A Tool for Converting Public Transport Schedules

Peterpaul Klammer

Supervisor: Prof. Johann Gamper
Free University of Bozen - Bolzano

October 11, 2011

Contents

1 Introduction and Problem Definition **4**
1.1 Background 4
1.2 Problem Definition and Contribution 5
1.3 Organization of Thesis 7

2 Related Work **7**

3 Converting SASA and SAD Schedules **8**
3.1 Overview 8
3.2 SASA Solution 9
3.2.1 Importing the Data into the Database 10
3.2.2 Importing the Geometry 11
3.2.3 Generating the Calendars 11
3.2.4 Correcting the Representation of Time 12
3.2.5 The Stop Sequence 13
3.2.6 Completing the Tables 13
3.2.7 The Export 14
3.3 The SAD Solution 14
3.3.1 Parsing 15
3.3.2 Update Geometry 16
3.3.3 Generate an Unique TripID 17
3.3.4 Generating the Stop Sequence 18
3.3.5 Generate Calendars 19

4 Implementation **21**
4.1 Automatization and Web Application 22
4.2 Validating and Visualizing the Feed 23

5 Conclusion and Future Work **24**

Abstract

South Tyrolean transport agencies (SASA[1], SAD[2]) provide their data in proprietary non-standardized formats. This data describe the schedules, trajectories, trips and routes of the different means of transport (bus, train, cable car, funicular, etc.). In order to use this data for public services (e.g. Google Bimodal Shortest-Path Queries or Isochrones Queries) there is the need to convert it into a standardized format. The Google Transit Feed Specification (GTFS)[3] defines a common format for public transportation schedules and associated geographic information.

The Google Transit Feed Specification allows to represent the data graphically and to work with it. The main goal of this thesis is the transformation of the given data input files into the Google Transit Feed Specification. The solution is based on the import of the raw data into a database. Once imported there, the data is transformed and rearranged in order to be used for the specification. Therefore the main goal of the thesis is to implement a solution that converts specific formats of SASA and SAD to the Google Transit format. A second important task of this thesis work is to test the integrity of the data after the transformation to the Google Transit format. The goal is a transformation with as much data as possible, but it is not possible to transform all of the provided data. For example, not all of the provided data of the SAD agency can be converted.

1 Introduction and Problem Definition

1.1 Background

Now a days in the field of Spatial Network Databases a lot of new applications are developed and used. Examples for applications of that type are "shortest path with Google", "kNN queries" and "Isochrone computations". In Figure 1 you can see a Google shortest path query using the feed of San Francisco(USA) and an isochrone computation developed by the University of Bozen - Bolzano.

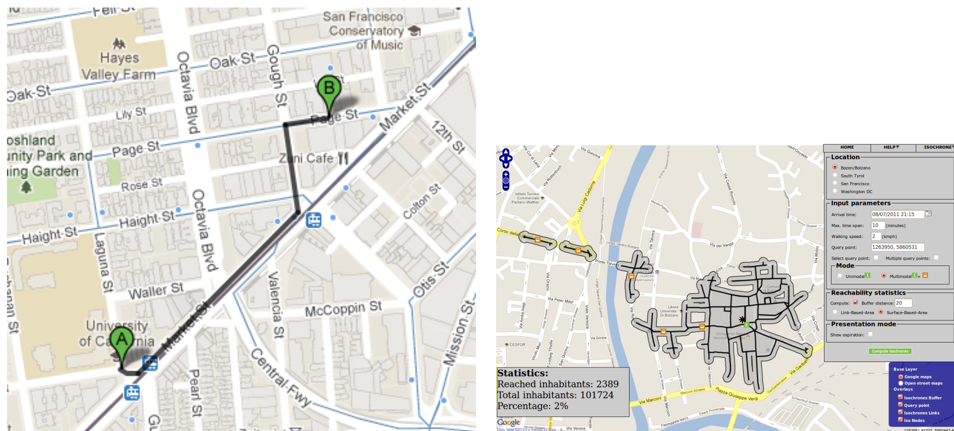

Figure 1: Shortest path and Isochrone computation

All of these applications have a common problem, they need the schedules of the public transport agencies if they should be considered for the computations. That is the reason why for public transport agencies it is very important to provide their schedules and their geographical data to their customers and users. South Tyrolean transport agencies such as SASA and SAD provide their data in a non-standardized format as many of the agencies are doing. Therefore there is the big need to change the raw data into a standardized format.

The goal is to develop a tool that converts different formats to a common one. The tool should start with one single command and then convert the given input data, so that the converted output called feed can be used. In practice a web application takes a given input zip file and uploads it to the server, where the data is converted to the Google Transit Feed Specification. After the converting step the web application provides a link to the location where the feed can be downloaded and in addition an email notification is sent to the transport agency. A second important step was to verify that the data after the converting is still consistent. The goal is to lose as less as possible data sets from the given input raw data that was provided by the

transport agencies and the University. Google defines the Google Transit Feed Specification, with which Google is able to visualize transit schedules. With this specification Google is distributing the work of generating the feed to the developers. Local developers then do the work, they implement a solution for the local areas.

The Google Transit Feed is a zip file composed of text files containing multiple comma separated fields. The structure of the feed is defined by the specification. The most important text files that have to be in the zip feed are the following ones:

- agency: contains agency_id, agency_name, agency_url and timezone.
- stops: contains stop_id, stop_name, stop_desc, stop_lat and stop_lon.
- routes: contains route_id, route_short_name , route_long_name and route_type.
- trips: contains route_id, service_id and trip_id.
- stoptimes contains: trip_id, arrival_time, departure_time, stop_id and stop_sequence.
- calendar contains: service_id, monday, tuesday, wednesday, thursday, friday, saturday, sunday, start_date and end_date.
- calendar_dates contains: service_id, date, exception_type

The transformed zipped feed can be shared to other developers or can be used for own purposes to work with the data. In this paper I will show how the transformation of the raw data to the finished working feed is done with the implemented utility step by step. Furthermore the used technologies are shown and explained.

1.2 Problem Definition and Contribution

The problem is that local transport agencies are providing their data in a non-standardized proprietary formats, so that only they can work with the data. Local transport agencies made a request to the University of Bozen - Bolzano to develop a tool to convert their raw data, so that they can change their format and system. The raw data of SASA differs from the one of the SAD, so that two different tools are needed to convert all the data. Therefore the main task is to provide a single tool, that converts data in an efficient way such that as less as possible data is lost. Second, the obtained new data should be consistent with the old data, the new data in the Google Transit Feed should be exactly the same.

The SASA input files are composed of three CSV files, which are the following ones:

- orariPassaggio.csv this file contains all the information about the stops and stop times. For example:

TripID	optionalLineID	StopID	Stop_sequence	Departure_time
1	643	5108	1	06:43:00
1	643	5110	1	06:44:00

- lineeCorse.csv this file contains all the information about the trips. For example:

TripID	RouteN	Location	Desc. Ita	Desc. De	Direction	Date	Calendar
1	1	BZ	Fago 1	Fagen Str. 1	A	01.04.11	0001...
2	1	BZ	S.Gen	Jenesien.	A	03.04.11	1101..

- linee.csv this file contains all the information about the routes. For example:

RouteID	Short Name	Long Name	Desc	Opt.RouteName	Opt.Location
1	BZ-ME	Bolzano-Merano	L. 201 Merano	B-M	Merano
2	City LA	City LA	CityB-Lana	C-L	Merano

- sasa_ge_busdata.kml this file contains the geometry entries for the stops. For example:

```
<Placemark>
  <name>553: Parcines - Partschins </name>
  <Point>
  <coordinates>
    11.0749697222222,46.6832566666667,0
  </coordinates>
  </Point>
  <description><![CDATA[13 ME / ]]>
  </description>
</Placemark>
```

The SAD input file is more complex. The data provided by the SAD is in a special format that the SAD defined by their own, it is named SNTSEL. The data is contained in files that need to be parsed in a precise way in order to get the right information. The first lines of the file are containing information about the version of the file and how long it is valid. Then at some point a tabulator with some number indicates how many lines of entries are following. Each line contains a certain number of entries that are separated by spaces. The following example shows that the tabulator with the number 785 indicates that 785 lines of various entries are following and need to be parsed.

```
Revision: 9.0
20110310 20110301 20110618
  785
10 34 1 Bolzano Bozen 021027 1 20110301 20110618
11 35 1 Resia Reschen 021027 1 20110301 20110618
12 55 1 Curon Graun 021027 1 20110301 20110618
```

The contribution of this thesis is the development and implementation of a conversion tool. With this tool the provided non-standardized data can be converted to a common format that is Google Transit Feed as already mentioned before in the introduction. Local transport agencies and the

University of Bozen - Bolzano can use this tool to convert the data and to work with the gained output data.

1.3 Organization of Thesis

In Section 2, related work is discussed, in particular repeated schedules are discussed. In Section 3 the solutions for the raw data provided by the local transport agencies SASA and SAD are shown. The most important steps are shown in chronological order. Section 4 shows how to use the tool and how to set up the property file. The use of the command line interface is explained and also the web interface. Finally Section 5 conclude the thesis. In this Section also some future work is discussed.

2 Related Work

The movement in public transport is organized according to schedules. These schedules have a number of regularities and irregularities. Cancelled or additional trips are the irregularities, such as holidays and strikes. This causes some difficulties to handle real world data[4] . Traditionally schedules are represented as regular schedules. But that is not always true because of occurring exceptions. The most actual solutions are linked to a static converting. The Google Transit Feed is solving this problem by storing a regular schedule for a certain period of time. The period is specified with a start and a clear end date. In the whole period the schedule is valid, if an exception occurs it is stored in the exceptions file. With that approach it is possible to represent trips with the corresponding schedules. This approach is linked to the discovery of common patterns of events. The algorithm is designed to find commonly occurring sequences[5] . One problem linked to this approach is to find the common pattern inside the period of validity, this problem is also occurring in this thesis. SASA and SAD both are representing their schedules in a bit string with a start and end date. The problem was to find a common pattern that is occurring frequently. How this problem is solved is shown later in this paper. The Google Transit Feed specifies the calendar so that for every weekly day a 0 or a 1 indicates if regularly on that day the service is traveling. If an exception is needed this exception is stored inside the exception file. This makes it possible to represent every kind of trip during the given period of validity.

Storing time-qualified data is causing the phenomenon of temporal repetitions i.e. the association of multiple time values with the same data[6]. Google tries with the Transit Specification to store the data in an efficient way. The schedules are split into routes, trips, stops, stop times and calendars. This avoids that data is stored twice inside the database or the files. Identifying common patterns is very useful, because agencies have

some trips that use the same schedules. Since the schedules are stored inside a separated file one schedule can be used for two or more trips that have the common schedule. This is also preventing redundant data entries in the files.

3 Converting SASA and SAD Schedules

3.1 Overview

The System Architecture is shown in Figure 2, it shows the general architecture, containing the most important parts of the solution. Apache Ant[7] is the main part of the solution, Ant is connecting all the single short pieces of Java code and SQL scripts and executes them in a chronological way. A property file containing all the relevant information is passing the parameters to the Ant build script, such as input file path, table names, output file path, etc. With these parameters Ant is executing SqlLoader, SqlPlus from Oracle and Java code. The single pieces of code then are interacting with the Oracle database. After the transformation and modification of the raw data the export can be performed. This is done by SqlPlus, that uses a command called "spool" to redirect the sql statement output into a file.

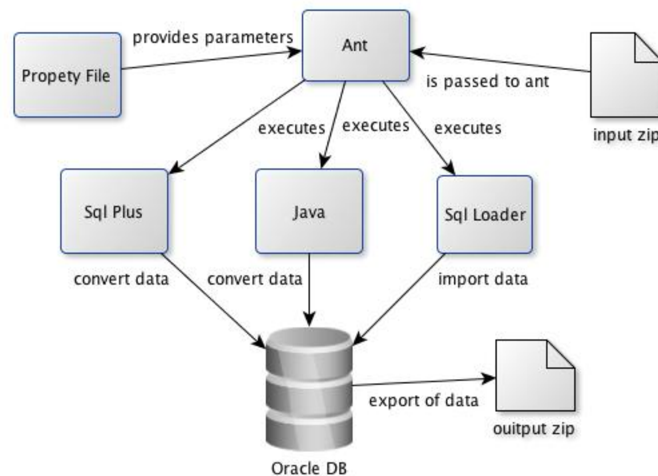

Figure 2: System Architecture

The general algorithm to solve the problem is to import the data into a database by parsing it with Java code or Oracle SqlLoader. Imported into the database the data can be joined and modified in the needed way, this is done by Java code or some SQL scripts that are executed with Oracle

SqlPlus. After transforming, the data can be exported with SqlPlus that provides the "spool" command. With these command the output of the SQL statement is redirected into a text file.
To summarize, the most important steps are:

- Import the data with Java or SqlLoader by parsing it line by line.
- Transform the data using Java and SqlPlus.
- Export the data with SqlPlus to a file.

Apache Ant is a Java library and command line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. With Ant it is possible to specify certain properties inside a property file, these properties are then red by Ant and passed as parameters to the called targets. This is really useful, because it is possible to specify dynamically the table name or feed names etc. This helps to make the utility dynamical and if a change is needed it is easier to apply changes to the code. The target that calls the solution is finally executing all the needed steps such as import, migrate, convert and export of the data. Ant is a very efficient way to combine multiple code snippets, for example this solutions uses multiple technologies to get the feed and is combining all of them to one consistent piece, that can be started with one command and generates the final output feed. The gained feed is then ready for use. For the SASA the "sasa2gtfs" task is shown in the Figure 3.

```
<target name="sasa2gtfs" depends="init">
 <antcall target="unzip" />
 <antcall target="migrate" />
 <antcall target="initPostFunctions" />
 <antcall target="complete" />
 <antcall target="exportTabls" />
 <antcall target="zipFeed" />
 <antcall target="terminate" />
</target>
```

Figure 3: sasa2gtfs target

3.2 SASA Solution

With the SASA raw data the first important step was to understand the concept how the data is stored inside the CSV files. The files are arranged in such a way that it is almost intuitive to get an understanding of the data. The data needs to be splitted up into smaller pieces, from the four input files in order to be exported correctly. In addition the imported data needs to be changed in some cases. For example the SASA uses a Solari date format, because they need that format to be able to display the trips at the stops with their monitors that are installed there. Such data needs to be modified to be able to use this data with the feed.

3.2.1 Importing the Data into the Database

The first implementation step is to make tables in order to be able to import the raw data into an Oracle database. The tables are created with SqlPlus executed by Apache Ant targets. The scripts contains placeholders that are replaced by the names contained in the property file. The property file has the big advantage that table names and attributes can be changed easily if some change is needed. Otherwise the whole tool would have to be modified in order to change some table names. After that step the data files are imported with the SqlLoader from Oracle, this step is very easy because the syntax of this utility is very easy and intuitive. An example template is shown in Figure 4, placeholders look like @spaceholder@ and are later replaced by an ant target as shown in Figure 5. The Ant target replaces the placeholders @CSV FILE@ and @TABLENAME@ and then executes the copied template with the SqlLoader. When the target is needed another time the placeholders are replaced by the properties that are indicated in the property file and then executed. After the import of the data into the temporary tables the data can be transformed.

```
LOAD DATA
INFILE '@CSV_FILE@'
INTO TABLE @TABLE_NAME@
FIELDS TERMINATED BY ";" OPTIONALLY ENCLOSED BY '"'
(
  ROUTE_ID SEQUENCE (1,1),
  ROUTE_SHORT_NAME,
  ROUTE_LONG_NAME,
  ROUTE_DESC,
  AGENCY_ID CONSTANT '1'
)
```

Figure 4: SqlLoader Script

```
<property name="tableName" value="@setme" />
  <property name="tempCtrlFile" value="@setme" />
  <property name="sourceFile" value="@setme" />
  <property name="dataFile" value="data.csv" />
  <property name="logFile" value="import_${tableName}.log" />
  <property name="ctrlFile" value="import_${tableName}.ctrl" />
    <copy file="${properties.scripts.importPath}/${tempCtrlFile}"
   tofile="${properties.scripts.importPath}/${ctrlFile}">
      <filterchain>
        <replacetokens>
   <token key="TABLE_NAME" value="${tableName}" />
   <token key="CSV_FILE" value="${dataFile}" />
 </replacetokens>
      </filterchain>
    </copy>
<exec dir="${properties.scripts.importPath}" executable="sqlldr">
  <arg value="${properties.db.user}/${properties.db.password}"/>
  <arg value="control=${ctrlFile}" />
  <arg value="log=${logFile}" />
</exec>
```

Figure 5: Ant target

3.2.2 Importing the Geometry

The specification requires that every stop provide the geometry where the stop is located. A valid WGS 84 latitude and longitude are required for every stop. The corresponding geometry provided by the SASA is stored inside a KML file, this KML file can be parsed with a piece of Java code that then stores the geometry into a geometry object inside the database. From that object the latitude and longitude can be extracted easily to the final tables. The following pseudo code describes the function of the Java code:

while $KML\ entry \neq null$ **do**
  $KML \rightarrow read$
  $KML \rightarrow toDB(with\ prepared\ Statement)$
end while

3.2.3 Generating the Calendars

Generating the calendars is another very important step of the transformation of the input data. Google transit requires for every trip a schedule entry, this means that every trip has a weekly schedule that indicates whether the service is valid for the given day. A table is generated that has an entry for Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday, a start date and an end date. A 1 indicated that the service is valid at that day, instead a 0 indicates that it is non-valid. In addition to this also an exception table is created, in this table the exceptions are stored that occur. The calendar issue is solved by a piece of java code that takes the input string that looks like:

0111111011111101111 (in reality much longer)
with start_date: 02.04.11

The 02.04.11 is a Sunday, therefore on Sundays the trips is not valid, because this is indicated by the 0. Instead on Monday the 03.04.11 it is valid. Therefore after the execution of the Java code it is known that the schedule regularly is valid from Monday until Saturday. Then after the 16.04.11 exceptions are occurring, and these exceptions are then inserted into the exception table, so that these exceptions then can be exported to the feed at the end as also the calendar itself. The algorithm looks as follows:

for every bit string **do**
  bitstring → *find most common weekly schedule*
  schedule → *write schedule to table*
  schedule → *find exceptions and write to table*
end for

The finished calendar for the example trip then looks like:

service_id	mon	tue	wed	thu	frid	sat	sun	start	end
5	1	1	1	1	1	1	0	20110403	201106010

3.2.4 Correcting the Representation of Time

The times need to be converted to a special format that Google Transit Feed uses. Trips that span multiple dates will have stop times greater than 24:00:00, in other words, if a Trip starts at 10:30:00PM and ends at 2:15:00AM on the following day the stop times would be 23:00:00 and 26:15:00. This issue is resolved by an SQL script that changes the timestamp to the special output format that is used by Google Transit Feed. The SASA agency is using a Solari date format, because they need to visualize their schedules on the monitors that are installed at their stops. On the monitors the next departures are shown, therefore they need that format. In order to use this date, there is the need to convert it with a given tool that is written in Java. The problem with the trips that start before midnight and end after midnight is solved by the mentioned SQL function. The script is shown in Figure 7, the function TIMESTAMP2STRING is created with that script. The function converts the time if it is needed. If a value is larger than 24:00:00 the amount of additional hours is added to the time. How this looks like is shown in Figure 6. The initial values are shown on the left, and on the right the values after converting them.

TripID	StopID	DepartureTime
5	5432	23:57:00
5	5433	23:59:00
5	5434	24:05:00
5	5434	01:01:00

TripID	StopID	DepartureTime
5	5432	23:57:00
5	5433	23:59:00
5	5434	24:05:00
5	5434	25:01:00

Figure 6: Timestamp2String example data

```
DROP TABLE TMP_MAX_DATE_STR;
CREATE TABLE TMP_MAX_DATE_STR AS
SELECT TO_CHAR(MIN(DEPARTURE_TIME),'YYYY-MM-DD')
AS MAX_DATE_STR FROM @TABLE_NAME@;
CREATE OR REPLACE
FUNCTION  TIMESTAMP2STRING(
P_TIME TIMESTAMP,DATE_PATTERN VARCHAR2) RETURN VARCHAR2
IS
  MAX_TIME TIMESTAMP;
   DATE_STRING VARCHAR2(10);
   TIME_STRING  VARCHAR2(10);
   HOUR_STRING  VARCHAR2(10);
BEGIN
   SELECT MAX_DATE_STR INTO DATE_STRING
   FROM TMP_MAX_DATE_STR WHERE ROWNUM=1;
   MAX_TIME:=TO_TIMESTAMP(CONCAT(DATE_STRING,'-23:59:59'),CONCAT(DATE_PATTERN,'-HH24:MI:SS'));
   IF P_TIME IS NULL
   THEN RETURN NULL;
   END IF;

   IF  P_TIME < MAX_TIME
   THEN
    TIME_STRING:=TO_CHAR(P_TIME,'HH24:MI:SS');
   ELSE
     HOUR_STRING:=24 + TO_NUMBER(TO_CHAR(P_TIME,'HH24'));
     TIME_STRING:=CONCAT(HOUR_STRING,CONCAT(':',TO_CHAR(P_TIME,'MI:SS')));
   END IF;
   RETURN  TIME_STRING;
END;
/
QUIT
```

Figure 7: Timestamp2String function

3.2.5 The Stop Sequence

The SASA is already providing a stop sequence, that saves a lot of time, because Google Transit Feed requires a stop sequence. A stop sequence indicates the order of stops for every particular trip. For example a certain trip first stops at stop A, so that A has stop sequence 1, later it stops at Stop B, that has stop sequence 2. The stop sequence is important, because as the name stop sequence already says, it guarantees that the stops of a single trip can be determined very easy in chronological order. The finished stop sequence of the shown initial SASA values can be seen in Figure 8.

TripID	StopID	ArrivalTime	DepartureTime	Stop_sequence
1	5108	06:43:00	06:43:00	1
1	5110	06:44:00	06:44:00	2
1	5112	06:44:30	06:44:30	3
1	5114	06:45:00	06:45:00	4
1	5115	06:45:30	06:45:30	5

Figure 8: Stop sequence example

3.2.6 Completing the Tables

In this step the data of the temporary table is rearranged in the final tables where the data is ready for the export. This works well with the SASA data, so that no data is lost. Most of the data is at the right place already

in the temporary tables so that the final export tables are looking almost identically compared with the temporary ones. At this step the only problem is to find unique keys, in order to be able to find a correlation between the data. If there would be no unique key it would not be possible to transfer the data to the Google Transit Feed. With the keys the data can be positioned in every table were the specification requires them. So that the data finally is ready for the export.

3.2.7 The Export

The export of the data is done with the SqlPlus from Oracle, with the command "spool" it is possible to redirect the output of the command line execution into a file. The file headers that are required by the Google Transit feed are copied with an Apache Ant target before the execution of the export, so that the output is added to the bottom of the header file. In Figure 9 the placeholders @OUTPUT_FILE@ and @TABLE_NAME@ are replaced before the execution by an ant target with the right table name.

```
SET TRIMSPOOL ON;
SPOOL "@OUTPUT_FILE@" append;
SELECT
 ROUTE_ID ||',"'||
 Rtrim(ROUTE_SHORT_NAME) ||'","'||
 Rtrim(ROUTE_LONG_NAME) ||'","'||  Rtrim(ROUTE_DESC) ||'",'||
 ROUTE_TYPE
FROM @TABLE_NAME@ ORDER BY ROUTE_ID ASC;
SPOOL OFF;
QUIT
```

Figure 9: Example output script

3.3 The SAD Solution

In general the procedure to generate the feed from the SAD input files is the same, only that it is a little more complicated to do it. The SAD data requires some thinking, because it is not that trivial to understand the data modeling by looking at it. The manual needs to be understand and some logistical problems need to be solved when the data is rearranged for the Google Transit Feed.

In general the main steps for the SAD data are:

- Import the data with Java parsing it line by line.
- Join and modify the data by using Java and SqlPlus.
- Exporting the data with SqlPlus to a file.

3.3.1 Parsing

This step with the SAD data is not very easy, because all the data is in one single file, and is listed after each other. The data is stored in the following order:

- Stops
- Routes
- Trips
- Stop Times

In addition there are two files containing the stops geometry, that also need to be parsed, they are parsed in the same way as the sntsel.txt file. The general algorithm to parse the sntsel.tot file is:

while $file\ entry \neq null$ **do**
  if line starts with TABULATOR **then**
    STATUS++
  end if
  if $STATUS == 1$ **then**
    Split Stops data and insert into db
  end if
  if $STATUS == 2$ **then**
    Split Routes data and insert into db
  end if
  if $STATUS == 3$ **then**
    Split Trips data and insert into db
  end if
  if $STATUS > 3$ **then**
    Split Stops Times data and insert into db
  end if
end while

After every tabulator the status is increased, so that it is possible to insert every data set into the right table. If the status is larger than 3 this means that the stop times are proceeded. Every stop has different stop times

from different trips. Each single stop time entry has an unique id and starts with a tabulator. Every entry is proceeded once at a time. All the stop times are inserted with the corresponding trip_id and route_id. After the import into the database the stop times table is looking as shown in Figure 10.

StopID	T_ARR	T_DEP	RouteID	TripId	PredNode	SuccNode
4513	20:16:00	20:16:00	519	8466	-1	4573
4001	20:58:00	20:58:00	519	8466	2139	-1
4573	20:20:00	20:20:00	519	8466	4513	467
606	20:41:00	20:41:00	519	8466	35	2115
2115	20:45:00	20:45:00	519	8466	606	2139
467	20:25:00	20:25:00	519	8466	4573	603
603	20:30:00	20:30:00	519	8466	467	35
35	20:38:00	20:38:00	519	8466	603	606
2139	20:49:00	20:49:00	519	8466	2115	4001

Figure 10: Stoptimes table example

3.3.2 Update Geometry

One problem that occurred was that there is no clear geometry, because the SAD uses so called "palines", what is meant by a "paline" is illustrated in Figure 11. Every point represents one stop inside one location called "paline". For example my home town Vintl - Vandoies is one paline containing 6 different stops. The problem is that every stop is referenced to one "paline", which means that 1 stop is referenced to several different geometries. This means that no unique geometry for the stops is given. The solution for solving this problem is to take a centered point inside the paline. With that approach one "paline" references to one stop.

Technically this is solved by the SQL function SDO_AGGR_CENTROID, that takes a set of points an searches the central point in the given set of points. After this step it is possible to reference the geometry into the temporary stops table with a script, shown in Figure 12.

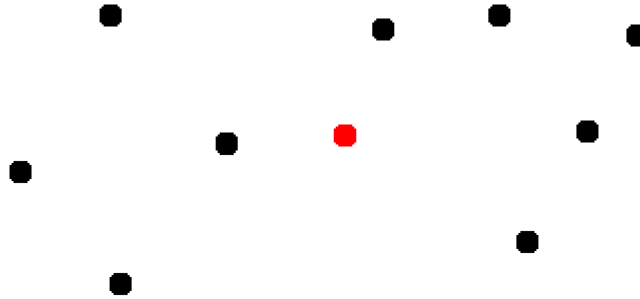

Figure 11: A Paline

```
update @BUS_STOPS@ S1 set S1.geometry=(
select SDO_AGGR_CENTROID(MDSYS.SDOAGGRTYPE(S2.geometry, 0.005))
from @BUS_STOPS_GEOMETRY@ S2 where S1.stop_id=s2.stop_id
);
```

Figure 12: Script for updating the geometry

But this approach has two disadvantage, the first that some of the given points can not be used and the second that the geometry is not 100% precise. The loss of data sets is around 5%, but without this compromise it would not be possible to transfer the data to the Google Transit Feed. Therefore the loss of data has do be taken into account.

3.3.3 Generate an Unique TripID

Another problem is that the SAD data provides no unique TripID, that makes it difficult to generate the feed. Because the specification requires a unique TripID for the whole set of data. Therefore there was the need to find a unique way to identify a set of attributes in order to be able to generate an unique TripID, that then can be referenced to the different tables that contain the old ID. It is possible to identify every single data entry by routeID, tripID and the first instance of the entry, with these attributes as unique identification it is possible to generate a new unique tripID in the trips table and later that new tripID can be updated in the stop times table. This approach has the disadvantage that some data sets are lost, but the amount of lost data is still acceptable because it is between 5 -10 %. This approach is needed, because without an unique trip id the feed could not be generated. Using only the first instance seemes to be the best solution for the transformation of the data, because from the SAD specification it was

not possible to understand the meaning of having two instances. And the number of two instances of one trip is very small, so that is makes sense to don't consider them. The script that generates the ID and references it is shown in Figure 13, and in Figure 14 a finished example is shown.

```
drop sequence SEQ_@T_TRIP_TAB@;
CREATE SEQUENCE SEQ_@TMP_TRIPS_TABLE@ INCREMENT BY 1;
UPDATE @T_TRIP_TAB@ T SET T.new_trip_id=SEQ_@T_TRIP_TAB@.NEXTVAL
where INSTANZA_TRIP_ID=1;
update @TMP_STOPTIMES_TABLE@ ST set new_trip_id=(
select T.new_trip_id
from @TMP_TRIPS_TABLE@ T
where T.new_trip_id is not null and
ST.route_id=T.route_id and ST.trip_id = T.trip_id);
```

Figure 13: Script for generating the ID

StopID	T_ARR	T_DEP	RouteID	TripId	PredNode	SuccNode
4513	20:16:00	20:16:00	519	5	-1	4573
4573	20:20:00	20:20:00	519	5	4513	467
606	20:25:00	20:25:00	519	5	35	2115
467	20:30:00	20:30:00	519	5	4573	603
603	20:38:00	20:38:00	519	5	467	35
35	20:41:00	20:41:00	519	5	603	606
2115	20:45:00	20:45:00	519	5	606	2139
2139	20:49:00	20:49:00	519	5	2115	4001
4001	20:58:00	20:58:00	519	5	2139	-1

Figure 14: Finished generated ID example

3.3.4 Generating the Stop Sequence

Generating the stop sequence with the SAD data is not that easy as with the SASA one, because the SASA agency already provided the complete stop sequence. Instead the SAD is only providing for every trip, the stops and the stop times (arrival and departure times). In order to be able to see in which order the stops are, the SAD gives also information for every stop such as stop predecessor and successor. If a stop has no predecessor it has the value -1 and when it has no successor it has the value -1. This makes it possible to generate the stop sequence with a piece of JAVA code. in Figure 15 is illustrated how the stop times of a trip can look like. The first entry has predecessor -1, so it is the first stop in the sequence. The first entry then has the successor stop 4573 therefore it is known that the entry with stop 4573 is the second stop in the sequence. Stop 4573 has successor stop 467 so it is the third stop in the sequence. And continuing this procedure the right stop sequence is generated.
An important thing is that in this approach already the new generated trip id is used, because with the old one the problem was much more difficult.

StopID	T_ARR	T_DEP	RouteID	TripId	PredNode	SuccNode	Stop_sequence
4513	20:16:00	20:16:00	519	5	-1	4573	1
4573	20:20:00	20:20:00	519	5	4513	467	2
467	20:25:00	20:25:00	519	5	4573	603	3
603	20:30:00	20:30:00	519	5	467	35	4
35	20:38:00	20:38:00	519	5	603	606	5
606	20:41:00	20:41:00	519	5	35	2115	6
2115	20:45:00	20:45:00	519	5	606	2139	7
2139	20:49:00	20:49:00	519	5	2115	4001	8
4001	20:58:00	20:58:00	519	5	2139	-1	9

Figure 15: Finished stop times example

The new id simplifies the whole problem a lot. The code first is reading all the relevant data from the database and then taking every single entry one by one. Each entry is proceeded after the other, if the predecessor is -1 a new object for that trip is generated in a vector that contains a vector of trips. If the predecessor is not -1 the trip is searched in the vector and there is added a new entry to the trip vector. When all the entries are proceeded the vector containing the trip vectors is red. Every trip vector is red after each other. With an update the sequence is generated, by updating where the data of the vector is equal to the data of the database entry. In that way the stop sequence is generated. This procedure is not very fast because the update of the database is very cost intensive, but there is the need for the stop sequence so the loss of time has to be considered into account. The following pseudo code describes how the stop sequence is generated.

while *result set with sequence data* $\neq$ *null* **do**
  if predecessor == -1 **then**
    TripEntry(stopId, tripId,newTrId)
  else
    TripEntry(stopId, tripId,newTrId, previousTripEntry)
  end if
end while
while *vector with elements* $\neq$ *null* **do**
  update StopSequence in DB where entry(stopId, tripId,newTrId, previousTripEntry) = stopID, TripID, nTripID, prevEntry
end while

3.3.5 Generate Calendars

Generating the calendars is one of the difficult parts with the SAD data, because the SAD provides a long string called bitfield for the period of validity specified in the first lines of the main document. The bitfield is a string that contains only the values 0 or 1, a 1 indicates that the trip is valid at the certain date instead a 0 indicates that it is not valid at that day. An example for a bitfield is the following one:

111111111111111111111111111111111111111111111111111111111111111111

In all the examples the start date is the 01.03.11 this remains the same, since the bit filed always has a unique start value for all the entries of the document, the start date is a Tuesday, therefore on that Tuesday the service starts and from there on the service is valid every day. A more interesting example would be the following one:

111000111110011101001111100111110011111001111100111110011111001111

This is an example for a service that is valid only from Monday to Friday and that is not traveling on Saturday and Sunday. On the fourth day there is an exception, this kind of exception needs to be stored into the calendar exemptions file, in order to be able to consider also that special cases. In this example the given service would be stored in the exceptions table with the entry

service_id = 2, date = 04.03.2011 exception_type = 2.

The 2 indicates that the service at this day is not valid, a 1 would indicate the opposite.
The problem was to generate the calendar with the given bitfield with given start date. This is realized by finding the most frequent weekly pattern in the bitfiled. This weekly pattern is stored as the weekly schedule in the calendar table, and all the exceptions are then stored inside the exception table. Later the calendar tables can be exported easy by Ant. The algorithm for solving this problem is the following one:

while $set\ of\ data \neq null$ **do**
  $bitfield \rightarrow identify\ start\ and\ end\ date$
  $bitfield \rightarrow identify\ most\ frequent\ weekly\ pattern$
  $write \rightarrow schedule\ with\ dates\ and\ service_id\ into\ database$
end while
while $calendar\ entry\ in\ db \neq null$ **do**
  for $i = 1 \rightarrow bitfield.length$ **do**
    if bitfield.at(i) != schedule.day **then**
      $write \rightarrow exception\ into\ exception\ table$
    end if
  end for
end while

4 Implementation

The solutions can be used in two ways, the whole code can be used by executing the ant target from the command line. This is done by navigating into the ant folder of the project where one of the following commands is executed:

- ant sasa2gtfs
- ant sad2gtfs

Secondary and more comfortable the web application can be used. The web application provides a html page that offers the possibility to upload the input file to the University host, and finally the generated output feed can be downloaded by a provided link to the feed. In addition to this an email is sent to the responsible of the transport agency. In Figure 16 the web interface is shown.

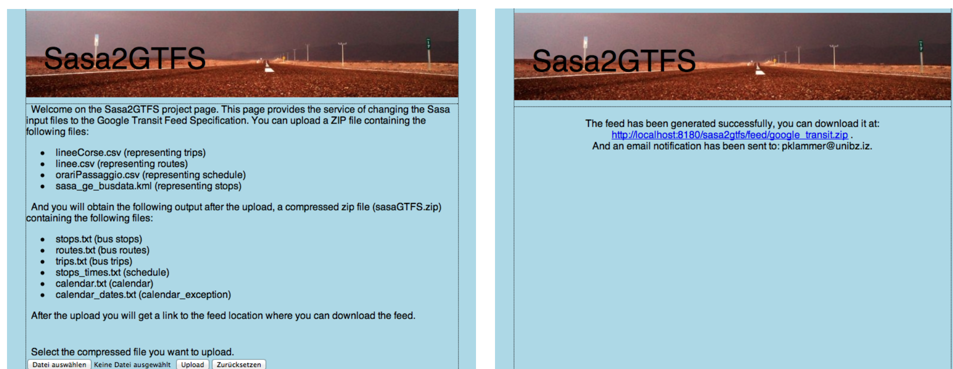

Figure 16: The web application

The property file contains all the needed information for the execution of the whole project. It is important to specify all the needed information in this xml file. Ant is extracting the data from the file and passing it to the methods and targets. The most important attributes contained in the property file are the paths of input and output files and the table names. Figure 17 shows how the property file looks like. The important attributes as inputPath, inputFile, outputPath, outputFile, srcDir, etc are shown. All the attributes are specified in this file, also the table names for creating the tables are set in this file. This guarantees that the tool is as dynamical as possible. The important settings are the following ones:

- project: input and output files.
- database: host with schema and user data.
- webapp: host, port and e-mail addresses.

- scripts: names of the used script templates.
- export: names of the used export scripts.
- parameters: here can be set that the tables are deleted after the export.

These attributes need to be set before the execution. Once set, the tool can be used unlimited. If needed, a change can be performed easily by changing the property file.

Element	Value
▼ properties	
▼ project	
name	SASA2GTFS
inputPath	./data/csv/
inputFile	googleSASA_20110628.zip
outputPath	./data/feed
outputFile	sasa-bolzano-it.zip
srcDir	src
feedName	feed
classDir	classes
libDir	lib
▶ database	
▶ webapp	
▶ tables	
▶ KML	
▶ correctTime	
▶ scripts	

Figure 17: Property file

4.1 Automatization and Web Application

The whole project is executed with Apache Ant, this means that a property file containing all the table information and the settings that are needed for the execution is delivering the information to Ant. The data is proceeded with an Ant build file so that the right parameters are passed to the Java code, the SqlLoader and SqlPlus. Ant is also responsible for the internal copying of data to the right folder, for example for the WEB application that works with Apache Maven[8] it is important to copy the whole ant project into a certain target directory, in order to be able to deploy the application.
The web application technically is realized with a servlet that gets called from a html page. The servlet takes the input file, and sets all the important properties in the property file, and then executes the ant target for the execution of the whole utility. The email notification is realized inside the servlet, the servlet uses a Java class to send the email.

4.2 Validating and Visualizing the Feed

The feed can be checked with a utility that is written in Python and also visualized with another one that also is based on Python. The validation of the feed is done with the python tool "feedvalidator.py", this tool takes as parameter the url to the feed that needs to be checked. After checking the data the browser opens a page that shows whether the feed is valid or not, and if there are some warning also they are shown. This looks like shown in Figure 18.

GTFS validation results for feed:
google_transit.zip
FeedValidator extension used: None

Agencies: SASA
Routes: 38
Stops: 901
Trips: 5128
Shapes: 0
Effective: April 03, 2011 to November 28, 2011

During the upcoming service dates Mon Sep 12 to Thu Nov 10:
Average trips per date: 462
Most trips on a date: 1523, on 1 service date (Mon Sep 12)
Least trips on a date: 124, on 8 service dates (Sun Sep 18, Sun Sep 25, Sun Oct 02, ...)

Found these problems:

74 errors	291 warnings
74 Missing Values	245 Duplicate IDs
	7 Invalid Values
	3 Stops Too Closes
	14 Too Fast Travels
	22 Unused Stops

Errors:

Figure 18: Feedvalidator output

After a successful validation of the feed it can be visualized with the "scheduleviewer.py", that is opening in the browser the feed, so that it can be used like if it would be on google maps. This offers the possibility to check the feed another time visually. Figure 19 shows how a feed looks like in the schedule viewer, a trip from Bolzano/Bozen to Merano/Meran is shown, all the stops are indicated with the stop times of that trip. This is very useful to check the feed again manually if it is correct or not. If a geometry would not be at the right place this can be seen easily with this python tool.

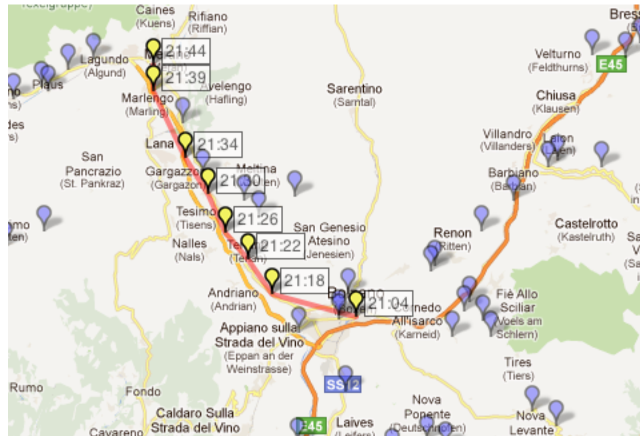

Figure 19: Scheduleviewer output

5 Conclusion and Future Work

The solution shown in this thesis is working quite well, even if there are some compromises that are needed in order to get a working feed. But more or less the loss of data taken into account is acceptable for the transformation from raw data to the feed. The provided data from SASA can be transformed 100% into the Google Transit Feed where the SAD data can be transformed only by 90% into the Google Transit Feed. The gained feed makes it possible to use the data that were provided by the local transport agencies. The feed can be used for Isochrone computations[9], kNN query processing, and for Google Shortest Path computations.

A possible step for the future could be the extension of the tool. It would be good if the tool could convert more formats to the common Google Transit Specification. Changing from SASA directly to SAD and vice versa would be nice. This could be realized by implementing a tool for changing Google Transit Feed to SASA and SAD formats. Also adding some new formats to the tool would make sense. For example the formats of other transport agencies such as the Austrian train agency or the German train agency could be added. The main goal for the future is to provide as much as possibe formats in the Google Transit Feed Specification.

References

[1] Sasa Bozen-Bolzano: http://www.sasabz.it/

[2] Sad - Trasporto Locale S.p.a.: http://www.sad.it/

[3] Google Transit Feed Specification:
http://code.google.com/transit/spec/transit_feed_specification.html

[4] Representing Public Transport Schedules as Repeating Trips - Romans Kasperovics, Michael H. Böhlen, Johann Gamper Free University of Bozen-Bolzano, June 2008

[5] An Algorithm to Discover Calendar-based Temporal Association Rules with Item s Lifespan Restriction - Geraldo Zimbro, Jano Moreira de Souza, Victor Teixeira de Almeida, Wanderson Arajo da Silva, Federal University of Rio de Janeiro

[6] Querying Multi-granular Compact Representations - Romans Kasperovics and Michael Boehlen Free University of Bozen-Bolzano.

[7] Apache Ant: http://ant.apache.org/

[8] Apache Maven: http://maven.apache.org/

[9] Computing isochrones in multi-modal, schedule-based transport networks (Demo paper) - V. Bauer, J. Gamper, R. Loperfido, S. Profanter, S. Putzer, and I. Timko Free University of Bozen-Bolzano