A1-D5

Ontology Driven Visualisation of Maps with SVG

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/A1-D5/D/PU/a1
Responsible editors:	H.J. Ohlbach
Reviewers:	Michael Rosner
Contributing participants:	Munich
Contributing workpackages:	A1
Contractual date of deliverable:	31 August 2005
Actual submission date:	29 September 2005

Abstract

In this work we demonstrate a particular use of ontologies for visualising maps in a browser window. The GIS data are represented in the OWL data format that corresponds to the ontology of transporation networks (OTN). These data are transformed into Scalable Vector Graphics (SVG). The transformation is specified symbolically as instances of a *transformation ontology*. This approach is extremely flexible and easily extendible to include all kinds of information in the generated maps.

Keyword List

semantic web, geospatial notions, ontologies, scalable vector graphics, visualisation

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Ontology Driven Visualisation of Maps with SVG

Hans Jürgen Ohlbach[1], **Bernhard Lorenz**[2]

[1] Department of Computer Science, University of Munich
Email: ohlbach@pms.ifi.lmu.de

[2] Department of Computer Science, University of Munich
Email: lorenz@pms.ifi.lmu.de

29 September 2005

Abstract
In this work we demonstrate a particular use of ontologies for visualising maps in a browser window. The GIS data are represented in the OWL data format that corresponds to the ontology of transporation networks (OTN). These data are transformed into Scalable Vector Graphics (SVG). The transformation is specified symbolically as instances of a *transformation ontology*. This approach is extremely flexible and easily extendible to include all kinds of information in the generated maps.

Keyword List
semantic web, geospatial notions, ontologies, scalable vector graphics, visualisation

Contents

1 Introduction

In Deliverable A1-D4 the Ontology of Transportation Networks OTN has been described. OTN was generated by making the concepts and structures which are implicitly contained in the Geographic Data Format (GDF) explicit as an OWL ontology. OTN contains all kinds of notions for transportation networks, roads, trains, ferries and much much more. In this deliverable A1-D5 we show one of the applications of an ontology like OTN, the ontology driven generation of displayable maps in Scalable Vector Graphics (SVG) [1, 2].

There are may different ways for generating maps as pictures on a computer. The most straightforward way is to read the geographic data from a file or a database and use a special purpose algorithm that transforms the data into some bitmap graphics format. These algorithms are complicated and not easy to change and extend. Only experts who are familiar with the details of the algorithms can do this. The algorithms depend very much on the particular data format of the geographic data, and they usually yield static pictures.

A second method is to generate instead of bitmap graphics, code in a graphics description language. An example for such a language is SVG (Scalable Vector Graphics). SVG is an XML-based language for describing geometric objects. There are special plugins for web browsers which can render SVG files in a browser window [2].

Compared to bitmap graphics, SVG has a number of advantages:

- since it is vector graphics, it is zoomable without loosing resolution on the screen;
- it has language constructs for describing dynamic changes of the graphics;
- since SVG objects have a DOM representation [3] in the browser, script languages like JavaScript can interact with it. SVG documents can therefore serve as GUIs to interact with the user. We used this to allow the user to interactively change the presentation of the maps;
- SVG renderers can adapt the generated picture to the output device. Therefore the SVG generator need not worry about the device characteristics;
- SVG documents are XML documents which can be read by humans. This is very useful during the development and test phase of SVG generators.

The generation of code in a graphics description language like SVG instead of bitmap graphics has therefore the advantages:

- the algorithms for transforming the geographic data are much simpler because the rendering of the graphical data is done by the browser;
- the generated graphics code is device independent. The renderer automatically adapts the graphics to the output device;
- if the graphics description language has constructs for dynamic pictures, they can be used directly without having to care about rendering image sequences;

- most kinds of interaction, zooming in and out to a certain extent, for example, is done by the browser, and need not be taken into account by the transformation algorithms.

The primary data sources for the visualisation are usually GIS databases in some of the standard formats, GDF [11] or GML [4], for example. This is not the only choice. In this paper we propose an alternative. We still use GIS data in some of the standard formats as primary source, but only because these are the only available data. The idea is to take an OWL ontology of transportation networks, in our case OTN, and to represent the data as instances of the concepts of this ontology. OWL provides a data format for instances of the concepts of the ontology, and we use the OWL data format for the GIS data. This is not just a syntactic reformulation. It offers completely new possibilities because the OWL data format is only loosely coupled with the OWL ontology. For example, consider an ontology containing the concept of a *road*. A road may have directions, at most two. If there is a particular road R with directions $= 1$ then OWL would classify R as a *road*. If, in a later step, the ontology is extended with the concept *one_way_road* as a *road* with directions $= 1$ then OWL would automatically reclassify R as a *one_way_road*. Thus, there is a certain degree of independence between the OWL data format and the ontology. The same data can be used for different ontologies.

The method for visualising maps proposed (and implemented) in this deliverable is now basically as follows:
There are three classes of input data

1. the concrete GIS data which has been transformed into the OWL data format (we used the map of Munich);

2. an ontology of transportation networks, in our case OTN;

3. a set of transformation rules which determine how the instances of the concepts in the ontology are to be transformed into SVG code.

The transformation algorithm now applies the transformation rules to all relevant instances of the concepts in the ontology and produces SVG documents as output. With this architecture it is extremely easy to change the visualisation. For example, if we want to distinguish one–way roads from ordinary roads, it is only necessary to introduce the concept of a one–way road in the ontology and to add a corresponding transformation rule.

If the data source is in XML, which is the case for GML or the OWL data format, and the target is also XML, which is the case for SVG, there is a further alternative for generating maps. One can develop an XSLT style sheet that transforms the GIS data into SVG [5]. This approach is, however, very limited and inflexible because the XSLT style sheet depends extremely on the structure of the XML data source. Moreover, it does not support rasterisation and levels of detail, which is extremely important for working with large maps.

In this document we describe the basic ideas and techniques of our approach. More details can be found in [6].

2 SVG Visualisation

We start with a description of the SVG visualisation technique because this motivates some of the design decisions for the transformation method.

2.1 The Final Result

The final result of the visualisation is illustrated in Figure 1. The browser window consists of a frame containing the SVG map and a HTML menu on the right hand side. The SVG map is zoomable in a wide range, more than the built–in SVG zooming facility allows. One can change the section to be displayed by just dragging the mouse over the window. The map may contain dynamic elements, for example buses or trains moving along the rails, clouds moving over the scene etc.

The menu allows the user to choose what he wants to see. It is divided into two main sections *Modules* and *Ontology*. The *Ontology* section corresponds to a *display ontology*, which is a tree of concepts in the transportation network realm. Checking or unchecking the appropriate box causes the corresponding items in the map to become visible or invisible. A *module* in the upper section of the menu consists of a set of elements from the display ontology. Any combination of elements from the display ontology can form a module. Checking or unchecking makes the whole group of items visible or invisible.

There is a further feature which is not part of the section of the browser window shown in Fig. 1. Below the map there is a text input section where one can type in a street name and the street is then highlighted in the map.

2.2 Dynamic Loading

All visualisation systems for maps have the same problem: the server has usually much more data than the user at the client side wants to see. Network bandwidth and computation capacity are not large enough to transfer all the data from the server to the client such that the client can decide what to show to the user and what not. Therefore it is necessary to partition the data at the server side and send only the relevant parts to the client side. If the user changes the section of the map to be shown or he zooms in and out, more data needs to the loaded dynamically during the user interaction.

SVG has no built-in facility for dynamically loading data from the server. The combination of the DOM representation of SVG data in the browser memory and the possibilities of scripting languages like JavaScript to modify the DOM at any time, however, makes it possible to program dynamic downloading of data from the server. The method works as follows: any SVG file may contain elements like this:

```
<g bBox="14848 8831 3144 5782"
   loadUrl="maps/MunichBackground/MunichBackground.svgz" />
```

These elements are ignored by the browser, but they may be manipulated by JavaScript. The `bBox` attribute contains the bounding box of the picture (a compressed SVG document) to be

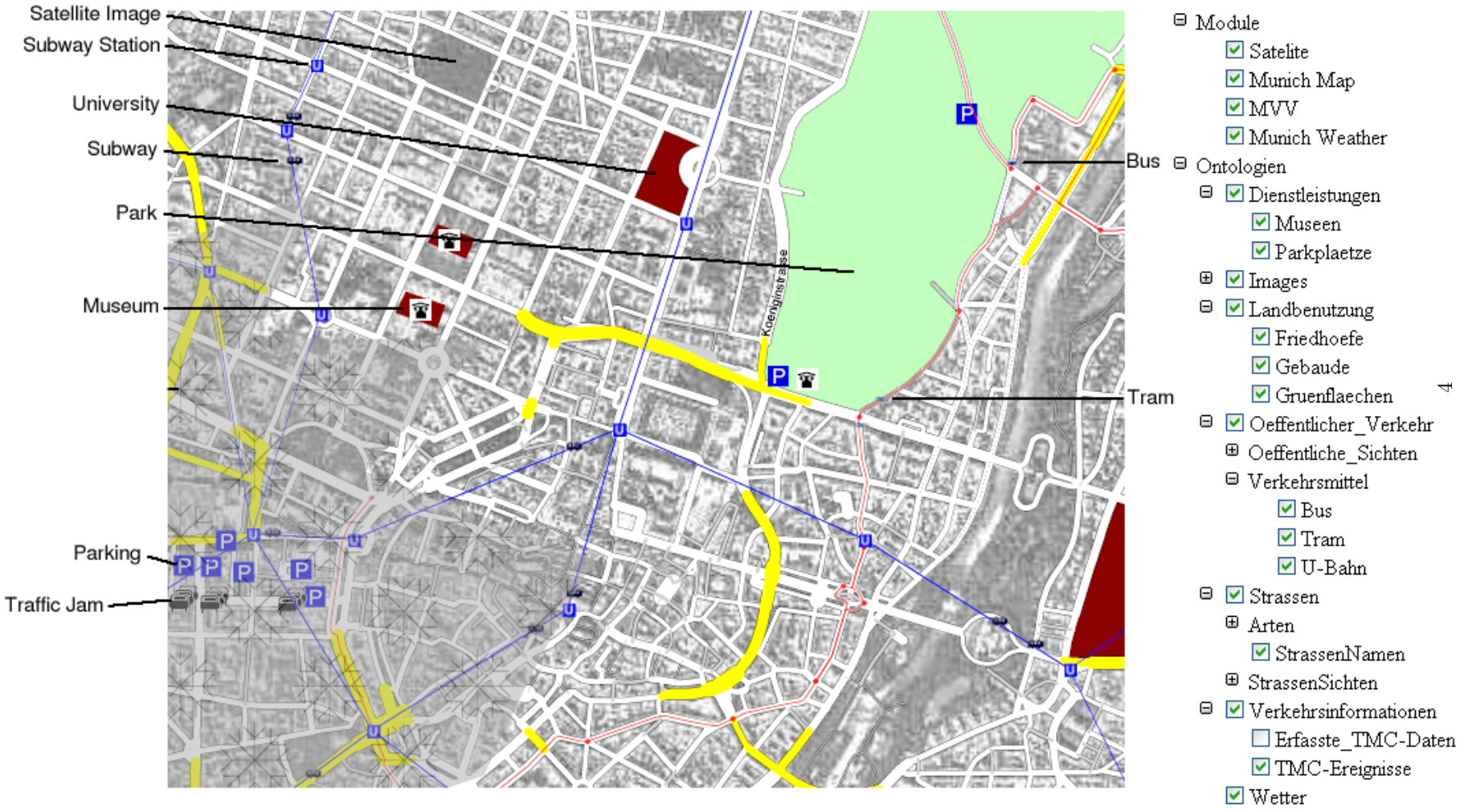

Figure 1: Visualisation in SVG

loaded from `loadURL`. The script uses the bounding box to decide when the other picture has to be loaded. When this is the case, it loads the file from the corresponding URL, parses it as an XML document into a further DOM tree and replaces the node that corresponds to the <g> element with the new DOM tree. The browser then automatically redisplays the modified picture.

In order to use this mechanism for loading only the actually needed parts of a big map, we have to solve two problems. The first problem is to divide a big map into small enough tiles which can be loaded independently. The second problem is to support zooming already at the server side. The problem here is that the same item must be displayed differently at different zoom levels. For example, if the whole map of Germany is to be shown, it makes no sense to display all the details of, say, the city of Munich. Munich should in this case be displayed only as a dot, or maybe as a very simple polygon filled with a uniform colour. If the user zooms into Munich, he wants to see of course more detail. These different views have to be prepared at the server side.

2.3 Rasterisation

There is a very simple solution for splitting a big map into smaller tiles: a fixed grid is imposed on the map and the map is split into the grid elements. The disadvantage is that this way the split parts may have very different size. There may be split parts with almost nothing in it and split parts in very densely populated areas which contain thousands or millions of items.

A much better distribution of equally sized split parts can be obtained with R-trees [14, 15]. An R-tree is a structure for storing 2-dimensional data. Each node in the tree contains data about its *minimum bounding rectangle*, i.e. the smallest rectangle which includes the node itself and all of it descendants. Leaves contain the data and nodes contain index information. Nodes can be further combined within other nodes, rectangles can be overlapping. The root node therefore contains all descending nodes and subsequently all leaves including the bounding rectangle of the whole tree. As a rule it is normally not possible to generate an optimal R-tree, since this would involve a complexity of $O(n) = 2^n$. Therefore, there are generally a number of different (equal) instances of an R-tree and some algorithms for maintaining its structure. This does, however, not have a significant impact on the rasterising process as we need it for our application.

Figure 2 shows a possible R-tree, which is rather self explanatory.

In our special case, using an R-tree means using the advantages of rectangular grid sections while eliminating the need for separate indexing or cumbersome preprocessing of data. The grid is comprised of the minimum bounding rectangles (which can be overlapping), whereas each element belongs to only one section and all sections contain a similar number of elements. Each node contains information about its children and the sizes of their respective rectangles. This allows for recursive search from the root along the different nodes, while for each node it can be quickly decided, whether it touches the area to be displayed (and therefore, whether data from its children has to be loaded). Elements can easily be distributed equally between grid sections and there is no need for a separate index file. Figure 3 shows a tile of a map of Munich which has been generated using an R-tree. This tile contains only data for a particular road type. We use the OTN ontology to separate the items in the tiles into instances of the

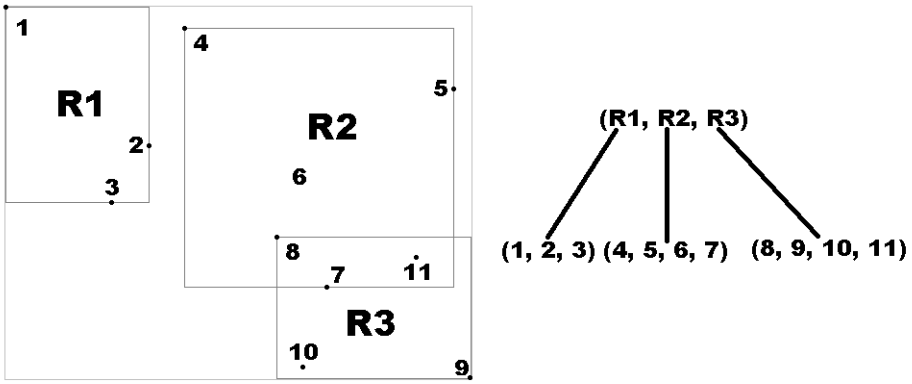

Figure 2: R-Tree Sample

same classes.

2.4 Levels of Detail

Yet another similar problem stems from the zoom mechanisms. Depending on the current zoom level, certain elements - mostly because of their size - cannot be displayed properly because they would be too small to be useful. Elements like this include smaller streets (which come in greater numbers as well), street names and similar things. Moreover it makes sense to simplify certain elements to simpler structures in order to improve readability and usability of the map. Cities might be reduced to circles or dots of different diameter (depending on other attributes, such as number of inhabitants). This form of presentation is much more useful than putting processing power into rendering irregular city boundaries which are too small to be identified as such.

Every element therefore contains the attributes *minDetail* and *maxDetail* which set the levels of detail in between which the element is to be visible. The level of detail is the minimum of both the vertical and horizontal resolution. On a map displaying from coordinates (200, 400) to (500, 1000) the minimum would be $min(300, 600) = 300$.

Our system therefore generates SVG files for the different tiles at the different zoom levels. If the user zooms in or out the corresponding files are automatically loaded.

2.5 Extensions to SVG groups

A JavaScript program supports in addition to the usual SVG attributes further attributes which are grouped within a `<g>` tag. All visible elements are tested for these attributes and the necessary processing is done, for example at time of loading. New attributes contain:

bBox the rectangle to which the elements of the group belong

minDetail minimum detail level of the group

Figure 3: A generated tile of a map

maxDetail maximum detail level of the group

loadUrl the url which provides the contents of the group

updatePeriod the delay in seconds after which a refresh is necessary

ontology a comma separated list of class names (of the visualisation ontology) which the element belongs to

module the name of the module which is displayed in the selection tree

3 From OTN to SVG

In a preparatory step the GIS data has been transformed into the OWL data format of the OTN ontology. All information about roads, bus lines, underground lines, parks, etc. are therefore stored as instances of the OTN ontology. In order to generate the many little SVG files which contain the tiles of the map at the various zoom levels, one could now write a thick program that reads the map and somehow generates the SVG files. This would be extremely complicated and inflexible. Therefore we took another route.

SVG has a relatively small fixed number of constructs for displaying graphical structures. These few constructs are also represented as an OWL ontology, the *transformation ontology*. The main parts are depicted in Fig. 4.

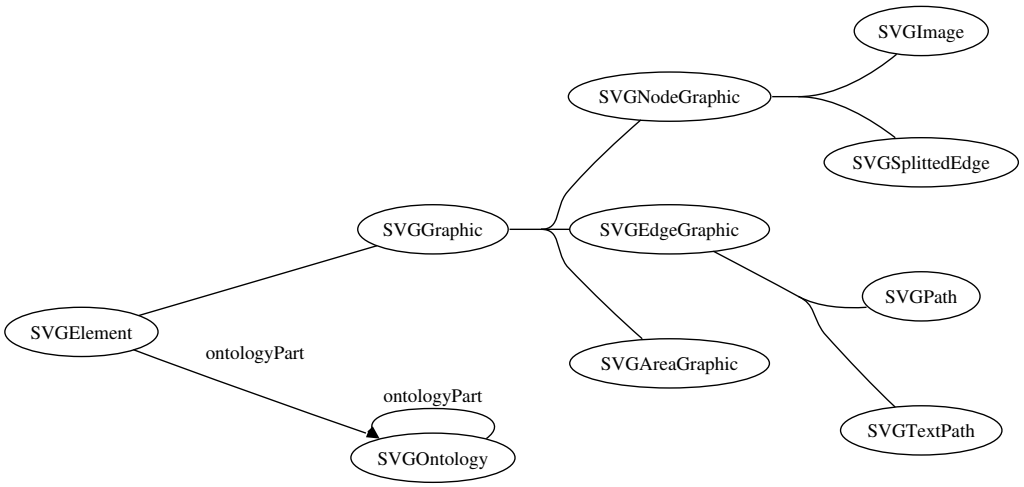

Figure 4: Transformation ontology for transformations from OTN to SVG

The *SVGOntology* class does not correspond to an SVG construct. It defines the structure of elements in the SVG document which are to be displayed under "ontology" in the menu on the right hand side of the browser window (see Fig. 1).

Each component of the map is to be transformed into one of these SVG elements. For example, a road may be transformed into an SVG path element. A railway or a bus line may also be transformed into an SVG path element. The idea is now to generate for each element of a map that is to be transformed into a particular SVG element an instance of the corresponding class of the transformation ontology. This instance must contain the information *how to* transform the map element into SVG.

To illustrate this, consider the following instance of SVGPath:

```
<SVGPath rdf:ID="BusLine">
  <useOnClass>Route_Link</useOnClass>
  <condition>=[public_Transport_Mode]=Bus</condition>
  <minDetail>0</minDetail>
  <maxDetail>40000</maxDetail>
  <paintingOrder>300</paintingOrder>
  <width>3</width>
  <groupAttributes>class="Bus"</groupAttributes>
  <elementType>path</elementType>
  <addId>false</addId>
  <ontologyPart rdf:resource="#oeffentliches_Verkehrsnetz_ontologyPart"/>
  <ontologyPart rdf:resource="#Bus"/>
</SVGPath>
```

It specifies how the OTN data `Route_Link` with `public_Transport_Mode` = `Bus`, which represents a segment of a bus line, is to be transformed into an SVG path element. The important

parts are <useOnClass>Route_Link</useOnClass> and <condition>=[public_Transport_Mode]=Bus</condition>. It means that the transformation is to be applied to all instances of the class Route_Link which satisfy the condition public_Transport_Mode=Bus. minDetail and maxDetail specify the zoom level for which this transformation is to be applied. The remaining elements of SVGPath specify geometric and other details to be inserted into the SVG path element. The actual coordinates for the path element are directly taken from the OTN data.

In the next example we want to put a little moving image of a bus onto the SVG path element of a bus line. SVG has features for generating dynamic graphics. Unfortunately it turned out that in the currently available browsers they slow down the rendering so extremely that they are just not usable. Therefore the system generates moving images on a map by periodically downloading a new version from the server[1]. This is specified in the next example.

```
<SVGImage rdf:ID="Bus">
   <useOnClass>Line</useOnClass>
   <condition>=[public_Transport_Mode] =Bus</condition>
   <minDetail>0</minDetail>
   <maxDetail>40000</maxDetail>
   <url>images/bus.gif</url>
   <updatePeriod>5</updatePeriod>

   <xCoord>=[x]-15</xCoord>
   <yCoord>=[y]-25</yCoord>

   <height>50</height>
   <width>30</width>

   <onClick>=IF [external_Link] THEN window.top.open ("[external_Link]")
   </onClick>
   <tooltip>=Bus|Linie [alternate_Name] |
                Departure Time: {TIME(3)@[startTime]} \- [starts_at].[ID] |
                Arrival Time: {TIME(3)@[endTime]} \- [ends_at].[ID] |
                Waiting Time: {TIME(2)@[waitingTime]} |
                Travel Time: {TIME(2)@[drivingTime]}</tooltip>

   <ontologyPart rdf:resource="#Bus"/>

   <ontologyPart rdf:resource="#aktueller_Betrieb_ontologyPart"/>
   <paintingOrder>10000</paintingOrder>
   <addId>false</addId>
   <viewbox>-30 -30 25878 23419</viewbox>
 </SVGImage>
```

This time we use an SVGImage element to insert the image images/bus.gif into the map. The

[1]Periodically downloading a new version of a file from a server is actually much more flexible than using the dynamic elements of SVG. The server can take a lot more information into account for computing these images than the client has available.

transformation is to be applied to OTN instances of `Line` with attribute `public_Transport_Mode = Bus`. The element `<updatePeriod>5</updatePeriod>` causes that the SVG file with the image is to be reloaded every 5 seconds. If the image is to be moved then this file must be updated at server side in regular intervals. `<xCoord>=[x]-15</xCoord>` shows an example for a special arithmetic language which is part of the transformation technology. `[x]-15` means that the `x` attribute of the corresponding OTN instance is to be subtracted by 15 in order to get the precise x-coordinate of the image. The elements `<onClick>` and `<tooltip>` show other features of this language. `<onClick>` causes an event listener to be inserted into the SVG element, and `<tooltip>` causes a tooltip to be inserted. `[external_Link]`, `[startTime]` etc. refer to elements and attributes in the OTN data source.

Putting it all together

Now we have the data source, i.e. the GIS data as OTN instances in the OWL format. We have the SVG graphics elements as the transformation ontology in OWL, and we have transformation rules as instances of the transformation ontology. This is the declarative part. The actual transformation is now done by a particular Java program. For each element of the transformation ontology (see Fig. 4) there is a corresponding Java class. They have methods which know how to match the OTN data with instances of the transformation ontology and how to generate SVG code from this.

For example, there is a Java class SvgImage. This class can be instantiated with the parameters of the SVGImage instances of the transportation ontology, the `Bus` instance from above, for example. Now we have a Java object whose methods are able to search through the OTN data and to identify the items for which SVG code is to be generated that inserts the little image of the bus. This information is inserted into an R-tree, and from the R-tree the system generates the SVG files for the tiles of the map.

All this sounds quite complicated, but it is extremely flexible. It allows to change or extend the displayed map by just changing or extending the instances of the transformation ontology. It is also quite straightforward to add new information from new data sources, for example symbols for traffic jams from TMC (Traffic Message Channel) data [13, 12]. The OTN ontology must be extended to contain the concept of traffic jams, the TMC data must be turned into the OWL data format, and a new SVGImage instance must be added to the transformation ontology.

3.1 Formulas

The transformation from the geographic data to the SVG data may require calculations which can be specified in the instances of the transformation ontology. We saw already some examples of the formula language which is used there. A formula always begins with =, followed by a number of operators and arguments. All strings not starting with a = are handled as static strings or text entries. If an operation leads to an invalid or no result, the resulting value is treated as 0. This can occur when the syntax of the formula is not correct or it cannot be calculated (e.g. division by zero, etc.). Since texts can often contain regular brackets "(" and ")" formulas contain curly braces instead ({}).

Operations +, −, ∗, / and % can be applied to numbers and text, although if applied to text

the argument are treated as 0. There are, however, exceptions. If + is applied to text, the arguments are concatenated, if $*$ is applied to a text and a number n, the text is concatenated n times.

Comparison operators are <, >, <=, >=, <> and =. These return 1 if successful, otherwise 0.

Type conversion is denoted by "@". The desired type is given directly before the @ (no whitespace in between) and the value follows. Predefined types are the following:

- INT@ conversion to an integer
- TIME@ extracts the time from a timestamp (which also includes a date)
- DATE@ extracts the date from a timestamp
- DATETIME@ extracts time and date from a timestamp

The attributes of features can be accessed using [ATTRIBUTE_NAME]. The ID of a feature can be accessed using [ID] although in a strict sense it does not represent an attribute. If the attribute is in turn a feature, its attributes can be accessed using a dot notation, such as [FEATURE_NAME]•[ATTRIBUTE_NAME]. An edge for example begins (*starts_at*) at a point in space which contains an x-coordinate. Its value can be accessed using `[starts_at].[x]`.

If the feature is a *Line*, for each vehicle travelling along this line a separate graphical symbol is generated. Furthermore, standard attributes can be accessed, such as x- or y-coordinates of the vehicles current position. The current line and the next (resp. previous) stop can be referenced through *[route_Section]*, *[starts_at]* and *[ends_at]*. The times of arrival and departure are found in *[startTime]* and *[endTime]*. *[waitingTime]* and *[drivingTime]* hold the idle time before departure and the duration of travel.

4 Further Services

The browser that renders SVG data has, via JavaScript, access to the data structures underlying the items on the generated image. This can be exploited to implement further services. One of the services we implemented is a road finder. Since every road has a name, the browser can build an index for the roads when they are downloaded. The user can now type in the name of a road and the browser uses the index to match the road name with the SVG elements that display the road. These elements are now highlighted by changing their colour attribute. So far this works only for roads. The reason is that different types of objects are usually represented by different combinations of SVG elements. A road, for example, consists of road segments which are displayed with one or two (or more) SVG path elements. This association is different for other types of objects and therefore has to be programmed in another way.

Highlighting particular roads is a service which can be executed at client side. We also implemented a prototype of a service where the client has to contact the server. This service searches the shortest path between too locations on a map. The client sends the two locations to the server, the server computes the shortest path, and sends back an SVG documents that shows

the shortest path in the browser window. So far, only the interface between client and server is implemented and only fixed test data are sent over the interface.

5 Summary and Outlook

In this work we illustrate a particular use of ontologies for dealing with geographic data. The geographic data are represented in the OWL data format that corresponds to the Ontology of Transportation Networks (OTN). Since there is a certain degree of independence between the data and the ontology, it is possible to adapt the ontology to the needs of the application and still work with the same data.

The transformation of the geographic data into SVG is also controlled by an ontology. The SVG elements are represented as concepts of a *transformation ontology* and the particular rules for transforming the data in a particular way are specified as instances of the concepts of the transformation ontology. By changing these instances or creating new instances one can change or extend the displayed maps very easily. Therefore this is an extremely flexible architecture which allows one to program the generation of maps by specifying the transformation in a symbolic way.

There are two main differences to other approaches for generating maps:

1. the programming technique is *semantic* with a significant symbolic specification part. This is much more flexible than other programming techniques. For example, if the presentation of the map is to be changed or extended, it is usually sufficient to start an OWL editor and change some classes or instances;

2. behind the displayed map there is always the document object model (DOM), and the elements of the DOM are still linked to the ontology. This enables interactive services where the ontology can be invoked.

SVG is a very expressive language with a number of quite powerful features. The renderers for SVG are therefore quite complicated and still seem not be optimised for large data sets. The dynamic features of SVG in particular slow down the rendering considerably. Rendering big maps with a lot of items is still slower than it could be. We are currently exploring the possibility to write a renderer in Java, not for full SVG, but for the special constructs needed for visualising maps.

In another ongoing work we want to integrate dynamic data sources into the visualisation mechanism. A particular dynamic data source is the Traffic Message Channel (TMC). It broadcasts information about traffic jams and other traffic events digitally over radio. This digital information can also be turned into SVG documents which can be downloaded into the client to show the actual status of the information. This will be the subject of one of the forthcoming deliverables.

Acknowledgement

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net).

References

[1] Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation, January 2003, http://www.w3.org/TR/SVG11/

[2] Adobe SVG Viewer, Version 3.01, September 2003, http://www.adobe.com/svg/

[3] Document Object Model (DOM) Level 2 Core Specification, November 2000, http://www.w3.org/TR/DOM-Level-2-Core/

[4] Geography Markup Language Version 3, http://www.opengis.org/docs/02-023r4.pdf

[5] Visualisierung von GML mit XSLT und SVG, Diploma thesis of Andreas Kupfer, September 2003, http://www-public.tu-bs.de:8080/~y0010824/svg/index.html

[6] Basisontologie und Anwendungs - Framework für Visualisierung und Geospatial Reasoning. Diploma thesis by Frank Ipfelkofer, 2005.

[7] OWL Web Ontology Language, W3C Recommendation, February 2004, http://www.w3.org/TR/owl-features/

[8] International Organisation for Standardisation http://www.iso.org

[9] Intelligent transport systems, ISO/TC 204, http://www.iso.org/iso/en/stdsdevelopment/tc/tclist/TechnicalCommitteeDetailPage.TechnicalCommitteeDetail?COMMID=4559

[10] Geographic Data Files 3.0 (GDF) Documentation, 1995, http://www.ertico.com/links/gdf/gdfdoc/gdfdocon.htm

[11] Intelligent transport systems - Geographic Data Files 4.0 (GDF) - Overall data specification, ISO/DIS 14825, August 2002, http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=30763&scopelist=PROGRAMME

[12] Radio Data System Forum http://www.rds.org.uk/rds98/rds98.htm

[13] Traffic Message Channel Forum http://www.tmcforum.com/

[14] R-Tree Portal, Juni 2003, http://www.rtreeportal.org/

[15] RTree Visualization Demo der National Technical University of Athens, November 1999, http://www.dbnet.ece.ntua.gr/~mario/rtree/