool that actually interprets the “scheme” as a template.

Thus, this technique is not appropriate for demonstrating the practicability of transformers or for real projects. However, the technique has the advantage that it requires only a minimum effort to get a rough prototype of a tool for defining and applying transformers. As described in [81], we have used such a prototype as test environment during the implementation of TBE-QML. The development of the external scheme editor has thereby been decoupled from the implementation of TBE-QML.

### 7.1.4 Defining an external scheme editor

If it is not possible or impractical to extend an existing CASE-tool towards defining and applying transformers, one can decide to develop an external editor for the schemes of the CASE-tool, which usually have a normative format. Besides editing schemes, this editor can then be specifically tailored to the transformer definitions and applications. The schemes of both editors, that of the CASE-tool and the external one, are interchangeable.

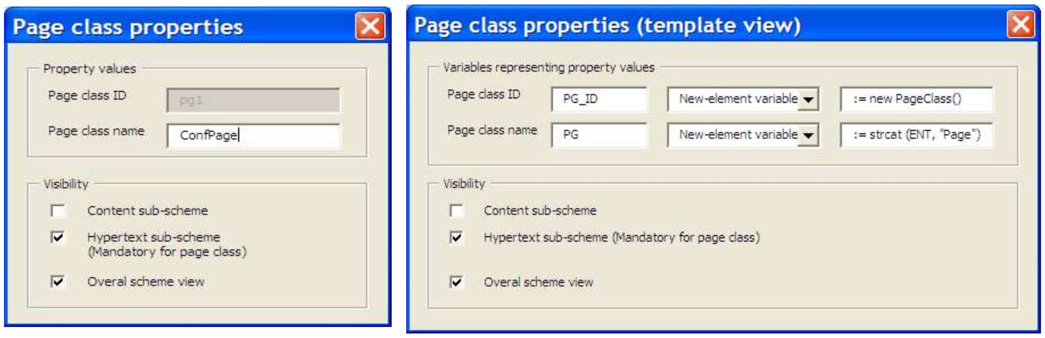
Of course, defining an external editor may require much development effort. However, the benefit of this approach is that it combines the advantages of (1) a third-party CASE-tool that provides for generating web applications for several target platforms out of conceptual schemes, and (2) an integrated *TBE* -system that supports conceptual modelling through defining and applying by-example transformers.

We demonstrate the use of an external editor for defining schemes by the example of *TransEd*, which is a prototype for a *TBE* -enabled external editor for schemes of the WebRatio CASE-tool. *TransEd* is currently under development as part of a master's thesis [136]. It provides a visual interface for editing WebML schemes based on the normative WebML DTD [138] and for defining and applying by-example transformers for WebML schemes. The schemes edited with WebRatio can subsequently be edited in *TransEd* and vice versa. Furthermore, transformers defined in *TransEd* as well as transformer applications specified in *TransEd* can also be viewed in WebRatio.

*TransEd* seamlessly integrates the specification of schemes and the specification of transformer definitions and applications. Besides fulfilling the minimum requirement of supporting scheme element skeletons, the tool also fulfills the other requirements introduced at the beginning of this section. In the following, we briefly show how these requirements are addressed by *TransEd*.

*TransEd* offers a scheme development mode and a transformer definition mode. In the scheme development mode, the editor is comparable to that of the commercial CASE-tool WebRatio. However, *TransEd* additionally supports the graphical application of transformers. In the transformer definition mode, the editor provides an overall-scheme view for defining templates. For defining transformers, two separate templates are displayed, one representing the query template and one representing the generative template.

*Scheme element skeletons.* Each scheme element is complemented by a scheme element skeleton that supports the definition of variables. For example, Fig. 7.3 illustrates forms for entering the properties of a scheme element



**Page class properties**

Property values

| | |
|---|---|
| Page class ID | pg1 |
| Page class name | ConfPage |

Visibility

- [ ] Content sub-scheme
- [✓] Hypertext sub-scheme (Mandatory for page class)
- [✓] Overal scheme view

**Page class properties (template view)**

Variables representing property values

| | | | |
|---|---|---|---|
| Page class ID | PG_ID | New-element variable | := new PageClass() |
| Page class name | PG | New-element variable | := strcat (ENT, "Page") |

Visibility

- [ ] Content sub-scheme
- [✓] Hypertext sub-scheme (Mandatory for page class)
- [✓] Overal scheme view

Figure 7.3: Page-class properties; Left: scheme development mode; Right: transformer definition mode.

of type “page class”. Please ignore section “Visibility” for the moment. The left part depicts the form in the mode of scheme development. This corresponds to a property form as usually provided by CASE tools. The right part illustrates the form in the mode of defining a transformer, where the page class is viewed as a page class skeleton and provides for entering variable names, specifying the kind of the variable, and attaching construction expressions, if desired.

*Graphical specification of directives.* In *TransEd*, TBE-directives are specified as defined by the *TBE*-language. This needs no explanation.

*Overall view.* In *TransEd*, transformers are defined in an overall scheme view, which may comprise scheme elements of different sub-schemes. Such an overall scheme view is a basic unit from which transformers and templates comprising sub-templates are built.

*Visualization at a glance. TransEd* allows to visualize each property, which is usually shown in a separate form, also directly in the graphical representation. Further, *TransEd* distinguishes different levels of visualizing details and allows thereby for zooming in and out. The graphical notation for visualizing details corresponds basically to that described in *TBE.*

**Example 90** *Fig. 7.4 depicts the definition of transformer* `DemoTrans`*, whose query template comprises page class skeletons as well as entity type skeletons. In the generative template, the property “Shown Attributes” is visualized. The transformer itself is only for illustration purpose and is therefore not explained.*

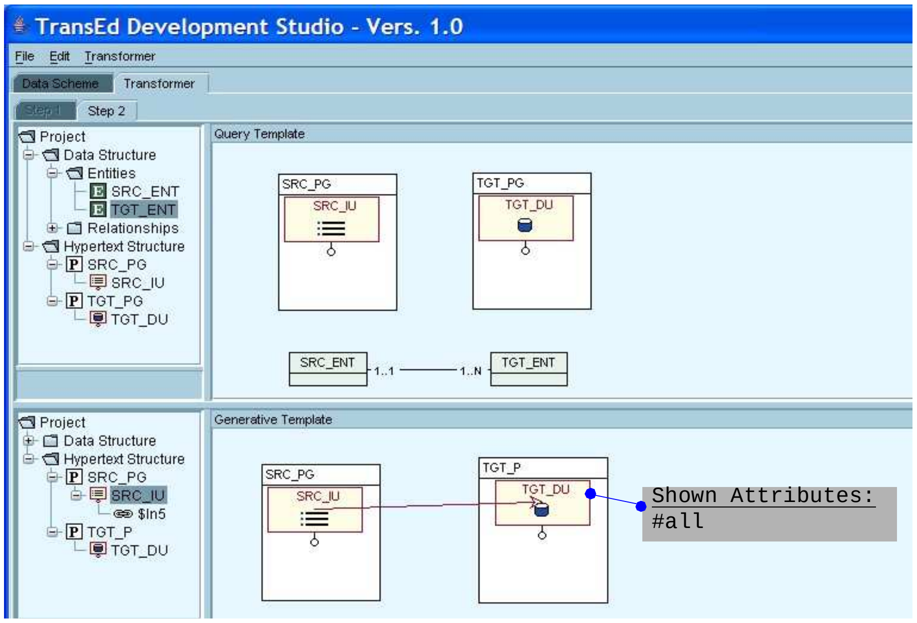
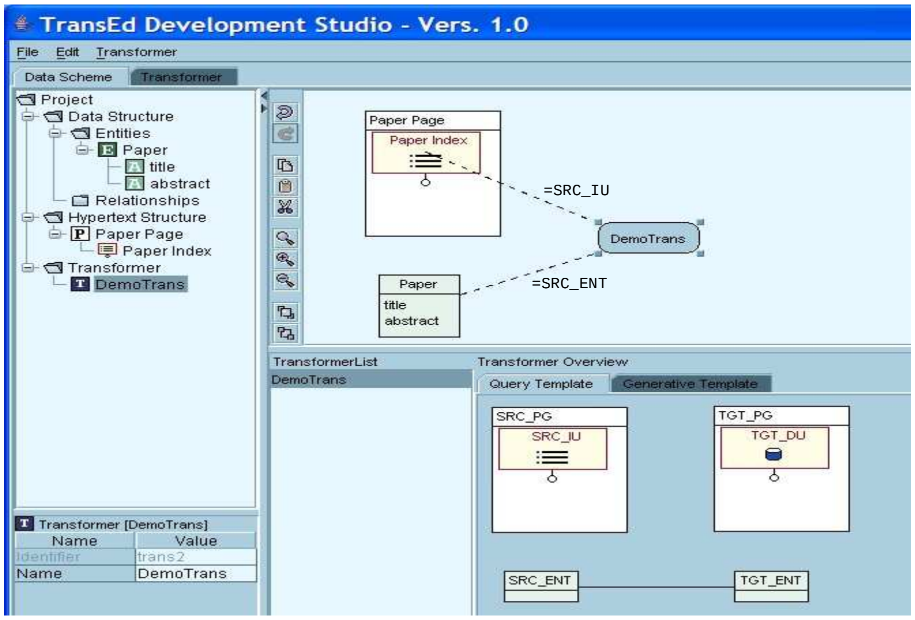


Figure 7.4: Transformer definition in *TransEd*: Overall scheme view.

*Checking syntax and semantics.* *TransEd* is aware of the syntax and semantics of transformers and can thereby assist modelers in defining transformers. Besides alerting errors, the system highlights interdependencies between different parts of a transformer. For example, if the mouse pointer is over a parameter variable of the generative template, the corresponding result variable of the query template is highlighted.

*Graphical applications of transformers.* The graphical application of transformers of *TransEd* differs from that of the *TBE* language in that currently no schematic representation of the transformer definition is provided within the graphical application. Thus, a scheme element can only be connected to the graphical application rather than to a (schematic) scheme element skeleton contained therein. This technique is not as comfortable as that suggested by *TBE* because the variable that is to be addressed by the line has to be annotated textually. However, this technique is much simpler to implement as no schematic scheme element skeletons have to be provided.

Further, in order to facilitate graphical applications of transformers whose templates mix scheme element skeletons of different sub-schemes, *TransEd* also provides an overall scheme view in the scheme development mode. This overall scheme view again may present scheme elements of different sub-schemes. This view is useful if several application-specific constraints that refer to elements of the content sub-scheme as well as to elements of the hypertext sub-scheme are to be specified. In order to express in which view a particular scheme element shall be presented, one can individually specify the visibility in the scheme element's property form.



Transformer [DemoTrans]

| Name | Value |
|---|---|
| Identifier | trans2 |
| Name | DemoTrans |

Figure 7.5: Graphical application of a transformer `DemoTrans` in *TransEd*; overall scheme view; repository explorer

**Example 91** *Fig. 7.5 illustrates the graphical application of transformer* `DemoTrans`*, which is represented by a rounded rectangle including the transformer's name. The transformer's definition is not repeated in this graphical shape but is shown in a separate window below. Application-specific constraints can be expressed by lines connecting scheme elements to the graphical application, and the transformer variable to be addressed is notated at the line.*

*For the graphical application depicted in Fig. 7.5, entity type* `Paper` *(content sub-scheme) as well as index unit* `Paper Index` *(hypertext sub-scheme) are to be attached to the graphical application of transformer* `DemoTrans`*. For that purpose, both scheme elements are shown in the overall scheme view. The lower part of Fig. 7.3, which depicts the* *property form of a page class, demonstrates how to specify an individual visibility for a particular scheme element.*

*Exploring the repository. TransEd* offers a transformer repository, which is implemented as a list of transformers imported into the current project. The definition of a transformer selected from this list is also shown and can be explored.

**Example 92** *The lower part of Fig. 7.5 illustrates the transformer repository, i.e. the list of transformers imported into the current project. The list comprises only one demo transformer* `DemoTrans`. *Its definition is split into a query template and a generative template, each illustrated next to the list in separate windows.*

| Requirement | Level of fulfillment | Comments |
|---|---|---|
| Support of scheme element skeletons | ✓ | |
| Graphical specification of TBE-directives | ✓ | |
| Overall-scheme view | ✓ | |
| Visualization at a glance | ✓ | |
| Checking transformer syntax and semantics | ✓ | |
| Graphical applications of transformers | ~ | No schematic transformer representation |
| Exploring the repository | ✓ | |

Figure 7.6: Requirements fulfilled by external scheme editor *TransEd*.

We have to point out that *TransEd* is a research prototype that aims at a demo development environment for evaluating the practicability of the *TBE* language. Thus, *TransEd* currently supports only a subset of the scheme elements provided by WebML. However, for web schemes utilizing this reduced set of scheme elements only, transformer definition and application is supported well as summarized in Fig. 7.6.

## 7.2 Implementing TBE-QML

The semantics of by-example transformers has been described independently of a particular modelling language by means of a relational representation of schemes and the relational data manipulation language TBE-QML. Thus, a straightforward approach to put by-example transformers to work is to implement them according to the basic architecture that has been described conceptually in Section 4.1.

In this section, the conceptual architecture of *TBE* is refined, and model-dependent parts are distinguished from model-independent ones. Model-independent parts can be implemented once for all, whereas model-dependent parts have to be implemented once for each modelling language that is to be enabled for *TBE*. The challenge is to design the model-independent parts such that implementations of model-dependent parts can be plugged in easily.

We give here an overview of the architecture and the design decisions, whereas the particular design and the implementation is described in [81]. The description comprises the following aspects: (1) The components that translate transformer definitions into TBE-QML statements, (2) the components that process transformer applications, and (3) the implementation of these components.

### 7.2.1 Components for compiling transformer definitions

A graphical transformer definition for a particular modelling language $L$ is given by a query template and a generative template, both notated using scheme element skeletons representing $L$'s scheme elements together with *TBE* directives. The *TBE* system compiles this graphical specification into a transformer definition in terms of TBE-QML.

Fig. 7.7 illustrates this overall process, where participating components (documents and processes) are distinguished into model-dependent components and model-independent ones. Model-dependent components depend on the particular modelling language and must be implemented once for each language $L$. These components are shown gray-shaded. The package of model-independent components, in contrast, is implemented only once for all modelling languages that shall be enabled for *TBE*.

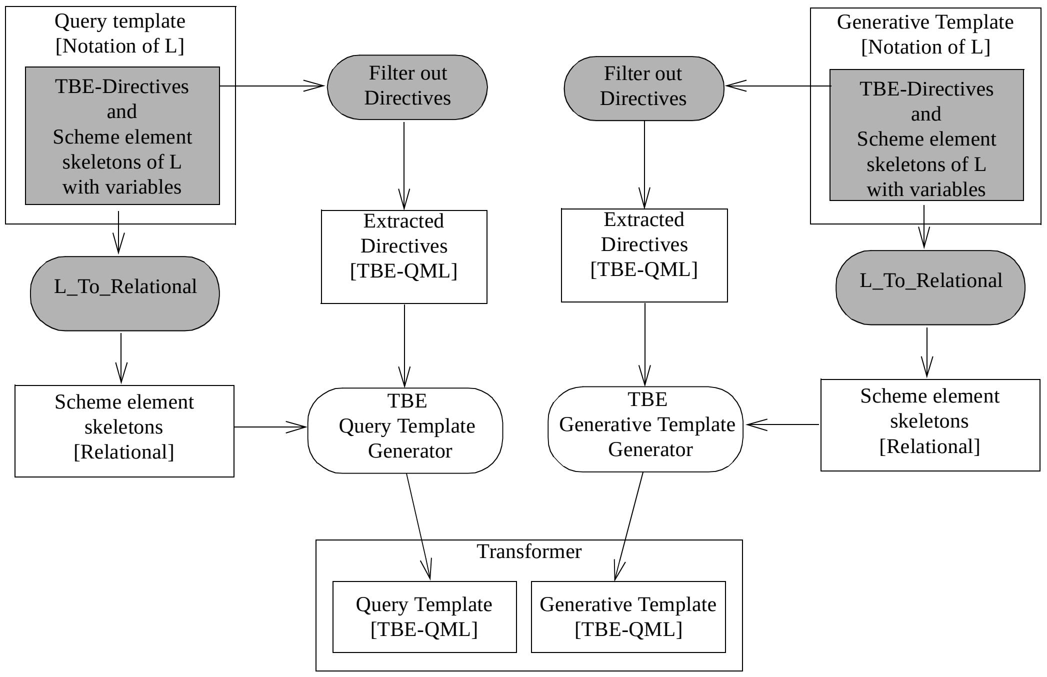

Query template
[Notation of L]
TBE-Directives and Scheme element skeletons of L with variables
Filter out Directives
Filter out Directives
Generative Template
[Notation of L]
TBE-Directives and Scheme element skeletons of L with variables
L_To_Relational
Extracted Directives [TBE-QML]
Extracted Directives [TBE-QML]
L_To_Relational
Scheme element skeletons [Relational]
TBE Query Template Generator
TBE Generative Template Generator
Scheme element skeletons [Relational]
Transformer
Query Template [TBE-QML]
Generative Template [TBE-QML]

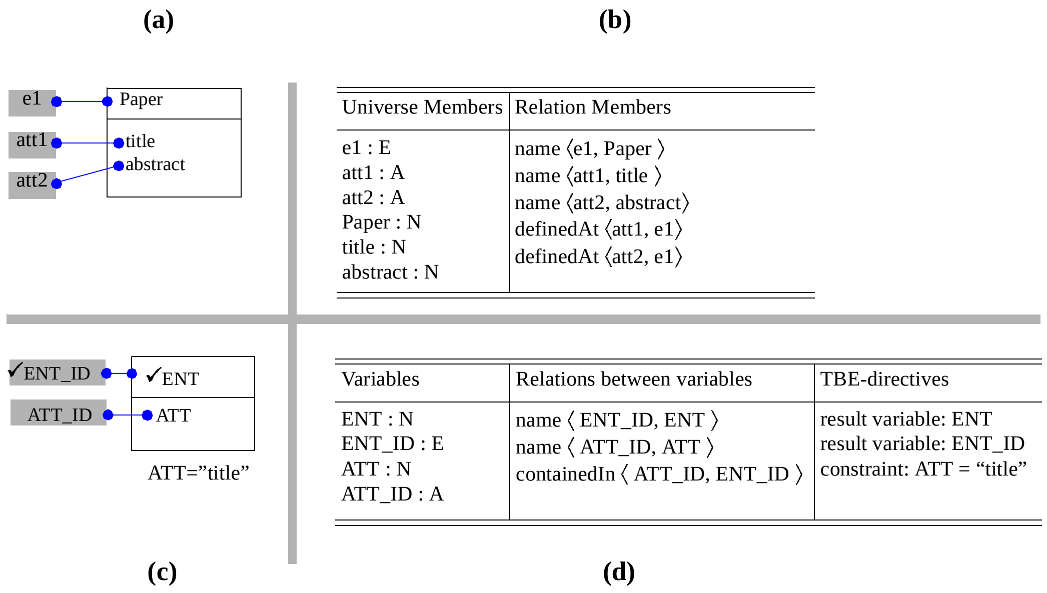
Figure 7.7: Overall process of compiling a by-example transformer definition into a TBE-QML statement.

The query template and the generative template in the representation of $L$ both are model dependent for two reasons: First, $L$ uses its own data structure for representing schemes. For example, WebML uses an XML-representation according to the WebML DTD. Consequently, scheme element skeletons are represented using $L$'s data structure, too. Second, as mentioned in Section 7.1, different modelling languages may incorporate *TBE*-directives in a different manner, e.g. as model primitives or simply as annotated text.

Both, *TBE*-directives and scheme element skeletons, are mapped to a standardized representation by using the processes `Filter out Directives` and $L$`_To_Relational`, respectively. These processes are model dependent because each language $L$ may use its own data structure. For the *TBE*-directives, i.e. symbols, comparison constraints, construction expressions, etc., process `Filter out Directives` achieves a textual notation of *TBE*-directives. The concrete syntax is not relevant here.

For the scheme element skeletons, process $L$`_To_Relational` achieves a relational representation. This mapping has already been explained for scheme elements (c.f. Section 4.2.1). For scheme element skeletons, which are just like scheme elements but have variables in place of concrete values, the same process can be applied. The resulting "universe members" are variables, and, consequently, the resulting "relations between universe members" denote relations between variables.

Reusing process $L$_`To_Relational` that way has the advantage that the resulting "relations" already reflect the final relationship constraints and relation constructors as required for TBE-QML query templates and generative templates, respectively. Only a few syntactic adaptations are required, which are performed by the two processes `TBE Query Template Generator` and `TBE Generative Template Generator`, respectively.



| Universe Members | Relation Members |
|---|---|
| e1 : E<br>att1 : A<br>att2 : A<br>Paper : N<br>title : N<br>abstract : N | name ⟨e1, Paper ⟩<br>name ⟨att1, title ⟩<br>name ⟨att2, abstract⟩<br>definedAt ⟨att1, e1⟩<br>definedAt ⟨att2, e1⟩ |

| Variables | Relations between variables | TBE-directives |
|---|---|---|
| ENT : N<br>ENT_ID : E<br>ATT : N<br>ATT_ID : A | name ⟨ ENT_ID, ENT ⟩<br>name ⟨ ATT_ID, ATT ⟩<br>containedIn ⟨ ATT_ID, ENT_ID ⟩ | result variable: ENT<br>result variable: ENT_ID<br>constraint: ATT = "title" |

Figure 7.8: (a) and (b): Graphical and relational representation of an entity type scheme element, respectively; (c) and (d): Graphical and relational representation of an entity type skeleton used in a query template, respectively

**Example 93** *In order to demonstrate that templates and schemes are similar also at the level of their relational representation, we first repeat the relational representation of a scheme and then give an example of a template's relational representation. Fig. 7.8(a) depicts entity type* `Paper` *in graphical representation, whereas Fig. 7.8(b) illustrates the corresponding relational representation as achieved by process* `WebML_To_Relational`.

*Fig. 7.8(c) depicts query template* `EntWithAttribs`*, which selects IDs and names of entity types that comprise attributes. It comprises an entity type skeleton where variables* `ENT_ID`*,* `ENT`*,* `ATT_ID`*, and* `ATT` *stand in place of concrete values. Fig. 7.8(d) presents query template* `EntWithAttribs`*'s relational representation, which is achieved as follows: First, the directives, i.e. the annotated symbols and comparison constraints are filtered out and stored as depicted in the third column. Then, process* `WebML_To_Relational` *achieves "universe members" that are variables, and "relations" that are relations between these variables. The mapping to the final TBE-QML query statement is then achieved by process* `TBE Query Template Generator`*. A relation of the form* $\mathrm{R}\langle v_1, \ldots, v_n\rangle$ *is simply transformed to a relationship constraint of the form* $\langle v_1, \ldots, v_n\rangle \in R$*. The result is shown in Fig. 4.6 on page 79.*

### 7.2.2 Components for processing transformer applications

A transformer is applied to a scheme by placing a graphical application within this scheme. The application may be individualized, e.g. by application-specific constraints. The *TBE*-system takes this scheme as input, filters out the *TBE*-directives specifying the transformer application, individualizes the transformer, and performs the transformations accordingly.

Fig. 7.9 illustrates the overall process of applying a transformer. Again, model-dependent components are gray-shaded. The input scheme comprises scheme elements in the representation of modelling language $L$ as well as *TBE*-directives specifying the (individualized) transformer application. Both, scheme elements and directives, are model-dependent and are therefore mapped to a standardized representation. Process *L*`_To_Relational` is the same as described with transformer definitions above.

Process `Filter out Directives` filters out the (individualized) application of the transformer, i.e. the directives that specify this application. The process derives (1) the name of the transformer to apply and (2) the optional individualization directives like application-specific constraints, etc. Again, the concrete syntax of the extracted directives is not relevant here. Process `Individualize Transformer` then loads the respective TBE-QML transformer definition from the repository and applies the specified individualizations.

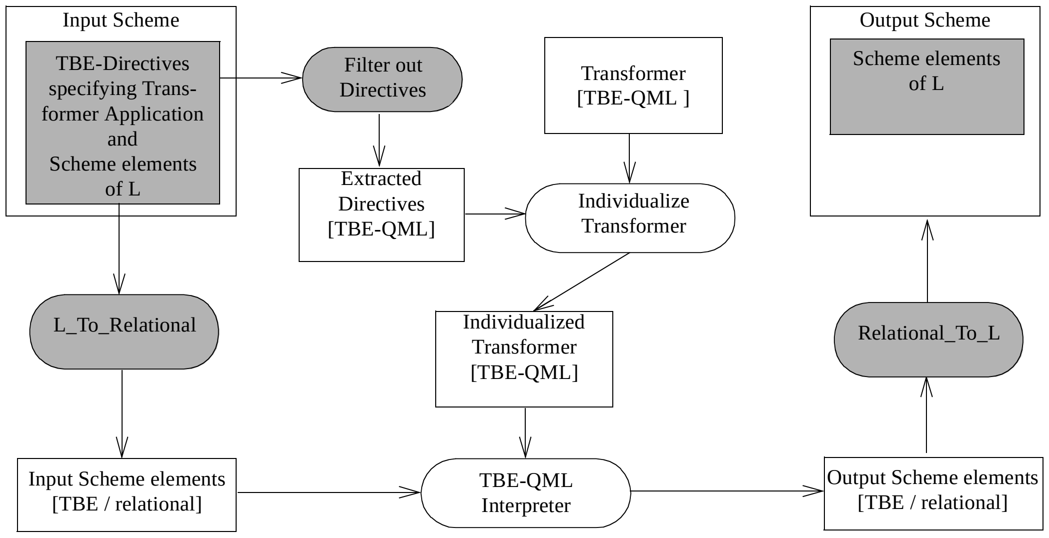

Input Scheme
TBE-Directives specifying Transformer Application and Scheme elements of L
Filter out Directives
Transformer [TBE-QML ]
Output Scheme
Scheme elements of L
Extracted Directives [TBE-QML]
Individualize Transformer
L_To_Relational
Individualized Transformer [TBE-QML]
Relational_To_L
Input Scheme elements [TBE / relational]
TBE-QML Interpreter
Output Scheme elements [TBE / relational]

Figure 7.9: Overall process of performing a (possibly individualized) transformer application.

Process `TBE-QML Interpreter` then performs the transformations as defined by the individualized transformer on the input scheme in relational representation. The result is an output scheme, still represented relationally. Thus, process `Relational_To_`$L$ maps this representation to the final result, i.e. an output scheme in the representation of modelling language $L$. Of course, process `Relational_To_`$L$ and the final result are model-dependent.

## 7.2.3 Component implementation

As mentioned at the beginning of this section, the main goal in the design of a realization of TBE-QML is to provide an implementation of the model-independent parts such that model-dependent parts can be plugged in easily. Thus, the primary design decisions concern those model-independent components that form an interface to the model-dependent ones. Depending on these decisions, the design of the other model-independent components is chosen.

Fig. 7.10 summarizes the components that are part of either of the processes “transformer compilation” and “transformer application”, categorized along the two orthogonal axes model-dependent/model-independent and document/process. The interface documents, i.e. those model-independent doc-

| Components | Model dependent | Model independent |
|---|---|---|
| Process | • L_to_Relational | • TBE_QueryTemplate_Generator |
|  | • Relational_to_L | • TBE_GenerativeTemplate_Generator |
|  | • Filter_out_Directives | • Individualize_Transformer |
|  |  | • TBE-QML_Interpreter |
| Document | • Input Scheme | Interface Documents: |
|  | • Output Scheme | • Input Scheme (relational) |
|  | • Query Template (in L) | • Output Scheme (relational) |
|  | • Generative Template (in L) | • Query Template (relational) |
|  | • Directives (in L) | • Generative Template (relational) |
|  |  | • Extracted directives |
|  |  | Other Documents: |
|  |  | • Transformer (TBE-QML) |
|  |  | • Individualized Transformer (TBE-QML) |

Figure 7.10: Model dependent/independent documents and processes of a TBE-system.

uments that are output or input of a model dependent process, are distinguished from other model-independent documents.

Notably, there are only two kinds of interface documents, namely (1) schemes/templates in relational representation and (2) *TBE* -directives in TBE-QML representation. Thus, a data structure capturing the TBE relational model and the TBE-representation of directives has to be defined first.

Both the TBE relational model and the TBE-directives are represented as XML data structures. XML has been chosen for the following reason: Most modelling languages either use XML directly for representing schemes or provide at least for exporting schemes to such a representation. Thus, if the data model used for the TBE-representation is also XML, then the model-dependent processes $L$`_To_Relational`, `Relational_To_`$L$, and `Filter_out_Directives` are concerned only with transforming XML-data. And for that purpose, extensive tool support exists, e.g. XML parsers, XSL-Transformation (XSLT) processors[1], and XML transformer generators [11, 100].

[1]For an overview over XML and XSLT tools available, confer, for example, the W3C home page at www.w3.org.

Consequently, XML has been chosen also as data structure for representing transformers, i.e. TBE-QML query templates and generative templates, and individualized transformers. Thereby, the processes `TBE_Query_Template_Generator`, `TBE_Generative_Template_Generator`, and `Individualize_Transformer` can be conveniently expressed with XSLT, too. Note that, as mentioned in Sections 7.2.1 and 7.2.2, the adaptations required for turning the relational representation of a template into TBE-QML statements as well as those required for individualizing a transformer are fairly simple.

Finally, process `TBE-QML_Interpreter` is to be implemented. This process takes as input (1) an input scheme in relational representation stored as an XML data structure, and (2) an TBE-QML statement represented basically as a set of variables, a set of constraints, a set of relation constructions, and a set of expressions. As this TBE-QML statement by itself is not executable, there are two basic choices to put them to work: (1) An ad-hoc interpreter can be defined that interprets these statements. (2) The statements can be translated into a script in terms of another language that then can be executed.

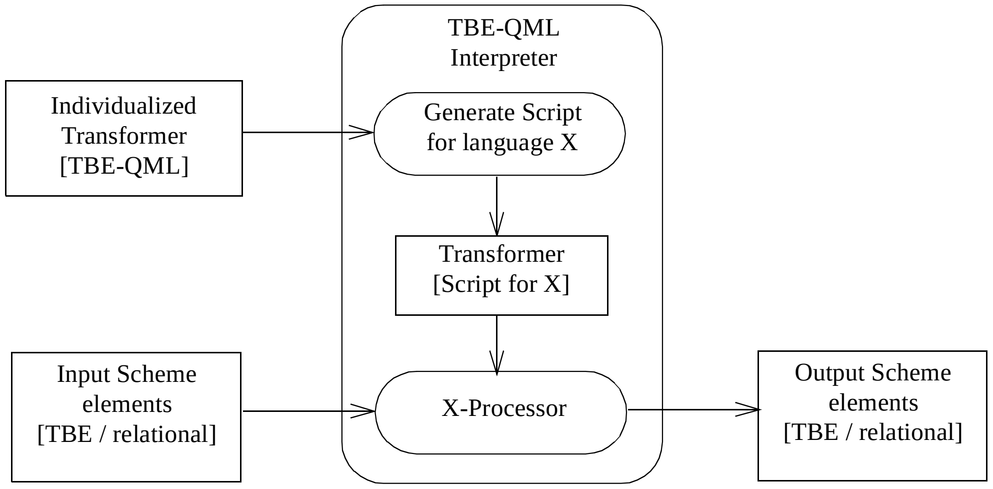

TBE-QML Interpreter
Individualized Transformer [TBE-QML]
Generate Script for language X
Transformer [Script for X]
Input Scheme elements [TBE / relational]
X-Processor
Output Scheme elements [TBE / relational]

Figure 7.11: Process `TBE-QML Interpreter` refined.

For quickly getting a prototype, we decided to follow the second approach, i.e. to generate a script of an existing language $X$. Thus, process `TBE-QML_Interpreter` is refined as shown in Fig. 7.11. A sub-process `Generate_Script_for_Language_`$X$ takes the TBE-QML statement as input and generates a transformer expressed in terms of language $X$. Subsequently, another sub-process $X$-`Processor` executes this script in order to perform the transformation.

For choosing a language $X$, the following requirements need to be met. First, language $X$ must be adequate for expressing TBE-QML statements, i.e. for evaluating query templates *and* for instantiating generative templates. Second, an $X$-processor must be available and should be executable within the same environment as that from which the overall transformer application was triggered, e.g. from a CASE-tool. Thus, an $X$-processor deployed in some portable code, e.g. Java Byte-code, is preferred.

Concerning query template evaluation, we have identified the following two options: (1) A deductive language like, for example, F-logic[82], Prolog [129], or Datalog [1] can be used. This option is straightforward since the query part of TBE-QML is based on domain relational calculus, which itself is based on first-order logic. (2) TBE-QML's declarative query specification can be translated into XSLT commands or XQuery statements. This option is beneficial because the documents to be transformed are represented as XML-documents.

Concerning generative template instantiation, any of these languages can be chosen as soon as it supports manipulation of relational data, i.e. adding/deleting relations and generating new values by means of expressions. Datalog and F-logic, which are basically data retrieval languages, are therefore not adequate; they do not (or only marginally) support expressions for deriving new values and for manipulating relations.

We decided to choose Prolog and XQuery. XQuery has been preferred to XSLT because XQuery is better suited for expressing complex queries. Furthermore, if desired, XQuery could be translated into XSLT [88]. For Prolog and XQuery, practicable processors are available, e.g. Jess [62, 119] and IPSI-XQ [60], respectively, which are both available as Java Byte-code. The implementation based on Prolog is currently being developed, whereas the implementation based on XQuery is described in [81].

The advantage of the variant based on Prolog is that a simple TBE-QML query template translates almost directly to a rule in Prolog and that the deductive engine optimizes rule evaluation. The drawback is that, especially when advanced TBE constructs like nested relations, pinned variables, etc. are to be processed, then extra rules are necessary, and the order of execution must be controlled, e.g. through rule priority; this can lead to inefficient evaluation strategies. XQuery, which offers also some procedural flavor in terms of loops and function calls, has the advantage that the generated code can be specifically tuned for such advanced TBE-constructs. However, the drawback is that process `Generate_Script_for_XQuery` must take care of producing optimized code even for simple TBE-QML statements, e.g. by avoiding cartesian products where possible.

# Chapter 8

# Use-Case: a department's web site

## Contents

In this chapter, we demonstrate the use of transformers by the example of a web scheme of a department's web site that is to be developed. In Section 8.1, we show how the scheme is defined manually, i.e. in absence of transformers.

Then, in Section 8.2, we identify recurrent modelling tasks that have been repeatedly applied during the development of the scheme, and we provide transformers that support these tasks. In Section 8.3, we demonstrate that the web scheme of the department's web site can be developed more rapidly and conveniently by utilizing these transformers.

Note that the transformers introduced in this chapter are of general purpose and are not defined for the department's web site only. Thus, modelers do not have to define them again and again for each project, and these transformers could also be predefined and provided through a library.

## 8.1 Developing the scheme without transformers

A simple web site of a university department is to be developed. The purpose of the site is to publish information about the department, the staff members, the publications, and the courses provided. The department web-site offers the same view for both (external) visitors of the site and (internal) staff members, and no login or registration is required.

### 8.1.1 Content scheme

The following requirements are defined for the content that is to be presented on the web site:

- The department itself is described by its name, the address, the location on the campus, and the office's fax number, phone number, and e-mail address. Further, the department has a head who is a staff member.
- Staff members are identified by their name and have a position, a phone number, office hours, and a photo.
- Publications are identified by a cite key and are further described by the year of publication, the type of publication (i.e. book, paper, technical report, and thesis), a text field containing the full citation, and a text field containing the abstract.

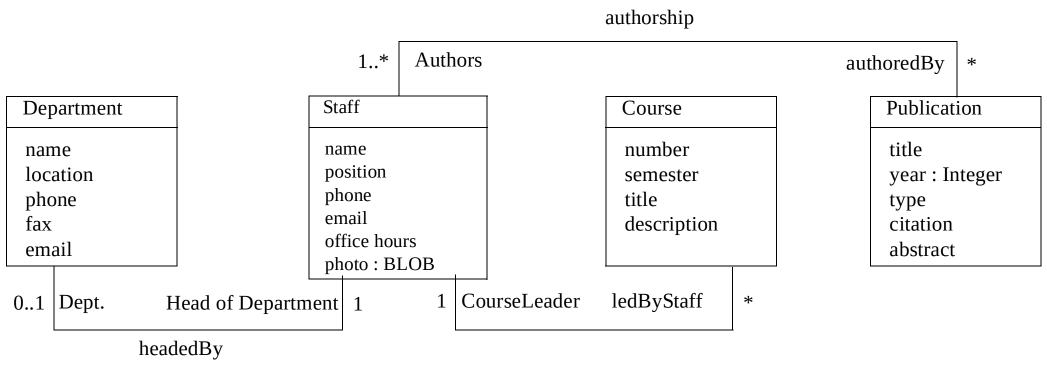


Figure 8.1: Content scheme of the department site

- Courses are identified by a course number and the semester in which they are held. Further, each course has a title and a description, and it is led by a staff member.

Fig. 8.1 depicts the content scheme, which directly corresponds to above mentioned data requirements. Entity types `Department`, `Staff`, `Course`, and `Publication` represent the department, staff members, courses, and the publications. The default data type of attributes is `String`. Thus, all attributes implicitly are of type `String`, except attribute `photo`, which is declared as `BLOB` (binary-large-object), and attribute `year`, which is declared as `Integer`.

Entity type department is a singleton, i.e. contains exactly one member. This is, however, not expressed in the scheme because WebRatio does not offer a primitive that marks an entity type as singleton. Relationship `headedBy` expresses that the department has exactly one head, who must be a staff member. Relationship `leads` between entity types `Staff` and `Course` expresses that a staff member may lead any number of courses, whereas one course is led by exactly one staff member. Relationship `authorship` between entity types `Staff` and `Publication` expresses that a staff member may be author (or co-author) of any number of publications, whereas a publication must have at least one author.

### 8.1.2 Hypertext scheme

The site shall present the content in a hypertext as follows:

- The home page shall display the department's details and the name of its head.
- For staff members, an overview page shall be provided that displays a list of staff members and allows to select one of them at runtime. Once selected, the details of the respective staff member are presented on a staff member's detail page. This detail page then displays all attribute values as well as a list of all courses held by the staff member. Further, a list of her publications shall be presented. Upon selection of a course or a publication, the respective details page is displayed as explained below.
- All courses are listed on a separate overview page, too. Again, once a course is selected therefrom, the user is redirected to the course's details page. Besides all attribute values of the respective course, this page then presents also the name of the staff member who leads the course and provides a link to her details page.
- Publications are also listed on an overview page. Further, because there are about 250 publications to present, an access structure that allows to filter the list by category and/or by year shall be provided. The details of a paper selected from the list are again presented on a details page. Thereon, all the attribute values of the publication as well as all the authors of the paper are listed.
- The home page as well as all the overview pages shall be reachable from any point in the web site.

Fig. 8.2 depicts the hypertext scheme that expresses above mentioned requirements. We first describe the page classes themselves and then explain the inter-page links.

Page class `Department Page` represents the site's home page, i.e. that page that is presented whenever a user enters the domain of the site. The page class contains a data unit `Department` that presents the sole member of entity type `Department` and a data unit `Head` that presents the head of this department, i.e. that staff member that is reachable from a department via relationship role `Head of Department`.

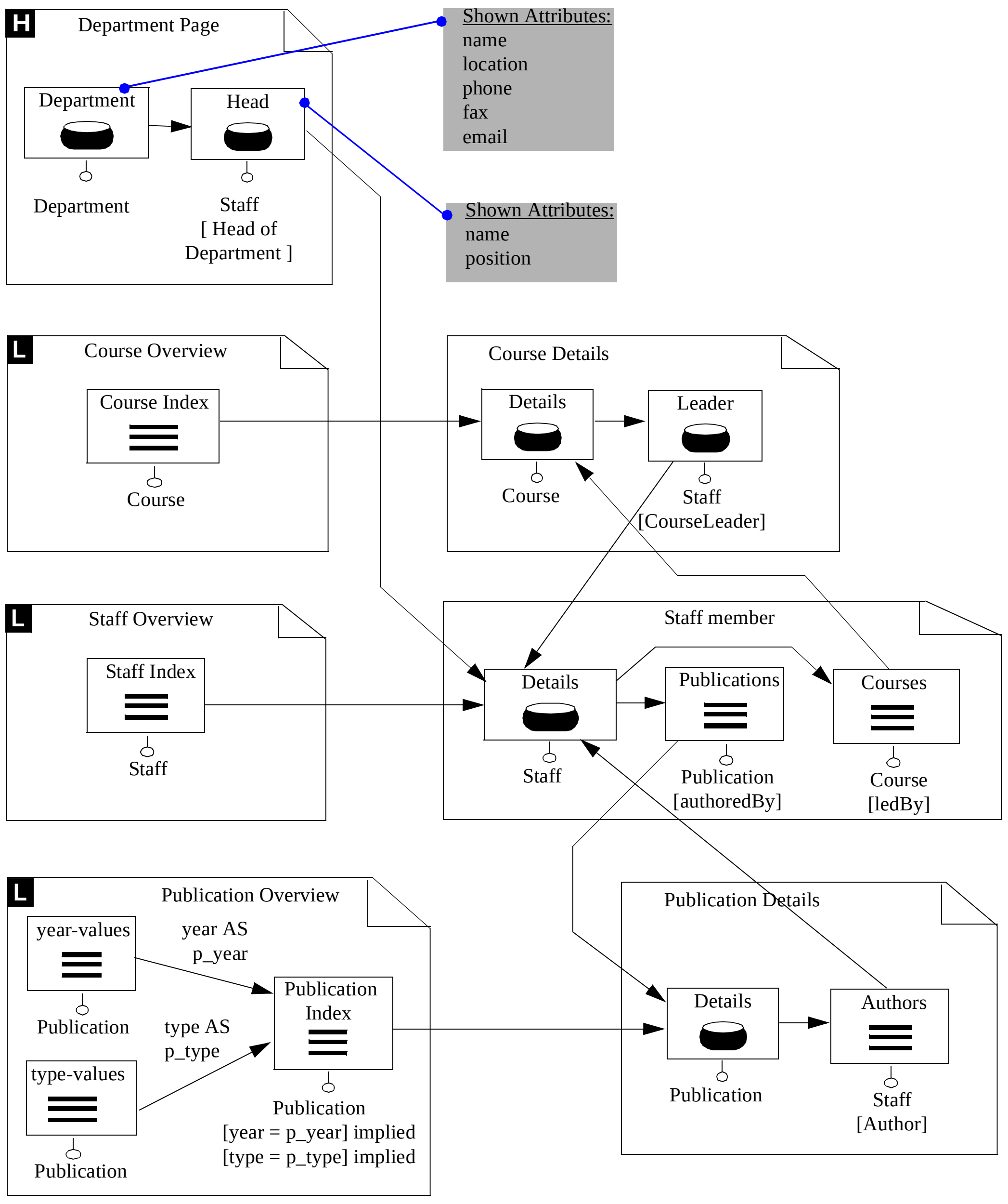


Figure 8.2: Hypertext scheme for university department web site

Data unit `Department` displays the attributes `name`, `location`, `phone`, `fax`, and `email` of entity type `Department` as declared in section "`Shown Attributes`" in the property window attached. Data unit `Head` displays only attributes `name` and `position` of entity type `Staff`. For reasons of conciseness, we have shown the property windows only for the data units `Department` and `Head`. For the rest of the data units and index units contained in the hypertext scheme, the reader may assume the lists of "shown attributes" to be set adequately.

Page class `Course Overview` comprises an index unit `Course Index` for showing a list of all courses. This page is declared as landmark such that is reachable from any other point in the web site through a menu that is permanently visible. Page class `Course Details` contains data unit `Details` for presenting the details of a course that is passed as parameter along the incoming link. Further, page class `Course Details` contains a data unit `Leader`, which presents the staff member who leads the course, i.e. that member of entity type staff that is reachable from a course via relationship role `CourseLeader`.

Page class `Staff Overview` comprises an index unit `Staff Index` for showing a list of staff members. Again, this page is declared as landmark. Page class `Staff member` contains a data unit `Details` for showing the details of a staff member that is passed along an incoming link. Further, the page class contains two index units: Index unit `Publications` presents the publications of the respective staff member, i.e. those publications that are reachable from a staff member via relationship role `authoredBy`. Index unit `Courses` presents the courses led by this staff member, i.e. those courses that are reachable via relationship role `ledBy`.

Page class `Publication Overview`, which is again a landmark, comprises an index unit `Publication Index` for showing a list of publications. Further, two index units, `year` and `type`, present a list of all the year-values and a list of all the type-values occurring in publications, respectively. Once a user selects a year out of this list, the respective value is passed as a parameter named `p_year` along the link to index unit `Publication Index`. According to the selection condition `year = p_year`, the list of publications is constrained to those having appeared in the selected year. Keyword `implied` next to the selection condition expresses that this condition is checked only if a parameter `p_year` is provided, i.e. if a user has selected a year in index unit `year`. Selecting a type-value has an analogous effect.

Page class `Publication Details` comprises data unit `Details` for presenting the details of the publication that is passed along the incoming link. Index unit `Authors` presents the authors of this publication, i.e. a list of those members of entity type `Staff` that are reachable from this publication via relationship role `Author`.

The inter-page links provide for navigating to more detailed information. Index units of overview page classes and other content units that present representative attributes only are connected to main data units of details page classes. Consider, for example, the link from index unit `Course Index` of page class `Course Overview` to data unit `Details` of page class `Course Details`. When a user selects a course from the course index, then the details of this course are presented in the respective details page. Similarly, data unit `Leader`, which presents only the name and the position of a staff member leading a course, is connected to the details page class of staff members. Thus, when traversing this link from a particular course page, then the staff member page presenting the leader of this course in detail is selected.

## 8.2 Transformers for recurrent modelling tasks

The schemes depicted in Fig. 8.1 and Fig. 8.2 represent the department's web site at the point where the process of scheme development has already been completed. Nevertheless, when analyzing the final scheme, it turns out that a number of patterns of extending and refining the scheme have repeatedly been applied. We have identified the following recurrent modelling tasks:

1. *Provide for overview and details pages.* Based on the entity types of the content scheme, page classes that give an overview of the members of entity types as well as page classes that present a member of an entity type in detail are defined.

2. *Provide for supplementary units.* Page classes presenting a main item in detail are extended by supplementary units, i.e. units for presenting members of other entity types that are directly related to the main item.

3. *Interconnect units.* Page classes or, more precisely, the units contained in different page classes are interconnected, again according to relationships defined in the underlying content scheme.

4. *Provide for category index.* In order to facilitate access, index units that present a large number of items are complemented by category indices that constrain the items to be presented.

In the remainder of this section, we describe these modelling tasks in more detail, illustrate where they have been applied in the development of the department's web site, and provide transformers that support these tasks.

### 8.2.1 Providing for overview and details pages

According to modelling task "Provide overview and details pages", for entity types of the content scheme, overview page classes and details page classes are defined. This task can be split in two parts, one for providing overview page classes and one for providing details page classes. We start with the task concerning the details page classes, which is simpler than the other one.

For providing details pages, usually one page class with a data unit is generated for each entity type. This data unit represents the main item of the page and displays all the attributes of the respective entity type.

**Example 94** *In the scheme of the department's web site, this task has actually been applied to every entity type. Note that the details pages got extended by further data units and index units later on. This extension, however, is subject to a separate task explained in Section 8.2.2.*

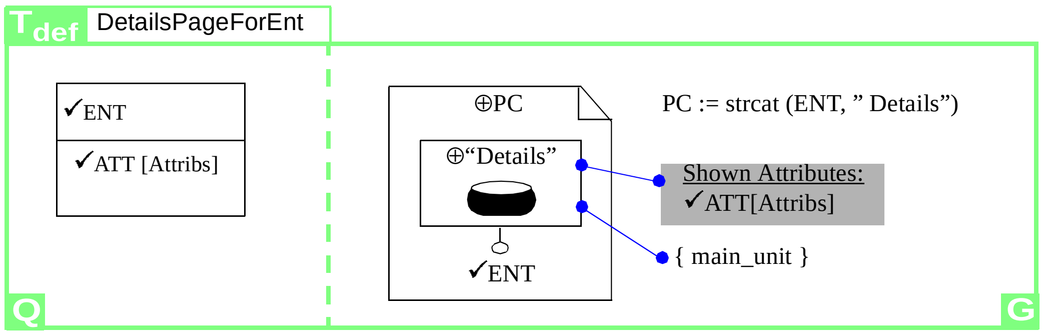


Figure 8.3: Definition of transformer `DetailsPageForEnt`

Fig. 8.3 depicts the definition of transformer `DetailsPageForEnt`, which supports modelling task "Provide for details pages". The transformer's query template selects entity types together with their attributes. The entity types are represented by result variable `ENT`. The attributes, which are stored in a nested relation named `Attribs`, are represented by result variable `ATT`. The generative template is then instantiated for each entity type `ENT` (and the corresponding nested relation `Attribs`) in that a page class with an embedded data unit is generated. The page class is named after entity type `ENT`, whereas the data unit is named "`Details`". The data unit's list of attributes to display, i.e. property `Shown Attributes`, is set to the list of attributes as provided through nested relation `Attribs`. Further, the newly generated data unit has a user-defined property "`main_unit`" attached in order to distinguish it from supplementary data units introduced later in this section.

For providing overview pages, usually a page class with an index unit is generated for each entity type. Such a page class is for providing an overview of the members of the corresponding entity type. The index units of overview page classes usually do not present all attributes of their underlying data type but neglect details. Further, the overview page classes usually become landmarks.

**Example 95** *In the scheme of the department's web site, this modelling task has been applied to every entity type except singleton entity type* `Department`. *Hence, overview page classes* `Course Overview`, `Staff Overview`, *and* `Publication Overview` *have been generated. The list of shown attributes for overview pages is not given in the figure. Yet we assume that the following attributes of the content scheme are not presented in overview pages: The course overview does not provide the detailed course description, the staff overview does not present a photo, and the publication overview does not show the publication's abstract.*

In order to declare which entity types and attributes are to be considered for overview pages, the following user-defined properties are introduced: (1) User-defined property `singleton` attached to an entity type expresses that this entity type is a singleton such that no overview page class is to be generated. (2) User-defined property `detail` attached to an attribute expresses that this attribute describes a detail to be presented in detail pages only. These properties are considered by transformer `OverviewPageForEnt` explained next.

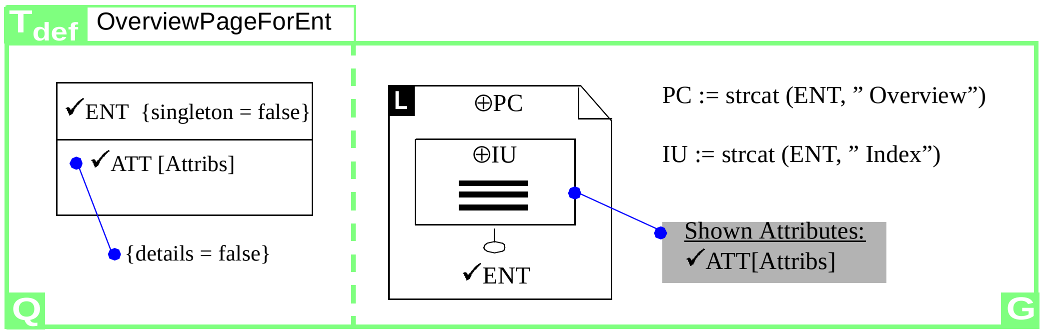


Figure 8.4: Definition of transformer `OverviewPageForEnt`

Fig. 8.4 illustrates the definition of transformer `OverviewPageForEnt`, which supports modelling task "provide for overview pages". The query template works analogously to that of transformer `DetailsPageForEnt` illustrated in Fig. 8.3, but selects only entity types that are not marked as "singleton", i.e. that either have no user-defined property named `singleton` attached or have this property set to `false`. Further, only those attributes that are not marked as "detail" are selected. The generative template is instantiated for each entity type `ENT` as follows: A landmark page class with an index unit is generated, both named after entity type `ENT`. The index unit's list of attributes to display, i.e. property `Shown Attributes`, is set to the list of attributes as represented by nested relation `Attribs`.

Of course, the content scheme has to be annotated with user-defined properties `singleton` and `detail` such that transformer `OverviewPageForEnt` works as intended. This requires some effort, which, however, pays off for the following reasons: (1) User-defined properties improve the content scheme's expressiveness because a particular intent of an entity type or attribute is explicitly expressed. (2) The provided annotations can be used several times in different transformers.

### 8.2.2 Providing for supplementary units

According to modelling task "provide for supplementary units", a details page class, which presents one item in detail, often is extended by supplementary content units that present information directly related to the page's main item. This can be referred to as modelling task "Provide supplementary units". The following two rules can be identified: (1) Let $\mathtt{DU}_{\mathtt{main}}$ be a details page's main data unit, and let $\texttt{E}_{\texttt{main}}$ be its content source. If $\texttt{E}_{\texttt{main}}$ is connected to another entity type $\texttt{E}_{\texttt{supp}}$ via a to-one relationship role `R`, then a data unit $\texttt{DU}_{\texttt{supp}}$ referring to this entity type is created, and $\texttt{DU}_{\texttt{main}}$ is connected to $\texttt{DU}_{\texttt{supp}}$. Further, data unit $\texttt{DU}_{\texttt{supp}}$ gets relationship role `R` attached as selection condition. Thereby, since data unit $\texttt{DU}_{\texttt{main}}$ presents one member $e_{main}$ of entity type $\texttt{E}_{\texttt{main}}$, data unit $\texttt{DU}_{\texttt{supp}}$ presents that member of $\texttt{E}_{\texttt{supp}}$ that is directly reachable from $e_{main}$ via the to-one relationship role `R`. (2) The first rule applies analogously to to-many relationship roles, yet an index unit is created instead of a data unit.

Note that "supplementary" data units are just ordinary data units. However, they are not used with the intent to present items in detail but are for presenting some representative attributes of these items. This is, as for index units in general, also true for supplementary index units.

**Example 96** *In the scheme of the university department's web site, modelling task "Provide supplementary units" has been applied to every details page. We pick out the definition of page class* `Course Details` *for explanation. This page class presents (1) the details of a course, and (2) supplementary, the name of the leader of this course. Point (1) is expressed by data unit* `Details`*, which has entity type* `Course` *as content source. Point (2) is expressed as follows: Entity type* `Course` *refers to entity type* `Staff` *via the to-one relationship role* `CourseLeader`*. Thus, a data unit* `Leader`*, which has entity type* `Staff` *as content source and which has relationship role* `CourseLeader` *attached, has been created. Data unit* `Leader` *displays only attributes* `name` *and* `position`*, which are assumed to be representative for a staff member.*

In order to declare which attributes are representative, user-defined property `representative` is introduced. Attributes marked with this property are the only attributes to be presented in supplementary units. Further, in order to distinguish main data units from other data units that might exist in a scheme, it is assumed that main data units are marked with user-defined property `main_unit`. Note that transformer `DetailsPageForEnt` generates details page classes with data units that have this property set. Both properties, `representative` and `main_unit`, are considered accordingly by transformer `GenerateSupplementaryUnits` explained next.

Fig. 8.5 depicts the definition of transformer `GenerateSupplementaryUnits`, which supports modelling task "Provide for supplementary units". The

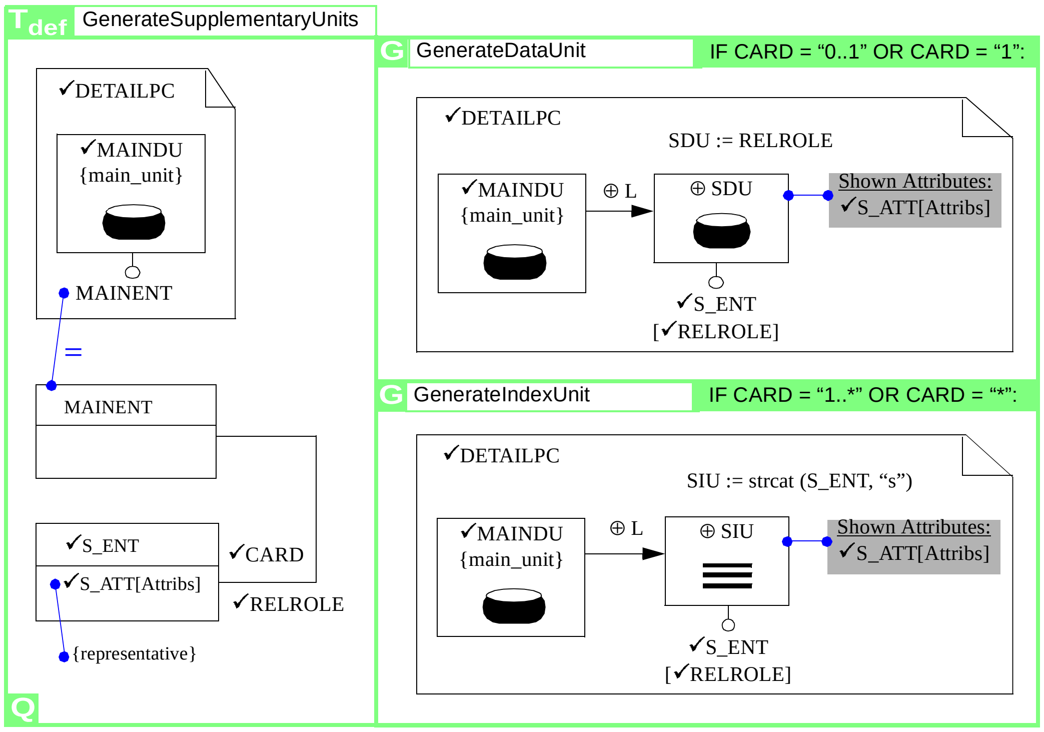


Figure 8.5: Definition of transformer `GenerateSupplementaryUnits`.

transformer comprises one query template and two conditional generative templates.

The query template selects detail page classes with main data units and entity types that are directly related to the content source of such a main data unit. This is expressed as follows: The details page class, which is represented by result variable `DETAILPC`, must contain a main data unit, which is represented by result variable `MAINDU`. A main data unit is characterized by user-defined property `main_unit`.

From all main data units, only those are considered whose content source is related to another entity type, which is assumed to provide supplementary information. The content source of the main data unit is represented by non-result variable `MAINENT`, whereas the other entity type is represented by result variable `S_ENT`. The relationship role that leads from `MAINENT` to `S_ENT` is selected, too, together with its attached cardinality. The relationship role and its cardinality are represented by result variables `RELROLE` and `CARD`, respectively. Further, the attributes of supplementary entity type `S_ENT` are stored in a nested relation `Attribs`. The attributes are represented by result variable `S_ATT`, where only those attributes are selected that have user-defined property `representative` set.

For each tuple of the query's result relation $Rel_{Qm}$( `DETAILPG`, `MAINDU`, `S_ENT`, `RELROLE`, `CARD`, *Attribs*(`S_ATT`)), the generative part of transformer `GenerateSupplementaryUnits` is instantiated. This generative part comprises two conditional generative templates, namely `GenerateDataUnit` shown in the upper part, and `GenerateIndexUnit` shown in the lower part. Which one of these two is instantiated depends on the value of the relationship role's cardinality represented by result variable `CARD`. If this cardinality is either `0..1` or `1`, then `GenerateDataUnit` is instantiated and a supplementary data unit is generated. Otherwise, i.e. if the cardinality is either `1..*` or `*`, then `GenerateIndexUnit` is instantiated and a supplementary index unit is generated. We explain only generative template `GenerateDataUnit`:

Generative template `GenerateDataUnit` extends a details page class `DETAILPC` as follows: A new data unit, represented by new-element variable `SDU`, is generated within `DETAILPC` and is named after relationship role `RELROLE`. The content source of this data unit is the supplementary entity type represented by `S_ENT`. Further, a selection condition referring to relationship role `RELROLE` is created. The attributes that are to be displayed by data unit `SDU` are those that are provided through nested relation `Attribs`, which contains the representative attributes of entity type `S_ENT`. Finally, the existing main data unit, which is represented by parameter variable `MAINDU`, is connected to the new data unit `SDU` by means of a newly generated link. This link is represented by new-element variable `L`.

**Example 97** *Suppose that transformer* `GenerateSupplementaryUnits` *is applied to a scheme that comprises a page class* `Staff Details` *with a main data unit* `Details` *that has entity type* `Staff` *as content source. Note that entity type* `Staff` *is related to three other entity types, i.e. (1) to entity type* `Publications` *via relationship role* `authoredBy` *of cardinality* `*`*, (2) to entity type* `Course` *via relationship role* `ledByStaff` *of cardinality* `*`*, and (3) to entity type* `Department` *via relationship role* `Dept` *of cardinality* `0..1`*.*

*In this setting,* `GenerateSupplementaryUnits`*'s query template would achieve a result as depicted in Fig. 8.6. The representative attributes of each related entity type* `S_ENT` *are stored in a nested relation* `Attribs(S_ATT)`*, i.e. attributes* `title`*,* `year`*, and* `type` *for* `S_ENT`

| Rel_Q | | | | | |
|---|---|---|---|---|---|
| DETAILPC | MAINDU | S_ENT | RELROLE | CARD | Attribs |
| | | | | | S_ATT |
| Staff Details | Details | Publication | authoredBy | * | title<br>year<br>type |
| Staff Details | Details | Course | ledByStaff | * | number<br>semester<br>title |
| Staff Details | Details | Department | Dept. | 0..1 | {empty} |

Figure 8.6: Result of evaluating `GenerateSupplementaryUnits`'s query template for page class `Staff Details`.

`= Publication`, *attributes* `number`, `semester`, *and* `title` *for* `S_ENT = Course`, *and the empty set for* `S_ENT = Department`.

*Consequently, the generative template is instantiated three times and extends page class* `Staff Details` *as follows: (1) For each relationship role of cardinality* `*`, *i.e. for* `authoredBy` *and* `ledByStaff`, *conditional generative template* `GenerateIndexUnit` *is instantiated. Thus, two new index units* `Publications` *and* `Courses` *are generated. (2) For relationship role* `Dept`, *which is of cardinality* `0..1`, *conditional generative template* `GenerateDataUnit` *is instantiated and generates a new data unit* `Depts`. *Note that entity type* `Department` *does not comprise any representative attribute such that data unit* `Depts` *does not present any information. We will come back to this point later in this section.*

### 8.2.3 Interconnecting units

According to modelling task "interconnect units", units contained in different page classes are interconnected based on relationships defined in the underlying content scheme. This modelling task comprises two parts, namely (1) to connect supplementary data units to main data units, and (2) to connect index units to main data units. We address these two parts separately, one after the other.

### Connect index units to main data units

Index units usually provide a link to a corresponding details page. The following rule can be identified: Let `IU` be an index unit and let `E` be the content source of this index unit. If a corresponding details page exists, i.e. a page class that contains a data unit $\texttt{DU}_{\texttt{main}}$ presenting members of entity type `E` in detail, then `IU` is connected to $\texttt{DU}_{\texttt{main}}$ via a link.

**Example 98** *In the scheme of the university department's web site, every index unit has been connected to a corresponding main data unit. For one thing, index unit* `Course Index` *in page class* `Course Overview` *has been linked to the details page of courses, i.e. to data unit* `Details` *in page class* `Course Details`*.*

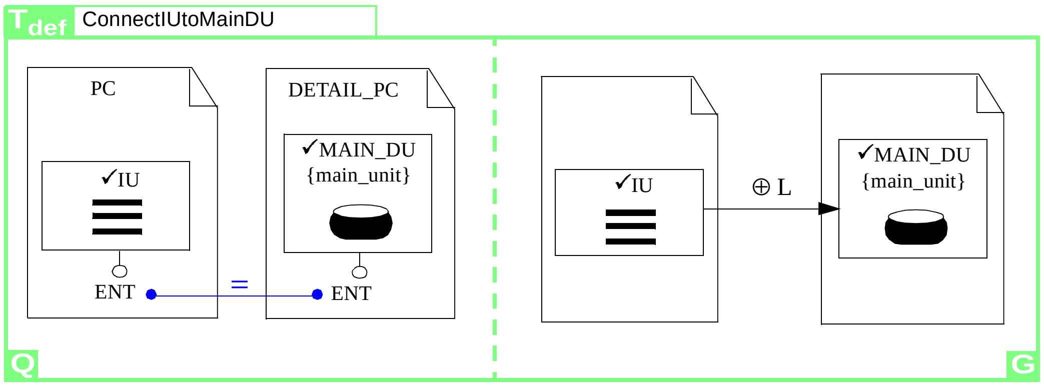


Figure 8.7: Definition of transformer `ConnectIUtoMainDU`.

Fig. 8.7 depicts the definition of transformer `ConnectIUtoMainDU`, which connects index units to main data units. The query template selects those combinations of index units and main data units that have the same content source. Index units and main data units are represented by result variables `IU` and `MAIN_DU`, respectively. Again, a main data unit is characterized by user-defined property `main_unit`. Both units must have the same content source, which is represented by non-result variable `ENT`. For each pair of an index unit `IU` and a data unit `MAIN_DU` selected by the query template, the generative template generates a link from `IU` to `MAIN_DU`. This link is represented by new-element variable `L`.

**Example 99** *Suppose that transformer* `ConnectIUtoMainDU` *is applied to a scheme that comprises two index units* `Staff Overview` *and* `Authors` *and a main data unit* `Details`*, where all these content units have entity type* `Staff` *as content source. In this setting, the query template of transformer* `ConnectIUtoMainDU` *achieves a result relation* $Rel_Q(\texttt{IU}, \texttt{MAIN\_DU})$ *comprising two tuples, i.e.* $\langle\texttt{Staff Overview}, \texttt{Details}\rangle$ *and* $\langle\texttt{Authors}, \texttt{Details}\rangle$*. Consequently, the generative template is instantiated and achieves two new links targeting data unit* `Details`*, one from index unit* `Staff Overview` *and one from index unit* `Authors`*.*

**Connect supplementary data units to main data units**

Supplementary data units usually provide a link to a corresponding details page. The following rule can be identified: Let $\texttt{DU}_{\texttt{abstr}}$ be a data unit that abstracts from details, and let `E` be the content source of this data unit. If there exists a corresponding details page, i.e. a page class that contains a data unit $\texttt{DU}_{\texttt{main}}$ presenting members of entity type `E` in detail, then $\texttt{DU}_{\texttt{abstr}}$ is connected to $\texttt{DU}_{\texttt{main}}$ via a link.

**Example 100** *In the scheme of the university department's web site, every supplementary data unit has been connected to a main data unit. For one thing, data unit* `Leader`*, which is placed in page class* `Course Details` *and which presents the name and the position of a course leader, has been linked to the details page of staff members, i.e. to data unit* `Details` *in page class* `Staff member`*.*

Fig. 8.8 depicts the definition of transformer `ConnectSupplDUtoMainDU`, which connects supplementary data units to main data units. This transformer works analogously to transformer `ConnectIUtoMainDU`, but instead of index units, the query template selects *supplementary data units* that have the same content source as a main data unit.

## 8.2.4 Providing for category index

According to modelling task "Provide for category index", index units that are to present a large number of items often get a category index attached.

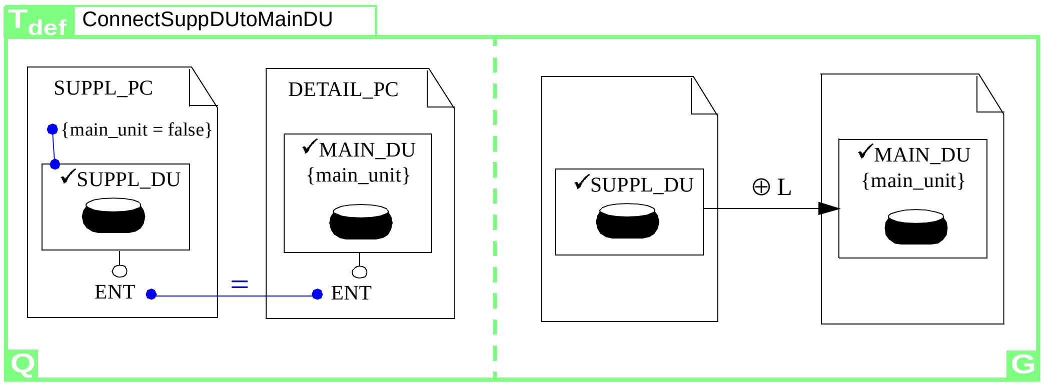


Figure 8.8: Definition of transformer `ConnectSupplDUtoMainDU`.

It is assumed that the set of items can be split into groups depending on the value of a particular attribute, which is referred to as a category attribute. Defining a category index works as follows: Let `IU` be an index unit presenting a list of members of entity type `E`, and let `CAT_ATT` be an attribute of `E` serving as category attribute. Then, an index unit `CAT_IU` is created that presents a list of all distinct `CAT_ATT`-values of entity type `E`. This category index is linked to index unit `IU`, where a link parameter carrying the selected category value is attached to the link. Index unit `IU` is then extended by a selection condition such that whenever a user selects a category, then items only from this category are presented.

**Example 101** *In the scheme of the university department's web site, modelling task "Provide for category index" has been applied twice for the publication overview, i.e. once with attribute* `year` *and once with attribute* `type` *serving as category attribute, respectively. The resulting page class* `Publication Overview` *is depicted in Fig. 8.2.*

Fig. 8.9 illustrates the definition of transformer `CategoryIndexForIU`, which supports modelling task "Provide for category index". The query template returns a relation $Rel_Q(\mathtt{PC}, \mathtt{IU}, \mathtt{ENT}, \mathtt{CAT\_ATT})$, where each of its tuples contains a page class `PC` and an index unit `IU`, which represents the index unit to be complemented by a category index, and `IU`'s content source `ENT` together with one of its attributes `CAT_ATT`, which serves as category attribute.

For each tuple of $Rel_Q$, the generative template extends page class `PC` by a new category index as follows: (1) A new index unit serving as category

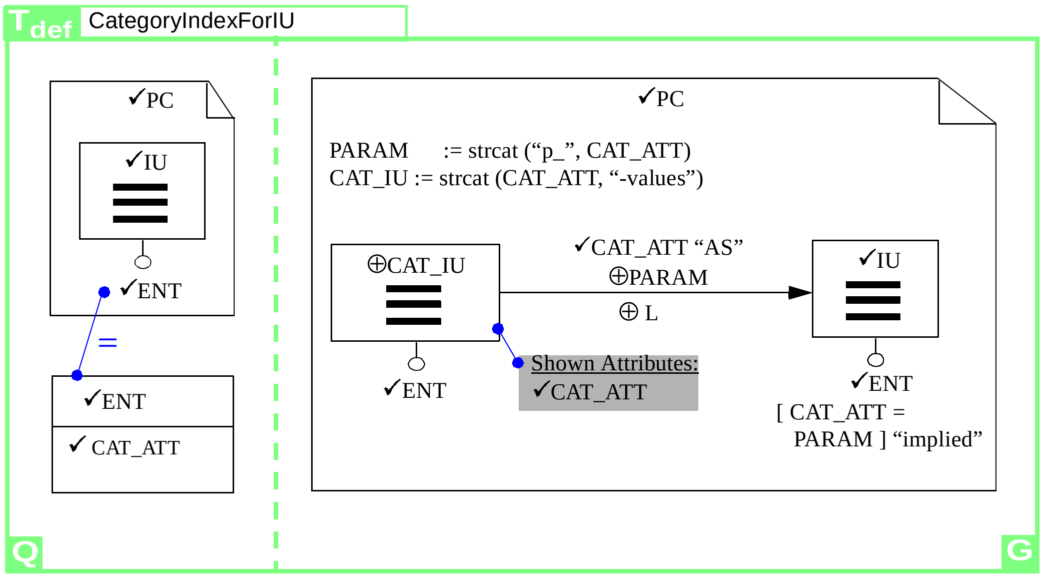


Figure 8.9: Definition of transformer `CategoryIndexForIU`.

index is generated. It is represented by new-element variable `CAT_IU`, and its content source is the same entity type as that of index unit `IU`, i.e. entity type `ENT`. Index unit `CAT_IU` displays exactly one attribute, i.e. the category attribute `CAT_ATT` provided by the query template. (2) A link from new index unit `CAT_IU` to index unit `IU` is generated. This link is represented by new-element variable `L`. Further, a link parameter, which transports the value selected by the user in the category index, is attached to the link. This parameter is represented by new-element variable `PARAM`, and its label is derived by preceding the name of the category attribute by literal "`p_`". (3) The existing index unit `IU` is extended by a new selection condition of the form [`CAT_ATT` = `PARAM`] `implied`.

**Example 102** *Suppose a scheme that contains a page class* `PublicationsOverview` *with an index unit* `Publication Index`*, where entity type* `Publication` *is the content source of this index unit and comprises, among others, two attributes* `year` *and* `type`*. Suppose that transformer* `CategoryIndexForIU` *is applied to this scheme having variable* `CAT_ATT` *constrained such that only entity types* `year` *and* `type` *are considered. In this setting, the query template achieves a result relation* $Res_Q$ (`PC`, `IU`, `ENT`, `CAT_ATT`) *containing two tuples, i.e.* ⟨`Publication Overview`, `Publication Index`, `Publication`, `year`⟩

*and* ⟨`Publication Overview`, `Publication Index`, `Publication`, `type`⟩*.*

*For each of these two tuples, the generative template is instantiated. The first tuple, i.e. that one having variable* `CAT_ATT` *bound to* `year`*, imposes the following result: (1) a new category index unit, which is named* `year-values` *and which is placed within page class* `Publication Overview`*; (2) a new link from index unit* `year-values` *to index unit* `Publication Index`*; (3) a new link parameter, which is named* `p_year` *and which is bound to attribute* `year` *of index unit* `year-values`*, (4) a new selection condition, which has the form* `year = p_year` *and which is attached to index unit* `Publication Index`*. The second tuple is treated analogously and again yields a new category index, a new link, a new parameter, and a new selection condition.*

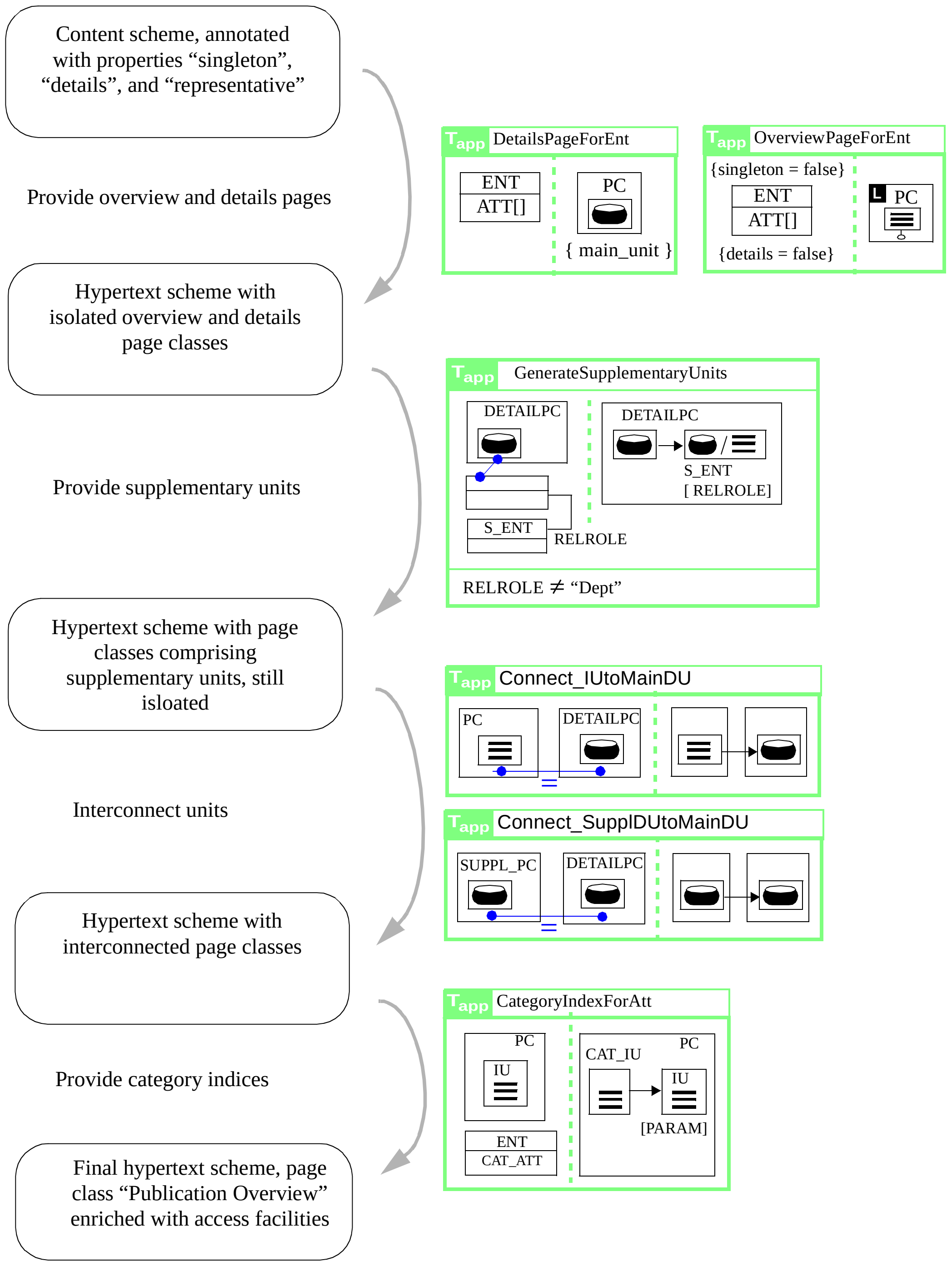

Content scheme, annotated with properties "singleton", "details", and "representative"
Provide overview and details pages
Hypertext scheme with isolated overview and details page classes
Provide supplementary units
Hypertext scheme with page classes comprising supplementary units, still isloated
Interconnect units
Hypertext scheme with interconnected page classes
Provide category indices
Final hypertext scheme, page class "Publication Overview" enriched with access facilities
Tapp DetailsPageForEnt
ENT
ATT[]
PC
{ main_unit }
Tapp OverviewPageForEnt
{singleton = false}
ENT
ATT[]
{details = false}
L PC
Tapp GenerateSupplementaryUnits
DETAILPC
S_ENT
RELROLE
DETAILPC
S_ENT
[ RELROLE]
RELROLE ≠ "Dept"
Tapp Connect_IUtoMainDU
PC
DETAILPC
=
Tapp Connect_SupplDUtoMainDU
SUPPL_PC
DETAILPC
=
Tapp CategoryIndexForAtt
PC
IU
ENT
CAT_ATT
CAT_IU
PC
IU
[PARAM]

Figure 8.10: Overview of the development process with transformers.

## 8.3 Developing the scheme with transformers

With the transformers developed in the previous section, sites like the department's web site can be developed more rapidly and conveniently. This section demonstrates how one modelling task after the other is performed by applying transformers, each time extending and refining the scheme until the final scheme is achieved. Fig. 8.10 gives an overview of this development process, which is explained in detail in the remainder of this section.

### 8.3.1 Defining the content scheme

The first step in developing the department's web site is to define the content scheme. This task has to be performed manually because it is the first task during conceptual design and, consequently, there does not exist any other part of the scheme from which the content scheme could be derived. The result of this first step is a content scheme defined manually as depicted in Fig. 8.1.

### 8.3.2 Providing for overview and details pages

The next step in the development of the web scheme is to provide overview pages and details pages by applying transformers `OverviewPageForEnt` and `DetailsPageForEnt`, respectively. Transformer `DetailsPageForEnt` generates a details page class simply for each entity type of the scheme. Transformers `OverviewPageForEnt` generates an overview page class for each entity type except for those that are marked as `singleton`. Further, only those attributes are considered that are not marked as `details`. Consequently, the content scheme has to be annotated with these user-defined properties such that transformer `OverviewPageForEnt` works as intended.

Fig. 8.11 depicts the content scheme of the university department's web site, now annotated with user-defined properties `singleton`, `detail`, and `representative`. Attributes marked with property `representative` are to be displayed in supplementary index units and data units. Property `singleton` is attached to entity type `Department` such that transformer `OverviewPageForEnt` does not consider this entity type. Property `detail` is attached to attributes `photo`, `description`, and `abstract`, which are intended to be presented only in details pages. Hence, these attributes are

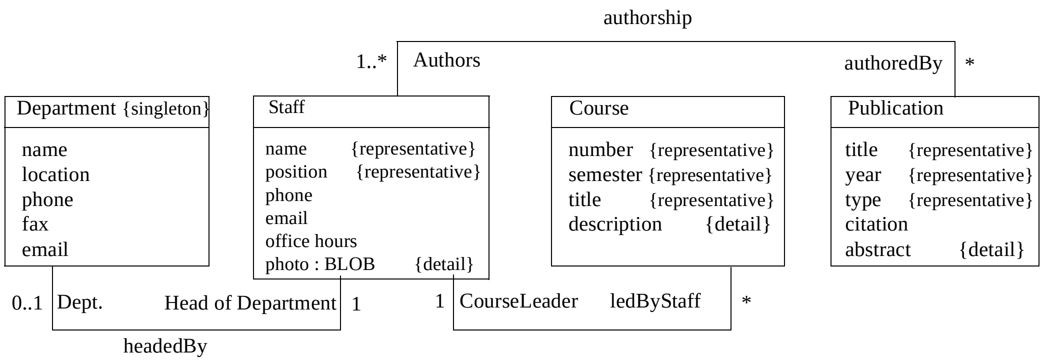


Figure 8.11: Annotated content scheme of department's web site.

considered only by transformer `DetailsPageForEnt` are neglected by transformer `OverviewPageForEnt`.

Fig. 8.12 illustrates the result of applying transformers `DetailsPageForEnt` and `OverviewPageForEnt` to the annotated content scheme. The application of transformer `OverviewPageForEnt` generates the landmark page classes `Course Overview`, `Staff Overview`, and `Publication Overview`. The index units define that all those attributes are to be displayed that have not been marked as `detail` in the content scheme. Note that no overview page class has been generated for singleton entity type `Department`. The application of transformer `DetailsPageForEnt` generates the page classes `Department Details`, `Course Details`, `Staff Details`, and `Publication Details`. The data units contained in these page classes have user-defined property `main_unit` set. Further, these data units define that all the attributes of their respective entity type are to be displayed.

Note that the page classes, data units and index units generated by the applications of transformers `DetailsPageForEnt` and `OverviewPageForEnt` do not yet correspond exactly to those of the scheme of Fig. 8.2 defined manually. For example, page class `Department Details` is not yet declared as the site's home page. To achieve a total correspondence, one has to rework the transformer's result as follows: Page class `Department Details` has to be marked as home page and has to be renamed to "Department Page". Further, data unit `Details` representing the sole department record has to be renamed to `Department`. These adaptations are illustrated in the illustrations of the hypertext scheme that follow.

Of course, the transformer applications could have been individualized such that above mentioned particularities would have been considered already by

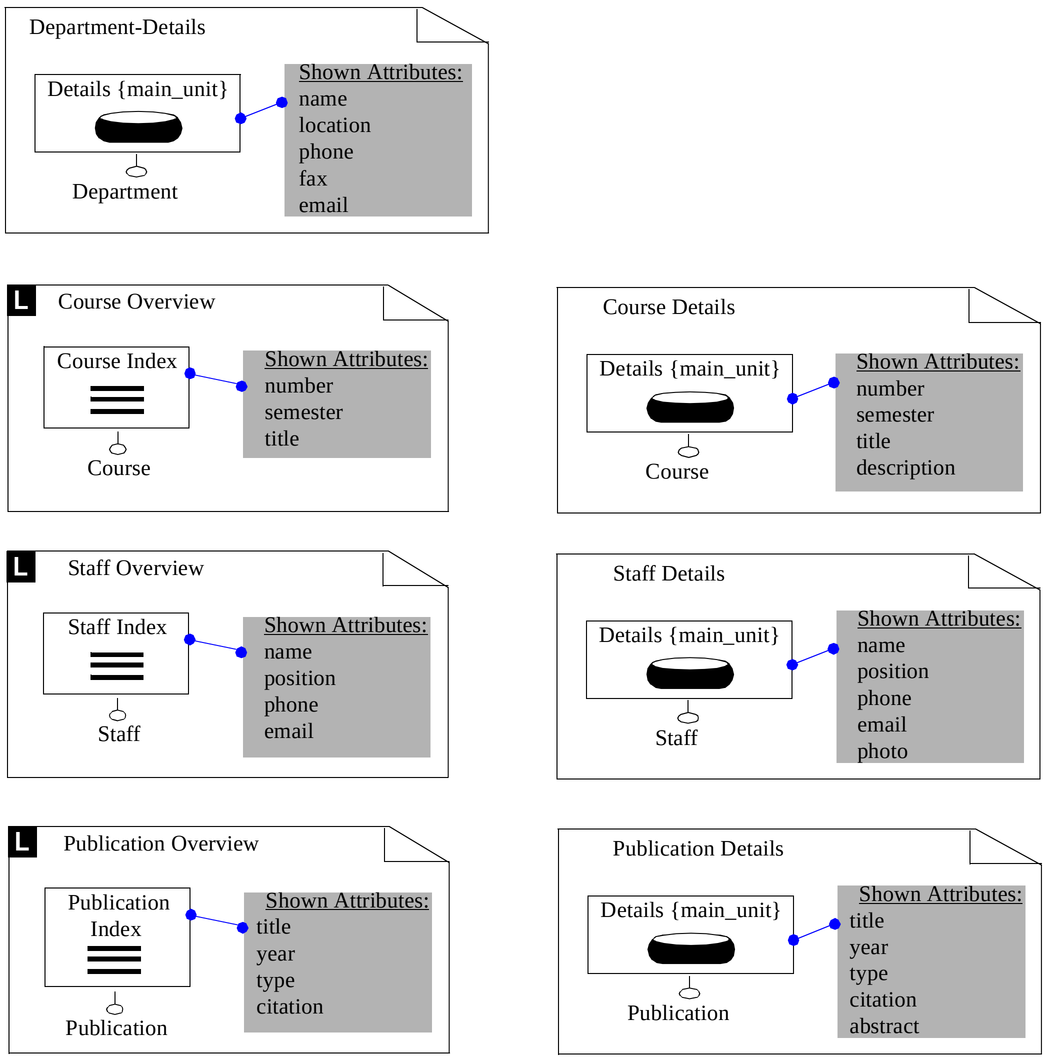


Figure 8.12: Hypertext scheme with isolated overview and details page classes.

the transformer. For example, the construction expression for the name of the department's details page could have been overridden. However, this individualization probably had required more effort than simply reworking the scheme after the transformer application.

### 8.3.3 Providing for supplementary units

In the next step, details page classes, which are for presenting one item in detail, are extended by supplementary content units. These content units present information that is directly related to the page's main item. For this purpose, transformer `GenerateSupplementaryUnits` is applied to the scheme that has resulted from the previous step. Note that supplementary content units are to display only representative attributes. These attributes are marked in the content scheme with user-defined property `representative`, which is considered by the query template of transformer `GenerateSupplementaryUnits`.

Fig. 8.13 illustrates the result of applying transformer `GenerateSupplementaryUnits` to the hypertext scheme of Fig. 8.12. Please ignore the gray links for the moment, they will be explained together with the next modelling task to be performed. The application of transformer `GenerateSupplementaryUnits` has generated supplementary data units `Head of Department` and `Course Leader`, and the supplementary index units `Publications`, `Courses`, and `Authors`.

Note that the transformer application is individualized by application-specific constraint `RELROLE` $\neq$ `"Dept"`, which excludes relationship role `Dept` from consideration. In absence of this application-specific constraint, page class `Staff Details` would get extended by a supplementary data unit representing the department that is headed by the respective staff member. Although this could make sense as well, it is probably not intended if the name of the sole department is presented in the header of the web site anyway. Thus, entity type department does not declare any attribute as `representative`, since no supplementary unit representing this entity type is to be generated.

The scheme of Fig. 8.13, which is the result of a transformer application, differs from the scheme of Fig. 8.2, which has been edited manually, only by different captions of two data units: Data units `Head of Department` and `Course Leader` in Fig. 8.12 are captioned `Head` and `Leader` in Fig. 8.2, respectively. Of course, if an individual caption is desired, a modeler may simply rework the transformer's result.

### 8.3.4 Interconnecting units

In the next step, links are generated from index units to main data units and from supplementary data units to main data units. Each data unit

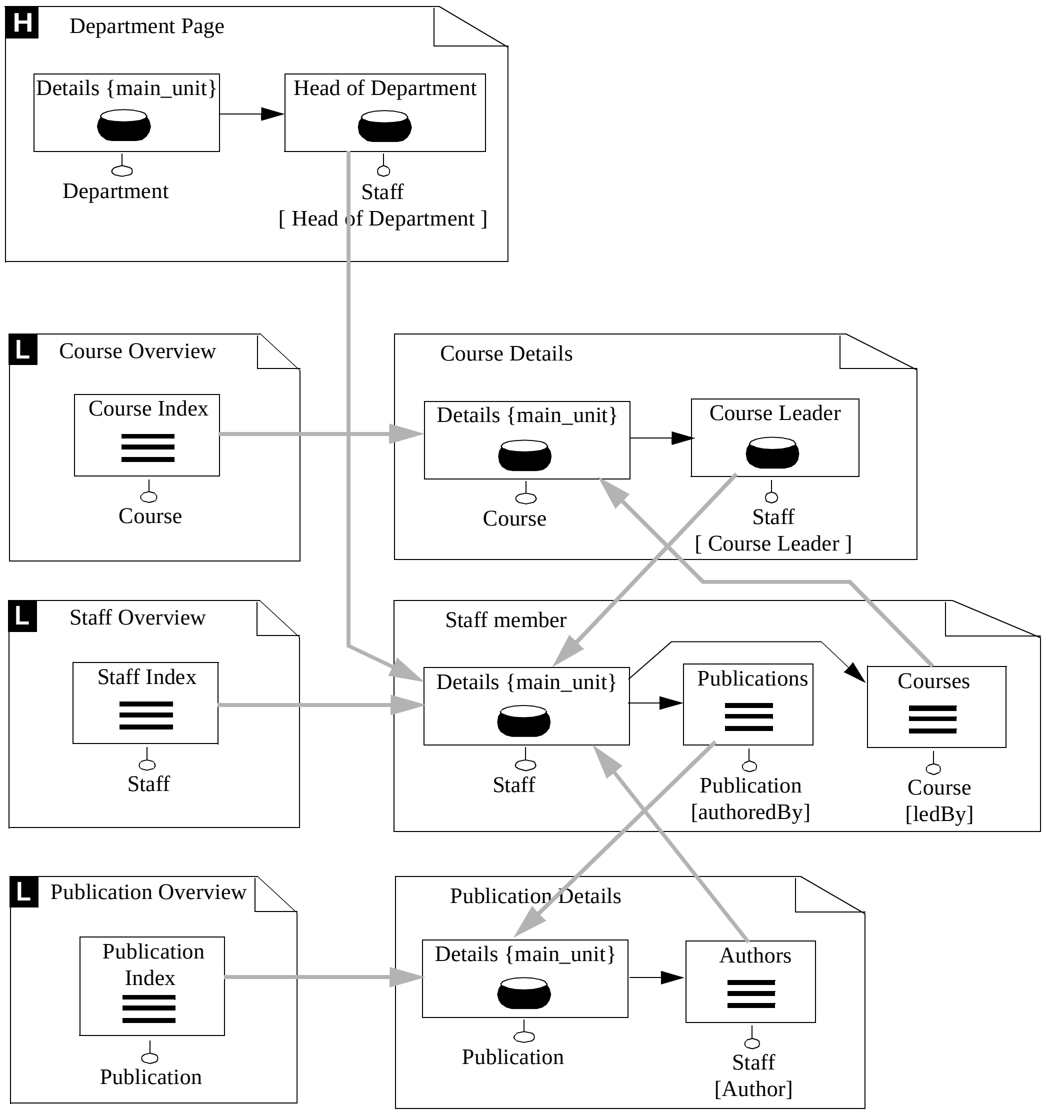


Figure 8.13: Hypertext scheme with page classes comprising supplementary units; Gray links: result of applying transformers `ConnectIUtoMainDU` and `ConnectSupplDUtoMainDU`.

that is not marked as a main data unit is considered as supplementary. These two tasks are captured by transformers `ConnectIUtoMainDU` and `ConnectSupplDUtoMainDU`, respectively, which are applied to the scheme resulting from the previous step. Fig. 8.13 illustrates the result of these applications, where the newly generated links are shown in gray color. Note that these links are ordinary links, and the gray color is used for illustration purpose only.

### 8.3.5 Providing for category index

In the last step, two index units `year-values` and `type-values` serving as category indices for index unit `Publication Index` are defined. Index unit `year-values` displays a list of year-values that, upon selecting one of these values, constrains the items displayed in index unit `Publication Index`. Index unit `type-values`, which displays a list of publication types, works analogously.

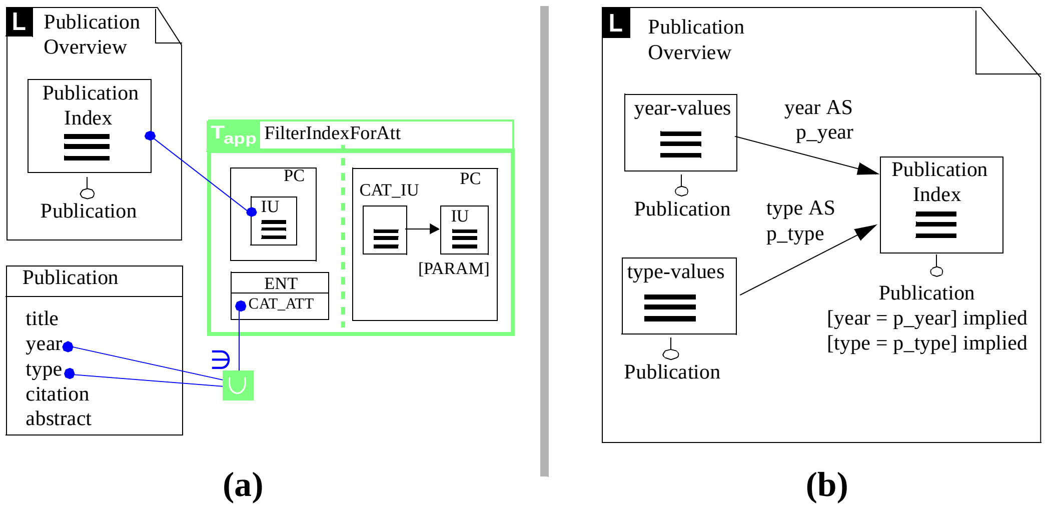


Figure 8.14: (a) Individualized application of transformer `CategoryIndexForIU`; (b) result of (a).

For generating these category filters, transformer `CategoryIndexForIU` is applied as shown in Fig. 8.14(a). The application-specific constraints express that only index unit `PublicationIndex` and, from all the attributes of the underlying entity type, only attributes `year` and `type` are to be considered. Fig. 8.14(b) illustrates the result of this application. It corresponds exactly to page class `Publication Overview` of the scheme that is shown in Fig. 8.2 and that has been defined manually.

# Chapter 9

# Conclusion

## Contents

This chapter summarizes the concepts of *TBE* as presented in this thesis and gives an outlook to future work.

## 9.1 Summary

We have presented the language *TBE* for defining and applying web scheme transformers. Such transformers facilitate the process of modelling web applications in that they assist modelers in performing recurrent modelling tasks. Each transformer is executable in the sense that, when applied to a scheme, it performs extensions and refinements as they would otherwise have to be performed by the modeler again and again in the same way.

For defining transformers, we have introduced the language *TBE* that combines a visual approach with a by-example approach. *TBE* is a visual language because it is based on visual scheme elements that are used by conceptual modelling languages for defining web schemes. *TBE* is a by-example language as a transformer is defined by giving a generic example of a scheme element configuration before transformation (the input configuration) and a scheme element configuration after transformation (the output configuration). These two parts are represented by a pair of a query template and a generative template, respectively, which make up the transformer's definition. From this definition, an executable transformer is derived as follows: From the query template, a query is derived that, when applied to a scheme, selects all scheme element configurations that match the transformer's input configuration. The operations derived from the generative template are applied to each scheme element configuration selected by the query template, each time generating an extended or refined scheme element configuration.

Defining transformers by-example facilitates understandability because the transformer definition by itself is a *comprehensive* and *comprehensible* description of the transformer's behavior.

For applying transformers, we have presented off-the-shelf applications, where the transformer is taken as is, and individualized applications, where the transformer's behavior is adapted specifically for each individual application. When applied off-the-shelf, a transformer behaves as described above. For adapting this behavior, one can individualize applications as follows: (1) Application-specific constraints further constrain the transformer's query template such that particular scheme element configurations can be excluded from consideration. (2) Application-specific expressions override expressions defined in the transformer's generative template in order to adapt, for example, naming policies for the scheme elements to be generated. (3) Individually pinned new-element variables adapt a generative template such that particular scheme elements are generated only once per transformer application rather than once for each scheme element configuration selected by the query template. (4) New-element variables can be individually turned into parameter variables such that the generative template extends or refines an existing scheme element instead of generating a new one.

Applications of transformers that may be individualized provide for flexibility and adaptability as necessary for using transformers in the process of web application development. Expressing individualizations in a practicable manner is particularly supported by graphical transformer applications.

In addition to the basic concepts of *TBE*, which cover many of the modelling tasks recurrently performed by modelers, advanced concepts have been developed that further increase the expressive power of *TBE*. We have introduced complex query templates, which allow to express disjunctions, negation, and universal quantification. We have presented complex generative templates, which provide delegation of details to separate, conditional, or hierarchically organized sub-templates. We have provided complex transformers, that (1) support nested query templates and nested generative templates for processing nested relations, and (2) cascading transformers for performing subsequent modelling tasks with a single transformer application.

The concepts of *TBE*, despite abstracting from a scheme's internal representation, provide enough expressive power for expressing typical modelling tasks. This has been demonstrated by developing a web scheme for a department's web site. It has been shown that, when developing the scheme manually, several modelling tasks are applied repeatedly. By utilizing transformers supporting these tasks, this web scheme can be developed more rapidly and conveniently.

Of course, as for any visual language, some modelling tasks cannot be expressed in a practicable manner with *TBE*. For example, *TBE* is not adequate for defining a transformer that derives an arrangement of scheme elements such that links do not cross each other. However, we feel that *TBE* is adequate for defining and applying transformers that considerably facilitate the process of conceptual modelling of web applications.

We have precisely defined the semantics of the visual by-example language *TBE* based on a relational representation of schemes, which is independent of the representation of schemes as defined by particular modelling languages, and based on a calculus called TBE-QML defined over such relational representations. Thereby, the concept of *TBE* can be easily employed for various modelling languages and not only for WebML, which has served as example for illustrating the use of by-example transformers.

We have investigated how *TBE* can be incorporated into existing CASE tools, i.e. how CASE tools can be extended towards defining and applying transformers. This extension requires to (1) allow for specifying by-example transformers and their application in a practicable manner, and (2) to implement the TBE-QML language, which has been used for defining the semantics of by-example transformers. For achieving a prototypical *TBE* system, two master's theses cover the aspects (1) and (2), respectively. In [136], *TransEd* has been developed, which is a prototype of an editor for WebML schemes that allows to define and apply transformers. In [81], a generic implementation of TBE-QML is presented, i.e. an implementation that can be easily adapted for transforming different schemes of different modelling languages.

| Approaches / Requirements | TBE | WebML's Built-In Trans-formation | Araneus' Built-In Trans-formations | OO-H Trans-formation Rules | Macro-Languages |
|---|---|---|---|---|---|
| Understandability | + | ~ | ~ | ~ | ~ |
| Flexibility | + | – | – | – | ~ |
| Adaptability | + | – | – | – | ~ |
| Usability | + | + | + | + | + |
| Adequate for user-defined Transformers | + | – | – | ~ | ~ |
| Generality | + | – | – | – | – |

Figure 9.1: Fulfilment of requirements for web scheme transformation: Comparing *TBE* to other approaches.

Fig. 9.1 summarizes the benefits of *TBE* when compared with existing approaches for web scheme transformation. These existing approaches and their benefits and drawbacks have been investigated in the introduction. They have the major drawback that they require the specification of operations over the internal representations of schemes and confine thereby understandability, adequateness, and generality. Further, existing approaches do not or only moderately fulfill the requirements of flexibility and adaptability. The benefit of *TBE* stems from abstracting from a scheme's internal representation and from providing individualized transformer applications.

## 9.2 Future Work

This thesis has focused on transformers for extending and refining conceptual schemes of web applications during the development process. Beyond this focus, research can be continued in several directions, which are briefly discussed in this section. In particular, we plan to (1) apply the concepts of *TBE* to different modelling languages, (2) to utilize query templates for asserting design criteria, and (3) to apply transformers as continuous in order to maintain dependencies between scheme elements.

### 9.2.1 Application to different modelling languages

The language constructs of *TBE* are not limited to *web* modelling languages but can be easily applied to other modelling languages as well. Transformers are particularly useful where one part of a scheme typically is derived from or depends on another part of the scheme. For example, as described in [126], the Unified Modelling Language (UML) provides various diagram types for describing a system from different perspectives or abstraction levels. Hence, various UML models of the same system are dependent and strongly overlapping. Based on these dependencies, *TBE* can be utilized for transforming one diagram into another one.

Further, the concepts of *TBE* can be used for expressing transformations between schemes of different modelling languages. This make sense where different modelling languages have to be used for modelling overlapping and interdependent aspects of the same system. Such transformations can be expressed easily in that a transformer's query template accesses a scheme of one modelling language whereas the generative template extends a scheme of another modelling language. For example, the web modelling language WebML (and its CASE tool WebRatio) supports the definition of different hypertext views for different groups of users. For the purpose of modelling the different user groups, use-case diagrams could be utilized, yet such diagrams are not part of WebML (and are not supported by WebRatio). However, in *TBE*, one could define a transformer whose query template extracts groups of users as defined in a use-case diagram drawn with a UML editor and whose generative template generates WebML hypertext views for each selected user group.

Moreover, *TBE* 's concepts could be beneficially employed for precisely describing software design patterns and for facilitating their instantiation. They could be used for transforming diagrams of general-purpose diagram editors like Microsoft Visio, or they could perform transformations between models describing business processes at different levels of abstraction.

### 9.2.2 Utilizing query templates for asserting design criteria

*TBE* 's query templates are capable of checking whether scheme element configurations exist that fulfill particular constraints, and consequently, whether scheme element configurations exist that violate such constraints. Thus, query templates could be used for asserting design criteria just like SQL-queries can be used for asserting data consistency criteria. If a query template $Q$ is applied to a scheme $S$ as an assertion of the form CHECK NOT EXISTS (Q), then an error would be raised whenever scheme $S$ is changed such that query template $Q$ returns any tuple. However, it would be impracticable if the assertion reacted on every atomic change applied to the scheme, because fulfilling a design criteria may require a couple of changes. Thus, reasonable events that trigger the assertion need to be defined, e.g. when the modeler decides to have a prototype generated from the scheme.

**Example 103** *Consider a query template* `NonReachPC` *that selects non-reachable page classes, i.e. page classes that are neither targeted by any link nor are a home page or a landmark. If a non-reachable page class is considered as violating a design criterion, then this query template could be applied as* CHECK NOT EXISTS (NonReachPC). *Henceforth, an error is raised whenever a page class becomes not reachable.*

*Note, however, that if this assertion would react on every change of the scheme, it would become impossible to define a new page class. Since a page class first must be defined before it can be targeted by a link,* every *page class newly created is isolated and causes the assertion to fail.*

This thesis has focused on transformers supporting modelling tasks, and query templates provide adequate expressive power for this purpose. However, expressing assertions may require additional expressive power. For example, it may be required to check whether a generalization hierarchy is cyclic or whether two page classes are connected transitively. For this purpose, query templates could be extended to support recursion.

### 9.2.3 Continuous transformer applications

Transformer applications as presented in this thesis “die” once they have been processed, and any correlation between the initial scheme elements and the scheme elements generated based on them is lost. Consequently, if the initial scheme elements are changed later on, these changes are not propagated to dependant scheme elements. In order to maintain such a dependance, continuous transformer applications could be introduced that, once applied, keep track of which scheme elements have been generated based on other ones. Then, if the initial scheme elements are changed, dependent scheme elements are adapted just as if the transformer had originally been applied to the (changed) initial scheme elements.

**Example 104** *Reconsider transformer* `EntryUnitForEnt`*, which generates an entry unit with entry fields based on entity types and their attributes. If this transformer is applied to an entity type* e*, a corresponding entry unit with entry fields is generated. However, if the entity type is changed later on, e.g. by adding a new attribute to* e*, then the corresponding entry unit is* not *extended by a new entry field.*

*In contrast, if the same transformer* `EntryUnitForEnt` *were applied as continuous to entity type* e*, then this application would maintain a coherence between the entity type* e *and the entry unit generated therefrom as follows: (1) new entry fields are added whenever entity type* e *is extended by new attributes; (2) entry fields or the entry unit are deleted whenever attributes of* e *or entity type* e *itself are removed, respectively, and (3) entry fields or the entry unit are renamed whenever the name of their corresponding attribute or the name of entity type* e *is changed.*

Technically, supporting continuous applications of transformers requires a "memory" such that the application remembers the query template's result relation and which scheme elements have been generated from which tuple of this relation. If the scheme changes such that re-evaluating the query template achieves a different result, then the following cases are distinguished: (a) for each tuple that has not yet been in the query result, new dependant scheme elements are generated; (b) for each tuple that is not contained in the query result any more, the dependant scheme elements are removed; (c) if it is identifiable that a tuple has changed, then the changes are propagated to the scheme elements generated based on this tuple.

Of course, it may be difficult to detect whether a scheme element is to be modified or whether one scheme element is to be deleted whereas another one is to be generated. We have also to keep in mind that a modeler might have modified scheme elements generated by the transformer or might have referred to them such that a conflict would arise if these scheme elements were adapted by the continuous transformer application. Furthermore, analogously to query templates that are applied as assertions, continuous transformer applications should not react on every change applied to a scheme. Discussing these issues in detail is subject to future work.

# List of Figures

# Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesly Publishing Company, Reading, Massachusetts, USA, 1995.

[2] Suad Alagić and Philip A. Bernstein. A model theory for generic schema management. *Lecture Notes in Computer Science*, 2397:228–239, 2002.

[3] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A graphical query language with recursion. In *Proceedings of the Third International Conference on Human-Computer Interaction*, volume 2 of *Designing and Using Human-Computer Interfaces and Knowledge Based Systems; Graphics*, pages 384–391, 1989.

[4] Michele Angelaccio, Tiziana Catarci, and Giuseppe Santucci. QBD$*$: A graphical query language with recursion. *IEEE Transactions on Software Engineering*, 16(10):1150–1163, October 1990. Special Section on Visual Programming.

[5] Paolo Atzeni and Alessio Parente. Specification of web applications with ADM-2. In *Proceedings of the First International Workshop on Web-Oriented Software Technology (IWWOST'01)*, June 2001.

[6] Nevzat Hurkan Balkir, Gultekin Özsoyoglu, and Z. Meral Özsoyoglu. A graphical query language: VISUAL and its query processing. *IEEE Transactions on Knowledge & Data Engineering*, 14(5):955–978, September 2002.

[7] Nevzat Hurkan Balkir, E. Sukan, Gultekin Özsoyoglu, and Z. Meral Özsoyoglu. VISUAL: A graphical icon-based query language. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 524–533. IEEE Computer Society, 1996.

[8] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, 16(3):311–322, December 1987.

[9] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings Publishing Company Inc., Redwood City, California, USA, 1992.

[10] Brigham Bell and Clayton Lewis. Chemtrains: A language for creating behaving pictures. In *Proceedings of the 1993 IEEE Workshop on Visual Languages, August 24-27, Bergen, Norway*, pages 188–195. IEEE Computer Society, 1993.

[11] Alexandru Berlea and Helmut Seidl. Transforming XML Documents Using fxt. *Journal of Computing and Information Technology*, 1(10):19–35, March 2002.

[12] Mark Bernstein. Patterns of hypertext. In *Proceedings of the ninth ACM conference on Hypertext and hypermedia*, pages 21–29. ACM Press, 1998.

[13] D. Blostein and A. Schuerr. Computing with graphs and graph rewriting. *Software Practice and Experience*, 29(3):1–21, 1999.

[14] Scott Boag, Don Chamberlin, Daniela Florescu, Jonathan Robie, and Jerome Siméon. "XQuery 1.0: An XML Query Language" (W3C Working Draft). available at http://www.w3.org/TR/2003/WD-xquery-20031112/, November 2003.

[15] Mario A. Bochicchio, Roberto Paiano, and Paolo Paolini. JWeb: An HDM environment for fast development of web applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, June 07 - 11, Florence, Italy*, volume II. IEEE Computer Society, 1999.

[16] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[17] Aldo Bongio, Stefano Ceri, Piero Fraternali, and Andrea Maurino. Modeling data entry and operations in WebML. In *Proceedings of WebDB 2000*, volume 1997 of *Lecture Notes in Computer Science, LNCS*. Springer, 2001.

[18] G. Booch. *Object Oriented Analysis and Design with Applications, Second Edition.* Benjamin/Cummings Publishing, Reading (MA), USA, 1994.

[19] Borland. *ModelMaker: the Borland Delphi Productivity / CASE tool. Version 7.20.* http://www.modelmakertools.com/mm.htm, 2004.

[20] Alan Borning, Robert Duisberg, Bjorn N. Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), October 4-8, 1987, Orlando, Florida*, pages 48–60, December 1987.

[21] Marco Brambilla, Stefano Ceri, Sara Comai, Piero Fraternali, and Ioana Manolescu. Model-driven specification of web services composition and integration with data-intensive web applications. *IEEE Bulletin of Data Engineering*, 4(25):53–59, December 2002.

[22] Marco Brambilla, Stefano Ceri, Sara Comai, Piero Fraternali, and Ioana Manolescu. Model-driven development of web services and hypertext applications. In *Proceedings of the 7th World Multi-Conference on Systemics, Cypernetics, and Informatics (SCI) 2003*, July 2003.

[23] Marco Brambilla, Stefano Ceri, Sara Comai, Piero Fraternali, and Ioana Manolescu. Specification and design of workflow-driven hypertexts. *Journal of Web Engineering (JWE)*, 1(2):163–182, April 2003.

[24] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, New York , NY , USA, 1996.

[25] Cristina Cachero, Jaime Gomez, Antonio Parraga, and Oscar Pastor. The OO-H method. In *Proceedings of the First International Workshop on Web-Oriented Software Technology (IWWOST'01)*, June 2001.

[26] Donatella Castelli and Serena Pisani. A transformational approach to correct schema refinements. In Tok Wang Ling, Sudha Ram, and Mong-Li Lee, editors, *Conceptual Modeling - ER '98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings*, volume 1507 of *Lecture Notes in Computer Science*, pages 226–239. Springer, 1998.

[27] T. Catarci, M.F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260, 1997.

[28] R. G. G. Cattell, Douglas K. Barry, and Dirk Bartels, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Series in Data Management Systems, 1997.

[29] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2003.

[30] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: a graphical language for querying and restructuring XML documents. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1171–1187, May 1999.

[31] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, and Letizia Tanca. Complex queries in XML-GL. In *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000), Como, Italy*, volume 2, pages 888–893, March 2000.

[32] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):137–157, 2000.

[33] Stefano Ceri, Piero Fraternali, Maristella Matera, and Andrea Maurino. Designing multi-role, collaborative web sites with WebML: a conference management system case study. In *Proceedings of the First International Workshop on Web-Oriented Software Technology (IWWOST'01)*, June 2001.

[34] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. Data-driven, one-to-one web site generation for data-intensive applications. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*, pages 615–626, San Francisco, September 1999. Morgan Kaufmann.

[35] Peter P. S. Chen. The entity-relationship model — toward a unified view of data. *Proceedings of the 1th Conference on Very Large Databases*, page 173, 1975. Also published in/as: ACM Transactions on Database Systems, Vol.1 No.1, Mar.1976, pp.9–36.

[36] James Clark. XSL Transformations (XSLT), Version 1.0, W3C recommendation. http://www.w3.org/TR/1999/REC-xslt-19991116, November 1999.

[37] Jim Conallen. Modeling Web application architectures with UML. *Communications of the ACM*, 42(10):63–70, 1999.

[38] Jim Conallen. *Building Web Applications with UML.* Number XXI in The Addison-Wesley Object Technology Series. Addison-Wesley Longman, Amsterdam, 2nd edition, 2003.

[39] Mariano P. Consens, Isabel F. Cruz, and Alberto O. Mendelzon. Visualizing queries and querying visualizations. *SIGMOD Record*, 21(1):39–46, 1992.

[40] Gennaro Costagliola, Filomena Ferrucci, and Rita Francese. Web engineering: Models and methodologies for the design of hypermedia applications. In S.K. Chang, editor, *Handbook of Software Engineering & Knowledge Engineering*, volume 2 of *Emerging Technologies*, pages 181–199. World Scientific, 2002.

[41] I. F. Cruz. User-defined visual query languages. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 224–231, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[42] I. F. Cruz and P. S. Leveille. Implementation of a constraint-based visualization system. In *Proceedings of the Symposium on Visual Languages*, pages 13–20, Los Alamitos, CA, USA, September 2000. IEEE Computer Society Press.

[43] Isabel F. Cruz. DOODLE: a visual language for object-oriented databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 71–80. ACM Press, 1992.

[44] Isabel F. Cruz, Michael Averbuch, Wendy T. Lucas, Melissa Radzyminski, and Kirby Zhang. Delaunay: A database visualization system. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 510–513. ACM Press, 1997.

[45] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration.* The MIT Press, 1993.

[46] Simon W. Day. Oracle 9i designer: Technical overview. Technical report, Oracle Corporation, April 2002.

[47] A. Diaz, T. Isakowitz, V. Maiorana, and G. Gilabert. RMC: A tool to design WWW applications. In *Proceedings of the Fourth International World Wide Web Conference, Boston, December 1995*, pages 11–14, 1995.

[48] Amnon H. Eden, Joseph Gil, and Amiram Yehudai. Automating the application of design patterns. *Journal of Object Oriented Programming (JOOP)*, 10(2):44–46, May 1997.

[49] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, 1992.

[50] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Inc., Reading, Massachusetts, USA, 3rd edition, 2000.

[51] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. STRUDEL: a Web site management system. In Joan M. Peckman, editor, *Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD 1997: May 13–15, 1997, Tucson, Arizona, USA*, volume 26(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 549–552, New York, NY 10036, USA, 1997. ACM Press.

[52] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: experiences with a web-site management system. In *SIGMOD*, pages 414–425, 1998.

[53] Mary F. Fernández, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Declarative specification of Web sites with Strudel. *VLDB Journal*, 9(1):38–55, 2000.

[54] Brian Foote, Neil Harrison, and Hans Rohnert. *Pattern Languages of Program Design 4*. Addison-Wesley, Reading, Massachusetts, USA, December 1999.

[55] Piero Fraternali. Web development tools: a survey. *Computer Networks and ISDN Systems*, 30(1-7):631–633, April 1998.

[56] Piero Fraternali. Tools and approaches for developing data-intensive Web applications: a survey. *ACM Computing Surveys*, 31(3):227–263, 1999.

[57] Piero Fraternali, Maristella Matera, and Andrea Maurino. Conceptual-level log analysis for the evaluation of web application quality. In *Submitted for publication; available at* http://www.webml.org, June 2003.

[58] Piero Fraternali and Paolo Paolini. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In Hans-Jörg Schek, Fèlix Saltor, Isidro Ramos, and Gustavo Alonso, editors, *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, volume 1377 of *Lecture Notes in Computer Science*, pages 421–435. Springer, 1998.

[59] Piero Fraternali and Paolo Paolini. Model-driven development of web applications: the AutoWeb system. *ACM Transactions on Information Systems (TOIS)*, 18(4):323–382, 2000.

[60] Fraunhofer Gesellschaft. IPSI-XQ - The XQuery Demonstrator. Version 1.3.2. Available at http://www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/index_e.html, February 2004.

[61] Fred. *Fred / JavaFrames project home page.* http://practise.cs.tut.fi/fred/index.html, 2004.

[62] Ernest Friedman-Hill. *Jess in Action.* Java Rule-based Systems. Manning Publications Co., July 2003.

[63] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Massachusetts, USA, 1994.

[64] F. Garzotto, P. Paolini, D. Bolchini, and S. Valenti. "modeling-by-patterns" of web applications. In P. Chen et al., editor, *Proceedings of the ER'99 Workshop on Advances in Conceputal Modeling*, volume 1727 of *Lecture Notes in Computer Science (LNCS)*, pages 293–306. Springer Verlag, November 1999.

[65] Franca Garzotto, Luca Mainetti, and Paolo Paolini. Adding multimedia collections to the dexter model. In *Proceedings of the ECHT'94 European Conference on Hypermedia Technologies*, Papers, pages 70–80. ACM Press, 1994.

[66] Franca Garzotto, Paolo Paolini, and Luciano Baresi. Supporting reusable web design with HDM-Edit. In *Proceedings of the 34th Annual*

*Hawaii International Conference on System Sciences, January 03-06, 2001, Maui, Hawaii*, volume 7, 2001.

[67] Franca Garzotto, Paolo Paolini, and Daniel Schwabe. HDM – A model-based approach to hypertext application design. *ACM Transactions on Information Systems*, 11(1):1–26, 1993.

[68] GOF. *The Hillside Group Home Page*. http://www.hillside.net.

[69] Jaime Gómez and Cristina Cachero. *Information modeling for internet applications*, chapter OO-H Method: extending UML to model web interfaces, pages 144–173. Idea Group Publishing, 2003.

[70] Jaime Gómez, Cristina Cachero, and Oscar Pastor. Extending a conceptual modelling approach to web application design. In *CAiSE'00*, pages 79–93, 2000.

[71] Jaime Gómez, Cristina Cachero, and Oscar Pastor. Conceptual modeling of device-independent Web applications. *IEEE MultiMedia*, 8(2):26–39, April 2001.

[72] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Architecture-oriented programming using FRED. In *Proceedings of the 23rd international conference on Software engineering*, pages 823–824. IEEE Computer Society, 2001.

[73] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Generating application development environments for java frameworks. In *Generative and Component-Based Software Engineering, Third International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings*, volume 2186 of *Lecture Notes in Computer Science*. Springer, 2001.

[74] Lutz J. Heinrich. *Management von Informatik-Projekten*. Oldenbourg Verlag, München, Deutschland, 1997.

[75] IBM. Eclipse platform technical overview, Version 2.1. www.eclipse.org, February 2003.

[76] Tomás Isakowitz, Edward A. Stohr, and P. Balasubramanian. RMM: a methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, 1995.

[77] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley Longman, Inc., Reading, Massachusetts, 1999.

[78] Andri Ioannidou Jim Gindling, Jennifer Loh, Olav Lokkebo, and Alexander Repenning. Legosheets: A rule-based programming, simulation and manipulation environment for the lego programmable brick. In *Proceedings of the 11th International IEEE Symposium on Visual Languages, September 5-9, 1995, Darmstadt, Germany*, pages 172–179. IEEE Computer Society, 1995.

[79] Timo Niemi Kalervo Järvelin and Airi Salminen. The visual query language CQL for transitive and relational computation. *Data & Knowledge Engineering*, 35(1):39–51, October 2000.

[80] Frank Kappe. Knowledge management with the Hyperwave eKnowledge Infrastructure. Technical report, Hyperwave Information Management, Inc., October 2001.

[81] Michael Karlinger. Defining and applying by-example transformers in WebML: concept and implementation. Master's thesis, University of Linz, Department of Data & Knowledge Engineering, Altenbergerstr. 69, 4040 Linz, Austria, To appear.

[82] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4):741–843, 1995.

[83] Anthony C. Klug. Abe: A query language for constructing aggregates-by-example. In Harry K. T. Wong, editor, *Proceedings of the First LBL Workshop on Statistical Database Management, Melno Park, California, USA, December 2-4, 1981*, pages 190–205. Lawrence Berkeley Laboratory, 1981.

[84] Nora Koch. A comparative study of methods for hypermedia development. Technical Report 9905, Ludwig-Maximilians-Universität München, November 1999.

[85] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, Reading, MA, 3rd edition, December 2003.

[86] Phyo Kyaw and Cornelia Boldyreff. A survey of Hypermedia design methods in the context of World Wide Web design. Technical Report

03-98, Computer Science Department, University of Durham, United Kingdom, 1998.

[87] Michel Lacroix and Alain Pirotte. Domain-oriented relational languages. In *Proceedings of the Third International Conference on Very Large Data Bases, October 6-8, 1977, Tokyo, Japan*, pages 370–378. IEEE Computer Society, October 1977.

[88] Stephan Lechner, Günter Preuner, and Michael Schrefl. Translating XQuery into XSLT. In Hiroshi Arisawa, Yahiko Kambayashi, Vijay Kumar, Heinrich C. Mayr, and Ingrid Hunt, editors, *ER 2001 Workshops, HUMACS, DASWIS, ECOMO, and DAMA, Yokohama Japan, November 27-30, 2001, Revised Papers*, volume 2465 of *Lecture Notes in Computer Science*, pages 239–252. Springer Verlag, 2002.

[89] Stephan Lechner and Michael Schrefl. Transformers-by-example: pushing reuse in conceptual web application modelling. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1654–1661. ACM Press, March 2004.

[90] Stephan Lechner and Micheal Schrefl. Defining web schema transformers by example. In V. Marik, W. Retschitzegger, and O. Stepankova, editors, *Database and Expert Systems Applications, Proceedings of the 14th International Conference (DEXA 2003), Prague, Czech Republic, September 2003*, Lecture Notes in Computer Science, LNCS Vol. 2736, pages 46–56. Springer Verlag, 2003.

[91] Mengchi Liu. Deductive database languages: problems and solutions. *ACM Computing Surveys*, 31(1):27–62, March 1999.

[92] Nikos A. Lorentzos and Konstantinos A. Dondis. Query by example for nested tables. In Gerald Quirchmayr, Erich Schweighofer, and Trevor J.M. Bech-Capon, editors, *Database and Expert Systems Applications, Proceedings of the 9th International Conference (DEXA 2003), Vienna, Austria, August 1998*, Lecture Notes in Computer Science, LNCS Vol. 1460, pages 716–725. Springer Verlag, 1998.

[93] Georges Louis and Alain Pirotte. A denotational definition of the semantics of DRC, A domain relational calculus. In VLDB Endowment, editor, *Very large data bases: eighth International Conference on Very Large Data Bases, Mexico City, Mexico, September 8–10, 1982*, pages 348–356, P.O. Box 2245, Saratoga, CA, USA, 1982. VLDB Endowment.

[94] John J. Marciniak, editor. *Encyclopedia of Software Engineering*, chapter Brad A. Myers: Program Visualization; also in [103]. John Wiley & Sons, March 2002.

[95] Maristella Matera, Andrea Maurino, Stefano Ceri, and Piero Fraternali. Model-driven design of collaborative Web applications. *Software Practice and Experience*, 33(8):701–732, July 2003.

[96] G. Mecca, P. Atzeni, A. Masci, G. Sindoni, and P. Merialdo. The Araneus Web-based management system. In *ACM SIGMOD'98*, pages 544–546. ACM Press, 1998.

[97] Paolo Merialdo, Paolo Atzeni, Marco Magnante, Giansalvatore Mecca, and Marco Pecorone. Homer: a model-based case tool for data-intensive web sites. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, page 586. ACM, 2000.

[98] Paolo Merialdo, Paolo Atzeni, and Giansalvatore Mecca. Design and development of data-intensive web sites: The araneus approach. *ACM Transactions on Internet Technology*, 3(1):49–92, 2003.

[99] B. Meyer. *Eiffel - the language.* Prentice Hall Object-Oriented Series. Prentice Hall, 1992.

[100] Microsoft Corporation. BizTalk Mapper Functinality (Part of the BizTalk Server 2004). available at http://www.microsoft.com/biztalk/.

[101] Tommi Mikkonen. Formalizing design patterns. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 115–124. IEEE Computer Society Press / ACM Press, 1998.

[102] Francesmary Modugno, Albert T. Corbett, and Brad A. Myers. Graphical representation of programs in a demonstrational visual shell - an empirical evaluation. *ACM Transactions on Computer-Human Interaction*, 4(3):276–308, 1997.

[103] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[104] Brad A. Myers. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI conference on Human factors in computing*

*systems*, pages 534–541. ACM Press/Addison-Wesley Publishing Co., 1998.

[105] Marc Nanard, Jocelyne Nanard, and Paul Kahn. Pushing reuse in hypermedia design: golden rules, design patterns and constructive templates. In *9th ACM conference on Hypertext and hypermedia*, pages 11–20. ACM Press, 1998.

[106] Levent V. Orman. Queries = examples + counterexamples. *Information Systems*, 21(8):615–635, December 1996.

[107] Z. Meral Özsoyoglu and Gultekin Özsoyoglu. Summary-table-by-example: A database query language for manipulating summary data. In *Proceedings of the First International Conference on Data Engineering, April 24-27, 1984, Los Angeles, California, USA*, pages 193–202. IEEE Computer Society, 1984.

[108] A. Papantonakis and P. J. H. King. Syntax and semantics of Gql, a graphical query language. *Journal of Visual Languages and Computing*, 6(1):3–25, 1995.

[109] Anthony Papantonakis and Peter J. H. King. Gql, a declarative graphical query language based on the functional data model. In *Proceedings of the workshop on Advanced visual interfaces*, pages 113–122. ACM Press, 1994.

[110] Henderik Alex Proper. Data schema design as a schema evolution process. *Data Knowledge Engineering*, 22(2):159–189, 1997.

[111] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2000.

[112] Rational. *IBM Rational Rose XDE Modeler.* http://www.rational.com, 2004.

[113] R.M.Stallman. Emacs: The extensible, customizable, self-documenting display editor, 1979.

[114] Gustavo Rossi, Fernando Lyardet, and Daniel Schwabe. Developing hypermedia applications with methods and patterns. *ACM Computing Surveys (CSUR)*, 31(4es):8, 1999.

[115] Gustavo Rossi, Daniel Schwabe, and Alejandra Garrido. Design reuse in hypermedia applications development. In *Proceedings of the eighth ACM conference on Hypertext*, pages 57–66. ACM Press, 1997.

[116] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet. Improving web information systems with navigational patterns. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[117] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems (TODS)*, 13(4):389–417, 1988.

[118] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley Longman, Inc., Reading, Massachusetts, 1999.

[119] Sandia National Laboratories. Jess - the rule engine for the java platform. available at http://herzberg.ca.sandia.gov/jess/.

[120] Stefan Schiffer. *Visuelle Programmierung.* Addison Wesley, 1998.

[121] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Visual programming with graph rewriting systems. In *Proceedings of the 11th International IEEE Symposium on Visual Languages, September 5-9, 1995, Darmstadt, Germany*, pages 326–333. IEEE Computer Society, 1995.

[122] Daniel Schwabe, Rita de Almeida Pontes, and Isbela Moura. OOHDM-Web: An environment for implementation of hypermedia applications in the WWW. *ACM SigWEB Newsletter*, 8(2), June 1999.

[123] Daniel Schwabe and Gustavo Rossi. An object oriented approach to web-based application design. *Theory and Practice of Object Systems*, 4(4), 1998.

[124] Daniel Schwabe, Gustavo Rossi, Luiselena Esmeraldo, and Fernando Lyardet. Web design frameworks: An approach to improve reuse in Web applications. *LNCS*, 2016:335–346, 2001.

[125] Daniel Schwabe, Patricia Vilain, Robson Guimaraes, and Gustavo Rossi. A conference review system in OOHDM. In *Proceedings of the First International Workshop on Web-Oriented Software Technology (IWWOST'01)*, June 2001.

[126] Petri Selonen, Kai Koskimies, and Markku Sakkinen. How to make apples from oranges in UML. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34), January 3-6, 2001, Maui, Hawaii - Track 3.* IEEE Computer Society, 2001.

[127] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems (TODS)*, 6(1):140–173, 1981.

[128] SQL. The SQL standard 1999 (SQL'99); ISO/IEC 9075:1999.

[129] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1993.

[130] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE). http://java.sun.com.

[131] Sun Microsystems. *Java Server Pages (JSP) Version 2.0*. http://java.sun.com/products/jsp/index.html, 2003.

[132] Sun Microsystems. Javabeans(tm) specification 1.01 final release. http://java.sun.com/products/javabeans/docs/spec.html, 2004.

[133] Toufik Taibi and David Chek Ling Ngo. Formal specification of design patterns - a balanced approach. *Journal of object technology (JOT)*, 2(4), July/August 2003.

[134] Kenji Takahashi and Eugene Liang. Analysis and design of web-based information systems. *Computer Networks and ISDN Systems*, 29:1167–1180, September 1997.

[135] A.U. Tansel, M.E. Arkun, and G. Ozsoyoglu. Time-by-example query language for historical databases. *Transactions on Software Engineering (TSE)*, 15(4):464–478, 1989.

[136] Andreas Wabro. Ein Editor für WebML Schemata und Template-basierte Schematransformer (in German). Master's thesis, University of Linz, Department of Data & Knowledge Engineering, Altenbergerstr. 69, 4040 Linz, Austria, To appear.

[137] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Object Technology Series. Addison Wesley, 1998.

[138] WebML.org. WebML Document Type Definition (DTD) Version 3.0 rc11. available at http://www.webml.org, January 2003.

[139] WebML.org. Acme Furniture Example. available at http://www.webml.org, 2004.

[140] WebML.org. Web Forum Example. available at http://www.webml.org, 2004.

[141] WebRatio. *WebRatio Site Development Studio, Version 3.2.10.* http://www.webratio.com, 2003.

[142] Lutz Wegner, Sven Thelemann, and Stephan Wilke. Qbe-like queries and multimedia extensions in a nested relational DBMS. In *Proceedings of the International Conference on Visual Information Systems*, 1996.

[143] M. Zloof. Query by example. In *Proceedings of the National Computer Conference (NCC), Anaheim, CA*, volume 44. American Federation of Information Processing Societies (AFIPS), May 1975.

[144] M. Zloof. Query-by-example: A database language. *IBM System Journal*, 16(4):324–343, 1977.

# Curriculum Vitae

## Personal Record

| | |
|---|---|
| Name: | Mag. Stephan Lechner |
| Date of Birth: | January 1, 1973 |
| Place of Birth: | Schärding am Inn |
| Citizenship: | Austria |
| Home Address: | A-4040 Linz, Schumpeterstraße 15/7 |

## University Education

| | |
|---|---|
| 1992–2000: | Study of Business Information Systems ("Wirtschaftsinformatik") at the Johannes Kepler University of Linz, Austria graduation to "Mag." in 2000 |
| 2000–2004: | Doctoral study ("Doktoratsstudium der Sozial- und Wirtschaftswissenschaften") at the Johannes Kepler University of Linz, Austria |

## Professional Experience

| | |
|---|---|
| 1994–2000: | Tutor for courses in Data & Knowledge Engineering at the Johannes Kepler University of Linz |
| 1994–1996: | Instructor for courses in programming languages, algorithms, and systems engineering at the Johannes Kepler University of Linz |
| 1999–2002: | Project manager and system developer with Microsoft Business Solutions – Navision, Naviconsult AG, Linz |
| since 2000: | University assistant at the University of Linz, Department of Business Information Systems |