Transformation selection for aptness-based web retrieval

B. van Gils basvg@acm.org
H.A. Proper erikp@acm.org
P. van Bommel p.vanbommel@science.ru.nl
P. de Vrieze p.devrieze@science.ru.nl

Radboud University Nijmegen
Institute for Computing and Information Sciences
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract

A myriad of resources can be found on the Web today, and finding (topically) relevant resources for a given information need is a daunting task. Even if relevant resources can be found, they may not be *apt* for the searcher in a given context: some properties of the resource may be "wrong" for his current context. Such issues can often be resolved by means of *transformations*. In this paper we discuss an algorithm for selecting candidate transformations for a given situation and present our first experiences with this algorithm.

1 Introduction

One of the challenges on the Web is to deal with *heterogeneity*: there are many forms / formats in which information is published on the Web, ranging from static webpages to movies and E-services. In (Gils, Proper & Bommel 2003, Gils, Proper, Bommel & Weide 2004*b*) we have presented a formal model for information supplied on the Web. The essence of this model is fairly straightforward and follows the lines of the RDF-initiative (Lassila & Swick 1999). In our model, resources on the Web:

- are typed
- are interrelated
- may have attributes
- are *about* something (See e.g. (Huibers, Lalmas & Rijsbergen 1996) for a treatment of aboutness)

When searching on the Web, it is not sufficient to merely look at *topical relevance*. In estimating how *apt* a resource is in a given situation, other factors play a crucial role as well. For example, the size of a resource, its file-format and price may also determine if a searcher is ultimately interested in a (topically relevant) resource. If some of these *attributes* are wrong, transformations may alleviate these problems. Transformations in the form of conversions between different file formats are well known. However, transformations can also affect other attributes. Examples include:

- Transform a file from HTML to HTML and remove all its hyperlinks
- Transform an image and lower its resolution
- Transform a ZIP-archive and remove its password

The need for such a broad definition of aptness is recognized in e.g. (Parker 2004):

> ... their definition of availability omitted the need for information to be in useful form.

In a retrieval setting, transformations *from* an input instance of a given type, with certain properties *to* an output instance of a certain type with certain properties may thus be used. If no *singleton* transformation is available, a *composed* transformation may be constructed by concatenating several transformations.

It may be the case that more than one (either a singleton or a composed) transformation is available for a given task. Selecting the "optimal" transformation may be difficult and developing an algorithm to aid us in doing so is the main goal of this paper.

The remainder of this paper is organized as follows. In Section 2 we briefly explain what properties are and how they can be represented. In Section 3 the basic properties of transformations are discussed as well as the effect that transformations may have on properties. Section 4 concerns composed transformations, including a discussion on *transformation patterns*. Finally, in Section 5 we discuss an algorithm for selecting transformations for a specific situation. Conclusions and future work are discussed in Section 6.

2 Properties of resources

Often, static and dynamic behavior of instances determine types. This is the case in e.g. object orientation and abstract data types (Goguen, Thatcher, Wagner & Wright 1977). In case of resources on the web, however, this is not entirely the case since resource types are determined independent of properties that an instance may or may not have.

Each resource on the Web has at least one type and may have more types because of subtyping. For example, any XML file is also an SGML file. These types have nothing to do with the properties that instances may have. An important observation is that instances *must* have types and *may* have certain properties. Remains the question: what are properties?

A property can be any statement about the type(s), relation(s) or attribute(s) that an instance may have. For example, attributes of resource r are:

- r has a specific type t

- r is related to another resource and this relation is of type *hyperlink*. Put differently, this property can be called *has hyperlinks*
- r is related to another resource s
- r has a version attribute
- r has a version attribute with a specific value

As has been stated before, each resource on the Web has at least one type. It can have more types because of subtyping. Let $\mathcal{RS}$ be the set of all resources on the Web and $\mathcal{TP}$ be the set of all resource types. Then $\mathsf{HasType} \subseteq \mathcal{RS} \times \mathcal{TP}$.

Properties can be thought of as predicates that may or may not hold for a certain resource. For example, r *has hyperlinks* is a (unary) property over resources, while r *is based on* s would be an example of a binary property. In general, properties can have any arity higher than one. To model this formally, we will presume a property to be represented in general as $\varphi(r, W)$, where r is a resource and W is a sequence of (zero or more) resources. If $\varphi(r, W)$, then resource r is said to have property φ with resources W.

To be able to evaluate the truth-assignment of φ for a given instance r we use the function Γ:

$$\Gamma(r, \varphi) \triangleq \{W \mid \varphi(r, W)\}$$

In other words, $\Gamma(r, \varphi)$ returns the set of sequences of resources W for which φ is true, given a resource r.

If r is a resource, r $\mathsf{HasType}$ HTML and r actually has hyperlinks (it is indeed possible to have a HTML without any hyperlinks) then $\varphi(r, W)$ indeed holds for any W.

Note that a formal language Φ for formulating what properties φ are is needed. In other work we have specified such a language on top of the model already mentioned. The reader is referred to (Gils, Proper, Bommel & Weide 2004*a*) for more details. As an example, the *has hyperlinks* property would be expressed as

$$\varphi(e, W) \triangleq \exists_{r \in \mathcal{RL}} [\mathsf{Src}(r) = e \wedge r \ \mathsf{HasType}\ \textit{hyperlink}]$$

Note that W does not occur anywhere on the right-hand side of this definition. This is indeed what one would expect from a unary predicate over resources.

In this example, $\mathcal{RL}$ is the set of all relations. Relations are presumed to be binary with a source and a destination. The Src function finds the source of a specific relation.

Note that *instances* may have properties. Also, note that if one instance of a certain type has a property, this does not imply that all instances of this type have this property. In case of the *has hyperlink* example, not every HTML file has hyperlinks. In other words (when considering properties at the typing level): properties are optional.

3 Transformations

With transformations, one resource can be transformed into another. The interesting thing, though, is that only some resources can be modified by a transformation. For example, it seems rather pointless to feed an audio file to a transformation that removes hyperlinks. In this section we explain the behavior of transformations, especially with respect to properties.

Usually, data transformation as considered in our paper, is distinguished from program transformation. The latter kind of transformation has a rich history of theory and practice. An overview has been presented in (Partsch 1990). Recent research results indicate that this area is still evolving into new directions, such as tool-supported adaptation of software systems (see e.g. (Lämmel 2004)).

Indeed our transformation theory has its focus in the retrieval of data resources. Our theory is particularly tailored to properties of data resources and effects of transformations, in a heterogeneous context such as the Web. An overview of concrete transformation rules operating on generic structures (e.g. graphs) are found in for example (Andries, Engels, Habel, Hoffmann, Kreowski, Kuske, Plump, Schürr & Taentzer 1999).

This section is organized as follows. In Section 3.1 we briefly describe some properties of transformations. In Section 3.2 we discuss the effects of transformations on properties. Finally, in Section 3.3 we will discuss how the effects of transformations may be learned.

3.1 Characteristics of transformations

An important characteristic of transformation is that any specific transformation has an *input type* and an *output type*. In other words, it transforms instances (resources) from its input type to its output type. Let $\mathcal{TR}$ be the set of all transformations and $\mathsf{Input}, \mathsf{Output} : \mathcal{TR} \rightarrow \mathcal{TP}$ be functions that find the input type and output type of a transformation. As an abbreviation we use

$$t_1 \xrightarrow{T} t_2 \triangleq \mathsf{Input}(T) = t_1 \wedge \mathsf{Output}(T) = t_2$$

to denote that transformation T has t_1 as its input type and t_2 as its output type.

This behavior of transformations at the *typing level* must have its reflection at the *instance level*. The semantics of any transformation T is that it transforms resources into resources. Let $\overrightarrow{T}$ denote the semantics of a transformation such that $\overrightarrow{T}(r) = s$ denotes that transforming r with T results in s. By definition we then have the following. Let r be a resource and r $\mathsf{HasType}$ t_1. Furthermore, let T be a transformation such that $t_1 \xrightarrow{T} t_2$. Then:

$$\overrightarrow{T}(r) = s \implies s \ \mathsf{HasType}\ t_2$$

3.2 Effects of transformations on properties

When discussing the effects of transformations on properties, a distinction must be made between the instance level and the typing level, similar to what we discussed in Section 3.1. We first discuss the instance level and then briefly elaborate on the typing level.

At the instance level, we discern four classes of effects:

1. A transformation is *neutral* with regard to some property. For example, a transformation that transforms HTML files to PDF may be neutral with regard to the price attribute.
2. A transformation may *alter* a certain property. For example, a transformation that lowers the resolution of an image may lower its price too.
3. A transformation may *remove* a certain property. For example, a transformation that transforms HTML files to ASCII may remove all hyperlinks.
4. A transformation may *introduce* a certain property. For example, a transformation may add a password to a ZIP file.

Let $\mathcal{EC}_i = \{neutral, alter, remove, introduce\}$ be the set of effect classes of transformation at the instance level. Using the Γ relation, it is straightforward to find out the effect class of a transformation $T \in \mathcal{TR}$ with regard to a specific φ. Let $\mathsf{Effect} : (\mathcal{TR} \times \mathcal{RS} \times \Phi) \rightarrow \mathcal{EC}_i$ be the function that finds the effect class of a transformation $T \in \mathcal{TR}$ on a resource $r \in \mathcal{RS}$ with regard to a property φ. This can be achieved by comparing the sets of objects that make Γ true for both the input and the output instance of the transformation:

- If these sets are equal, then for this (input) instance, the transformation is *neutral* with regard to this specific φ. For example, if $\varphi = \mathsf{HasType}$, and both the input instance and output instance have the same types then the transformation is neutral with regard to data resource types.
- If the input set is a subset of the output set, then the transformation, for this (input) instance, apparently is *introducing* with regard to this φ. In case of $\varphi = \mathsf{HasType}$ this means that the input type of the transformation is a subtype of its output type.
- Similarly, if the output set is a subset of the input set, then the transformation, for this (input) instance is *removing* with regard to this φ. In case of $\varphi = \mathsf{HasType}$, this means that the output type of the transformation is a subtype of its input type.
- If neither of the above applies then, for this (input) instance, the transformation is said to be *altering* with regard to this φ. We describe this as follows: let e be the input instance of the transformation and $\overrightarrow{T}(r) = s$ be the output instance of the transformation T. In case of $\varphi = \mathsf{HasType}$ this implies the following. Let $\tau(r) \triangleq \{t \mid r\ \mathsf{HasType}\ t\}$ be a relation that finds the types of a recourse.
  - The sets $\tau(r)$ and $\tau(s)$ overlap such that $\tau(r) \not\subseteq \tau(s) \wedge \tau(s) \not\subseteq \tau(r)$. For example, r and s do have a supertype in common (both are files) but apart from that they are completely different.
  - The sets $\tau(r)$ and $\tau(s)$ are disjoint. This implies that r and s have no (super)type in common.

Summarizing, the effect of transformation T on resource r with regard to property φ is the following:

$$
\begin{array}{lll}
\mathsf{Effect}(T, r, \varphi) \triangleq & & \\
\textbf{if} & \Gamma(r, \varphi) = \Gamma(\overrightarrow{T}(r), \varphi) & \textbf{then}\ \ \textit{neutral} \\
\textbf{if} & \Gamma(r, \varphi) \subset \Gamma(\overrightarrow{T}(r), \varphi) & \textbf{then}\ \ \textit{introduce} \\
\textbf{if} & \Gamma(r, \varphi) \supset \Gamma(\overrightarrow{T}(r), \varphi) & \textbf{then}\ \ \textit{remove} \\
\textbf{else} & \textit{alter} &
\end{array}
$$

Recall that transformations have an input *type* and an output *type* and that (some) properties are optional (at the typing level). Because properties are optional, the effect classes of a transformation regarded at the typing level are: $\mathcal{EC}_t = \{neutral, hybride, remove, introduce\}$. Following the line of reasoning for the instance level:

- If a transformation is *neutral* with regard to a φ for all instances of a given data resource type then, at the typing level, the transformation is said to be *neutral* with regard to this specific φ.
- It seems apparent that, at the type level, a transformation is *introducing* for a given φ if the transformation is *introducing* for every instance of this type. This is, however, not the case. If a transformation is *introducing* with regard to a φ for at least *one* instance and *neutral* for *all* others, then at the typing level the transformation is said to be *introducing* with regard to this specific φ.
- For similar reasons, if a transformation is *removing* with regard to a φ for at least one instance and *neutral* for all others, then at the typing level the transformation is said to be *removing* for this specific φ.
- Again, it may seem that at the typing level a transformation is *altering* with regard to a property if it is *altering* for all instances of this type. However, this is not the case. Other situations may occur also, for example: a transformation may be *introducing* for one instance, and *altering* for another. This occurs when a transformation sets the version attribute to the value 2.6, regardless of the fact that data resource already had a version attribute. If it did, the transformation is likely to be *altering* for this property. If it didn't, the transformation would be *introducing*. In this case, we're *indecisive* about the effect that a transformation has on a certain property.

Summarizing, the effect of a transformation T with regard to a property φ, considered at the *type level* is the following:

$$
\begin{array}{lll}
\mathsf{Effect}(T, t, \varphi) \triangleq & & \\
\textbf{if} & \forall_{r \in \pi(t)} \left[\Gamma(r, \varphi) = \Gamma(\overrightarrow{T}(r), \varphi)\right] & \textbf{then}\ \ \textit{neutral} \\
\textbf{if} & \forall_{r \in \pi(t)} \left[\Gamma(r, \varphi) \subseteq \Gamma(\overrightarrow{T}(r), \varphi)\right] & \textbf{then}\ \ \textit{introduce} \\
\textbf{if} & \forall_{r \in \pi(t)} \left[\Gamma(r, \varphi) \supseteq \Gamma(\overrightarrow{T}(r), \varphi)\right] & \textbf{then}\ \ \textit{remove} \\
\textbf{else} & \textit{hybride} &
\end{array}
$$

3.3 Learning the effects of transformations

In real applications using a transformation framework as described, many types, instances, and transformations will be used. Regarding properties, a

choice must be made: either a fixed (and predetermined) set of properties exists in such an application, or they may be specified at all times.

In both cases, though, it may be the case that the effect of a transformation on a certain property is unknown at a certain point in time. These effects can be *learned* in the following manner:

- Initially, it is assumed that a transformation is *neutral* with regard to every property, similar to the notion of being innocent until proven otherwise.
- After a transformation is performed on an instance, the properties of the input instance and the output instance are compared to study the effects:
  - We may discover a new property of a type. For example: before the transformation was executed we didn't have a single instance of the PDF type with a price but after the transformation we do. This implies that the next time we compose a transformation involving the PDF-type we can use this additional knowledge.
  - We may discover that a transformation is not neutral with regard to some property; i.e. it may alter, remove or add certain properties. For example, we may learn that a transformation from HTML to POSTSCRIPT removes all hyperlinks.

We are aware of the fact that transformation of web resources is necessary for a variety of purposes, including authoring, presentation, and querying. We do not consider all possible purposes in the current paper. As an example, deterministic approaches for document querying are considered in (Che 2003). Those transformations aim at optimization of query efficiency.

4 Composing transformations

For transformations, the input type and output type are known. Using this information it is possible to *compose* transformations by concatenating them. In this section we will study how this can be done by showing several common *combination patterns*. We do not provide a complete / exhaustive overview.

It is only possible to concatenate two transformations if the output type of one of them equals the input type of the other. Thus,

$$\begin{array}{ll} \textbf{if} & T_1, T_2 \in \mathcal{TR} \text{ such that } t_1 \xrightarrow{T_1} t_2 \text{ and } t_2 \xrightarrow{T_2} t_3 \\ \textbf{then} & \exists_{T_3} \left[t_1 \xrightarrow{T_3} t_3 \wedge T_3 = T_2 \circ T_2 \right] \end{array}$$

By combining transformations in this manner, a *directed graph* of transformation is created in which the nodes are the resource types and the edges are possible transformations between them. Since we are discussing transformations, this situation closely resembles that of morphisms in *category theory*[1]

For our purposes it is important to select the right transformation from this "transformation graph". An algorithm for doing so is discussed in Section 5. In the remainder of this section we will discuss several *patterns* of how transformations can be combined.

The first pattern to be discussed is that of a transformation from a type to the same type. An example would be a transformation from HTML to HTML that removes hyperlinks or some header information. Figure 1 graphically depicts this.

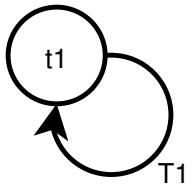

Figure 1: Transformation to the same type

An important aspect in this respect is the question of *loops*: does it make sense to traverse the same node or path in the transformation graph more than once? Traversing the same path more than once means that the same (series of) transformation(s) will be executed over and over again. This does not make sense. Traversing the same node (i.e. the same resource type) more than once *does* make sense, though. The above example with a transformation from HTML to HTML that removes hyperlinks is a good example. In other words, paths through the transformation graph must be *simple* but need not be *elementary* (See e.g. (Grassman & Tremblay 1996)).

Figure 2 depicts the simple concatenation pattern. In this case there is only one (composed) transformation from type t_1 to type t_3.

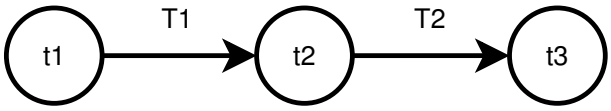

Figure 2: Simple concatenation

The situation becomes slightly more complex if there are several ways to get from type t_1 to t_2 and from t_2 to t_3. This is depicted in Figure 3. In this case there are f possible transformations for the first step and $(n - g + 1)$ possible transformations for the second step. This means that there are $f \times (n-g+1)$ possible ways to transform instances of type t_1 to type t_3.

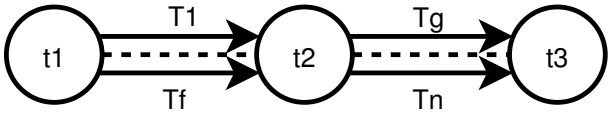

Figure 3: Concatenation

This pattern may be combined with the first pattern: it is possible to transform from type t_1 to t_2, then transform from t_2 to t_2 to achieve a certain ffect on some property, and finally transform from t_2 to t_3. Figure 4 depicts this. There are $2 \times f \times (n-g+1)$ possible ways to transform instances of type t_1 to type t_3,

[1] http://en.wikipedia.org/wiki/Category_theory

assuming that transforming from t_2 to itself is only done once.

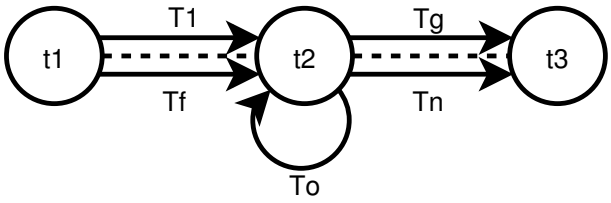

Figure 4: Concatenation with a repeating type

Last but not least, it is also possible that not only a composed transformation from t_1 to t_3 exists, but also a direct transformation. Combined with the first pattern this yields the situation depicted in Figure 5

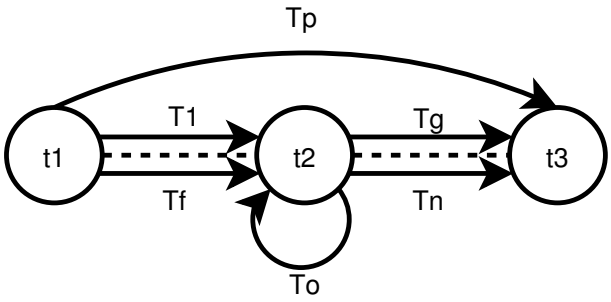

Figure 5: Direct and composed transformations

These patterns form the (theoretical) basis for finding all paths through the transformation graph. This is the topic of the next section.

5 Selection

The main topic for this section is to devise an algorithm that takes the possible transformation paths through the transformation graph under consideration. Because of the fact that there may be many *possible* paths, our algorithm must, somehow, reduce the number of *acceptable* paths. We will use a penalty-mechanism for this. The configuration of this penalty mechanism can be used to formulate the desired properties of the transformation paths (for example: the algorithm can be tweaked to return exactly *one* optimal path). In this section we will present the algorithm. Fine-tuning the (parameterized) algorithm is part of future research.

In this section we will use the situation as depicted in Figure 6. This figure shows 10 types and 21 possible singleton transformations. We will search for transformations from RTF to PS. For clarity, the names of the transformations have been omitted.

5.1 Naive path finder

In the simplest case we search for all possible paths from the input type to the output type, i.e. perform a depth first exhaustive search. The algorithm is rather straightforward:

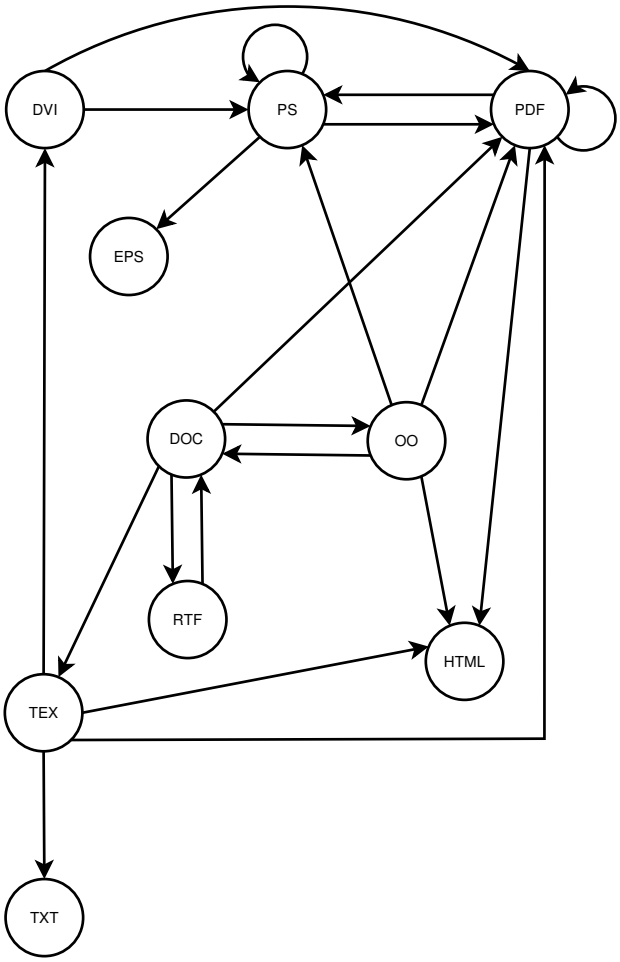

Figure 6: Example transformation graph

1. take the start type and take all transformations that have this type as its input type.
2. loop over these transformations and check if this transformation has been performed already[2].
3. if the transformation has *not* been performed yet, check if the target type is reached with the current transformation. If it has been reached then we have found a transformation path. If it has not been reached, make the current output type the new input type and start with step 1 again to recursively find the target.

We implemented this algorithm in the *Python*[3] programming language in order to be able to experiment with it. The pseudo-code in Figure 7 exemplifies the above algorithm.

Performing this algorithm on the above example leads to the following transformation paths:

1 rtf → doc → oo → ps
2 rtf → doc → oo → pdf → ps
3 rtf → doc → oo → pdf → pdf → ps
4 rtf → doc → doc → tex → dvi → ps
5 rtf → doc → tex → dvi → pdf → ps
6 rtf → doc → tex → dvi → pdf → pdf → ps
7 rtf → doc → tex → pdf → ps
8 rtf → doc → tex → pdf → pdf → ps
9 rtf → doc → pdf → ps
10 rtf → doc → pdf → pdf → ps

[2] We assume that it is rather pointless to perform the same transformation more than once. This also prevents endless looping.

[3] `http://www.python.org`

```
function getPath(from, to, currentPath)
begin
  // findTransformationsFrom finds all
  // transformations starting with input
  // type from
  candidates := findTransformationsFrom(from);
  results := new List();

  foreach transformation in candidates
  begin
    if transformation in currentPath then
      break;
    if transformation.resultType = to then
      results.append(currentPath + transformation)
    else
      results.append(getPath(transformation.resultType,
                  to, currentPath + transformation));
  end;

  return results;
end;
```

Figure 7: Pseudo code for path finder

With all possible transformation paths known, it is straight forward to figure out what happens to properties during transformations. If the effect that each transformation has on a given property is known then this knowledge should be used: follow the path and, in each node, determine if the property still holds or not.

However, if the effects that some transformations have on a given property are unknown, the only way to be absolutely sure which path should be selected would be to perform every transformation path (and thus learning the effects on this property for future use too).

We consider the composition of transformations. A sequence of transformations may compose a new 'overall' transformation. This raises the question of transformation performance, since several different transformation sequences may transform a given input type into a given output type. In our project, the transformation performance is considered by taking a shortest path view of web resource transformation. We illustrate this in section 4 and 5, setting the context for a full treatment of web transformation performance as found in other areas of transformation (see e.g. database transformation in (Rahayu, Chang, Dillon & Taniar 2001)). Note that in our shortest path view we do not necessarily require a single shortest path to be found. Rather, we aim at a reduction of the possible paths in order to yield a selected set of candidate transformation compositions. We have successfully exploited reduction in transformations in earlier projects, such as database transformation (see e.g. (Bommel & Weide 1992)).

5.2 Penalty-based approach

The approach discussed in the previous section has some serious disadvantages. First of all, as the number of types and singleton transformations grow, the number of possible paths through the transformation graph is likely to explode. Determining all possible paths from a given input type to an output type at runtime will take an increasingly amount of time. The situation is even worse if properties may be composed dynamically at runtime (see Section 3.3): after finding the possible paths, they must all be executed to determine what happens with the newly composed properties.

A similar problem exists in the world of (relational) databases: performing a join *before* doing a selection is computationally heavier than performing the join *after* doing the selection. Therefore, a push-down selection scheme should be adopted (See e.g. (Ullman 1989)). Translated to our problem of walking through the transformation graph: determining which transformation paths are not feasible should be done as soon as possible as opposed to removing the unwanted paths after figuring out all possible paths. Simply put: figure out which paths are likely to be infeasible *while finding all possible paths through the graph*. As soon as it is likely that following a path will lead to no good, that path should be abandoned and a new one tried; i.e. break the current loop and go on with the recursive search. This will not only lower the time that it takes to perform the search but, hopefully, will lower the number of paths that are found.

The question that remains is: what criteria should be used to estimate the likelihood that a path will not be feasible. We propose to use a penalty-based approach:

- Short paths are likely to be better (for example: faster in terms of execution time) than long paths. Therefore, each step is penalized. This is particularly apparent when, for example, execution time plays a role: every step takes extra time (if, in a certain situation, the execution time does not play a role than this penalty can be set to 0). However, this is not the only reason. Transformations may also reduce the "quality" of the input resource which can also be a reason to increase the penalty for this transformation.
- If a property must be retained during transformation, removing it along the way will result in a penalty. If the property is added along the way, this will result in a negative penalty. Similarly, if a property must be removed during transformation, adding it will lead to a penalty and removing it will lead to a negative penalty.
- As soon as the current penalty for a path surpasses a certain boundary then it is assumed that this path is likely to be not feasible: therefore, it will no longer be followed and a new path must be tried.

We also implemented this algorithm in the *Python* programming language. The pseudo code in Figure 8 shows the outline of this implementation and exemplifies the algorithm.

We extended the above mentioned example with penalties such that every transformation has a penalty of 0.1 because of execution time. However, the transformations dvi $\rightarrow$ pdf, pdf $\rightarrow$ pdf, oo $\rightarrow$ ps and oo $\rightarrow$ pdf have a penalty of 0.2 and tex $\rightarrow$ pdf has a penalty of 0.3 because these are presumed to be heavier in terms of computation. Also, we know that the transformations oo $\rightarrow$ html is *removing* with regard to a certain property φ and doc $\rightarrow$ tex is *introducing* for this same property. Since we wish to retain this property, the former transformation re-

```
function getPath(from, to, maxPenalty, currentPath)
begin
  // findTransformationsFrom finds all
  // transformations starting with input
  // type from
  candidates := findTransformationsFrom(from);
  results := new List();

  foreach transformation in candidates
  begin
    if ((transformation in currentPath)
        or (transformation.penalty > maxPenalty)) then
      break;
    if transformation.resultType = to then
      results.append(currentPath + transformation)
    else
      // penalty is a function that calculates the
      // penalty of this transforamtion
      results.append(getPath(transformation.resultType,
              to, maxPenalty - penalty(transformation),
              currentPath + transformation));
  end;

  return results;
end;
```

Figure 8: Pseudo code for penalty based path finder

ceives a negative penalty of 0.2 and the latter receives a positive penalty (bonus) of 0.2.

Running this algorithm and, thus, taking into account the above mentioned penalties results in the following paths:

path number	path
1	rtf → doc → oo → ps
2	rtf → doc → tex → dvi → ps
3	rtf → doc → tex → dvi → pdf → ps
4	rtf → doc → tex → pdf → ps
5	rtf → doc → pdf → ps

Selecting the "optimal" path from these transformations still needs to be done. It is tempting to simply select the path with the lowest penalty, but this may not always be the best path because the total effect that the transformation(path) has on the properties must be taken into account. For the above example, the penalties and effects are the following:

path number	penalty	effect
1	0.4	*neutral*
2	0.2	*introducing*
3	0.4	*introducing*
4	0.4	*introducing*
5	0.3	*neutral*

If the effect that a composed transformation has on properties is taken into account, as well as the penalty this transformation receives then the second path is to be selected since:

1. It is introducing for a property that we wish to retain, so we're 100% sure that the property will hold after this transformation path is executed on any given input instance.
2. It has the lowest penalty.

5.3 Reality check

The above example is, obviously, extremely simplistic and very small. To see if the general idea behind our algorithm works, we conducted a larger experiment. The goal of this "reality check" is to find out if the algorithm indeed selects less paths, shorter paths and executes faster.

- There are 100 types.
- We're looking for a transformation from type t_{12} to type t_{89}.
- We assume the existence of three properties: p_1, p_2 and p_3. The output instance must have properties p_1 and p_2, but may not have property p_3.
- There are 161 singleton transformations.
- Every singleton transformation will result in a penalty of 0.1.
- For 59 transformation-property combinations we know the effect (i.e. there are 59 statements in the form: Transformation t has effect e for property p), spread out over 46 transformations.
- If we don't know the effect of a transformation on a property, we will assume that it is neutral.
- The average penalty (either positive or negative) is 0.207.
- The maximum penalty that a transformation path may have is set to 2.5.

The graph with all transformations is depicted in Figure 9. For clarity, the names of the transformations have been omitted. Note that we did not include "transformations to self" (see Figure 1). For purposes of this experiment this does, at least conceptually, not make a difference. After running both

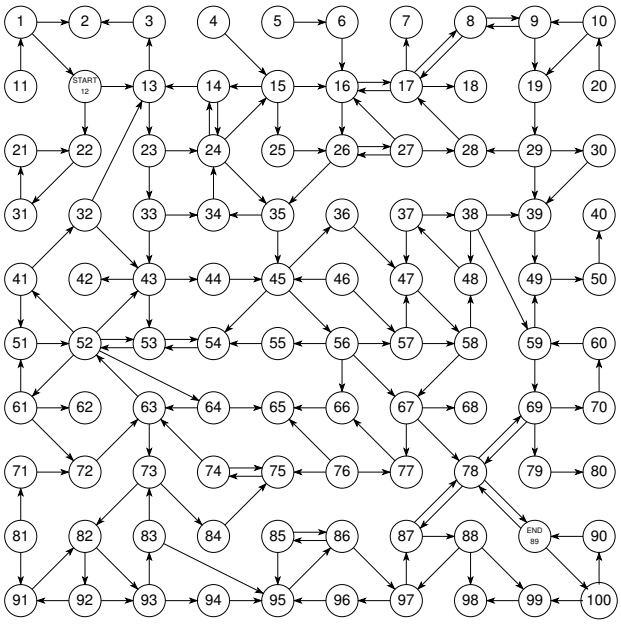

Figure 9: Larger example

the naive path finding algorithm and the more complex penalty based approach we observe the following:

- The path finder algorithm finds a total of 915 *possible* paths through the graph. Using the penalty based approach, this is reduced to 82 *acceptable* paths.
- The average length of a path for the path finder algorithm is approximately 27, whereas the av-

erage length in the penalty based approach is approximately 18.

- For this particular example, the penalty based approach is approximately 6 times faster than the naive algorithm.

The above suggests that, at least for this example, the penalty based algorithm performs better in terms of execution speed as well as in the number / length of the paths that it returns.

5.4 Discussion & Issues

Recall that the goal of this article is to describe an algorithm that finds composed transformations that increase the *aptness* of a resource on the Web. In other words, the algorithm should select transformations that manipulate resources such that the user will consider them more apt.

This immediately leads to several interesting, yet unsolved, problems such as: How can one find out what the user wants? How can one test if the transformation has indeed increased the aptness for this specific user in this specific situation? These questions are traditionally covered in the realm of user modeling and user profiles. Even though it is crucial to be able to answer this question for any real application it remains unanswered still in our theory. Finding a good way to get this (kind of) information (for example by deploying a *query by navigation*-like approach, see (Bruza & Weide 1992)) from the user is part of future research.

In theory, the properties as described in this article are a nice way to describe both the resources on the Web (i.e. resource r has some property p) as well as a query formulation for the resource that a certain user is interested in (i.e. a resource r about x with property p_1 and without property p_2). In practice, though, it may not be so easy to work with these properties. The first property-related issue has to do with the question: does the application support a fixed set of predefined properties, or can any property be formulated at run-time. The latter is, conceptually, nice because it allows more flexibility. However, in that case it will be very hard (in terms of computation) to determine if a transformed resource has this property or not. The only way to find this out is to actually perform the transformation. This brings the second issue with properties to the fore: for every property that is known to the system, a tool (software) must exist that (quickly) tests whether a resource has that property or not. In other words, a trade-off has to be made between (conceptual) flexibility and (operational) availability.

An issue with both the properties and the proposed (penalty based) algorithm has to do with the fact that the relative importance of properties can not be indicated. That is, suppose a user indicates that s/he is looking for a resource r with a certain property p_1 and without a property p_2. In our present approach it is not possible to express the fact that having property p_1 is more important, to this user, than not having property p_2.

It is possible to have *parameterized* transformations. An example of such a transformation would be a transformation that lowers the resolution of an image with n percent. In this case $\overrightarrow{T}(r, 10)$ would denote the fact that transformation T transforms resource r (an image) and lowers its resolution by 10%. If we would facilitate such types of transformation then optimizations might be possible along the lines of:

- maximize the value of property p_1
- minimize the value of property p_2

In this paper we did not include details about this approach.

6 Conclusions & Future research

The goal of this paper was to find an algorithm for selecting (one or more) transformations in order to increase the aptness of resources on the Web. Such a transformation framework can be used in a retrieval setting on the Web where traditionally only / mainly *topical relevance* is used to select resources that may satisfy the users information need. We propose to use a "push-down selection"-like[4] approach in which first the resources that are topically relevant are selected (for example by a search engine like GOOGLE) after which transformations may be used to increase the *aptness* of these selected resources. Such a strategy is needed since a set of transformations in combination with the large set of resources available to us directly via the Web, yields an even larger set of resources. In terms of (Ullman 1989), the set of resources available on the web can be seen as a (large!) *extensional* database. Use of the above discussed transformations yields a practically infinite *intensional* database. Searching through the latter database can only be done practically if branch-and-bound like optimization strategies are used to reduce search space.

In earlier work (e.g. (Gils et al. 2003, Gils et al. 2004*b*)) we have presented a formal model for information supplied on the Web and explained how *properties* can be used to describe both resources on the Web, and (the non-informational aspects of) one's information need. For example, a property of an image-resource on the Web could be its resolution. Similarly, the resolution / quality of an image can be part of the information need of a searcher. These properties are an important factor when trying to find "acceptable transformation(path)s" for increasing the aptness of a resource.

Simply put: transformations transform one resource into another. More specifically, a transformation will transform instances from its input type to instances of its output type. Furthermore, transformations may have an effect on the properties of the input instance. For example, consider the transformation from HTML to Postscript. Input instances may have hyperlink-properties. Output instances of this transformation will not have this property. In other words, this transformation is *removing* with regard to the property *has hyperlinks*.

Since both resources on the Web and desired resources (formulated in terms of an information need)

[4] As also used in e.g. database query optimization strategies (Ullman 1989).

are formulated in terms of these properties, our algorithm must take these effects into account when determining which transformations / transformation paths are *acceptable*. For this we use a penalty-based approach which is an extension of a simple depth first exhaustive search which finds all *possible* transformations. That is, *while* recursively finding all paths we try to prune those paths that are likely to be not feasible. For this we make the following assumptions:

- Longer paths are likely to be less good than shorter paths. Therefore, each step through the graph (i.e. performing a single 1-step transformation) will result in a penalty.
- Manipulating properties may either result in a penalty or a bonus. If a transformation is *removing* with respect to a property that must hold for the output instance then this will result in a penalty. If it is *introducing* for a property that must hold then this will result in a bonus.
- If the total penalty of a path reaches a certain level then we consider the path not feasible.

In two (small) experiments we have shown that such an algorithm can indeed work and reduces both the number of paths found (when comparing the penalty based approach with the normal path finding approach) as well as the time it takes for the algorithm to finish. The algorithm's results (a set of transformation paths) must either be interpreted manually or the algorithm must be run again under a modified configuration.

Such a decision mechanism, which is closely related to a parameterized tuning mechanism that may be used to steer the working of the algorithm, is currently under investigation. There are some other issues with our algorithm that need further attention. First of all, finding out what the user wants (in terms of properties) is a complex task traditionally dealt with in the field of user modeling. We are currently investigating a *Query by Navigation*-approach to deal with this. Using this approach we also hope to tackle the issue of the relative importance of properties. For example, it may be more important (for a specific user) to retain a certain property than it is to lose another.

To summarize: we are trying to extend our approach as well as develop tools to see how well our approach works in real world situations.

References

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A. & Taentzer, G. (1999), 'Graph transformation for specification and programming', *Science of Computer Programming* **4**(1), 1–54.

Bommel, P. v. & Weide, T. v. d. (1992), 'Reducing the search space for conceptual schema transformation', *Data & Knowledge Engineering* **8**(4), 269–292.

Bruza, P. & Weide, T. v. d. (1992), 'Stratified Hypermedia Structures for Information Disclosure', *The Computer Journal* **35**(3), 208–220.

Che, D. (2003), Implementation issues of deterministic transformation system for structured document query optimization, *in* 'Proceedings of 2003 International Database Engineering & Application Symposium', Hong Kong, pp. 268–277.

Gils, B. v., Proper, H. & Bommel, P. v. (2003), A conceptual model for information suppy, Technical Report NIII-R0313, Nijmegen Institute for Information and Computing Sciences, University of Nijmegen, Nijmegen, The Netherlands, EU. Accepted for publication in Data & Knowledge Engineering.

Gils, B. v., Proper, H., Bommel, P. v. & Weide, P. v. (2004*a*), Typing and transformational effects in complex information supply, Technical report, Radbout university Nijmegen Institute for Computing and Information Sciences. (To be published).

Gils, B. v., Proper, H., Bommel, P. v. & Weide, T. v. d. (2004*b*), Transformations in information supply, *in* J. Grundspenkis & M. Kirikova, eds, 'Proceedings of the Workshop on Web Information Systems Modelling (WISM'04), held in conjunctiun with the 16th Conference on Advanced Information Systems 2004 (CAiSE 2004)', Vol. 3, Faculty of Computer Science and Information Technology, Riga, Latvia, EU, pp. 60–78.

Goguen, J. A., Thatcher, J. W., Wagner, E. G. & Wright, J. B. (1977), 'Initial algebra semantics and continuous algebras', *Journal of the ACM (JACM)* **24**(1), 68–95. ISSN: 0004-5411.

Grassman, W. K. & Tremblay, J.-P. (1996), *Logic and Distrete Mathematics*, Prentice Hall, Upper Saddle River, New Jersey.

Huibers, T. W. C., Lalmas, M. & Rijsbergen, C. J. v. (1996), 'Information retrieval and situation theory', *ACM SIGIR Forum* **30**(1), 11–25.

Lämmel, R. (2004), 'Transformations everywhere, editorial', *Science of computing, Special Issue* .

Lassila, O. & Swick, R. R. (1999), Resource description framework (rdf) model and syntax specification, Recommendation, W3C.
URL: *http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/*

Parker, D. B. (2004), 'The folk art of information security needs an upgrade', *Communications of the ACM* **47**(8), 11–12.

Partsch, H. (1990), *Specification and Transformations of Programs*, Springer-Verlag, Berlin.

Rahayu, J. W., Chang, E., Dillon, T. S. & Taniar, D. (2001), 'Performance evalutation of the object-relational transformation methodology', *Data & Knowledge Engineering* **38**(3), 265–300.

Ullman, J. (1989), *Principles of Database and Knowledge-base Systems*, Vol. I, Computer Science Press, Rockville, Maryland.