CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Civil Engineering
Department of Mapping and Cartography

DIPLOMA'S THESIS

IMPLEMENTATION OF THE GUI FOR GNU PROJECT GAMA

Author:	Bc. Jiří Novák
Thesis supervisor:	prof. Ing. Aleš Čepek, CSc.
Date:	January - December 2011

Here should be placed the submission paper!!!

Acknowledgements

I would like to thank all the people who gave me relevant pieces of advice while working on my diploma's thesis.

At the first place it would be prof. Ing. Aleš Čepek, CSc. for his guidance, for his proposals and suggestions, which enriched this paper and for his help with developing the Gama computational facade API.

Last, but not least, I have to thank to my family for their encouragement during the time of my studies and writing of this paper.

Manifestation

I declare that I elaborated this diploma's thesis on my own with the exploitation of the literature mentioned in the Bibliography section.

In Prague, 20th of December 2011

.....
Jiří Novák

Name of the paper:	**Implementation of the GUI for GNU project Gama**
Author:	Bc. Jiří Novák
Study program:	Geodesy and cartography
Branch of study:	Geoinformatics
Thesis supervisor:	prof. Ing. Aleš Čepek, CSc.
Consultant:	-
Abstract:	Implementation of the graphical user interface (GUI) for the GNU project Gama dedicated to the adjustment of the local geodetic networks. Implementation is based on the general SQL database scheme for storing points and clusters of measurement. System is written in C++ with the exploitation of Qt libraries and focus to the portability issues (Windows, Linux) and internationalization. It consists of the dialogs for editing local network measurement configurations, graphical network overview and several output formats of the adjustment results (XML, TXT, HTML). GUI is significantly modular, on the basis of proper plugin framework providing flexibility in the further application development.
Keywords:	*GNU, Gama, geodesy, networks, adjustment, C++, Qt, XML, XSLT, design patterns, plugin framework, MVC, SQL, threads*

Název diplomové práce:	**Implementace GUI systému GNU Gama**
Diplomant:	Jiří Novák
Studijní program:	Geodézie a kartografie
Studijní obor:	Geoinformatika
Vedoucí diplomové práce:	prof. Ing. Aleš Čepek, CSc.
Konzultant:	-
Abstrakt:	Implementace grafického uživatelského rozhraní (GUI) pro systém vyrovnání lokálních geodetických sítí GNU Gama. Implementace vychází z obecného SQL databázového schématu pro ukládání bodů a skupin měření. Systém je naprogramován v C++ s využitím Qt knihoven a důrazem na přenositelnost (Windows, Linux) a vícejazyčnost. Přináší editační dialogy konfigurací měření v lokální síti, grafický náhled na síť a výsledky vyrovnání v několika výstupních formátech (XML, TXT, HTML). GUI je do značného stupně modulární, základem je vlastní pluginovací framework, poskytující flexibilitu při dalším vývoji aplikace.
Klíčová slova:	*GNU, Gama, geodézie, vyrovnání, lokální sítě, C++, Qt, XML, XSLT, návrhové vzory, pluginovací framework, MVC, SQL, vlákna*

Contents

List of Figures

Prologue

Although being in the era of global navigation satellite systems, the adjustment of local geodetic networks remains to be a base for geodesy and its related engineering disciplines. *GNU Gama* is one of few software solutions concerned to this problematic allowing us to adjust very general sets of measurement. It comes with the idea of *clusters*, groups of measurement with a common variance-covariance matrix. This together with the support for wide range of observation types and a possibility to choose the algorithm of numerical solution allows us to use *Gama* for many non-standard applications.

Despite of the benefits listed above, it has also several imperfections. The biggest one for sure is the missing graphical user interface accompanied with the requirement to manual creation of the input XML file and complicated support for Windows platform.

During the last year and half, *Gama* was under an intensive development during which several new features were added. A breakthrough was the introduction of SQL schmema for storing the input data and the incorporation for *SQLite* database as an alternative data input for the classical XML approach. The file-based project management idea implemented in my bachelor thesis was thus abandoned and a decision to make *QGama* a powerful database browser was adopted.

This diploma thesis is dedicated to the implementation of such a graphical interface above the *GNU Gama*'s SQL schema.

The author tried to get use of what he had learned during the 6 month scholarship on *Facultad de Ingeniería, Universidad de Buenos Aires, Argentina* where he attended several courses dedicated to the architecture of software, design patterns, methods of software development and relational database theory and make thus the developed application as much professional-looking as possible. Another aspect he was trying to fulfil is to write it easily extensible so that another students can also easily contribute to the development.

The paper is separated logically into several chapters. The author starts by an overview of what *GNU Gama* offers and which structures it uses for storing the observation data. He also provides a description of its XML input schema and the process of porting it into SQL followed by a detailed description of all the containing tables, attributes and constraints.

Second chapter is dedicated to the design and implementation issues of the application. It is written in the form of a developers guide. It explains step by step the structure of the project, how to compile it from source, what architecture the software uses. Every part is explained first theoretically followed by a practical example of the implementation adopted and a description of problems the author had to face and overcome. The theory of dynamic libraries, logging and plugins framework and creation of graphic scenes in the Qt / C++ environment is also covered.

The appendix then contains a user guide with some practical tutorials of the most common program usage with several screenshots included.

Chapter 1

GNU Gama

GNU Gama is a C++ computational library dedicated to the adjustment of geodetic networks (*Gama* is an acronym for words *Geodesy and Mapping*). It enables both:

- adjustment in *local Cartesian coordinate systems* (*gama-local*) and
- adjustment in *global coordinate systems* (*gama-g3*).

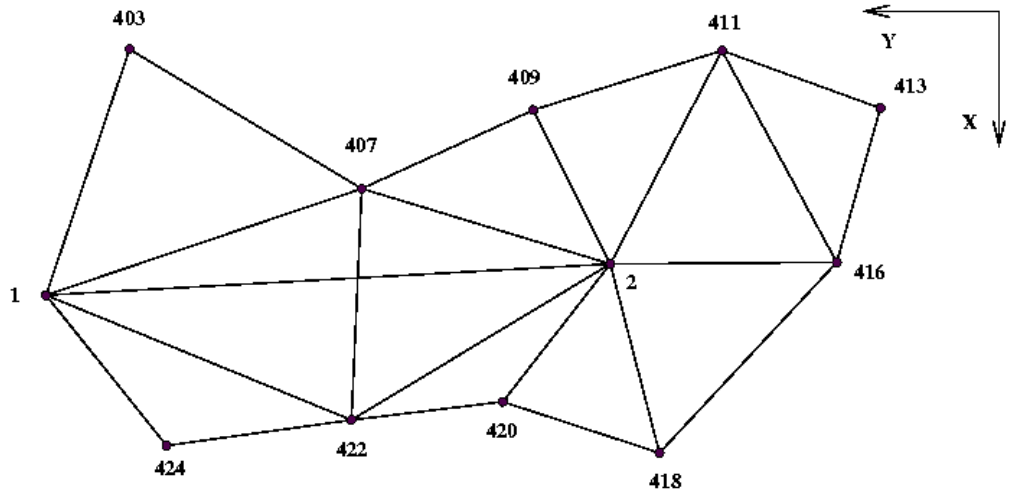

Figure 1.1: Example of local network configuration. [12]

The project emerged back in the year 1998 under the leadership of *Aleš Čepek*. He is the project maintainer as well as the main developer. Among the other contributors who enriched the project within the years we should not forget to name at least our faculty's (ex)students: *Jiří Veselý*, *Petr Doubrava*, *Jan Pytel* or *Václav Petráš*. The complete list of all project contributors is provided within the documentation.

GNU Gama provides both the "classical" adjustment and a lot of supplementary analyses of the adjusted network. One of the features, appreciated by the surveyors all around the world, is the possibility of choosing between various numerical algorithms to be applied for the matrix inversion. Thus, when one approach turns out to be numerically unstable

(because of some uncommon network configuration), the user has the option to compute using a different technique. Currently *GNU Gama* supports 4 different algorithms:

- Singular Value Decomposition (SVD) – the default,
- Gram-Schmidt Orthogonalization block matrix algorithm (GSO),
- Cholesky decomposition of semi-definite matrix of normal equations and
- Cholesky decomposition with the *envelope* reduction of the sparse matrix.

Project equations are solved directly without forming *normal equations* in the case of the first two algorithms.

Furthermore *Gama* is able to compute with 7 different types of observations [1] :

- horizontal directions,
- horizontal distances,
- horizontal angles,
- slope distances,
- zenith angles,
- height differences (levelling networks),
- observed coordinates (i.e. coordinates with given variance-covariance matrix) and
- observed coordinate differences (vectors).

Gama is a part of GNU free software [2] and is hosted on their official servers:

`http://www.gnu.org/s/gama/`

Latest source codes can be downloaded from an anonymous read-only Git [3] repository (current stable version at the time of writing this thesis was **1.11a**):

```
git clone git://git.sv.gnu.org/gama.git
```

A collection of sample networks in the XML input format is available at the same location and could serve as a good starting point for experimenting with *GNU Gama*. It can be cloned with the following command:

```
git clone git://git.sv.gnu.org/gama/examples.git
```

Although it was mentioned that *Gama* supported also adjustment in the global coordinate systems, this is not covered in the thesis, because there still does not exist a complete stable version of `gama-g3` library. The text describes only the `gama-local` part of the library.

[1] Bc. Václav Petráš is supposed to incorporate new ones as a part of his diploma thesis.

[2] The GNU Operating System Homepage: `http://www.gnu.org/`

[3] Good starting point while beginning with Git is the community documentation: `http://book.git-scm.com/`

1.1 New features in GNU Gama

In bachelor's thesis[13] (June 2010) an initial design of the GUI fundamentals was laid. Nevertheless from that time, dozens of important changes took place in the underlying `GNU Gama`'s computational library.

Let us look at them briefly. Two versions were released: **1.10** and **1.11a** [4] . Those included [5] :

1. Numerous bug fixes.
2. Several optimizations.
3. Incorporation of Spanish translation files (provided by Jokin Zurutuza).
4. Redefinition of the tree structure of exception classes to have a common base class and virtual methods `clone()` and `raise()`. This was necessary to allow the incorporation of the *SQLite* database [6] support introduced in [14].
5. Optional support for *SQLite* database as the *gama-local*'s data input.
6. First draft of string functions `Utf8::length()` and `Utf8::leftPad()` – intended to provide aligned well-formed text output also for non-latin languages.
7. Code refactorization (as a preparation for adding new observation types).

For the development of the graphical user interface the previously mentioned changes meant that the former system of XML based project was completely abandoned and a new strategy of focusing fully on the SQL approach was adopted.

1.2 Gama observation data structures

Before the newly introduced `gama-local` SQL scheme will be discussed, we have to recapitulate how *Gama* stores its observation data internally. The data structures are very general, *designed to enable adjustment of any combination of possibly correlated observations (like angles derived from observed directions or already adjusted coordinates from a previous adjustment). To achieve that, it uses the concept of* `clusters`*. Cluster is an object with a common variance-covariance matrix and a list of pointers to observation objects (distances, directions, angles, etc.)*[12].

All clusters are on the same time joined in a common object `ObservationData`. The reciprocal relations are also present: observation objects have a pointer to the cluster to which the observation belongs and each cluster contains a pointer to its parent `ObservationData` object.

[4] Current development version is marked *1.11b*.

[5] Full list of the newly introduced features is provided in the *ChangeLog* file in the official Gama's git repository – `http://git.savannah.gnu.org/cgit/gama.git/tree/ChangeLog`

[6] SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine implemented in C. Homepage: `http://www.sqlite.org/`.

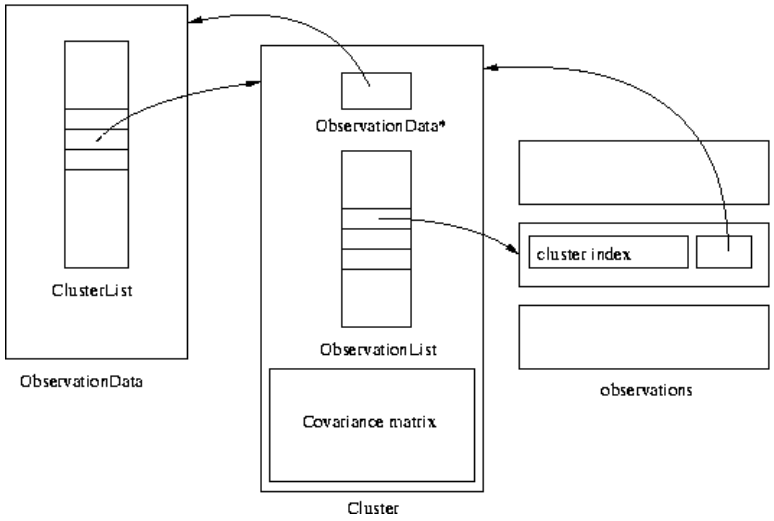

Figure 1.2: Gama observation data structures [12].

1.3 XML schema

`Gama-local`'s input XML corresponds to the internal data structures. [7]

- `<gama-local>` is the enclosing root tag (required) – contains `<description>`, `<parameters>` and `<points-observations>` tags.
- `<description>` pair tag contains the textual description of the network – required.
- `<parameters>` pair tag contains network parameters (optional – defaults took when not specified).
- `<points-observations>` pair tag contains all of the observations made (and its variance-covariance matrices if known) – required.
  - Following sections can appear repeatedly several times or does not have to appear at all (all, but `<point/>` are pair tags which could be composed by a single category-specific observation type tag, optionally followed by a pair `<cov-mat>` tag if the observations are correlated).
  - `<point/>` single tags containing the points entering the adjustment.
  - `<coordinates>` pair tag containing single `<point/>` tags.
  - `<obs>` pair tag containing single `<z-angle/>`, `<s-distance/>`, `<distance/>`, `<direction/>`, `<angle/>` tags.
  - `<vectors>` pair tag containing single `<vec/>` tags.
  - `<height-differences>` pair tag containing single `<dh/>` tags.

For better illustration, the same idea is subsequently expressed in the XML pseudo-code.

[7] Full XML schema definitions of the `gama-local` input and output XML are available within the source code of *QGama* application (`src/libs/qgama/resources/xml`).

```
<gama-local>
  <network>
    <description> ... </description>
    <parameters ... />
    <points-observations>
   <point .../>
   <coodinates>
      <point .../>
      <cov-mat ...>
      ...
      </cov-mat>
   </coordinates>
   <obs ...>
      <z-angle .../> | <distance .../> | <direction .../> | <s-distance> | <
          angle>
        <cov-mat ...>
      ...
      </cov-mat>
   </obs>
   <heigh-differences>
      <dh .../>
      <cov-mat ...>
      ...
      </cov-mat>
   </height-differences>
   <vectors>
      <vec .../>
        <cov-mat ...>
      ...
      </cov-mat>
   </vectors>
     </points-observations>
  </network>
</gama-local>
```

1.4 SQL schema

`Gama-local` SQL shema was designed to be as simple as possible so that it would work with a wide range of database engines [8] . It avoids using non-standard data types and any other advanced SQL features. [9] It consist of 8 tables corresponding logically to the structure of the formerly described XML schema. [10] The mapping which has been done while porting from XML tags to SQL schema followed this strategy.

- Network parameters were stored in the **gnu_gama_local_configurations** table.
- Description text was stored in a separate table **gnu_gama_local_descriptions** – this was to achieve better portability between database engines. It assumed the

[8] So far this scheme was successfully tested against *SQLite3*, *PostgreSQL 8.4*, *Oracle 10g XE* and *MySQL 5.1.*.

[9] From the same reason, the creation of a separate database scheme (not supported by *SQLite3*) was replaced by the policy of using very long table identifiers.

[10] Complete list of `DDL`s statements of the *GNU Gama*'s SQL schema are available in the annex B.

texts being cut into 1000 characters long pieces to avoid using CLOBs [11] which could not be present in some of the vendors implementations.

- Points entering the adjustment were stored in the **gnu_gama_local_points** table.
- Variance-covariance matrices were stored in the **gnu_gama_local_covmat** table.
- Relations between configurations – clusters and covariance matrices were stored in the **gnu_gama_local_clusters** table.
- Individual observations were stored in the separate tables: **gnu_gama_local_obs** (`distance`, `direction`, `s-distance`, `angle`, `z-angle` and `dh` observation types), **gnu_gama_local_vectors** (`vec` observation type) and **gnu_gama_local_coordinates** (`point` observation type).

The final mapping to the relational database schema thus would looked like this:

XML tag(s)	SQL schema table
`<parameters>`	`gnu_gama_local_configurations`
`<description>`	`gnu_gama_local_descriptions`
`<points-observations>` children of type `<point/>`	`gnu_gama_local_points`
`<points-observations>` children of the types `<obs>`, `<vectors>`, `<coordinates>` `<height-differences>`	`gnu_gama_local_clusters`
`<cov-mat>`	`gnu_gama_local_covmat`
`<z-angle/>`, `<distance/>` `<direction/>`, `<angle/>` `<s-distance/>`	`gnu_gama_local_obs`
`<vec/>`	`gnu_gama_local_vectors`
`<point/>`	`gnu_gama_local_coordinates`

1.4.1 Units in SQL tables

- Distances are stored in `meters`, its standard deviations in `millimetres`.
- Angular values as well as its standard deviations are stored in `radians` and converted to `gons` or `degrees` if needed using the following formula:

$$\text{rad} = \text{gon} \cdot \frac{\pi}{200} = \text{deg} \cdot \frac{\pi}{180}$$

.

1.4.2 Table `Configurations`

This table contains both, all the network parameters specified within the XML `<parameters/>` tag as well as some of the command-line parameters of `gama-local` console utility.

[11] Character Large Objects: `http://en.wikipedia.org/wiki/Character_large_object`.

Primary key: (`conf_id`)
Foreign keys: -

Column	Description
`conf_id`	Unique network (configuration) identifier within the database.
`conf_name`	Unique configuration name within the database.
`sigma_apr`	Value of the *a priori* reference standard deviation (square root of the reference variance).
`conf_pr`	Confidence probability used in the statistical tests.
`tol_abs`	Tolerance for the identification of gross absolute terms in the project equations.
`sigma_act`	Actual type of the reference standard deviation used in the statistical tests.
`update_cc`	Defines if coordinates of constrained points should be updated in the iterative adjustment. *If test on linearization fails, Gama tries to improve approximate coordinates of adjusted points and repeats the whole adjustment.*[12]
`axes_xy`	Orientation of axes `x` and `y`. Value `ne` means that axis `x` is oriented to the north and axis `y` to the east. For `left-handed` coordinate systems `ne, sw, es` or `wn` are acceptable and for `right-handed` `en, nw, se` or `ws` are acceptable.
`angles`	Defines whether observed angles and / or directions are measured in a counter-clockwise (`righ-handed`) or clockwise (`left-handed`) manner.
`epoch`	Epoch of the measurement (preparation for the adjustment and analysis of deformations).
`algorithm`	Specifies the numerical method used for the solution of the adjustment. Implicitly *Singular Value Decomposition* (`svd`) is used, but you can opt between the *František Charamaza*'s block matrix algorithm `GSO` based on the *Gram Schmidt Orthogonalization*, *Cholesky decomposition of the semi-definite matrix of normal equations* (`cholesky`) and *Cholesky decomposition with the envelope reduction of the sparse matrix* (`envelope`).
`ang_units`	Angular units to be used in the `gama-local`'s adjustment output.
`latitude`	Mean latitude of the network area.
`ellipsoid`	Name of the ellipsoid. Complete list of the supported ellipsoids is available in the *Gama*'s manual.[12]

[12] `http://www.gnu.org/s/gama/manual/gama.html#Supported-ellipsoids`

Column	Type	Nullable	Default	Constraints
conf_id	integer	N	-	-
conf_name	varchar(60)	N	-	unique
sigma_apr	double	N	10.0	check > 0
conf_pr	double	N	0.95	check > 0 and < 1
tol_abs	double	N	1000	check > 0
sigma_act	varchar(11)	N	'aposteriori'	check in ('apriori', 'aposteriori')
update_cc	varchar(3)	N	'no'	check in ('yes', 'no')
axes_xy	varchar(2)	N	'ne'	check in ('ne', 'sw', 'es', 'wn', 'en', 'nw', 'se', 'ws')
epoch	double	N	0.0	-
algorithm	varchar(12)	N	'svd'	check in ('svd', 'gso', 'cholesky', 'envelope')
ang_units	integer	N	400	check in (400, 360)
latitude	double	N	50.0	-
ellipsoid	varchar(20)	Y	-	-

1.4.3 Table Descriptions

As it was already mentioned, network description is held separately from the gnu_gama_local_configurations table, because of the effort of not to rely on the character large objects, which some of the database engines do not support. Every description is thus cut into 1000 characters long parts and concatenated while being read.

Primary key: (conf_id, indx)
Foreign keys: (conf_id)
references gnu_gama_local_configurations (conf_id)
ON DELETE CASCADE

Column	Description
conf_id	Id of the configuration which the description (text) belongs to.
indx	Sequence number of the text chunk within one configuration's description.
text	1000 characters of the configuration's description.

Column	Type	Nullable	Default	Constraints
conf_id	integer	N	-	-
indx	integer	N	-	check ≥ 1
text	varchar(1000)	N	-	-

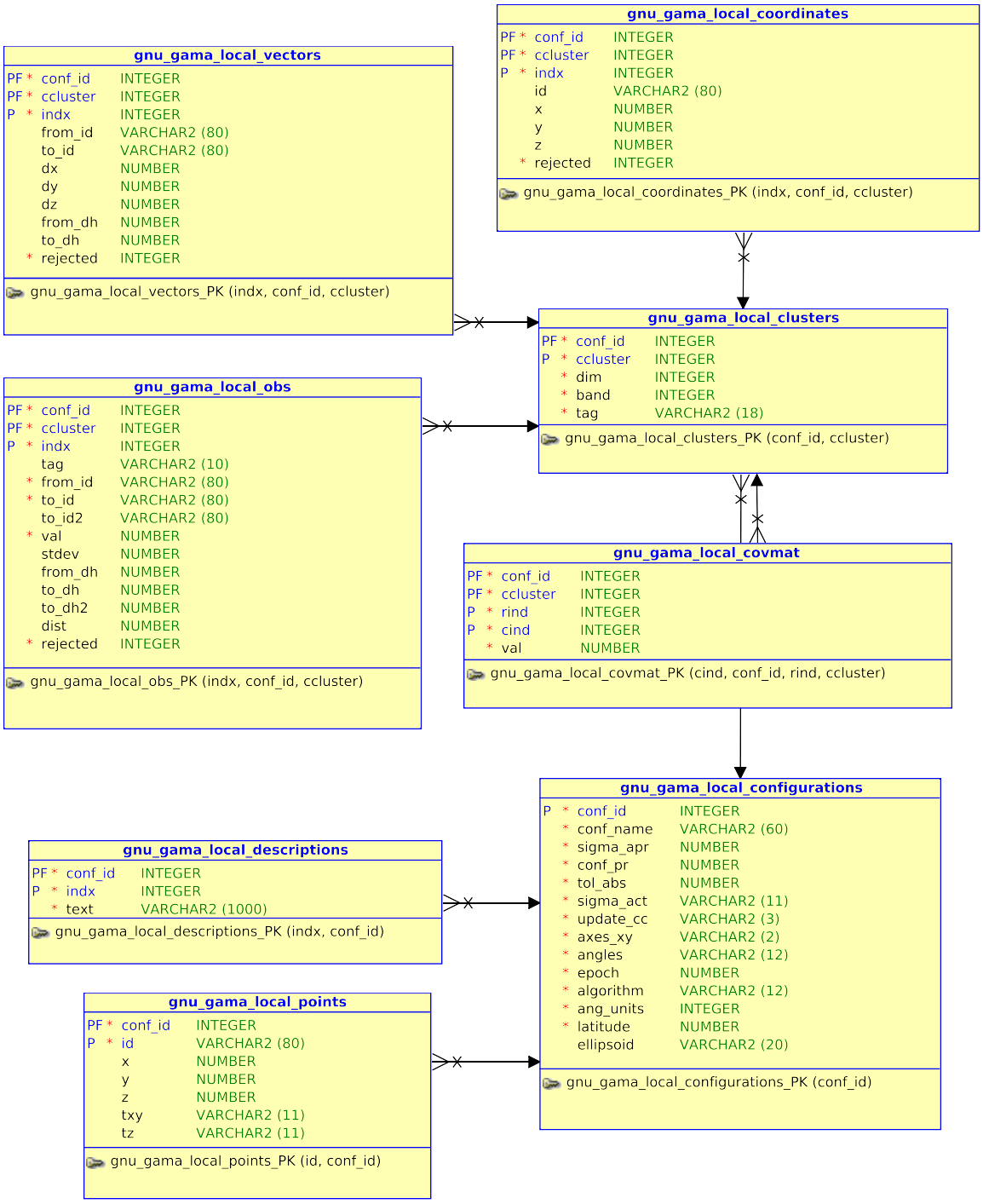

Figure 1.3: *GNU Gama*'s relational model diagram (created using *Oracle SQL Data Modeller*). P stands for *primary key*, F for *foreign key*.

1.4.4 Table `Points`

Points entering the adjustment could have its type specified as:

fixed Point coordinates are not changed in the adjustment.

adjusted Point coordinates are going to be adjusted (unknown parameters in the adjustment).

constrained Point coordinates are used for the regularization of free networks. *If the network is not free (fixed network), the constrained coordinates are interpreted as other unknown parameters. In classical free networks, constrained points define the regularization constraint:* $\sum dx_i^2 + dy_i^2 = \min$, *where dx and dy are adjusted coordinate corrections and the summation index i goes over all constrained points. In other words, the set of the constrained points defines the adjustment of the free network (its shape and size) with a simultaneous transformation to the approximate coordinates of selected points.*[12]

If no type is specified, the point will be interpreted as *adjusted* (unknown parameter).

Primary key: (`conf_id, id`)
Foreign keys: (`conf_id`)
references `gnu_gama_local_configurations` (`conf_id`)
`ON DELETE CASCADE`

Column	Description
`conf_id`	Id of the configuration which the point belongs to.
`id`	Point identification - all the printable characters can be used.
`x`	Coordinate X.
`y`	Coordinate Y.
`z`	Coordinate Z.
`txy`	Marks whether the XY coordinates should be `fixed`, `constrained` or `adjusted` in the adjustment.
`tz`	Marks whether the Z coordinate should be `fixed`, `constrained` or `adjusted` in the adjustment.

Column	Type	Nullable	Default	Constraints
`conf_id`	integer	N	-	-
`id`	varchar(80)	N	-	-
`x`	double	Y	-	-
`y`	double	Y	-	-
`z`	double	Y	-	-
`txy`	varchar(11)	Y	-	check in (‘`adjusted`’, ‘`fixed`’, ‘`constrained`’)
`tz`	varchar(11)	Y	-	check in (‘`adjusted`’, ‘`fixed`’, ‘`constrained`’)

1.4.5 Table `Clusters`

Cluster is a group of observations with common covariance matrix. The covariance matrix allows to express any combination of correlations among the observations in the cluster (including uncorrelated observations, where covariance matrix is diagonal).

In the database the observations are stored in three tables depending on their type:

- `gnu_gama_local_obs` (tags `<obs>` and `<height-differences>`),
- `gnu_gama_local_coordinates` (tag `<coordinates>`),
- `gnu_gama_local_vectors` (tag `<vectors>`).

Cluster's variance-covariance matrix is stored in the `gnu_gama_local_covmat` table. **Every observation of any supported type has to be in some cluster!**

Primary key: (`conf_id, ccluster`)
Foreign keys: (`conf_id`)
references `gnu_gama_local_configurations` (`conf_id`)
`ON DELETE CASCADE`

Column	Description
`conf_id`	Id of the configuration which the cluster belongs to.
`ccluster`	Sequence number of the cluster within the configuration.
`dim`	Dimension of the covariance matrix.
`band`	Bandwidth of the covariance matrix (fully-populated covariance matrix has a bandwidth of `dim-1` and diagonal matrix of `0`)
`tag`	Tag attribute specifies the type of observations in the cluster. It also implies the table where it will be physically stored.

Column	Type	Nullable	Default	Constraints
`conf_id`	integer	N	-	-
`ccluster`	integer	N	-	check > 0
`dim`	integer	N	-	check > 0
`tag`	varchar(18)	N	-	check in (`'obs'`, `'coordinates'`, `'vectors'`, `'heigh-differences'`)

1.4.6 Table `Covmat`

`Covmat` table contains individuals cluster variance-covariance matrices. Attributes (`conf_id, ccluster`) identify the specific matrix, (`rind, cind`) identify the position of the field in the matrix [13] and `value`, the corresponding variance or covariance. Missing record in the matrix is interpreted as 0. Attributes `rind` and `cind` have to respect the corresponding matrix dimension and bandwidth.

Primary key: (`conf_id, ccluster, rind, cind`)
Foreign keys: (`conf_id, ccluster`)
references `gnu_gama_local_clusters` (`conf_id, ccluster`)
`ON DELETE CASCADE`

Column	Description
`conf_id`	Id of the configuration which the variance-covariance matrix belongs to.
`ccluster`	Id of the cluster which the variance-covariance matrix belongs to.
`rind`	Row number of the variance-covariance matrix.
`cind`	Column number of the variance-covariance matrix.
`val`	Concrete variance or covariance.

Column	Type	Nullable	Default	Constraints
`conf_id`	integer	N	-	-
`ccluster`	integer	N	-	-
`rind`	integer	N	-	check > 0
`cind`	integer	N	-	check > 0
`val`	double	N	-	-

[13] Of course it would be perfectly possible to use an unidimensional index instead. The two indexes approach was opted with regard to better user's orientation.

1.4.7 Table Obs

Table gnu_gama_local_obs contains observations of type:

- horizontal distance (tag <distance/>),
- horizontal direction (tag <direction/>),
- horizontal angle (tag <angle/>),
- slope distance (tag <s-distance/>),
- zenith angle (tag <z-angle/>),
- levelling height differences (tag <dh/>).

Primary key: (conf_id, ccluster, indx)
Foreign keys: (conf_id, ccluster)
references gnu_gama_local_clusters (conf_id, ccluster)
ON DELETE CASCADE

Column	Description
conf_id	Id of the configuration which the observation belongs to.
ccluster	Id of the cluster which the observation belongs to.
indx	Sequential number of the observation within the cluster.
tag	Type of the observation.
from_id	Id of the standpoint. Must not differ within the cluster if observation type is 'direction'.
to_id	Id of the target.
to_id2	Id of the second target. Must be present if the observation type is 'angle'.
val	Observed value.
stdev	Value of the standard deviation. Just for backward compatibility, not used in *QGama* application.
from_dh	Standpoint's height.
to_dh	Target's height.
to_dh2	Second target's height.
dist	Distance of the levelling section. Must be present if the observation type is 'dh'.
rejected	Specifies whether the observation is rejected (1) or not (0).

Column	Type	Nullable	Default	Constraints
conf_id	integer	N	-	-
ccluster	integer	N	-	check > 0
indx	integer	N	-	check > 0
tag	varchar(10)	Y	-	check in ('direction', 'distance', 'angle', 's-distance', 'z-angle', 'dh')
from_id	varchar(80)	N	-	-
to_id	varchar(80)	N	-	-
to_id2	varchar(80)	Y	-	check (tag = 'angle' and to_id2 is not null)
val	double	N	-	-
stdev	double	Y	-	-
from_dh	double	Y	-	-
to_dh	double	Y	-	-
to_dh2	double	Y	-	-
dist	double	Y	-	check (tag = 'dh' and dist is not null)
rejected	integer	N	0	-

1.4.8 Table `Vectors`

Table `gnu_gama_local_vectors` contains coordinate differences (vectors).

Primary key: (`conf_id, ccluster, indx`)
Foreign keys: (`conf_id, ccluster`)
references `gnu_gama_local_clusters` (`conf_id, ccluster`)
`ON DELETE CASCADE`

Column	Description
`conf_id`	Id of the configuration which the vector belongs to.
`ccluster`	Id of the cluster which the vector belongs to.
`indx`	Sequence number of the vector within the cluster.
`from_id`	Id of the standpoint.
`to_id`	Id of the target.
`dx`	Coordinate difference in X.
`dy`	Coordinate difference in Y.
`dz`	Coordinate difference in Z.
`from_dh`	Standpoint's height.
`from_dh`	Target's height.
`rejected`	Specifies whether the observation is rejected (`1`) or not (`0`).

Column	Type	Nullable	Default	Constraints
`conf_id`	integer	N	-	-
`ccluster`	integer	N	-	check > 0
`indx`	integer	N	-	check > 0
`from_id`	varchar(80)	Y	-	-
`to_id`	varchar(80)	Y	-	-
`dx`	double	Y	-	-
`dy`	double	Y	-	-
`dz`	double	Y	-	-
`from_dh`	double	Y	-	-
`to_dh`	double	Y	-	-
`rejected`	integer	N	`0`	-

1.4.9 Table `Coordinates`

Table `gnu_gama_local_coordinates` contains control (known) coordinates which enter the adjustment.

Primary key: (`conf_id, ccluster, indx`)
Foreign keys: (`conf_id, ccluster`)
references `gnu_gama_local_clusters` (`conf_id, ccluster`)
`ON DELETE CASCADE`

Column	Description
`conf_id`	Id of the configuration which the coordinate belongs to.
`ccluster`	Id of the cluster which the coordinate belongs to.
`indx`	Sequence number of the coordinate within the cluster.
`id`	Point identification - all the printable characters can be used.
`x`	Coordinate X.
`y`	Coordinate Y.
`z`	Coordinate Z.
`rejected`	Specifies whether the observation is rejected (`1`) or not (`0`).

Column	Type	Nullable	Default	Constraints
`conf_id`	integer	N	-	-
`ccluster`	integer	N	-	check > 0
`indx`	integer	N	-	check > 0
`id`	varchar(80)	Y	-	-
`x`	double	Y	-	-
`y`	double	Y	-	-
`z`	double	Y	-	-
`rejected`	integer	N	`0`	-

1.4.10 Implementation issues

While implementing *QGama* application and testing the database schema it have been found that in some of the database engines the foreign key support was not enabled by default (e.g. the `ON DELETE CASCADE` clause was ignored). Author did not want to get rid of its benefits and therefore decided to handle its enforcement manually. That meant to call `PRAGMA foreign_keys = ON` for *SQLite* and add the `" engine=innodb"` clause in the end of the `CREATE TABLE ()` statements for *MySQL*.

Chapter 2

QGama 1.0.0 developer's guide

QGama is an easily scalable framework providing graphical user interface (GUI) and several other features to the *GNU Gama* adjustment library. It is written in C++ and *Qt framework* from the *Nokia* corporation. *QGama* is a skeleton that is extended with plugins. It works above the underlying relational database shema of *GNU Gama* (described in 1.4), providing configuration editing dialogs and a simple network overview. Project is prepared to be internationalized.

Source codes are hosted on the internal *Git* of *Department of Mapping and Cartography* and could be retrieved by executing the following command:

```
git clone git://geo102.fsv.cvut.cz/qgama.git
```

Project also has an *issue tracker* and *wiki* hosted on the same server:

`http://geo102.fsv.cvut.cz/trac/qgama`.

Doxygen documentation can be found at:

`http://josef.fsv.cvut.cz/~novakj62/qgama/doc/html/`.

2.1 Coding conventions

Throughout the source code author tried to adhere to the following conventions:

- Every *class* is defined in its separate header (`.h`) and class (`.cpp`) file.
- *Member variables* (attributes) are prefixed with `m_`, *class* (`static`) *variables* with `s_`.
- *Interfaces* (non-instantiable *abstract classes* with the presence of some *pure virtual methods*) are prefixed with `I`.
- Descendants of *abstract classes* mentioned above hold suffix `Impl`.
- Each *library / plugin* is defined in separate namespace.
- *CamelCase* naming convention was used for identifiers.
- When using pointer parameters, always test them with `Q_ASSERT()` macro to ensure non-null value before calling a method on them. [1]
- When a sub-project defines some constants, they are gathered into the `constants.h` file.
- Unified marking of problematic parts of code (`// FIXME:`) and markings for future improvements (`// TODO:`).

2.2 Project structure

QGama has the following directory structure:

`dist` Version *ChangeLogs* and a list of *known issues*.

`help` Online help pages (in HTML format), that are called from the dialogs within the GUI.

`src` Application source codes.

`src/app` Source code of the `main()` function.

`src/libs` Source code of the:

Log4Qt 3dparty shared library (Qt port of the famous `Log4j` by the *Apache Foundation*).[2]

Scripts Several utilities making possible to build *Gama* as a shared library (will be discussed in more detail in the *Compilation* section).

Gama Computational library for the adjustment of geodetic networks, compiled shared, with the *Gama Adjustment API* and several helper functions as its *facade*.

[1] This behaviour can be suppressed in production by defining `DEFINES += Q_NASSERT` symbol in the project file. Preprocessor will thus expand any `Q_ASSERT` macro invocations in an empty expression.

[2] *Log4j* homepage: `http://logging.apache.org/log4j/1.2/`

QGama shared library (containing the plugin extension system, thread-safe global object pool, persistent settings and several other utilities).

`src/plugins` Source code of the:

CorePlugin GUI main window, dialogs and infrastructure for establishing and persisting database connection, creating *GNU Gama* SQL schema, definition and filling mechanisms of the data models, adjustment and conversions worker threads, etc.

SQLEditor All the rest of editing dialogs and widgets.

NetworkOverview The network overview graphics/view classes.

`tests` Unit tests of the application. Shares the same directory structure as the `src` folder.

`translations` Application translation files.

Other important files

`Doxyfile`	Configuration file for *Doxygen*.[3]
`LICENSE.GPL`	Text of the GNU GPL v3 license under which the project is distributed.
`README.TXT`	Brief compilation instructions.
`TODO.txt`	List of the features to be implemented in the future releases.
`qgama.pri`	File with some helper functions used within the sub-projects. Also defines output and include paths.
`qgama.pro`	Main project file (ensures Qt version at least 4.7.3, starts compilation of the `src` and `test` folder).
`src/src.pro`	Wrapper for compiling `libs`, `app` and `plugins` directory.
`src/qgamalibrary.pri`	File with some basic settings valid for every *shared library* within the project.
`src/qgamaplugin.pri`	File with some basic settings valid for every plugin within the project.
`src/rpath.pri`	Settings of the runtime library search path of the particular program being compiled on the Unix platform (`ld`'s feature).[4]

[3] *Doxygen*'s homepage: `http://www.stack.nl/~dimitri/doxygen/`

[4] `Ld`'s manual: `http://linux.about.com/library/cmd/blcmdl1_ld.htm`

2.3 Compiling from source

2.3.1 Git installation

QGama project uses *Git Version Control System* for managing its source codes. Thus when a developer wants to compile *QGama* from sources, there is the need to install corresponding packages first. Their names might differ slightly, for *Debian*-based systems, the following command will install *Git* together with *Gitk* – a simple GUI wrapper:

```
sudo apt-get install git gitk
```

On Windows use `msysgit` - the latest release can be downloaded from: `http://code.google.com/p/msysgit/downloads/list`.

2.3.2 Qt SDK installation

Once having the source codes, the developer will need Qt libraries and corresponding compiler (*QGama* was tested so far with `g++` on *Linux* and `mingw` on *Windows*). Author would recommend to use the `Nokia's Qt SDK online installer`, because it:

- brings everything bundled including the *QtCreator IDE*,
- maintains the installation with the package approach, enabling thus to install / uninstall new features easily,
- offers library updates once available.

Qt SDK installer can be downloaded in the online and offline version from the following URL: `http://qt.nokia.com/downloads`.

2.3.3 Initialization of the git sub-modules

As it was already mentioned *QGama* depends on two third-party libraries: *Gama* and *Log4Qt* – both of them have its own separate repositories.

Git sub-modules is a way to define a dependency on those projects without including its source codes. The advantage is that the source code is not duplicated and that the user whenever compiling will get the latest version of the dependent projects.

Sub-modules are defined in the `.gitmodules` file in the project's root directory. From the following listing it can be seen the author declared two dependencies: *GNU Gama* project cloned from the official GNU repository into the `src/libs/3dparty/gama/gama` subdirectory and *Log4Qt* fork from `Gitorious.org` into the `src/libs/3dparty/log4qt/loq4qt` subdirectory.

```
$ cat .gitmodules
[submodule "src/libs/3dparty/gama/gama"]
  path = src/libs/3dparty/gama/gama
  url = git://git.sv.gnu.org/gama.git
[submodule "src/libs/3dparty/log4qt/log4qt"]
  path = src/libs/3dparty/log4qt/log4qt
  url = git://gitorious.org/log4qt/log4qt.git
```

Before proceeding to the compilation of *QGama* project, sub-modules have to initialized and updated explicitly:

```
$ git submodule init
Submodule 'src/libs/3dparty/gama/gama' (git://git.sv.gnu.org/gama.git) registered
    \
  for path 'src/libs/3dparty/gama/gama'
Submodule 'src/libs/3dparty/log4qt/log4qt' (git://gitorious.org/log4qt/log4qt.git
   ) \
  registered for path 'src/libs/3dparty/log4qt/log4qt'

$ git submodule update
Cloning into src/libs/3dparty/gama/gama...
remote: Counting objects: 9627, done.
remote: Compressing objects: 100\% (2023/2023), done.
remote: Total 9627 (delta 7421), reused 9627 (delta 7421)
Receiving objects: 100\% (9627/9627), 2.14 MiB | 369 KiB/s, done.
Resolving deltas: 100\% (7421/7421), done.
Submodule path 'src/libs/3dparty/gama/gama': checked out \
  'cb24f7d8031b3a41388a366b46f0ade5062009a1'
Cloning into src/libs/3dparty/log4qt/log4qt...
remote: Counting objects: 351, done.
remote: Compressing objects: 100% (270/270), done.
remote: Total 351 (delta 216), reused 82 (delta 47)
Receiving objects: 100% (351/351), 196.45 KiB, done.
Resolving deltas: 100% (216/216), done.
Submodule path 'src/libs/3dparty/log4qt/log4qt': checked out \
  'd0abc2d3011c54a5ff1d7fea96198525cab6dbb8'
```

2.3.4 Compilation

`Makefile` should be generated from `qgama.pro` in the project root and run [5] (or the project could be opened in *Qt Creator IDE* and the build button pressed). Project can be also built in a separate directory (so called shadow build).

Compilation will proceed as follows:

1. **version** script from `src/libs/3dparty/scripts/version` will be compiled, linked and executed.
   - This takes the `src/libs/3dparty/gama/gama/configure.ac` GNU Autotools file, extracts `GNU Gama` version and generates a `src/app/config.h` file including version defines:

```
1 #define VERSION "1.11a"
2 #define GAMA_VERSION "1.11a"
3 #define QGAMA_VERSION "1.0.0"
```

2. **libgama_files** script from `src/libs/3dparty/scripts/libgama_files` will be compiled, linked and executed.
   - This takes the `src/libs/3dparty/gama/gama/lib/Makefile.am` GNU Autotools file, extracts the headers and sources conforming the `GNU Gama` compu-

[5] If the compilation is performed on a multi-core hardware, `"-j<number of cores>"` hint could be used to accelerate the compilation process.

tational library and generates a `src/libs/3dparty/gama/gama_files.pri` – sources and headers listing in the format of `QMake` project include file.

3. **Log4Qt** from `src/libs/3dparty/log4qt` will be compiled as a shared library and placed into the destination directory `<build-dir>/libs/qgama`.

4. **Gama** from `src/libs/3dparty/gama` will be compiled as a shared library (with the Adjustment API and several helper classes as a `facade`) and placed into the destination directory `<build-dir>/libs/qgama`.

5. **QGama** from `src/libs/qgama` will be compiled as a shared library and placed into the destination directory `<build-dir>/libs/qgama`.

6. **QGama application** from `src/app/main.cpp` will be compiled, linked against `Log4Qt, QGama and Gama` and placed into the destination directory `<build-dir>/bin`.

7. **CorePlugin** from `src/plugins/coreplugin` will be compiled as a shared library and placed into the destination directory `<build-dir>/libs/qgama/plugins/cz.ctu.fce.dmc`. [6]

8. **SQLEditor** plugin from `src/plugins/sqleditor` will be compiled as a shared library and placed into the destination directory `<build-dir>/libs/qgama/plugins/cz.ctu.fce.dmc`.

9. **NetworkOverview** plugin from `src/plugins/networkoverview` will be compiled as a shared library and placed into the destination directory `<build-dir>/libs/qgama/plugins/cz.ctu.fce.dmc`.

On Windows platform, shared libraries (excluding plugins) are also copied into the `<build-dir>/bin` directory, because there is no way to set path, where dynamic libraries specific to the binary should be found by the system.

2.4 Architecture overview

As already mentioned QGama consist of two main components: **libraries** and **plugins**.

Looking at the source code of the main application (`src/app/main.cpp`) shows that it only initializes the logging framework and application translators. Next, it tries to load the plugins and then it passes control to the main Qt event loop.

```
1  int main(int argc, char *argv[])
2  {
3      QApplication app(argc, argv);
4
5      setupLog4Qt();
6      setupTranslators(app);
7
8      int result = loadPlugins(app);
9      if (result == CORE_COULD_NOT_START)
10         return 1;
```

[6] `<build-dir>` stands for the directory, where the compilation process takes place, `cz.ctu.fce.dmc` is a plugin's provider identification - in this case: `CZ.CzechTechnicaUniversityInPrague.FacultyOfCivilEngineering.DepartmentOfMappingAndCartography`

```
11
12      // start the event loop
13      m_logger->info(QObject::tr("Starting main window event loop."));
14      return app.exec();
15  }
```

Plugins Framework will be covered in more detail in a separate section 2.6. Here it is enough to say, that the main application will try to load all of the plugins in a specified directory. This comes defined by `SettingsImpl` class and will be discussed in the subsection 2.7.4. If it succeeds at least with the `Core` plugin, it starts the main application's window. If anything goes wrong during the plugins loading process, user will be notified correspondingly.

In the figure 2.1 an overview of *QGama*'s principal components could be seen. The core of the application's common functionality is in the **QGama** library – it has the main access point `ApplicationContext` class with several public static functions.

It provides access to:

- the thread-safe `Global Objects Pool` a kind of very simple `QObject`s application container,
- `ActionsManager` a tool for dynamic generation of the main window menus,
- a convenient method for showing the online help page – `showHelpPage()`,
- application persistent settings map,
- application non-persistent variables map and
- `PluginsManager`.

Besides that, `QGama` library brings also several utilities like custom `ProgressDialog`, `TextEditor`, `HtmlViewer`, etc. *QGama* depends on *Log4Qt* library.

Rest of the functionality is brought by plugins. There are 3 principal plugins so far (`CorePlugin, SQLEditor` and `NetworkOverview`), although others are planned to be implemented soon. The first two are essential for running the application, the third one is optional.

Every plugin depends on *QGama* library. `CorePlugin` additionally depends on *Gama* library, because `CorePlugin` contains the worker threads for network adjustment and output conversions. It also contains all of the necessary dialogs and infrastructure for database access, model definitions, mechanisms of filling them and notifying all of the observers when changed.

`SQLEditor` plugin depends on `CorePlugin` and it brings all of the editing dialogs and widgets for points, clusters and measurements.

`NetworkOverview` plugin depends on `CorePlugin` and it brings the very simple graphical network overview (has to be completed in future release to provide better interaction).

2.5 Logging framework

Almost every class within the *QGama* project uses logging framework. This section introduces some of its concept and describes its basic usage.

Inserting log statements into the source code is a technique which, based on author's opinion and experience, increases code readability and helps a lot in the development process, especially when you are working with threads, whose tracing with debugger could be tricky.

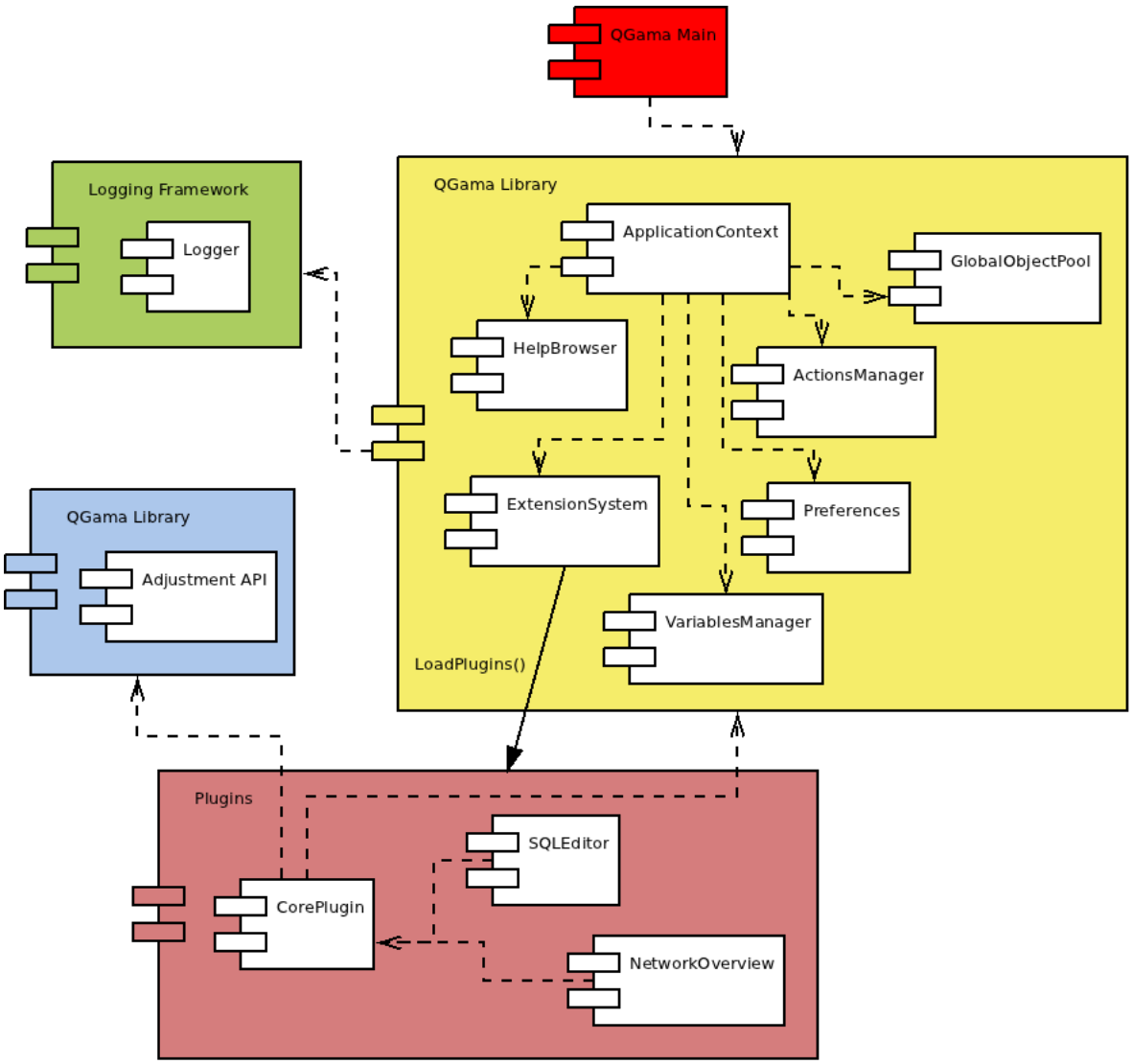

Figure 2.1: Components diagram (designed in Dia).

The traditional approach in Qt would be using the `qDebug(), qWarning(), qFatal()` functions defined in `<QtGlobal>` header. Using that has several drawbacks:

- It cannot be suppressed without recompilation (by specifying `DEFINES += QT_NO_DEBUG_OUTPUT` in the project file).
- The verbosity level of messages cannot be chosen.
- Output is produced only to the console.

With logging framework it is possible to control logging behaviour at runtime or by editing a configuration file, without modifying the application binary. It is designed in a

way that the statements could remain in the code without incurring a heavy performance cost.

Developer is thus equipped with detailed context of what is occurring inside the application. Thanks to the concept of log levels and logger hierarchies (which both will be discussed in more detail following paragraphs / sections), one can also control in great detail which log statements are displayed. For example, testers and developers would use maximum verbosity while production release would display only serious problems.

Moreover, the log output can be easily redirected into a file or other output stream, database, remote logging deamon or send by SMTP. For details documentation should be consulted:

`http://logging.apache.org/log4j/1.2/publications.html`.

2.5.1 Log4j and its ports

Log4j is a Java-based logging utility developed by Apache Software Foundation. Throughout the time it became one of the most popular logging frameworks on the Java platform and de-facto standard. There exists ports to many programming languages including C, C++, PERL, JavaScript, Ruby, PHP, SH etc.

The official `Log4cxx` [7] C++ port could be used, but there would be a problem of portability. Separate compilation of `log4cxx` and `apr` (*Apache Runtime*) libraries with *MSVC* would be needed on the *Windows* platform. Therefore author finally opted for the fully Qt port, which can be compiled and distributed within the rest of the source codes and brings no portability complications.

2.5.2 Log4Qt

`Log4Qt` does not implement all of the original `Log4j` Java package functionalities and some features are implemented using a different approach, nevertheless the basic concepts and usage remain the same. For the detailed list of differences *Log4Qt* documentation available at can be consulted:

`http://log4qt.sourceforge.net/html/index.html`.

2.5.3 Loggers, appenders and layouts

`Log4Qt` has three main components:

loggers define type of the log message,

appenders define destination of the log message,

layouts define format of the log message.

The biggest advantage of logging API over the plain `std::cout` or `qDebug()` is in its ability to disable certain log statements while allowing others to be printed unchanged. This assumes logging space to be categorized based on some developer-chosen criteria. For this purpose serve **loggers**. Loggers have hierarchical structure expressed in a similar way as the Java package naming convention – using the dots as parent-child separator. For example the logger named `QGama` is a parent of a logger named `QGama.Main`, etc.

[7] *Log4Cxx*'s homepage: `http://logging.apache.org/log4cxx/`

There also exist a special root logger, which resides at the top of the logger hierarchy. This logger always exists and cannot be retrieved by name.

For obtaining a logger, there are two conventional static methods:

- `Log4Qt::Logger::logger` and
- `Log4Qt::Logger::rootLogger`.

Each logger has 6 standard **levels** (ordered ascending in their importance): `TRACE, DEBUG, INFO, WARN, ERROR, FATAL`.

The way author use `Log4Qt` in his project is, that he creates a logger instance for each structure which needs logging. Logger is created as a private constant static pointer with the name respecting the fully qualified class name (including the namespace). An example follows.

Header file:

```
...
namespace QGama {

  class QGAMA_EXPORT SettingsImpl : public ISettings {
    ...

    private:
      ...
          /***** LOGGER */
          const static Log4Qt::Logger *s_logger;
  }; // class SettingsImpl

} // namespace QGama
...
```

Source file:

```
...
const Log4Qt::Logger* SettingsImpl::s_logger =
       Log4Qt::Logger::logger("QGama.SettingsImpl");
...
```

Wherever in the code of the class, anything has to be logged, a call to the corresponding member function of the `s_logger` pointer should be invoked. For example:

```
s_logger->debug("Starting adjustment...");
```

Appenders (output destination) define where the log messages will be printed. Among others there are appenders to *console, file, database, system log* or *telnet*. More than one appender can be attached to a logger. There is also an option to log asynchronously (a different thread is then used for the deferred logging).

Layouts enable the programmer to configure precisely the format of the logged messages.

2.5.4 Configuration

Configuration of the whole framework could be done in several ways:

- A separate text file `log4qt.properties` could be used and placed into the same directory as the application binary. Although we are in Qt/C++, the configuration file keeps using the original Java properties file syntax and `Log4j` variables. Both console and file appenders for the root logger are defined there. It means that every message from `INFO` level higher will be printed on console and every message from `WARN` level higher will be appended to file.

```
# Console output
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern = %d{yyyy-MM-dd HH:mm:ss} %c
    {1} [%p] %m%n
log4j.appender.console.Threshold=INFO

# File output
log4j.appender.file = org.apache.log4j.DailyRollingFileAppender
log4j.appender.file.File = /tmp/qgama.log
log4j.appender.file.DatePattern = "'.'yyy-MM-dd"
log4j.appender.file.layout = org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern = %d{yyyy-MM-dd HH:mm:ss} %c{1}
     [%p] %m%n
log4j.appender.file.Threshold=WARN

# Root logger options
log4j.rootLogger = ALL, console, file
```

- Another option specific to *Log4Qt* is to use `QSettings`. *Log4Qt* will automatically scan it for the presence of configuration and initialize itself according to it.

- Configuration could be made also explicitly in the source code, by instantiating corresponding classes.

Within *QGama* there is used the `QSettings` approach with a slight modification. Because of the `SettingsImpl` class is hidden under `ISettings` interface [8] , *Log4Qt* does not detect the presence of configuration automatically and it have to be initialized manually. This is the meaning of `QGama::setupLog4Qt()` function in `src/app/main.cpp`.

However this demonstrates only the basic configuration. More advanced features are described in the manual of *Log4Qt*, *Log4cxx* or `Log4j`.

2.5.5 Example of the output

Example output as recorded during the application startup is shown below.

```
2011-12-14 14:11:01 QGama.Main [INFO] Log4Qt logging framework initialized!
2011-12-14 14:11:01 QGama.Main [INFO] Loading bundled translation files.
2011-12-14 14:11:01 QGama.Main [WARN] Loading of QGama translator file failed: :/
    qgama_en
```

[8] More about this topic will be covered in 2.7.4

```
2011-12-14 14:11:01 QGama.Main [WARN] Loading of Qt translator file failed:
    qt_en_US
2011-12-14 14:11:01 QGama.Main [INFO] Loading plugins.
2011-12-14 14:11:01 QGama.PluginsManagerImpl [DEBUG] Reading settings.
2011-12-14 14:11:01 QGama.ISettings [DEBUG] Group changed to: Plugins
2011-12-14 14:11:01 QGama.ISettings [DEBUG] Group reseted to:
2011-12-14 14:11:01 QGama.PluginsManagerImpl [DEBUG] Processing provider: cz.ctu.
    fce.dmc
2011-12-14 14:11:01 QGama.PluginsManagerImpl [DEBUG] Processing information xml:
    Core_info.xml
2011-12-14 14:11:01 QGama.PluginsManagerImpl [DEBUG] Processing information xml:
    SQLEditor_info.xml
2011-12-14 14:11:01 QGama.PluginsManagerImpl [TRACE] Done with XML reading.
2011-12-14 14:11:01 QGama.PluginInfo [TRACE] Plugin 'Core': Resolving
    dependencies.
2011-12-14 14:11:01 QGama.PluginInfo [TRACE] Plugin 'Core': Successfully resolved
    .
2011-12-14 14:11:01 QGama.PluginInfo [TRACE] Plugin 'NetworkOverview': Resolving
    dependencies.
2011-12-14 14:11:01 QGama.PluginInfo [TRACE] Plugin 'NetworkOverview':
    Successfully resolved.
2011-12-14 14:11:01 QGama.PluginInfo [TRACE] Plugin 'SQLEditor': Resolving
    dependencies.
...
```

2.6 Plugins framework

The heart of *QGama* application is the plugin framework. It is inspired by the plugin mechanisms of *Qt Creator IDE* (`http://qt.gitorious.org/qt-creator`) and *Qtilities* project (`http://gitorious.org/qtilities`) although it does not want to be such general, multifunctional and configurable as they are. Author was trying to make everything as simple as possible to satisfy his needs.

This section starts with an overview of what Qt offers in the area of plugins and why that was not enough for *QGama* application. Then will the author go step by step through the finally adopted implementation explaining the key concepts.

2.6.1 Qt plugins

Qt plugin is a shared library with several benefits over the classical shared library. It can be loaded at runtime using `QPluginLoader` instance. `QPluginLoader` checks if the plugin is linked against the same version of Qt as the application and tries to access the library's root component (method `instance()`). After that it remains to test if the root component implements the expected plugin's interface using `qobject_cast()`. [9] If it succeeds, interface implementation brought by plugin is ready to use.

In *QGama* project, there is one general interface for tying together the application and plugins, the `IPlugin` defined in `src/libs/qgama/extensionsystem/iplugin.h`.

[9] Plugin should have visible only the symbols confirming the public interface! For better explanation of the shared libraries and the export of its symbols, section 2.7 shall be consulted.

```
namespace QGama {

    class QGAMA_EXPORT IPlugin : public QObject
    {
        Q_OBJECT

        public:
            virtual ~IPlugin() {}

            /***** INITIALIZATION AND FINALIZATION */
            virtual bool initialize(QString &errorString) = 0;
            virtual bool initializeExtensions(QString &errorString) = 0;
            virtual void finalize() = 0;

            /***** GLOBAL OBJECT POOL */
            void addObject(QObject *object);
            void removeObject(QObject *object);
    }; // class Plugin

} // namespace QGama

Q_DECLARE_INTERFACE(QGama::IPlugin, "cz.ctu.fce.dmc.QGama.IPlugin/1.0")
```

It consists of 3 pure virtual methods that each plugin has to implement.

- Method `initialize(QString &errorString)` is called by the *plugins manager* in the initial phase. In this method a plugin is supposed to initialize its part which does not depend on any other plugin and register its proper extension points (interfaces) into the global objects pool. For this purpose serves the `addObject(QObject *object)` convenience method. [10]

- Method `initializeExtensions(QString &errorString)` is called by the *plugins manager* in the phase where all of the other plugins are initialized. In this method a plugin is supposed to initialize its dependencies.

- Method `finalize()` is called by the *plugins manager* before a plugin will be stopped and deleted. It serves for a clean-up - i.e. removing object registrations from the *global object pool*.

  The `Q_DECLARE_INTERFACE` macro in the end tells the Qt's *meta-object system* about the interface existence. [11]

Plugin on the other side (in a separate project) has to:

- Define a plugin access-point class inherited by `QObject` (in *QGama*'s case satisfied yet in the `IPlugin` interface declaration) and plugin interface defined in the application (`IPlugin`).

[10] This is just an overview how the things work, it will be discussed in much more detail in section 2.6.3 and 2.6.4.

[11] If the reader wishes to know exactly to which commands this and other *plugin*-related macros are expanded, [13] shall be consulted, at the section 2.9. this topic is covered.

```
1  namespace QGama { namespace Core {
2
3      class QGAMA_CORE_EXPORT CorePlugin : public IPlugin {
4          Q_OBJECT
5          Q_INTERFACES(QGama::IPlugin)
6
7          public:
8              CorePlugin();
9              virtual ~CorePlugin();
10
11             virtual bool initialize(QString &errorString);
12             virtual bool initializeExtensions(QString &errorString);
13             virtual void finalize();
14
15         private:
16             MainWindow *m_mainWindow;
17             static Log4Qt::Logger *s_logger;
18     };
19
20 } } // namespace QGama::Core
```

- Use the `Q_INTERFACES()` macro (line 5) to tell the Qt's *meta-object system* about the interface existence.
- Export the plugin using the convenience `Q_EXPORT_PLUGIN2()` macro.

```
1 Q_EXPORT_PLUGIN2(Core, QGama::Core::CorePlugin)
```

The plugin approach described so far nevertheless does not satisfy yet the requirements which were laid on `QGama`: *"allow almost everywhere to add almost everything"*. To satisfy them, the concept of a *global object pool* has to be added. Both previously mentioned projects (*Qt Creator IDE* and *Qtilities*) also use it. Advantages of the *global object pool* will be discussed in more detail in a section 2.6.5.

2.6.2 QGama plugins

In the source code, plugins could be found in the `src/plugins` sub-directory. Every plugin includes basic definitions from `src/qgamaplugin.pri`. There are the basic settings of the destination directory, plugin shared library and rpath (on Unix). It also includes several helper functions. Some of them were took from the *Qt Creator IDE* source code with several local modifications. Every such usage is properly marked with Nokia's GPL license attached.

Below is an excerpt from a project file for the `Core` plugin.

```
1 TARGET=$$qtLibraryTarget(Core)
2 DEFINES+=QGAMA_CORE_LIBRARY
3
4 include(../../qgamaplugin.pri)
5 include(core_dependencies.pri)
6
7 QT += sql xml xmlpatterns
8
```

```
9 include(core_sources.pri)
```

Each plugin also has to define a simple meta-data-like XML file named `<plugin-name>_info.xml` with the following structure:

```
1 <plugin name="Core" version="1.0.0" compatibilityVersion="1.0.0">
2     <provider>cz.ctu.fce.dmc</provider>
3     <copyright>(c) Jiri Novak</copyright>
4     <license>GNU GPL v3</license>
5     <category>QGama</category>
6     <description>The core plugin for the QGama GUI of GNU project Gama.</
          description>
7     <url>http://www.fsv.cvut.cz</url>
8 </plugin>
```

If there is a need to use refer to some variables from the project file, there is a way. The file shall in that case be stored under the `<plugin-name>_info.xml.in` name and the variables called explicitly.

```
1 <plugin name=\"Core\" version=\"$$QGAMA_VERSION\" compatibilityVersion=\"$$
      QGAMA_VERSION\">
2 ...
```

QMake has one very useful command: `QMAKE_SUBSTITUTE`. If called within a project (`.pro`) file, given a file it substitutes all the variables inside it. [12]

2.6.3 PluginInfo

`PluginInfo` (`src/libs/qgama/extensionsystem/plugininfo.h`) is a class through which *plugins manager* controls the individual plugins.

Each plugin can be in several states during its life cycle.

Invalid	Initial state, plugin XML was not even parsed yet.
Read	Plugin XML was successfully parsed, every piece of information included in it is now accessible via `PluginInfo` class.
Resolved	All of the dependencies specified in the `<dependencies>` tag were found and they were not circular and within the compatibility range specified, their list is now accessible via the `dependencies()` method.
Loaded	Plugin's shared library was successfully loaded (Qt version, interface correspondence were verified), plugin interface is now accessible via the `plugin()` method.
Initialized	Plugin's `initialize()` method was invoked and returned true (no errors).
Running	Plugin's dependencies were also initialized and `initializeExtensions` method was invoked and returned true (no errors).
Stopped	Plugin's `finalize` method was called.
Deleted	Plugin instance was deleted.

[12] Official documentation for *QMake* variables is available at: `http://doc.qt.nokia.com/latest/qmake-variable-reference.html`, useful wiki for "undocumented features" of *QMake* at: `http://www.qtcentre.org/wiki/index.php?title=Undocumented_qmake`.

For switching between states, corresponding methods exist:

1. `read()`,
2. `resolveDependencies(const QList<PluginInfo*> pluginInfos)`,
3. `load()`,
4. `initialize()`,
5. `initializeExtensions()`,
6. `stop()`,
7. `kill()`.

If in any step of the initialization process any error occurs, it will be saved in the `PluginInfo`'s error string and plugin will stay in its current state (ignoring the rest of the steps performed on it). `PluginInfo` has two important getter functions for checking if something went wrong: `hasError()` and `errorString()`.

2.6.4 Plugins manager

Plugins manager implementation is hidden behind a `IPluginsManager` interface, which has the following declaration:

```
namespace QGama {

    class IPlugin;
    class PluginInfo;

    class QGAMA_EXPORT IPluginsManager : public QObject {
        Q_OBJECT

        public:
            virtual void loadPlugins() = 0;

            // getters
            virtual PluginInfo* pluginByName(const QString &name) const = 0;
            virtual QList<PluginInfo*> plugins() const = 0;
            virtual QHash<QString, QList<PluginInfo*> *> pluginCategories() const
                = 0;
            virtual QStringList disabledPlugins() const = 0;
            virtual QStringList forcedEnabledPlugins() const = 0;
            virtual QList<PluginInfo*> nonProblematicLoadOrder() const = 0;
            virtual bool hasError() const = 0;

        signals:
            void pluginsChanged();

        protected slots:
            virtual void shutdown() = 0;
    }; // IPluginsManager

} // namespace QGama
```

There is the essential `loadPlugins()` method, several conventional getters (whose function is obvious from their names), `pluginsChanged()` signal and the `shutdown()` slot. This has to be connected to the main application's `aboutToQuit()` signal.

```
QObject::connect(&app, SIGNAL(aboutToQuit()), &pluginsManager, SLOT(shutdown()));
```

In its constructor, *plugins manager* scans the application settings for the presence of group `Plugins` and entries:

directory Path to the directory from which plugins will be loaded.

disabled Determines which plugins should not be loaded on start-up.

forced Determines which plugins are essential.

Once having the name of the directory to be scanned, `readPluginInfos()` method is called. This iterates through all subdirectories in the path (each subdirectory is intended to represent a different plugin *provider*) and looks up the `<plugin-name>_info.xml` files. It will parse them and fill the inner lists of plugins and its categories. It will also try to resolve plugin dependencies.

When `loadPlugins()` method is invoked, *plugins manager*:

1. Finds non-problematic load order (*topological order* algorithm with the detection of possible circular dependencies).
2. Using the order calculated in step 1 it calls `loadPlugin()` method for each of the plugins. This checks if the given plugin does not have any errors, is enabled, is in the required state (`Resolved`) and all of its dependencies are already `Loaded`. If satisfied, plugin will be loaded, if not, no action will be taken.
3. Using the order calculated in step 1 it calls `initializePlugin()` method for each of the plugins. This checks if the given plugin does not have any errors, is enables, is in the required state (`Loaded`) and all of its dependencies are already `Initialized`. If satisfied, plugin will be initialized, if not, no action will be taken.
4. Using reversed order calculated in step 1 it calls `initializeExtensions()` method for each of the plugins. This checks if the given plugin does not have any errors, is enabled and in the required state (`Initialized`). If satisfied, plugins extensions will be initialized, if not, no action will be taken.
5. Emits `pluginsChanged()` signal in the end.

The `shutdown()` slot on the other hand, does the following.

1. First it looks if user disabled explicitly some plugin, so that the application would not load it on the next start-up, and stores this list into the application's persistent settings.
2. Finds non-problematic unload order.
3. Using the order calculated in step 2 it calls `stopPlugin()` method for each of the plugins. This checks if the given plugin does not have any errors, is enabled, is in the required state (`Running`) and all of its dependencies are already `Stopped`. If satisfied, plugin will be stopped (finalized), if not, no action will be taken.

4. Using reversed order calculated in step 2 it calls `deletePlugin()` method for each of the plugins. This checks if the given plugin does not have any errors, is enabled and in the required state (`Stopped`). If satisfied, plugin will be killed (deleted), if not, no action will be taken.

As it was mentioned in the `shutdown()` slot description, user can access plugins overview via *Edit -> Plugins* menu entry (figure 2.2). Plugins which are required for the application run are greyed out (it is not possible to disable them). ✔ sign means plugin is ok, ✗ means plugin is disabled and ✖ means plugin has errors (in this case another tab called *Plugin Errors* will be visible) – figure 2.5.

QGama has two *forced enabled* plugins: `Core` plugin and `SQLEditor` plugin. Those plugins cannot be disabled by the user. Moreover if an error occurs in the `Core` plugin (which brings among others the application's main window), application will not start at all, it will just display an *error overview* dialog and quit.

We can verify this behaviour from the main window's `loadPlugins()` function implementation listed afterwards.

1. It requests the *plugins manager* instance (*singleton*) and calls `loadPlugins()` on it.
2. Connects application `aboutToQuit()` signal to the *plugins manaager's* `shutdown()` slot.
3. Checks if any error occurred, if so, will display an *error overview* dialog indicating what went wrong (figure 2.3).
4. Checks if the `Core` plugin's state is `Running`. If not, will display a critical message dialog and exit the application (figure 2.4).

```
1  int QGama::loadPlugins(QApplication &app)
2  {
3      // load plugins
4      m_logger->info(QObject::tr("Loading plugins."));
5      IPluginsManager &pluginsManager = ApplicationContext::pluginsManager();
6      pluginsManager.loadPlugins();
7
8      // if an error occured, inform about it
9      if (pluginsManager.hasError()) {
10         PluginsErrorOverviewDialog dialog;
11         dialog.exec();
12     }
13
14     // check if core plugin is running
15     PluginInfo *corePlugin = pluginsManager.pluginByName("Core");
16     if (corePlugin->state() != PluginInfo::Running) {
17         QString message = QObject::tr("Cannot load 'Core' plugin. Application
               quits. Check logs for errors.");
18         m_logger->fatal(message);
19         QMessageBox::critical(0, QObject::tr("QGama Plugins Manager"), message);
20         return CORE_COULD_NOT_START;
21     }
22
23     // connect aboutToQuit signal to the destroy slot
```

```
24      QObject::connect(&app, SIGNAL(aboutToQuit()), &pluginsManager, SLOT(shutdown
            ()));
25
26      return CORE_SUCCESSFULLY_LOADED;
27  }
```

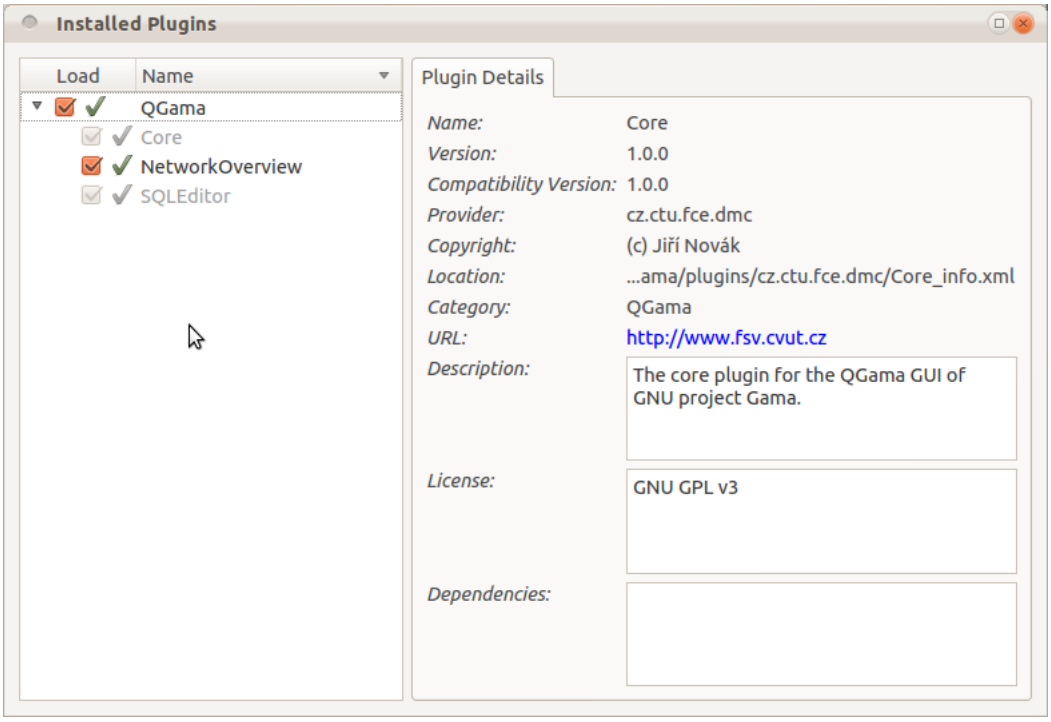

Figure 2.2: Plugins View Dialog example.

2.6.5 Thread-safe global object pool

Thread-safe global object pool is a wrapper class around a simple list of pointers to `QObject` instances, which is protected against simultaneous access from the different threads and provides several useful methods. It is implemented as *singleton* and its instance is intended to be accessed via `ApplicationContext` *factory method* `globalObjectPool()`.

The meaning of existence of the global object pool is simple: to provide a "storage" where individual plugins can "register" their extend points (*interfaces* and *implementations*) and provide a set of getters which will make possible to fulfil requests like:

- *"Give me all of the instances implementing specified interface."*
- *"Give me instance named MyClass".*

One could also said that it is a very simple application container.

Plugins are supposed to register their extension points with the `addObject(QObject *object)` method (or better with an equally-named member method of the `IPlugin` interface) in the `initialize()` method and unregister in the `finalize` method. Global object pool does not take over the ownership of inserted objects (`removeObject(QObject *object)` has to be called explicitly).

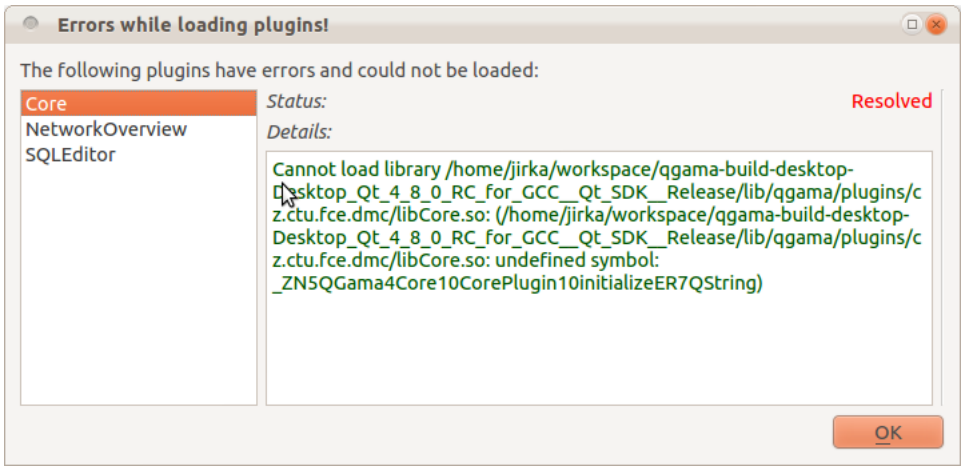

Figure 2.3: *Plugins error overview* dialog - an error in `Core` plugin (undefined symbol) causes that dependent plugins will also fail.

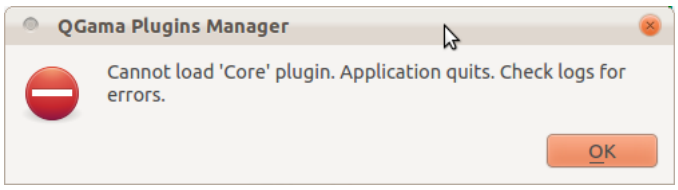

Figure 2.4: Error displaying problem with loading the `Core` plugin.

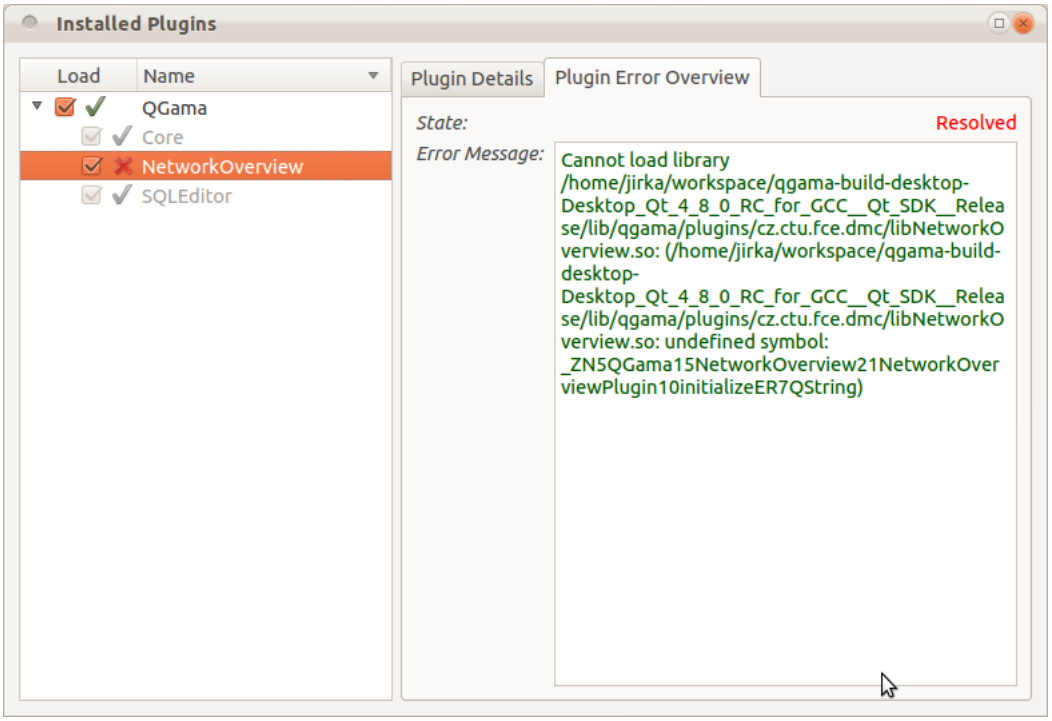

Figure 2.5: *Plugins view* dialog – error in the optional plugin.

If plugins need to interact with some component from a different plugin (which was registered previously in the global object pool), they can do it in the `initializeExtension()` method or anywhere in the source code that will follow after the execution of this method.

This approach enables that whenever the programmer feels there could be various implementations in the future, he will create an interface abstracting the concrete functionality, register its implementation into the global object pool and once generating the resulting dialog, instead of using the concrete implementation directly, he will ask the global object pool to give him / her all of the implementations of the required interface and build thus the dialog's layout at runtime based on "what is available".

This approach was used at different places within the code:

- The selection mode for the network configurations about to be opened: for sequential adjustment it will require to allow selecting more than one configuration at the same time and reorganization of the GUI's navigation panel would be also required. **Solution**:

  1. Definition of an interface which all of the selection modes will have to implement.

```
namespace QGama { namespace Core {

    class QGAMA_CORE_EXPORT IConfigurationChooserView : public
        QListView {
        Q_OBJECT

        protected slots:
            void selectionChanged(const QItemSelection &selected,
                                  const QItemSelection &deselected);
            void contextMenuEvent(QContextMenuEvent *event);

        public:
            explicit IConfigurationChooserView(QWidget *parent = 0);
            virtual ~IConfigurationChooserView() {}

            virtual QString name() const = 0;
            virtual void accepted() = 0;
            QModelIndexList selectedModelIndexes() { return
                selectedIndexes(); }

        signals:
            void selectionChanged();
    }; // class IConfigurationChooserView

} } // namespace QGama::Core
```

  2. Making the dialog aware of that interface and force it to populate via *global objects pool* in its constructor.

```
ConfigurationChooserDialog::ConfigurationChooserDialog(QWidget *parent)
    :
    QDialog(parent),
    m_ui(new Ui::ConfigurationChooserDialog)
{
    m_ui->setupUi(this);
```

```
6
7       // populate stack widget with the registered implementations of
8       // IConfigurationChooserView
9       QList<IConfigurationChooserView*> views =
10          ApplicationContext::globalObjectPool().objects<
                IConfigurationChooserView>();
11
12      foreach (IConfigurationChooserView *view, views) {
13          m_ui->stackedWidget_SelectConfiguration->addWidget(view);
14          m_ui->comboBox_EditMode->addItem(QIcon(ICON_FILE_DRAFT), view->
                name());
15
16          connect(view, SIGNAL(selectionChanged()),
17                  this, SLOT(reactToSelectionChange()));
18      }
19
20      s_logger->debug(tr("%1 configuration chooser view implementations
            found.").arg(views.size()));
21
22      // connect to the datamanager to handle model updates
23      connect(&ICore::instance().dataManager(),
24              SIGNAL(configurationsAndDescriptionsModelInitialized()),
25              this, SLOT(updateModels()));
26
27      // initialize dialog
28      reactToSelectionChange();
29  }
```

3. Register the implementation in the plugin's `initialize()` method.

```
1   bool SQLEditorPlugin::initialize(QString &errorString)
2   {
3       Q_UNUSED(errorString);
4
5       // register interfaces
6       ApplicationContext::globalObjectPool().addObject(
            m_singleNetworkChooser);
7
8       return true;
9   }
```

- Another place where to use it is the *Edit -> Preferences* dialog. Once again there will be an interface `IPreferencesPage`, whose implementations will be brought by plugins and registered into the *global object pool*. `Core` plugin will then construct the dialog in its `extensionsInitialized()` method. Because of the Qt's *parent-child system* parent will take care of deletion of all its children. This implies, we have to be careful not to cause double delete. Therefore the dialog should not be created every time it is showed, but only once after all plugins have been loaded.

- The same approach was used for the `NetworkOverview` plugin and the graphics scene / view it brings within. If the plugin is loaded, network's overview will be visualized

in the main window's *central widget*, otherwise the widget will remain empty.

Getters

Global object pool offers the following methods (a listing with self-explanatory commentaries is provided).

```
// get all instances
QList<QObject*> allObjects() const;

// get instance by objectName match
QObject* objectByName(const QString &name) const;

// get all instances of class
template <typename T> QList<T*> objects() const { ... }

// get first instance of class
template <typename T> T* object() const { ... }
```

2.6.6 Writing QGama's plugin

In this section the author will revise how to write a QGama's plugin from the scratch. Let him assume that a developer wants to add a plugin for *deformations analysis* (where the configuration chooser dialog, left project navigation panel and some of the dialogs need to look differently).

1. Latest *QGama*'s source code should be cloned from git.
2. It should be switched to the `src/plugins` directory.
3. New folder should be created there, in this example it will be called `deformationssanalysis`.
4. `src/plugins/plugin.pro` has to be edited to make it aware of the newly created plugin.
   - New target has to be add to the SUBDIRS.

```
SUBDIRS = plugin_coreplugin \
  plugin_sqleditor \
  plugin_networkoverview \
  plugin_deformationsanalysis
```

   - New target's subdir and dependencies has to be defined (assuming it will depend on the `Core` and `SQLEditor` plugins).

```
plugin_deformationsanalysis.subdir = deformationsanalysis
plugin_deformationsanalysis.depends = plugin_coreplugin
    plugin_sqleditor
```

5. It should be switched into the plugin directory. (`src/plugins/deformationsanalysis`).
6. Plugins project (`deformationsanalysis.pro`) file with the following content has to be created.

```
TARGET=$$qtLibraryTarget(DeformationsAnalysis)
DEFINES+=QGAMA_DEFORMATIONSANALYSIS_LIBRARY

include(../../qgamaplugin.pri)
include(deformationsanalysis_dependencies.pri)

QT += sql

include(deformationsanalysis_sources.pri)
```

7. The included `deformationsanalysis_dependencies.pri` file has to be created.

```
include(../../plugins/coreplugin/coreplugin.pri)
include(../../plugins/sqleditor/sqleditor.pri)
```

8. The included `deformationsanalysis_sources.pri` file has to be created.

```
HEADERS += \
  deformationsanalysisplugin.h \
  deformationsanalysis_global.h

SOURCES += \
  deformationsanalysisplugin.cpp

FORMS += \
```

9. A convenience `deformationsanalysis.pri` file has to be created.

```
include(deformationsanalysis_dependencies.pri)
LIBS *= -l$$qtLibraryName(DeformationsAnalysis)
```

10. Plugin info XML specification (`deformationsanalysis_info.xml.in`) has to be created.

```
<plugin name=\"DeformationsAnalysis\" version=\"$$QGAMA_VERSION\"
    compatibilityVersion=\"$$QGAMA_VERSION\">
    <provider>cz.ctu.fce.dmc</provider>
    <copyright>(c) Jiri Novak</copyright>
    <license>GNU GPL v3</license>
    <category>QGama</category>
    <description>DeformationsAnalysis plugin brings the GUI features to
        support deformation analysis computations.</description>
    <url>http://www.fsv.cvut.cz</url>
    <dependencies>
  <dependency name=\"Core\" version=\"$$QGAMA_VERSION\"/>
  <dependency name=\"SQLEditor\" version=\"$$QGAMA_VERSION\"/>
    </dependencies>
</plugin>
```

11. The `deformationsanalysis_global.h` has to be created as follows. [13] .

[13] The need for this step is described in section 2.7

```
#include <QtCore/qglobal.h>

#if defined(QGAMA_DEFORMATIONSANALYSIS_LIBRARY)
#  define QGAMA_DEFORMATIONSANALYSIS_EXPORT Q_DECL_EXPORT
#else
#  define QGAMA_DEFORMATIONSANALYSIS_EXPORT Q_DECL_IMPORT
#endif

#endif // QGAMA_DEFORMATIONSANALYSIS_GLOBAL_H
```

12. The `DeformationsAnalysis` class has to be created (does nothing so far).

    - Header:

```
#ifndef QGAMA_DEFORMATIONSANALYSIS___DEFORMATIONSANALYSISPLUGIN_H
#define QGAMA_DEFORMATIONSANALYSIS___DEFORMATIONSANALYSISPLUGIN_H

#include <plugins/deformationsanalysis/deformationsanalysis_global.h>
#include <qgama/extensionsystem/iplugin.h>

using namespace QGama;


namespace QGama { namespace DeformationsAnalysis {

    class QGAMA_DEFORMATIONSANALYSIS_EXPORT DeformationsAnalysisPlugin
        : public IPlugin {
        Q_OBJECT
        Q_INTERFACES(QGama::IPlugin)

        public:
            DeformationsAnalysisPlugin();
            virtual ~DeformationsAnalysisPlugin();

            bool initialize(QString &errorString);
            bool initializeExtensions(QString &errorString);
            void finalize();
    }; // class DeformationsAnalysisPlugin

} } // namespace QGama::DeformationsAnalysis

#endif //QGAMA_DEFORMATIONSANALYSIS___DEFORMATIONSANALYSISPLUGIN_H
```

    - Source:

```
#include <qgama/qgama.h>
#include <plugins/deformationsanalysis/deformationsanalysisplugin.h>

using namespace QGama;
using namespace QGama::DeformationsAnalysis;

DeformationsAnalysisPlugin::DeformationsAnalysisPlugin() {}

DeformationsAnalysisPlugin::~DeformationsAnalysisPlugin() {}
```

```
10
11 bool DeformationsAnalysisPlugin::initialize(QString &errorString)
12 {
13     Q_UNUSED(errorString);
14     return true;
15 }
16
17 bool DeformationsAnalysisPlugin::initializeExtensions(QString &
       errorString)
18 {
19     Q_UNUSED(errorString);
20     return true;
21 }
22
23 void DeformationsAnalysisPlugin::finalize() {}
24
25 Q_EXPORT_PLUGIN2(DeformationsAnalysis, QGama::DeformationsAnalysis::
       DeformationsAnalysisPlugin)
```

13. Application should be compiled and run at this moment, the developer should see the plugin listed in the *Edit* -> *Plugins* dialog (figure 2.6).

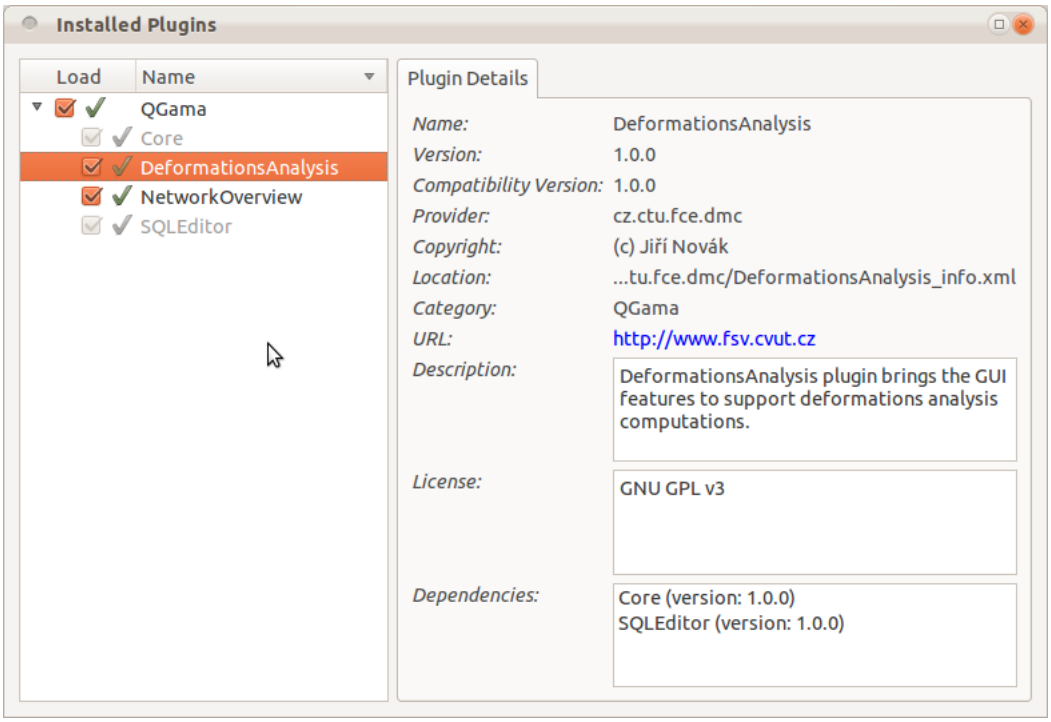

Figure 2.6: Newly added plugin visible in the *Edit* -> *Plugins* dialog.

14. New class (`DeformationsAnalysisConfigurationChooser` in our example) should be added to the plugin. This class will inherit from `IConfigurationChooser` interface from the `Core` plugin and implement the pure virtual functions in the way that another navigation widget is provided and offering different options, dialogs and widgets.

2.7 Dynamic libraries

2.7.1 Exporting symbols

While creating a shared library it has to taken into account that every symbol (function, variable or class) contained within the library and intended to be used by *clients* (that is application or other libraries), has to be marked in a special way. Otherwise the dynamic linker or similar program would not be able to find them. In other words, we have to export **public symbols** explicitly to make them visible (accessible from "outside"). The rest of the symbols should remain hidden.

On some platforms there is also required a special *import* declaration while using a shared library from within the client.

Qt provides a couple of conventional macros that are expanded to the necessary platform-specific definitions.

- `Q_DECL_EXPORT` macro that has to be added before the declaration of the symbols of a public interface to be exported while compiling a shared library.
- `Q_DECL_IMPORT` macro that has to be added before the declaration of the symbols of a public interface when compiling a client that uses the shared library.

We could achieve the right macro to be invoked in both cases by creating a separate header file with the following definition:

```
1  #ifndef QGAMA_CORE___GLOBAL_H
2  #define QGAMA_CORE___GLOBAL_H
3
4  #include <QtCore/qglobal.h>
5
6  #if defined(QGAMA_CORE_LIBRARY)
7  #  define QGAMA_CORE_EXPORT Q_DECL_EXPORT
8  #else
9  #  define QGAMA_CORE_EXPORT Q_DECL_IMPORT
10 #endif
11
12 #endif // QGAMA_CORE_GLOBAL_H
```

And adding a line to the library project file:

```
1  DEFINES += QGAMA_CORE_LIBRARY
```

This ensures that the right macro is expanded when using by library and clients (library has defined QGAMA_CORE_LIBRARY and client not). The typical usage then would look like:

```
1  #include <plugins/coreplugin/core_global.h>
2
3  class QGAMA_CORE_EXPORT ICore ...
```

File name convention `<project-name>_global.h` is used for those header files in each sub-project (*library*, *plugin*).

2.7.2 QGama libraries

In the same manner as it was in the case of *QGama*'s *plugins*, when the developer is creating a new `QGama library`, it should be created in the `src/libs` folder and the inclusion of `qgamalibrary.pri` file with the basic definitions should not be forgotten.

A typical project file of a library then looks like this:

```
TEMPLATE = lib
TARGET = QGama
DEFINES += QGAMA_LIBRARY
include(../../qgamalibrary.pri)
include(qgama_dependencies.pri)

QT += xml sql webkit

include(qgama_sources.pri)

RESOURCES += \
    qgama.qrc \
    ../../../help/help.qrc
```

Each sub-project additionally defines:

- its source and headers file in a separate `<project-name>_sources.pri` file,

```
HEADERS += \
  ...

SOURCES += \
  ...

FORMS += \
  ...
```

- its dependency includes in a separate `<project-name>_dependencies.pri` file,

```
include(../../libs/qgama/qgama.pri)
include(../../libs/3dparty/gama/gama.pri)
```

- a file `<project-name>.pri` containing project dependencies file and library linkages.

```
include(core_dependencies.pri)
LIBS *= -l$$qtLibraryName(Core)
```

2.7.3 Gama library

Gama library consist of the *GNU Gama*'s computational library compiled dynamically with the *Adjustment API* and few other classes put as its *facade*.

QGama::Exception

`QGama::Exception` is the base class for all of the exceptions within *QGama* project. For convenience it is derived from the `GNU_gama::Exception::base` and provides two parameters (title, text) and its corresponding getters.

GNU Gama Adjustment API

Adjustment API currently consist of a class `QGama::Adjustment`. It was developed by the thesis supervisor Aleš Čepek and serves as an interface for:

- Fetching the configuration related data from database into the inner structures (method `read_configuration()`).
- The actual adjustment (method `exec()`).
- Obtaining XML with the results (method `xml()`) or several other convenience getters of the calculated values.

Furthermore there is a `label()` signal used to inform about the calculation progress.

Xml2Txt

`Xml2Txt` is a helper class, which for the given XML *input*, *language*, *encoding* and *angular units* generates a resulting TXT format with the adjustment results. Basically deals the same code as the `gama-xml2txt` utility distributed within *GNU Gama*.

2.7.4 QGama library

`QGama` library is a library providing common, reusable utilities to the rest of the program. Its design is very general and could be reused for any other Qt based application.

All the library constants are stored within the `constants.h` file. It also bundles two separate resources files.

- `src/libs/qgama/qgama.qrc` which includes `css`, images, and XMLs from `src/libs/qgama/resources` directory.
- `translations/translations.qrc` which includes `.qm` files with translations from `translations` directory.

As already discussed in the section 2.4, it is conformed by the following components (everything defined within the `QGama` namespace).

ExtensionSystem	Provides infrastructure for the plugins mechanism. Already discussed in the section 2.6. • `IPlugin`, • `PluginInfo`, • `PluginsErrorOverviewDialog`, `PluginsViewDialog`, • `IPluginsManager` and `PluginsManagerImpl` classes.
Global Object Pool	Already discussed in the section 2.6.5.
ActionsManager	Provides a manager for dynamic creation of the main menu entries and application shortcuts. • `IActionsManager` and `ActionsManagerImpl` classes.

Editors	Provides HTML Viewer and Text Editor classes with a common interface for storing files, detecting changes and setting/retrieving content. • `Document`, `HtmlViewer`, `TextEditor` classes.
Preferences	Application's persistent storage of various settings. • `ISettings` and `SettingsImpl` classes.
HelpBrowser	A simple browser for the online help pages. • `HelpBrowser` class.
Utils	XML Syntax highlighter for Text Editor and custom Progress Dialog. • `ProgressDialog`, `XMLSyntaxHighlighter` classes.
VariablesManager	Global non-persistent map for storing variables during the application's runtime. • `IVariablesManager` and `VariablesManager` classes.

Application context

`ApplicationContext` class is an entry point of the `QGama` library. It is a non-instantiable class with *factory methods* returning the concrete implementations as the reference to their interface. [14] Thus when there would be the need of changing the implementation, it shall be the only place to touch.

Its public interface offers following getters.

```
...
public:
   static ISettings& settings();
   static IPluginsManager& pluginsManager();
   static GlobalObjectPool& globalObjectPool();
   static IActionsManager& actionsManager();
   static IVariablesManager& variablesManager();
   static void showHelpPage(const QString &page);
...
```

Preferences

For storing the application settings `QSettings` is currently used. It is a class providing persistent platform-independent storage of application settings. By default it stores the key-value pairs in the system registry on Windows, in XML preferences files on Mac OS and INI text files on Unix. There is also a possibility to enforce specific format on all

[14] `SettingsImpl`, `PluginsManagerImpl`, `GlobalObjectPool`, `ActionsManagerImpl`, `VariablesManagerImpl` are implemented as *singletons*

platforms. QGama's application settings thus are stored in the INI format on all platforms. Default location of the configuration file is:

- ~/.config/cz.ctu.fce.dmc/QGama.ini on Unix and
- C:\Documents and Settings\<User>\Application Data\cz.ctu.fce.dmc\QGama.ini on Windows.

QSettings uses internally a QMap indexed by QString and storing QVariants. When using QSettings organizations and application name has to be set (either in constructor or transitively by QCoreApplication global methods setOrganizationName(), setApplicationName()).

QVariant QVariant is a class that works in a similar way as the standard C++ unions. It permits to store most common Qt data types, holding always a single value of a single type (including lists, hashes, maps, etc.) at a time. It is possible to:

- Get the type QVariant currently holds by the type() method.
- Convert it to the different type with the convert() method (there also exist convenience methods for the most frequent types: toSize(), toString(), toStringList(), toHash(), etc.).
- Confirm if it could be converted to specified type with the canConvert() method.

For more detailed description Qt documentation should be consulted.

Storing custom types with QSettings There is also a way how to store custom data types (e.g. user-defined structure) into QSettings.

Let the author demonstrate it with the following structure.

```
1 struct Employee
2 {
3     QString name;
4     qint32 jobId;
5 };
6 Q_DECLARE_METATYPE(Employee)
```

The meta-type registration in the last row of the header file is essential. Every type which provides a public default constructor / destructor / copy constructor can be defined as meta-type. This causes that it will be possible to store it into the QVariant using the qVariantFromValue() global function and retrieve it back with the member value<T>() QVariant template method. It is very useful when there is for example the need of storing a custom pointer as the action data or similar situations.

Nevertheless this is not all, if the custom type should be also QSettings-aware. For enabling this, the custom type has to define two QDataStream operators to let Qt know how to serialize / deserialize it.

```
1 QDataStream &operator<<(QDataStream &out, const Employee &emp)
2 {
3     out << emp.name << emp.jobId;
4     return out;
5 }
```

```
6
7  QDataStream &operator>>(QDataStream &in, Employee &emp)
8  {
9      in >> emp.name >> emp.jobId;
10     return in;
11 }
```

It also requires run-time registration of the data-type before the first instantiation of the `QSettings` object will take place.

```
1 qRegisterMetaType<CustomStructure>("Employee");
2 qRegisterMetaTypeStreamOperators<CustomStructure>("Employee");
```

The author was about to use this technique while facing the storing of the database connection parameters in the `Core` plugin, but he could not use it because it was impossible to satisfy the last condition – `QSettings` is being used for the first time in the application's `main` function when retrieving the logging framework settings and at that time it cannot know anything about any custom types which plugins will define. Therefore: when a developer needs to store a custom data-type into application settings, non-elegant transform functions to convert it for example to `QHash` and back should be written. That is what was adopted in the case of `DbParameters` struct in the `Core` plugin.

ISettings and SettingsImpl classes `QSettings` class is not used directly, instead another level of indirection was introduced.

- An abstract interface `ISettings` with its proper inner `QMap<QString, QVariant>` was created.
- Its implementation `ISettingsImpl` was created. It fills the inner map from the persistent `QSettings` file in its constructor and saves it back there in the destructor. [15]

As it was already mentioned, `SettingsImpl` is implemented as singleton and its accessible via `ApplicationContext::settings()`.

An example of storing a value into it and retrieving it back is listed bellow. It is an excerpt from the `SettingsImpl` constructor, where the basic settings for plugins (path, forced-enabled plugins), logger (console / file appender, log level) and default window size (800 x 600) are stored.

```
1  ...
2  /***** PLUGINS DEFAULT SETTINGS */
3  beginGroup("Plugins");
4  // set plugin directory to the standard application folder
5  if (!contains("directory"))
6      set("directory", QString::fromAscii(QGAMA_PLUGIN_PATH));
7  // set the plugins which has to be loaded all the time
8  if (!contains("forced"))
9      set("forced", QStringList() << "Core" << "SQLEditor");
10 endGroup();
11 ...
```

[15] This approach allows easier future changes when for example the data would be stored in the database.

Actions manager

`ActionsManager` is a class providing a support for dynamic menu creation. Every plugin can specify in its `initialize()` method, which action and where wants to add and which shortcut within the application it should have.

A small example of its usage - the definition of the "File" menu creation follows.

```
void MainWindow::createMenuFile()
{
    IActionsManager& am = ApplicationContext::actionsManager();

    // Open Connection
    am.addAction(FILE_OPEN_CONNECTION,
                 tr("Open connection"),
                 tr("Open recently defined database connection."),
                 QKeySequence::Open,
                 QIcon(ICON_DATABASE_CONNECT),
                 MENU_EDIT);
    connect(am.action(FILE_OPEN_CONNECTION), SIGNAL(triggered()),
            this, SLOT(onMainWindowOpened()));

    // Disconnect
    am.addAction(FILE_DISCONNECT,
                 tr("Close connection"),
                 tr("Disconnects from the active database connection."),
                 QKeySequence::Close,
                 QIcon(ICON_DATABASE_DELETE));
    connect(am.action(FILE_DISCONNECT), SIGNAL(triggered()),
            this, SLOT(disconnectFromDb()));

    // Separator
    am.addSeparator(FILE_SEPARATOR_CONNECT);

    // Exit
    am.addAction(FILE_QUIT,
                 tr("Quit"),
                 tr("Quit the application."),
                 QKeySequence::Quit,
                 QIcon(ICON_QUIT));
    connect(am.action(FILE_QUIT), SIGNAL(triggered()),
            this, SLOT(close()));
}
```

The position in the menu is defined by the dot separators in the constants - that is if one is creating an action "File.Quit" a "File" menu is automatically created and action "Quit" is added to it. Same approach is used when specifying where the action should be added, the place is once again determined by a plain string comparison. Constants are defined with `QT_TR_NOOP()` macro to force thus their inclusion into the translation files.

Help browser

`HelpBrowser` is a very simple Online Help viewer. Every dialog is designed with a `Help` button, which should display a corresponding online help HTML page. Because this is not a critical feature, only one page was created as a proof of concept.

In the figure 2.7 you can see the *Create or edit configuration* dialog and the corresponding HTML page displayed in the `HelpBrowser`.

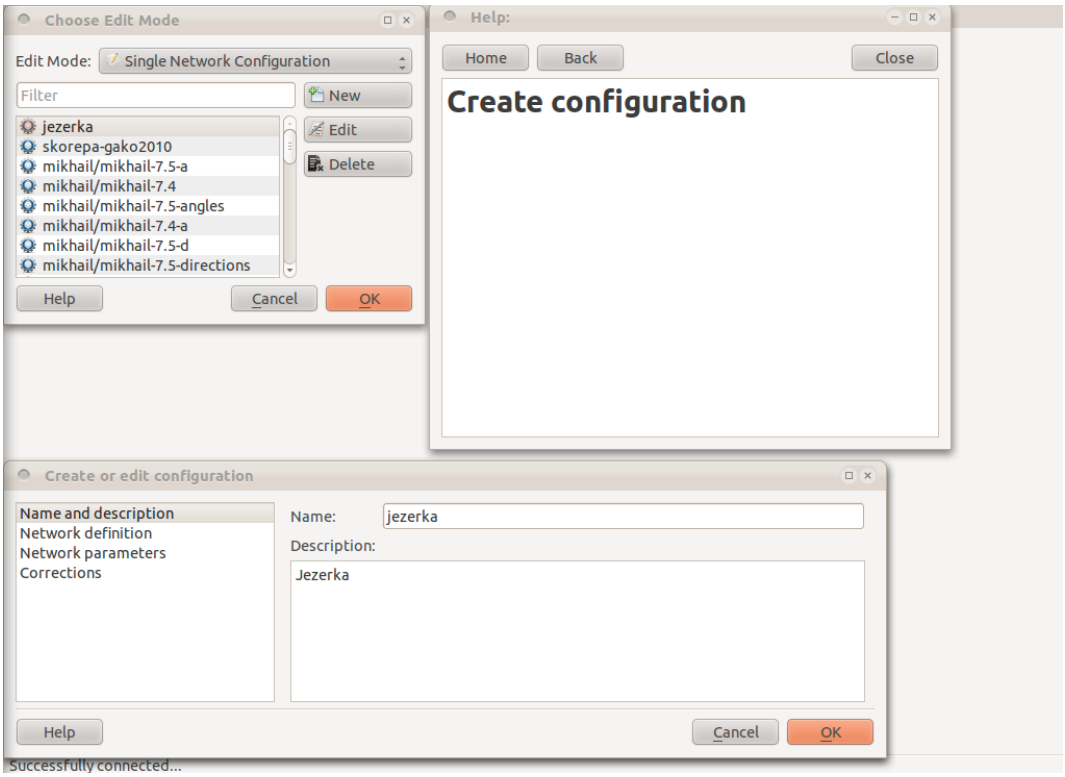

Figure 2.7: QGama's HelpBrowser - simple online help viewer.

If a developer wants to get use of the `HelpBrowser`, it is very simple.

- The HTML page with the dialog-related content has to be created and saved into the `help` folder in the root of `QGama`'s project.
- HTML page has to be registered in the `help.qrc` resource list.
- New private slot has to be added into the developed dialog class.

```
1 void CreateOrEditConfigurationDialog::showHelp()
2 {
3     ApplicationContext::showHelpPage(":/create_configuration.html");
4 }
```

- The dialog's button box signal `helpRequested()` has to be connected to the recently created slot.

```
1 // help
2 connect(m_ui->buttonBox, SIGNAL(helpRequested()),
3         this, SLOT(showHelp()));
```

Translations

Translations for all of the components of `QGama` (main application, `QGama` library, all of the plugins) are held now in one translation file (there was created a fake project containing all of the mentioned source codes).

```
TEMPLATE = app
DEPENDPATH += \
  ../src/app \
  ../src/libs/3dparty/gama \
  ../src/libs/qgama \
  ../src/plugins/coreplugin \
  ../src/plugins/networkoverview \
  ../src/plugins/sqleditor

include(../src/app/app_sources.pri)
include(../src/libs/3dparty/gama/gama_sources.pri)
include(../src/libs/qgama/qgama_sources.pri)
include(../src/plugins/coreplugin/core_sources.pri)
include(../src/plugins/networkoverview/networkoverview_sources.pri)
include(../src/plugins/sqleditor/sqleditor_sources.pri)

TRANSLATIONS = \
  qgama_cs.ts
```

If a developer wants to add a new language, he should follow these steps:

1. Add a new entry into the `translations/translation.pro` file, under the `TRANSLATIONS` target.
2. Run **lupdate translation.pro** to generate a corresponding `.ts` file.
3. Do the translation with the Qt's **linguist** tool.
4. Generate production binary with the **lrelease** command - this will generate the `.qm` file.
5. Add the newly created `.qm` file into the `translations.qrc` resource.

More about the internationalization process in Qt could be found in [13], section 5.8.

2.8 Plugins

2.8.1 CorePlugin

`Core` is the place where all of the infrastructure for database access, definition of a new connections / configurations, filling of models of the Model-View-Controller pattern, adjustment computation and format conversions take place.

`Core` plugin is declared inside the `QGama::Core` namespace and has the following basic components.

ICore interface

Singleton providing access to most important features of the `Core` plugin.
It provides access to:

- the applications important widgets (`mainWindow()`, `statusBar()`, `navigationDockWidget()`, `centralWidget()`),
- data manager which handles the filling of data models with corresponding data,
- active database parameters (to enable workers to establish another connection in the background thread).

It also emits signals informing about the current application's state:

coreOpened() is emited when the `Core` plugin's `MainWindow` is initialized,

coreAboutToConnect() emited while opening the *Recent connections* dialog,

coreConnected() emited if the connection to a database was successfully established,

configurationModelIndexesSelected() emited when user selects configuration(s) to be edited in the *Configuration Chooser* dialog,

coreAboutToDisconnect() emited when user requested to disconnect from the active database,

coreAboutToClose() emited from the `Core` plugin's `finalize()` method,

coreModelsReady() emited when *data manager* finishes fetching data of the selected configuration(s).

Data manager and models

`Core` plugin uses extensively the Model-View-Controller design pattern. It enable us to store data once and enable different, synchronized views on them.
`DataManagerImpl` is a singleton class which takes care of:

- Providing SQL DDL statements list for creating `GNU Gama`'s SQL schema if needed.
- Providing a global access-point to the data models and managing its life-cycle.
- Providing information about the currently edited configuration(s) features.

Once again, it is hidden behind an interface called `IDataManager`, which defines a couple of signals informing about the implementation state (its names are self-explanatory).

- `configurationsAndDescriptionsModelInitialized()` and
- `restOfTheModelsInitialized()`.

A common ascendant of all the models is `GamaDataModel` class. This inherits the `QSqlTableModel` which is a convenient class providing a higher-level interface for the database table access. It has several drawbacks.

- It has to operate only above one table (joins not accepted) and it is filled synchronously while `select()` method called
- In Qt, database connection cannot be shared between threads.
- There is a need to have models in the main event loop thread, because all of the views are there.
- Application should not momentarily stop responding while fetching a large amount of data (e.g. extend network configuration from the remote database - which is the worst case).

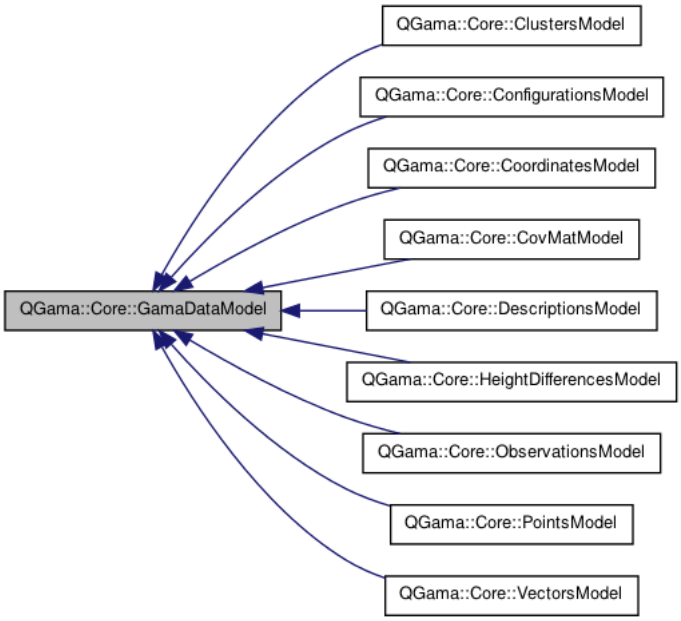

Figure 2.8: `GamaDataModel` is a common ascendant of all models.

Unfortunately, Qt does not solve this issue. A possible solution to that problem published Wysota (Qt enthusiast) on his blog [16] . In *QGama* currently this is not implemented and all of the models are filled within the main event loop thread.

It of course causes delays, which are compensated with busy cursors, working-indicating progress bars and calling to `qApp->processEvents()` to stay responsive. On the other hand, 99% of the QGama users will use the build-in *SQLite* (lightweight file-based database) support where no contention is noticed even when working with a large datasets (containing all of the `gama-local`'s examples from the git repository).

After a database connection is established (detailed description is provided within the `User Guide`, annex A), only `Configurations` and `Descriptions` models are fully fetched. Rest of the models are fetched after the selection of a specific configuration and are limited just to that configuration (which is obvious from the following code snippet).

```
1 // initialize model
2 setTable(TABLE_OBSERVATIONS);
3 setEditStrategy(QSqlTableModel::OnRowChange);
```

[16] `http://blog.wysota.eu.org/index.php/2006/12/26/remote-models/`

```
4 setFilter(whereClause + " and tag != 'dh' order by ccluster asc, indx asc");
5 select();
```

Each model defines also a sort filter proxy model available via `sortFilterModel()` method. This model is an wrapper to the original model enabling us to easily make sorting and filtering of the original model data. `GamaDataModel` also contains a static method `localizableEnumModel()`, which given the list of pairs (localized string, identifier) returns a `QStandardItemModel`. This is used whenever there is an enumeration of values that can be stored in some field of the database, for showing the user a localized string, but internally work with a fixed identifier (which is what is going to be actually stored in the database).

There are also 4 important data manipulation functions (pure virtual): `appendEntry()`, `insertEntryBefore()`, `insertEntryAfter()`, `deleteEntry()` and 1 virtual slot `presetFields()` returning a map of column integer and its value (it is intended to be called from within the functions for inserting a new entry to the model for calculation of the values of indexes - by default it returns an empty map).

Each individual model contains a public enumeration for referencing the columns. Each model also defines header data for its columns to support better descriptive and internationalized title than the original database identifiers. When some field has its own enumeration it is provided with the usage of the `localizableEnumModel()` and there exists a public getter for the pointers of those models. If some integer / double valued field has certain range that has to be satisfied, model offers also corresponding validator getters. Rest of the constraints like if the value is not empty are tested within dialogs and if not satisfied, the `Ok` button is disabled. The idea was not to bother user with the error dialog which would arise while trying to commit an invalid field into the underlying database.

Each model is also set to commit the data from cache to the database only when explicitly requested and does not work with on the level of individual fields, but records.

If a model needs to visualize and edit the data in a different format that they are physically stored there is a way. Let the author demonstrate the case on the `Observations` model, where exactly the same is required when dealing the angular values (they are stored in *radians*, but has to be visualized in *gons* or *degrees* [17]). Only thing that has to be done is to reimplement `QSqlTableModel`'s `data` and `setData` methods to behave as requested.

```
1  QVariant ObservationsModel::data(const QModelIndex &index, int role) const
2  {
3      if (role == Qt::TextAlignmentRole) {
4          if (index.column() == ObservationsModel::val) {
5              return QVariant(Qt::AlignRight | Qt::AlignVCenter);
6          } else {
7              return QVariant(Qt::AlignHCenter | Qt::AlignVCenter);
8          }
9      }
10
11     const QString tag = GamaDataModel::data(index.sibling(index.row(),
12                                             ObservationsModel::tag)).toString();
13
14     if (role == Qt::DisplayRole &&
```

[17] The same approach is adopted also while visualizing the standard deviation from `CovMat` model - physically a *variance* is stored in the model.

```
15          (tag == "direction" || tag == "z-angle") &&
16          index.column() == ObservationsModel::val) {
17          return GamaDataModel::data(index).toDouble() * m_angularUnits / M_PI;
18      }
19
20      return GamaDataModel::data(index, role);
21  }
```

The first `if` statement only gives some alignment related hints to the visualizer widget - in concrete that values should be aligned horizontally to the right and all the rest of field should remain centred.

The advertised conversion from radians (for all of the values of `direction` or `z-angle`) takes place at row 17. For all the rest of the cases we just call the default (inherited) implementation.

```
1   bool ObservationsModel::setData(const QModelIndex &index,
2                                   const QVariant &value,
3                                   int role)
4   {
5       const QString tag = GamaDataModel::data(index.sibling(index.row(),
6                                               ObservationsModel::tag)).toString();
7
8       if (role == Qt::DisplayRole &&
9           (tag == "direction" || tag == "z-angle") &&
10          index.column() == ObservationsModel::val) {
11          double val = value.toDouble() * M_PI / m_angularUnits;
12          return GamaDataModel::setData(index, val, role);
13      }
14
15      return GamaDataModel::setData(index, value, role);
16  }
```

The reverse `setData` method adopts exactly the same approach.

NullAwareItemDelegate To achieve interpreting a non-filled value (empty string) as a `null` value in the database, a developer has to subclass the `QStyledItemDelegate`, reimplement its `setEditorData()` and `setModelData()` methods and use this class whenever the model's data need to be shown. This implies either the using of convenience Qt view classes or `QDataWidgetMappers` through which it is possible to map the model's data into any widget. In both cases there is a setter method `setItemDelegate()`).

`NullAwareItemDelegate` uses Qt's *properties system* (which is based on Qt's *meta-object system* – kind of Qt's *reflexion* through which also the *signal-slot* mechanism is implemented). It searches the editor's meta-object for one of the properties *text* or *plainText*. When editor is a combo-box, reading and storing data is handled separately. In both cases nevertheless empty strings are handled as invalid `QVariant`s which posted to the model will be handled as `null` values.

MainWindow

`MainWindow` is the application's principle widget. It handles the main menu initialization, creating, (un)registering and populating of the central and navigation widget (figure 2.8.1),

emitting of the application state related `ICore`'s signals. It also defines all of the *about dialogs* and other features.

Connection and Configuration related dialogs

`Core` plugin also brings essential dialogs for initialization of each of the `QGama`'s sessions. Because `QGama` is basically spoken just a customized visualizer / editor of the database data, there has to be CRUD (Create / Replace / Update) features for database connections and configurations stored inside them.

In `src/plugins/coreplugin/dialogs` there are several classes:

RecentConnectionsDialog	`QGama`'s initial dialog, serves for the persistent management of the database connections. Connections parameters are defined within the `DbParameters` structure and stored together with the rest of the application settings inside the `.ini` file as specified in the subsection 2.7.4.
CreateOrEditConnectionDialog	Serves for defining a new database connection or editing the existing one. When confirming this dialog, it tries to establish the connection and inform user about the result. Furthermore it looks up the user-visible tables for the presence of the tables of `GNU Gama`'s SQL schema and if any of them not found, pop-ups a dialog where it will request the confirmation with its creation.
NewFileDialog	Serves when user wishes to create a new file-based database (*SQLite*).
CreateOrEditConfigurationDialog	Serves for defining a new network configuration or editing the existing one.
ConfigurationChooserDialog	Provides a filtered list of available configurations within the database, enabling the selection of those which should be opened (this is done by providing all the `IConfigurationChooserView` implementations found at run-time while creating the dialog).[18] Also enables the creation / deletion of new configurations and editing parameters of the current ones.

[18] As already discussed in the section 2.6.5 dedicated to the *global object pool*.

Principal widgets

`Core` plugin has two principal widgets (figure 2.9):

- `CentralWidget` (main window's central widget dedicated to visualize various graphical network views) and
- a dockable `NavigationDockWidget` (providing a central point for the navigation while editing configuration(s)).

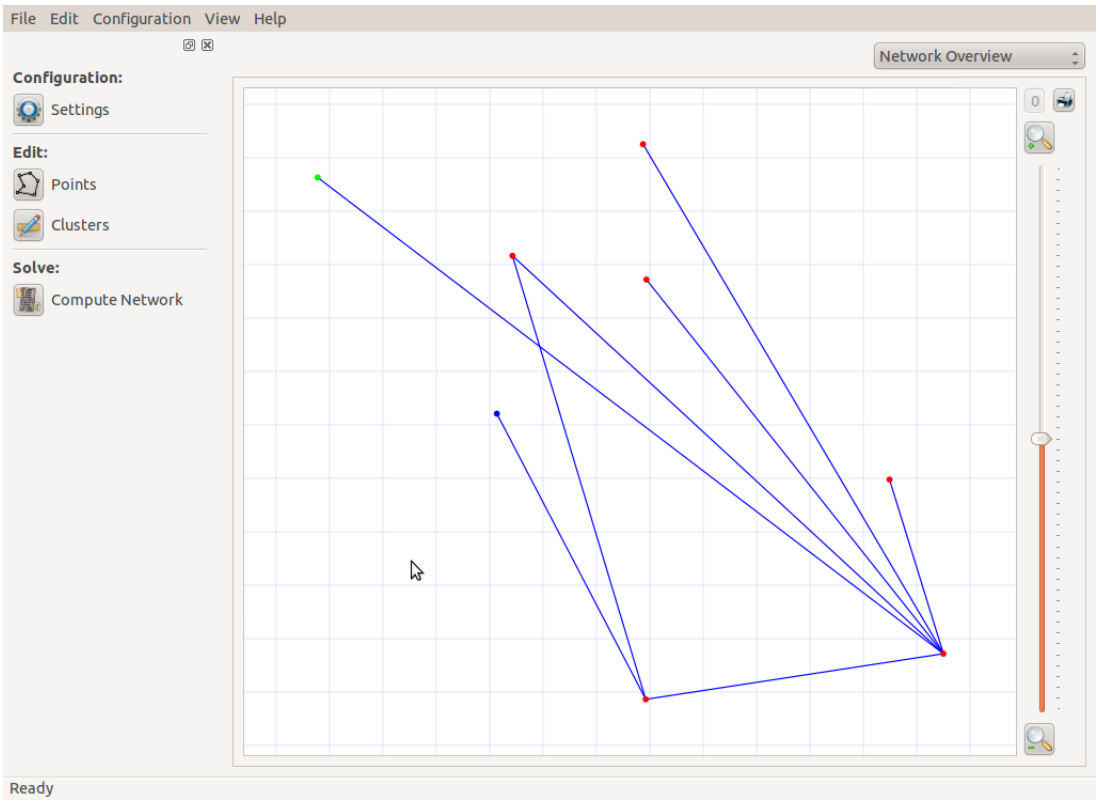

Figure 2.9: `QGama` application overview - `NavigationDockWidget` on the left, `CentralWidget` on the right, main menu above.

NavigationDockWidget is inherited from `QDockWidget` (to add it the dockable features) and contains an instance of `QStackedWidget` - a class which provides a stack of widgets when always just one widget is visible at a time. It has two public functions for setting the sub-widgets and making them visible:

- `addSubWidget(QWidget *widget)` for adding a navigation widget if it is not already in the stack and
- `setSubWidgetVisible(QWidget *widget)` for making some of the stacked widgets visible.

Those are used typically in the `accepted()` method implementation of the `IConfigurationChooserView` interface as we can see for example in the `SingleNetworkConfigurationChooserView` class of the `SQLEditor` plugin.

```
1  void SingleNetworkConfigurationChooserView::accepted()
2  {
3      s_logger->info(tr("Configuration selection accepted."));
4
5      static SingleNetworkConfigurationNavigationWidget *navigationWidget =
6              new SingleNetworkConfigurationNavigationWidget(this);
7
8      ICore::instance().navigationDockWidget()->addSubWidget(navigationWidget);
9      ICore::instance().navigationDockWidget()->setSubWidgetVisible(
           navigationWidget);
10 }
```

On the fifth row there is created (on the first method invocation) a single-network-configuration-specific navigation widget and every time a configuration is selected and opened in this configuration-chooser mode, this widget is made visible in the stackable `NavigationDockWidget`.

CentralWidget is composed by a `QStackedWidget` and a combo-box for switching the contained widgets. It contains an `initializeExtensions()` method called from the equally-named `MainWindow`'s method which populates the stacked widget with the run-time found implementations of `IConfigurationView` - another example of the *global object pool*'s usage.

```
1  void CentralWidget::initializeExtension(QString &errorMessage)
2  {
3      Q_UNUSED(errorMessage)
4
5      QList<IConfigurationView*> views =
6          ApplicationContext::globalObjectPool().objects<IConfigurationView>();
7
8      s_logger->info(tr("%1 configuration views found.").arg(views.size()));
9
10     foreach (IConfigurationView *view, views) {
11         m_stackedWidget->addWidget(view);
12         m_comboBox->addItem(view->name());
13     }
14
15     // if no implementation found, hide the implementation-chooser combobox
16     if (views.size() == 0)
17         m_comboBox->setVisible(false);
18 }
```

So far the only implementation of the `IConfigurationView` interface brings the `Network-Overview` plugin, but it is planned to provide several once (for example a view with the adjustment-resulting error ellipses).

Worker threads

The last substantial classes, forming the `Core` plugin are worker threads:

- **SolveNetworkTread** in which adjustment of the network takes place,

- **XsltTransformThread** in which XSLT transformation of the adjustment results from the input XML format to the XSL-defined output format takes place and
- **Xml2TxtTransformThread** which copies the functionality of the original `gama-xml2txt` command line utility.

Each of the listed classes is inherited from `QThread` exploiting its very useful `terminated()` signal, which is connected to the custom private slot with the following implementation:

```
1 void XSLTTransformThread::onTerminate()
2 {
3     s_logger->debug(tr("Deleting conversion thread."));
4     delete this;
5     s_logger->debug(tr("Conversion thread deleted."));
6 }
```

This ensures that the thread will delete itself automatically when its execution terminates. It is very convenient because while using the thread class, the programmer only needs create it on the heap, connect its *succeded* / *failed* signals to corresponding slots, call `start()` on it and that is all.

SolveNetworkThread takes the unique configuration name as the constructor parameter. In its `run()` implementation:

1. It tries to initialize a new database connection [19] (Qt does not support sharing the connection between threads).
   - If unsuccessful a `solvingFailed(QString, QString)` signal is emitted (first parameter stands for error title / category, second for error message) and the thread terminates.
   - If successful, an `Adjustment` instance is created, its `label()` signal is connected to the thread's `label()` signal and the `read_configuration()` and `xml()` methods are called.
2. If adjustment succeeds, `solved(QString)` signal will be emitted (providing the resulting XML string as the parameter). If the adjustment fails at any phase, exception is caught and the `solvingFailed(QString, QString)` signal will be emitted with the exception's title and text as its parameters.
3. If the database connection in step 1 was successfully established, it will be closed in the end.

XsltTransformThread takes input data and XSL definition as parameters of its constructor. In its implementation:

1. It creates an `QXmlQuery` with the `QXmlQuery::XSLT20` parameter saying we want to use it as a XSLT processor.
2. It sets input stream and XSL definition correspondingly, it also installs a message handler (`MessageHandler` class).

[19] Active database connection parameters are obtained via the `ICore::instance().activeDbParameters()` method invocation.

3. The `evaluateTo()` method is called.
4. If query's `isValid()` method returns true, `converted(QString)` signal will be emitted with the output format as its parameter.
5. Otherwise `conversionFailed(QString, QString)` is emitted.

Xml2TxtTransformThread takes input data, required language, encoding and angular units as parameters of its constructor. In its implementation:

1. It creates an instance of `Xml2Txt` class from `Gama` library, connect its `label()` signal to the thread's `label()` signal and calls the instance `txt()` method.
2. If exception is caught, `conversionFailed(QString, QString)` will be emitted with the exception's title and text as its parameters.
3. Otherwise `converted(QString)` signal will be emitted with the formatted output as parameter.

2.8.2 SQLEditor

`SQLEditor` plugin is the second of so-called *forced-enabled* (required) plugins. It brings, so far the only one, `Core` plugin's `IConfigurationChooserView` implementation (dedicated to the single-network editing). It offers:

- Dialog for editing / creating / deleting of the points entering the adjustment.
- Dialog for editing / creating / deleting of the observation clusters entering the adjustment.
- Dialog for choosing output format of the adjusted network (`ChooseOutputFormatDialog`).
- Dialog for TXT output parameters (`TxtOutputDialog`).

Edit dialogs and widgets

Editing dialogs for points and clusters, both inherits from the `IEditDialog` interface, which is basically just a `QDialog` with some of its events redefined, a `QTabWidget` and two important public functions `addTab()` and `setCategoryCount()`.

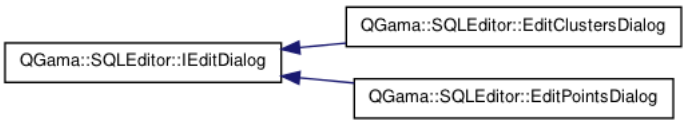

Figure 2.10: `IEditDialog` is a common ascendant of all editing dialogs.

All the logic is implemented inside the `EditDialogPageWidget` class which takes an `IEditWidget` pointer, a list of column numbers which should be hidden in the view, a list of column numbers which should be rounded in the view (by default to 4 decimal places) and an optional parent pointer as its constructor parameters.

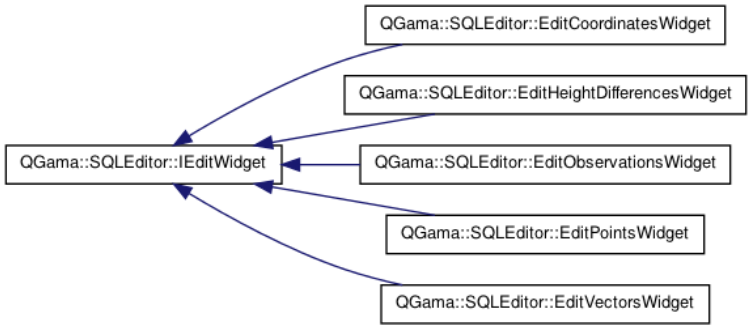

Figure 2.11: `IEditWidget` is a common ascendant of all editing widgets.

It also includes an important `updateSourceModel()` public slot updating the source model of the visible sort filter proxy model and a signal `rowCountChanged()`, which is emitted every time user add / delete some entry from the model.

In the figure 2.12 basic components of such an `EditDialogPageWidget` can be appreciated. In the upper section there is a `SortPanelWidget` and a `CustomTableView` enabling user to filter entries based on a regular expression matching values in a specified column (in a case sensitive or insensitive way).

The lower section is formed by an `ActionsPanelWidget` (including buttons for *Editing*, *Saving*, *Deletion* of the selected entry and an extensible context-aware action combo-box) accompanied by the concrete `IEditWidget` subclass (see figure 2.11).

`IEditWidget` provides only a common interface for all the widgets, through which it is integrated into the `EditDialogPageWidget` interactions.

A typical constructor of an `IEditDialog`'s subclass then looks like:

```
1  EditPointsDialog::EditPointsDialog(QWidget *parent) :
2      IEditDialog(parent)
3  {
4      // create pages
5      m_pagePoints = new EditDialogPageWidget(new EditPointsWidget(this),
6                                              QList<int>() << PointsModel::conf_id,
7                                              QList<int>() << PointsModel::x
8                                                           << PointsModel::y
9                                                           << PointsModel::z,
10                                             this);
11
12     // add pages
13     addTab(m_pagePoints, QIcon(ICON_NETWORK_POINTS), "Points");
14
15     // connect model changes
16     connect(&ICore::instance().dataManager(),
17             SIGNAL(restOfTheModelsInitialized()),
18             this,
19             SLOT(updateModels()));
20     connect(m_pagePoints, SIGNAL(rowCountChanged(int)),
21             this, SLOT(updatePointsCount(int)));
22
23     // set title
24     setWindowTitle(tr("Edit Points"));
25 }
```

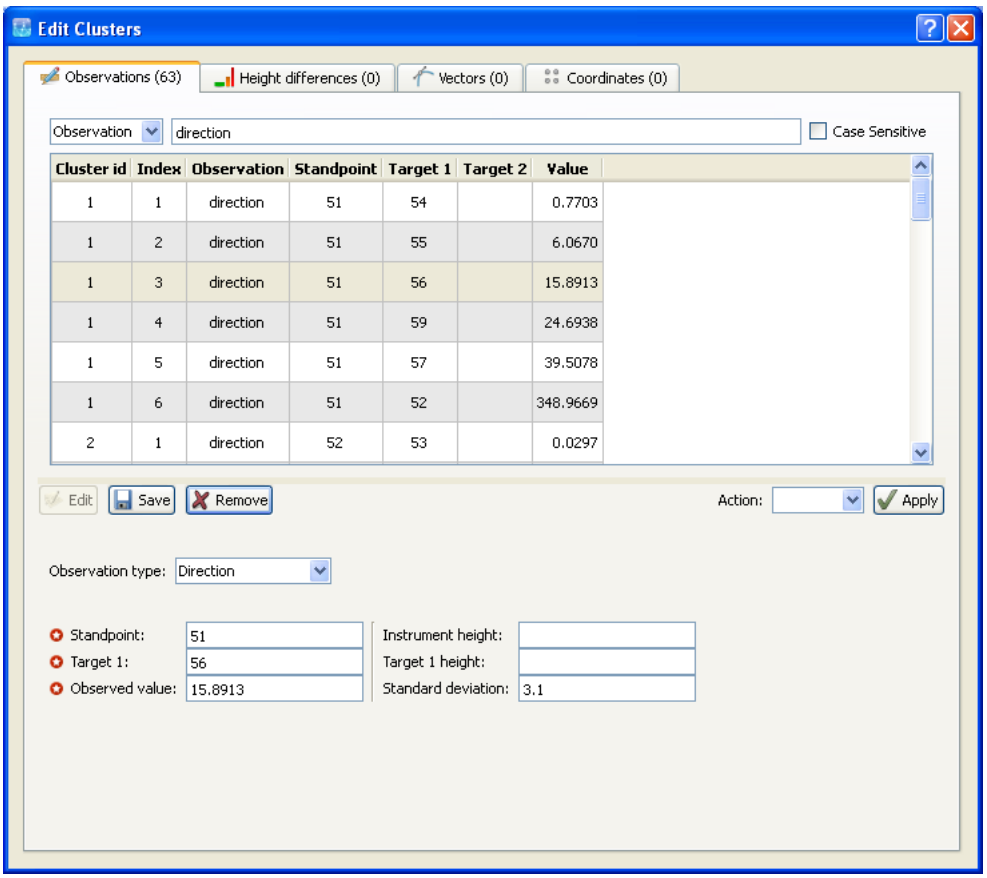

Figure 2.12: An example of the implementation of `IEditDialog`.

Several important actions should be noticed in the code listed above:

- the creation of a tab for the points model at line 5,
- adding this tab to the tab widget's stack at line 13,
- connecting the `DataManager`'s model changes to the slot which takes care of updating the corresponding model explicitly at line 16,

```
1  void EditPointsDialog::updateModels()
2  {
3      m_logger->debug(tr("Updating points model."));
4      m_pagePoints->updateSourceModel(ICore::instance().dataManager().
           pointsModel());
5  }
```

- connecting the `EditDialogPageWidget`'s `rowCountChanged(int)` signal to the `updatePointsCount(int)` slot at line 20.

```
1  void EditPointsDialog::updatePointsCount(int count)
2  {
3      setCategoryCount(0, count);
4  }
```

SingleNetworkConfigurationNavigationWidget

Nevertheless, the most important part of the `SQLEditor` plugin is `SingleNetworkConfigurationNavigationWidget`. It is a global crossroad offering the functionalities of this plugin. It contains the corresponding slots where the adjustment and conversion worker threads are started and is a parent for all of the editing dialogs.

2.8.3 NetworkOverview

`NetworkOverview` is the last plugin implemented so far. It does not have ambitions to provide complete interaction between the `Core` plugin's data models and graphical features, so far the only thing it does is to display a simple network overview (with zooming and printing features) once the configuration is loaded. An example is provided in the figure 2.9.

Network scene, view and items

The implementation is based on the Qt's `QGraphics Scene / View` framework. In practice it implements also the MVC pattern, but this time composed by a `QGraphicsScene` (containing `QGraphicsItem`s as entities) and several synchronized yet independent `QGraphicsView`s (which could be transformed - scaled, rotated, moved, etc.). An excellent starting point is Qt Documentation:

`http://developer.qt.nokia.com/doc/qt-4.8/graphicsview.html`.

`NetworkOverview` plugin is composed by the following classes:

NetworkScene	Most important class, contains pointers to the `Core` data models, draws background grid and points and observations once the models are filled.
NetworkView	`IConfigurationView` subclass, containing `QGraphicsView` and features for zoom in / out and printing the scene.
PointItem	`QGraphicsItem` subclass for representing points entering the adjustment (*adjusted* points are painted in red, *constrained* in blue, *fixed* in green and unknown types in yellow). `PointItem` is selectable with the `hover effect` defined.
MeasurementItem	`QGraphicsItem` subclass for representing the measurements between the points with known coordinates.

Interaction with Model / View framework Unfortunately in Qt there is no direct way how to interconnect the `Model / View framework` with `QGraphics Scene / View` framework. Theoretically it should be sufficient just to interconnect the following signals & slots between model and scene subclass (and in the slots create / edit / delete the `QGraphicItem`s correspondingly) [20] :

- connect(model, SIGNAL(`modelReset()`), scene, SLOT(`reset()`)),
- connect(model, SIGNAL(`layoutChanged()`), scene, SLOT(`layoutChanged()`)),

[20] As pointed out Joachim Schiele at his blog: `http://invalidmagic.wordpress.com/2010/10/05/qgraphicsscene-used-as-a-qabstractitemview-iii/`.

- connect(model, SIGNAL(`rowsInserted(const QModelIndex&, int, int)`), scene, SLOT(`rowsInserted(const QModelIndex&, int, int)`)),
- connect(model, SIGNAL(`rowsAboutToBeRemoved(const QModelIndex&, int, int)`), scene, SLOT(`rowsAboutToBeRemoved(const QModelIndex&, int, int)`)),
- connect(model, SIGNAL(`dataChanged(const QModelIndex&, const QModelIndex&)`), scene, SLOT(`dataChanged(const QModelIndex&, const QModelIndex&)`)).

And vice-versa if there would be need to provide also a way to add / edit / delete entries from within the graphical view. Adding support for this features is beyond the scope of this thesis.

2.9 Known issues

This is a list of known issues, which have to be repaired in the future releases:

- XSLT transformation with `QXmlPatterns` does not work with Qt version higher than 4.7.3. Reported at Nokia's bug list: `https://bugreports.qt.nokia.com/browse/QTBUG-22076`.
- Crash of `QGama::Xml2Txt` under Linux complaining about *"Invalid read of size 8"* from inside the `GNU Gama` library.
- Although application translator (including localized strings for all of the components - libraries, plugins) is successfully installed, only strings from the main application are translated.

2.10 Features to be implemented

A list of features to be implemented in the future releases.

- More interactive `NetworkOverview` plugin.
- Display results of the adjustment in a separate model, but similar views/dialogs as the input data - display the correspondent relations between adjusted and not adjusted values.
- Add plugin for displaying error ellipses.
- Complete online help pages and improve source code documentation for Doxygen.
- Rewrite models to be asynchronously filled from the worker thread.
- Solve the problematic of rounding the decimal values in editing widgets.
- Implement *Edit* `->` *Preferences* dialog.
- Prepare `.deb` package with Linux binaries.
- Redesign translation files structure to allow plugins bring their own translation files.

Epilogue

The objective of this thesis was to create a user-friendly, object-oriented graphical and portable interface for *GNU Gama*'s computational library for adjusting the local geodetic network. Author believes that was fulfilled.

QGama application as a powerful database front-end working above the *GNU Gama*'s SQL schema was created. It offers the standard features of the original `gama-local` console application and brings several new ones like HTML format of the adjustment results or a graphical overview of the network being adjusted.

From the developer point of view, *QGama* was written in a significantly modular manner – almost every feature is a plugin or shared library. This allows both: the third-party developers to contribute the application with an extension specific to their particular needs or very ease change of e.g. editing dialogs appearance without having to deal with the rest of the application logic.

Although many features remain to be properly tested or are currently provided only as a prove of concept, author believes *QGama* is ready to be deployed and tested by its first users. I hope it will not last long and that *QGama* would start to be used in both: the educational process at our faculty and a common surveyor's praxis.

Author was trying to write this paper as a handbook for any possible advanced Qt developer who would like to join the *QGama* development. He explained all the important features step by step and thus it should not be difficult for any possible follower to contribute.

Unfortunately the scope of this thesis did not allow the author to cover and explain all of the topics which would also deserve to be included. Either he refers to the recapitulation of the Qt way of C++ and its non-standard building process (including meta-object compiler, user-interface compiler, resource compiler), meta-object and properties system (Qt answer to the reflection pattern), signal-slots mechanism or some advanced Qt features like: smart pointers, threading, model / view framework, graphics scene / graphics view framework, undo framework, unit testing, problematic of asynchronous database access etc. The involved design patterns and tools used during the development phase (git, valgrind memory leak checker) would deserve their corresponding chapters.

Although this thesis is author's final work of the master programme, he would like to continue on the *QGama* development also in the future, because he is not unconcerned about its faith.

Bibliography

[1] BLANCHETTE, Jasmin; SUMMERFIELD, Mark: *C++ GUI Programming with Qt 4, 2nd edition*, Prentice Hall, 2006

[2] THELIN, Johan: *Foundations of Qt Development*, Apress, 2007

[3] SUMMERFIELD, Mark: *Advanced Qt Programming - Creating Great Software with C++ and Qt 4*, Prentice Hall, 2010

[4] EZUST, Alan; EZUST, Paul: *An Introduction to Design Patterns in C++ with Qt 4*, Prentice Hall, 2007

[5] MOLKENTIN, Daniel: *The Book of Qt 4 - The Art of Building Qt Applications*, Open Source Press GmbH, 2007

[6] GAMMA, E. et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 2004

[7] ECKEL, Bruce: *Myslíme v jazyku C++, knihovna programátora*, Grada Publishing, 2000

[8] ECKEL, Bruce; ALLISON, Chuck: *Myslíme v jazyku C++, 2. díl knihovna zkušeného programátora*, Grada Publishing, 2006

[9] BRADLEY, Neil: *XML, kompletní průvodce*, Grada Publishing, 2000

[10] HOLZNER, Steven: *XSLT, příručka internetového vývojáře*, Computer Press, 2002

[11] KOSEK, Jiří: *XSLT v příkladech* [online], last update 2004 [quoted 2010-05-01], accessible from WWW: `http://www.kosek.cz/xml/xslt/`,

[12] ČEPEK, Aleš: *GNU Gama Project Homepage* [online], last update January 20, 2009 [quoted 2011-12-13], accessible from WWW: `http://www.gnu.org/s/gama/`,

[13] NOVÁK, Jiří: *Bachelor's thesis: Object - Oriented GUI for GNU Gama*, Department of Mapping and Cartography, Faculty of Civil Engineering, Czech Technical University in Prague, 2010

[14] PETRÁŠ, Václav: *Bachelor's thesis: Support of SQLite database in program Gama-local*, Department of Mapping and Cartography, Faculty of Civil Engineering, Czech Technical University in Prague, 2011

[15] CREATE LOGIC: *Writing Qt Creator Plugins (Beta)* [online], last update September 11, 2009 [quoted 2011-12-13], accessible from WWW: `http://www.vcreatelogic.com/downloads/files/Writing-Qt-Creator-Plugins.pdf`,

[16] WIKIPEDIA.org: *Valgrind* [online], last update December 8, 2011 [quoted 2011-12-17], accessible from WWW: `http://en.wikipedia.org/wiki/Valgrind`,

Appendix A

QGama 1.0.0 user guide

A.1 Installation

For Windows platform an installer which could be downloaded from the project's homepage is available.

`http://geo102.fsv.cvut.cz/trac/qgama`.

For Linux so far does not exist another way than compile the source code [1] as described in the section 2.3.

Installation is simple:

1. Download the `qgama_1.0.0_setup.exe` from the project's homepage.
2. Double click on it, a language-selector will be displayed. Confirm the selection with the `Ok` button.

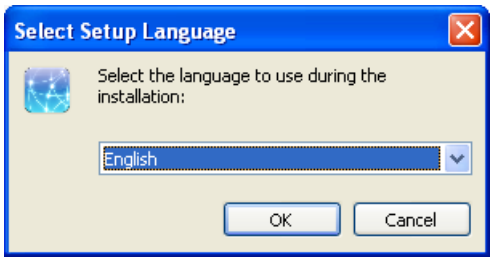

Figure A.1: Installation process - select setup language.

[1] Although a creation of distributable `.deb` binary package is planned.

3. Wizzard's welcome screen is displayed, continue with clicking on the `Next` button.

Figure A.2: Installation process - welcome screen.

4. Read the licence (GNU GPL v3) and accept it by clicking on the `Next` button.

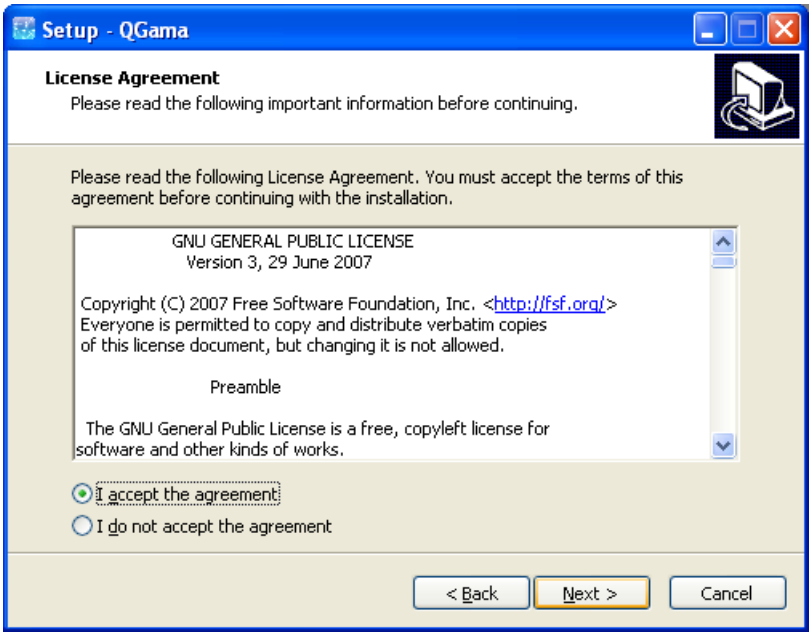

Figure A.3: Installation process - license agreement.

5. Select the destination directory. By default program is installed into `C:\Program Files\QGama`. Confirm with the `Next` button.

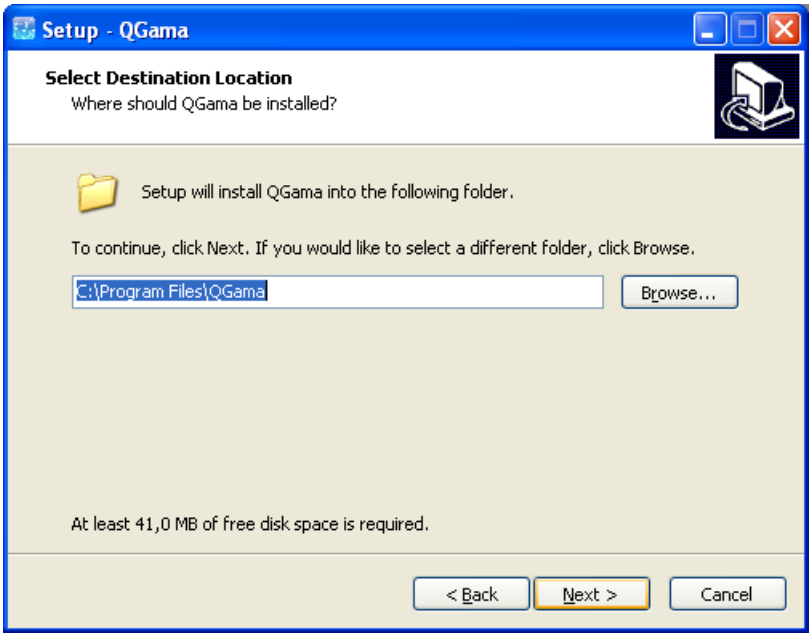

Figure A.4: Installation process - select destination location.

6. Select under which folder the shortcut in *Start menu* should be created. By default is is `QGama`. Confirm with the `Next` button.

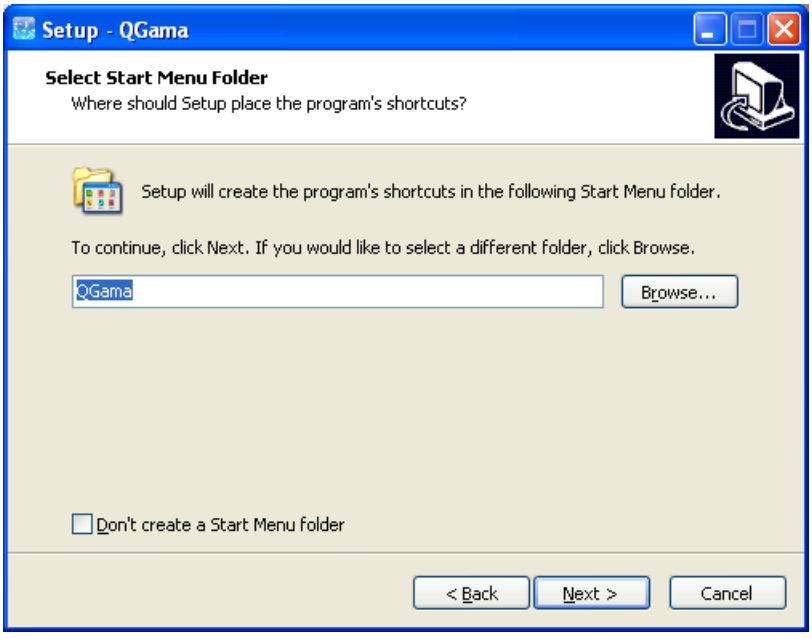

Figure A.5: Installation process - select start menu folder.

7. Check whether to create *desktop* and / or *quick launch* icons. By default both are unchecked. Confirm with the `Next` button.

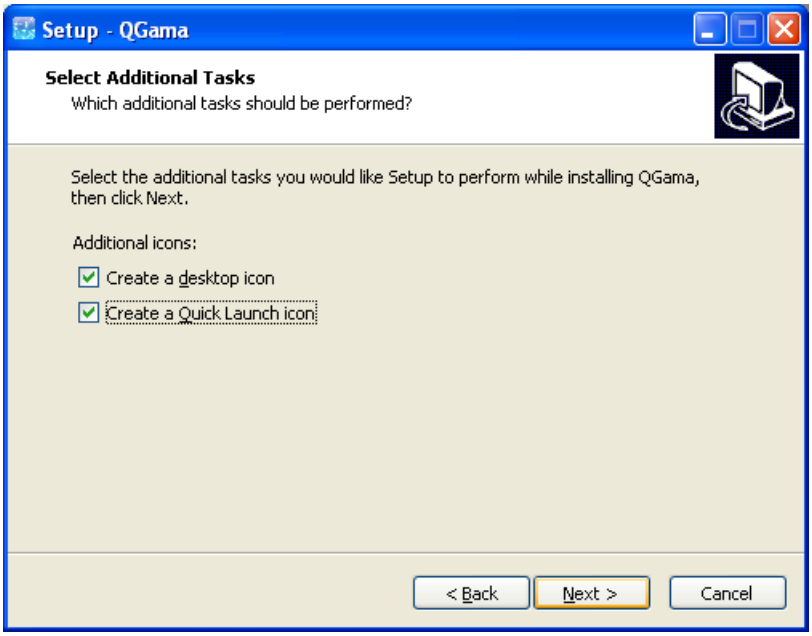

Figure A.6: Installation process - select additional tasks.

8. Recapitulation screen, confirm the installation by clicking on the `Install` button or return to correct the selection by clicking on the `Back` button.

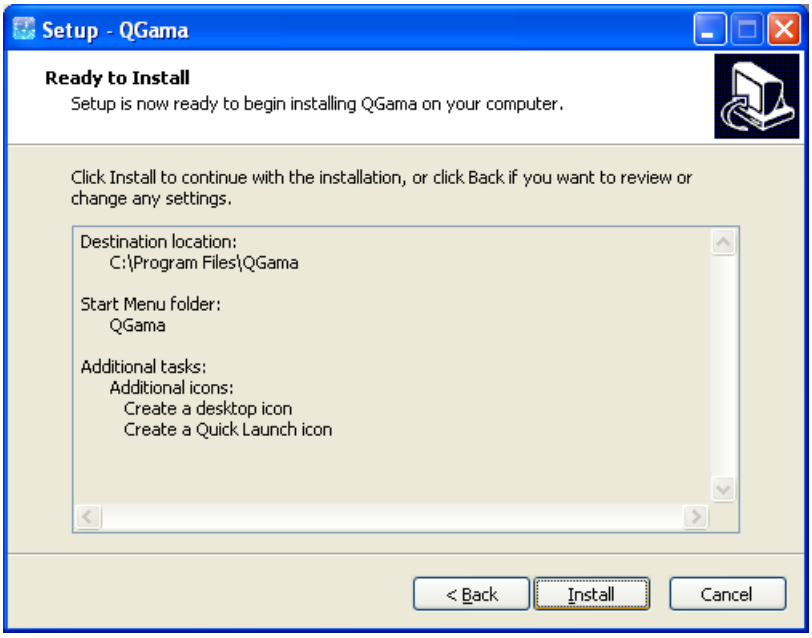

Figure A.7: Installation process - ready to install.

9. Wait until the installation completes with copying files.

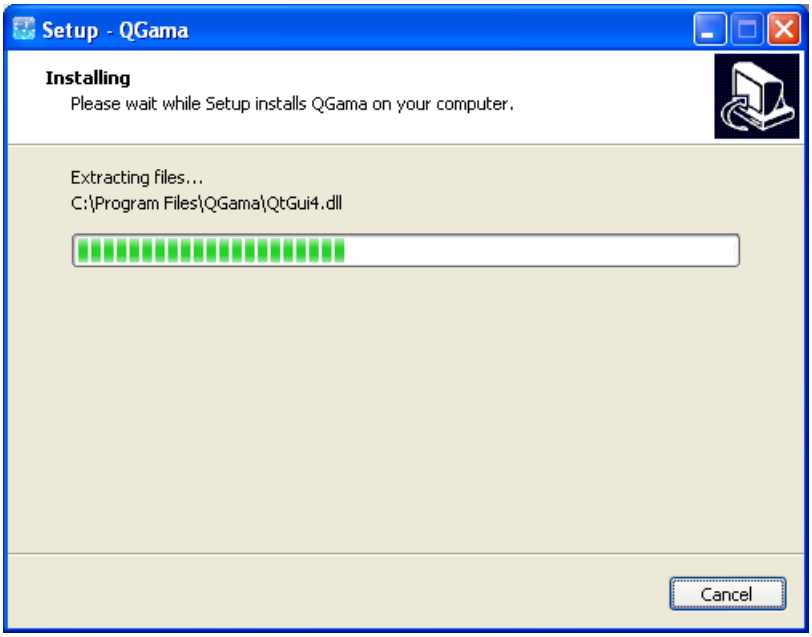

Figure A.8: Installation process - installing.

10. Installation was successful, check or uncheck the option to launch `QGama` immediately and click on the `Finish` button.

Figure A.9: Installation process - completed.

A.2 Defining new connection

On every `QGama`'s startup, the *Recent connections* dialog is opened. As this is the first application run, there are no connections defined yet.

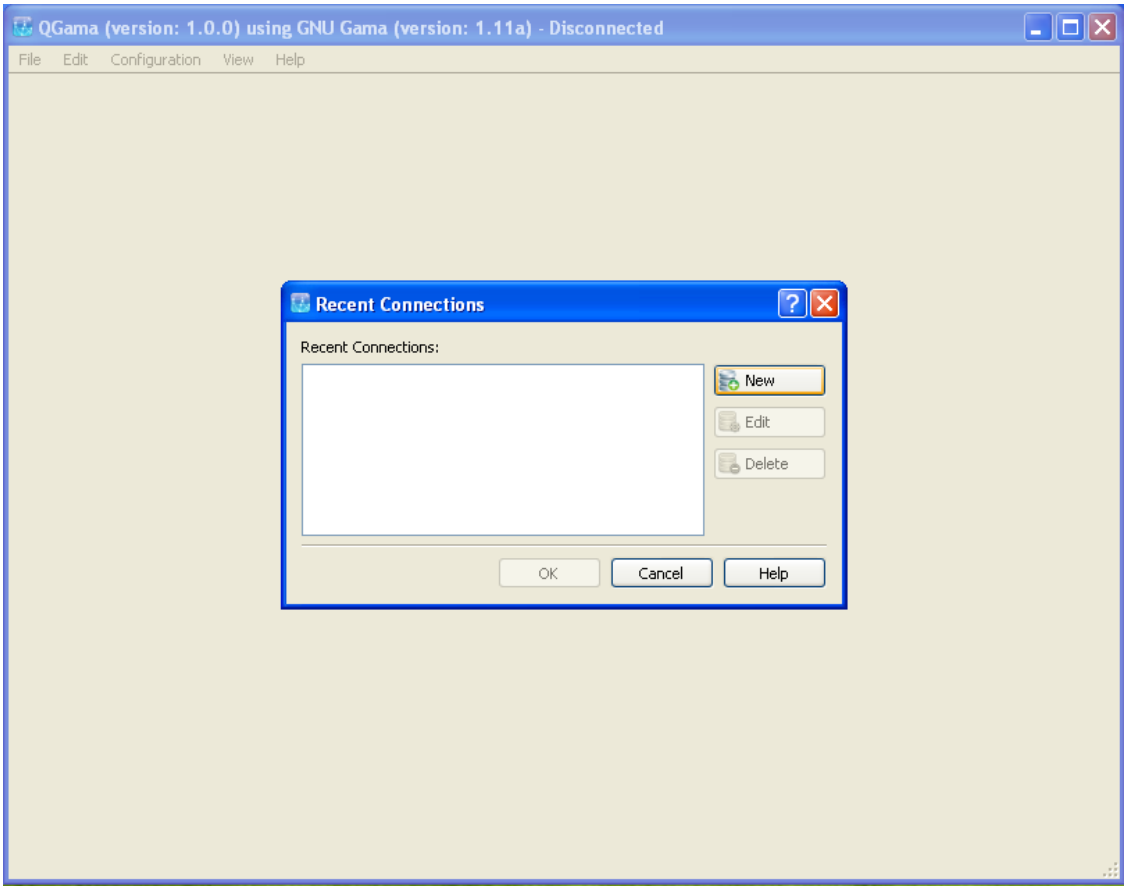

Figure A.10: `QGama`'s startup screen.

Recent connections dialog serves for creating, editing or deleting database connections. For defining a new one, the following steps should be followed:

1. Click on `New` button on the right.
2. *Create or edit connection* dialog is opened.
   - Depending on the Qt's *SQL driver plugins* found on the system [2] (`QGama` is distributed only with the `SQLite` support) the corresponding driver choices are shown in the *Server Type* combo-box.
   - Depending on the database driver chosen, different pieces of information are required to be entered.

[2] For information how to add support for your favourite database, Qt's documentation should be consulted:
`http://developer.qt.nokia.com/doc/qt-4.8/sql-driver.html`.

- For *SQLite* only *file name* and *label* fields are required.

Figure A.11: *Create or edit connection* dialog - SQLite.

- For the rest of the relational database management systems *hostname*, *port*, *database*, *username*, *password* and *label* are required.

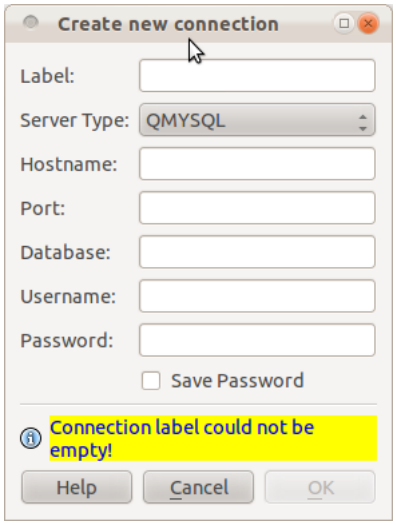

Figure A.12: *Create or edit connection* dialog - MySQL.

3. Let the author demontrate a connection using the `QSQLITE` driver. A *label* has to be filled and either  icon for creating a new file (figure A.13) or  for opening an existing file (there is an example set of configurations distributed together with `QGama` which can be found at `C:\Program Files\QGama\examples\readdemo.db`) invoked.

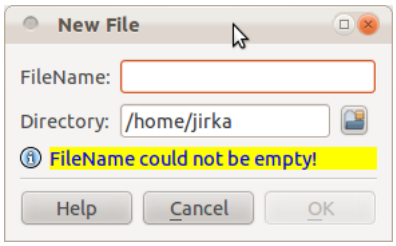

Figure A.13: *Create new file* dialog.

4. Confirm the selection by clicking on the `Ok` button.

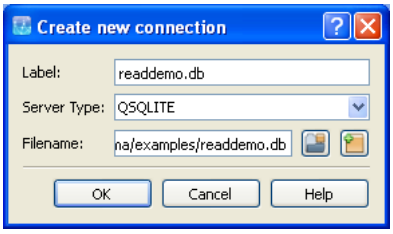

Figure A.14: *Create or edit connection* dialog - confirmation.

5. Connection is tested and notification about the result appears.

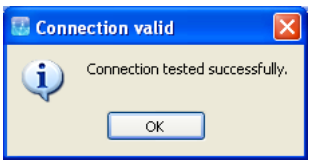

Figure A.15: Connection tested successfully.

6. If creating a new *SQLite* database, a prompt to confirm creation of `GNU Gama`'s SQL schema tables appears. Click on the `Yes` button to continue.

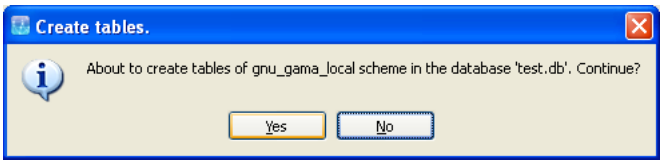

Figure A.16: Create tables of the `GNU Gama` SQL schema.

7. Progress bar informs about which tables are being created.

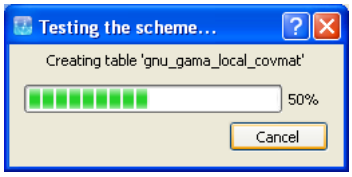

Figure A.17: Create tables progress bar.

8. Connection was successfully added, it can be seen now in the *Recent connections* dialog list.

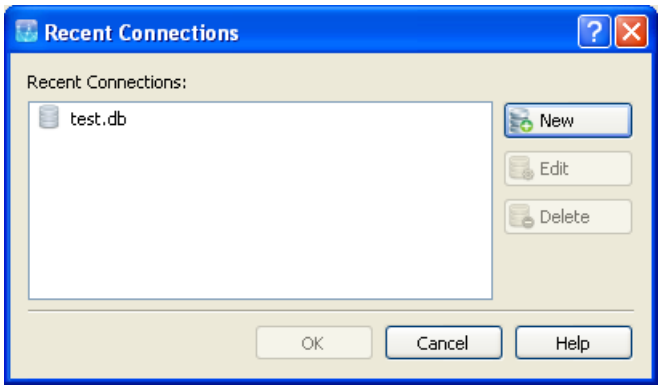

Figure A.18: New connection successfully added.

9. Select the recently defined configuration and click on the `Ok` button. *Choose edit mode* dialog will be shown.

A.3 Creating configuration

1. In the *Choose edit mode* dialog either an existing configuration can be edited or deleted or a new one created. If creating a new one, it will be added automatically to the list with name *Untitled configuration*.

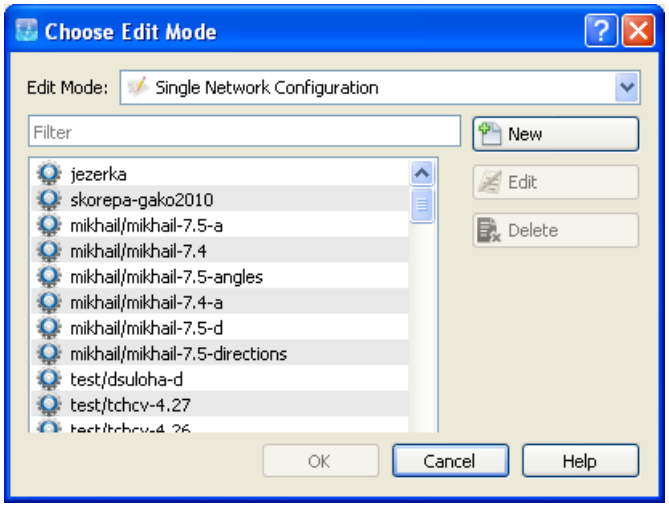

Figure A.19: *Choose edit mode* dialog.

2. *Choose edit mode* dialog offers also a filtering option, it is thus easy to locate the required configuration and edit it.

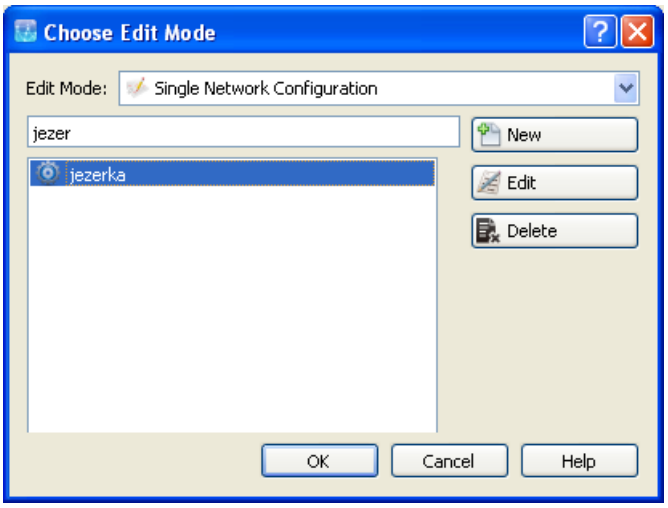

Figure A.20: Filtering configuration.

3. *Edit configuration* dialog is separated into 4 pages. First one allows us to specify configuration name and description.

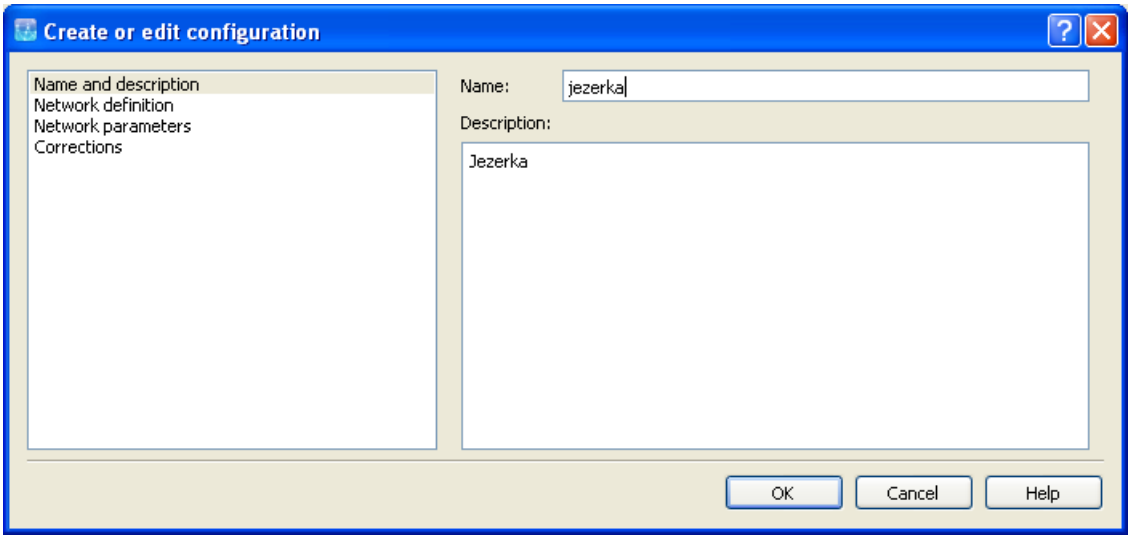

Figure A.21: *Edit configuration* dialog - name and description.

4. Second page contains network definition parameters (*a-priori standard deviation*, *confidence probability*, *tolerance for gross term identification*, *numerical algorithm* to be used, whether *constraint coordinates* should be updated and which *standard deviation* should be taken).

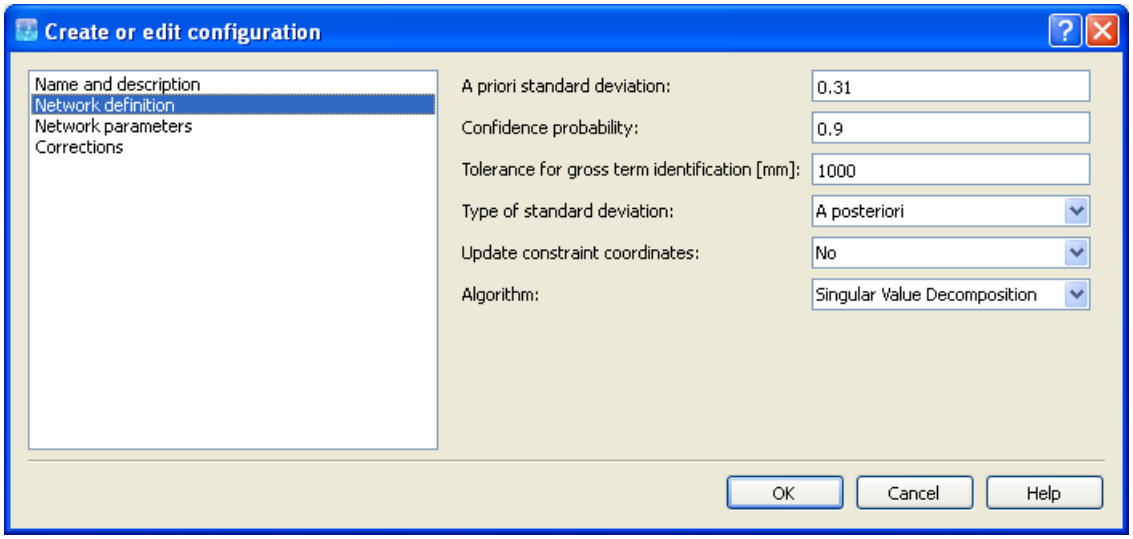

Figure A.22: *Edit configuration* dialog - network definition.

5. Third page contains network parameters (*orientation of the axes*, *angular units* to be used, *observation direction* and observation *epoch*).

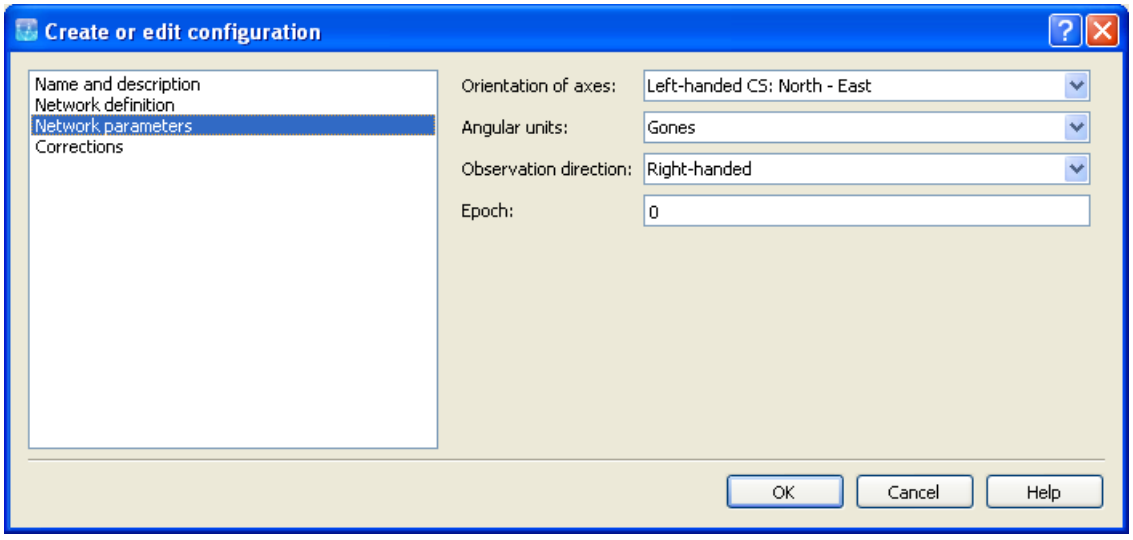

Figure A.23: *Edit configuration* dialog - network parameters.

6. Fourth page contains optional corrections which could be used when observed vertical and / or zenith angles need to be transformed into the projection plane. Implicit value for latitude is 45 degrees (50 gons) and implicit ellipsoid is *WGS84*.

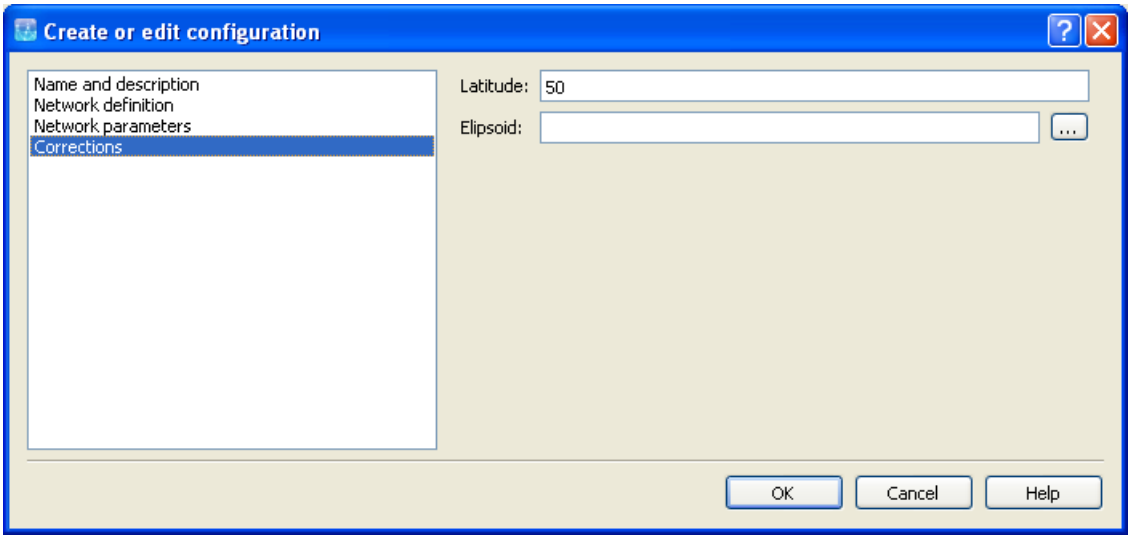

Figure A.24: *Edit configuration* dialog - corrections.

7. A different ellipsoid to be used in the corrections can be selected.

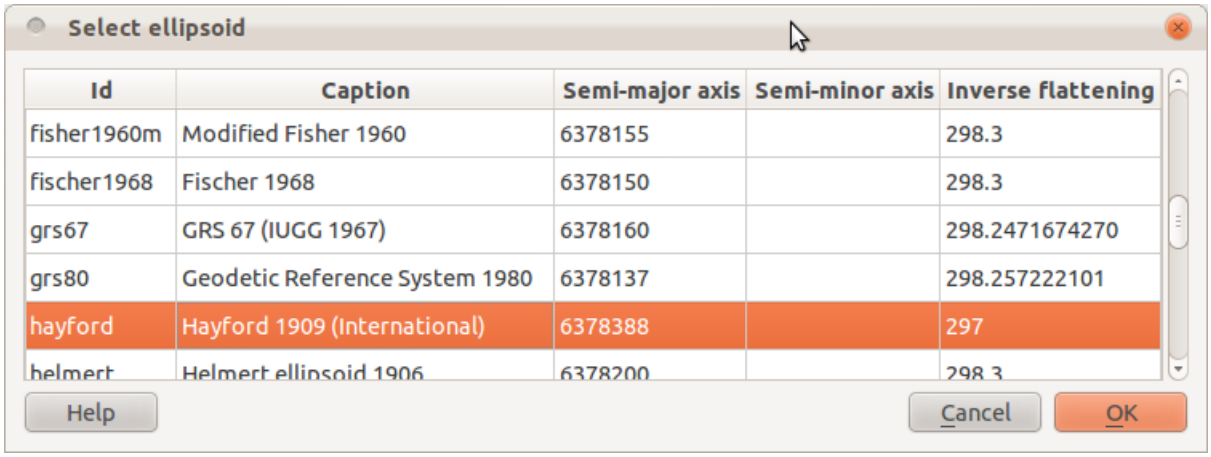

Figure A.25: *Edit configuration* dialog - select Ellipsoid.

A.4 Editing points

1. Dialog for editing points conforming the network configuration is accessible from the main window's *navigation panel*.

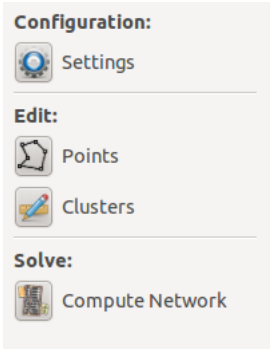

Figure A.26: *QGama*'s navigation panel.

2. *Edit points* dialog is formed by a filtering widget, simplified table view, actions panel and editing widget. Select the row to edit and click on the *Edit* button.

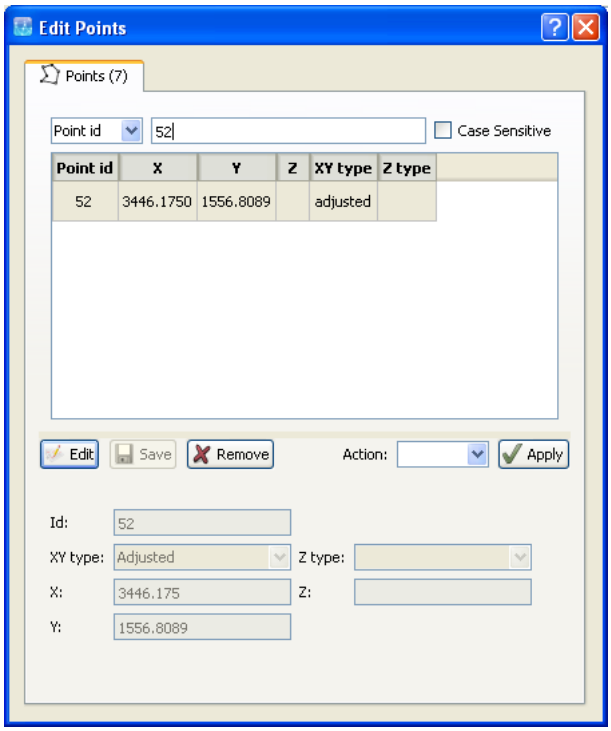

Figure A.27: *Edit points* dialog - overview.

3. Editing widget fields got enabled and its values can be changed now. To save the changes made, click on the `Save` button.

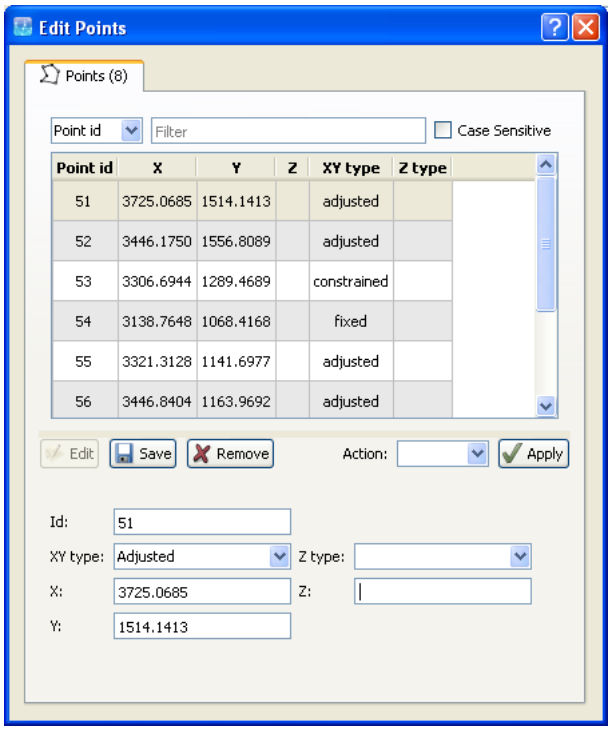

Figure A.28: *Edit points* dialog - editing an entry.

4. While deleting an entry, the confirmation is needed.

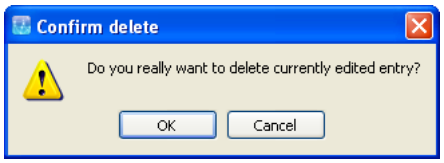

Figure A.29: *Edit points* dialog - deleting an entry.

5. Adding a new entry (appended in the end, inserted before / after some specific entry) should be accessible through the *Action* combo-box. Unfortunately it was not implemented yet in the time of the writing this paper, so the author could not provide a corresponding screenshot here.

A.5 Editing clusters

1. Dialog for editing clusters of observations shares the same structure as the *Edit points* dialog. There are tabs for *Observations*, *Height Differences*, *Vectors* and *Coordinates*. Corresponding editing widgets shares the philosophy of:
   - having the corresponding terms on the same line and
   - mark the required terms with ⭘.
2. Adding a new entry (appended in the end of the current cluster, appended before / after some specific entry within the current cluster, adding a new cluster) and editing widget of the cluster's *variance - covariance matrix* should be accessible through the *Action* combo-box. Unfortunately it was not implemented yet in the time of writing this paper, so the author could not provide a corresponding screenshot here.
3. Screenshots of all the tabs *Edit clusters* dialog offers follows.

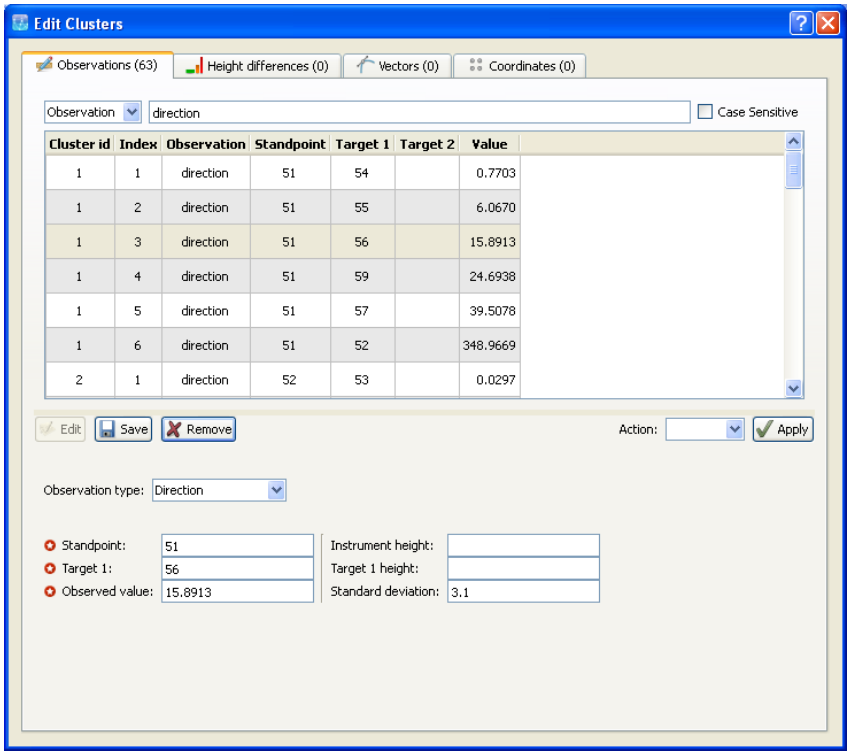

Figure A.30: *Edit clusters* dialog – *Observations* tab.

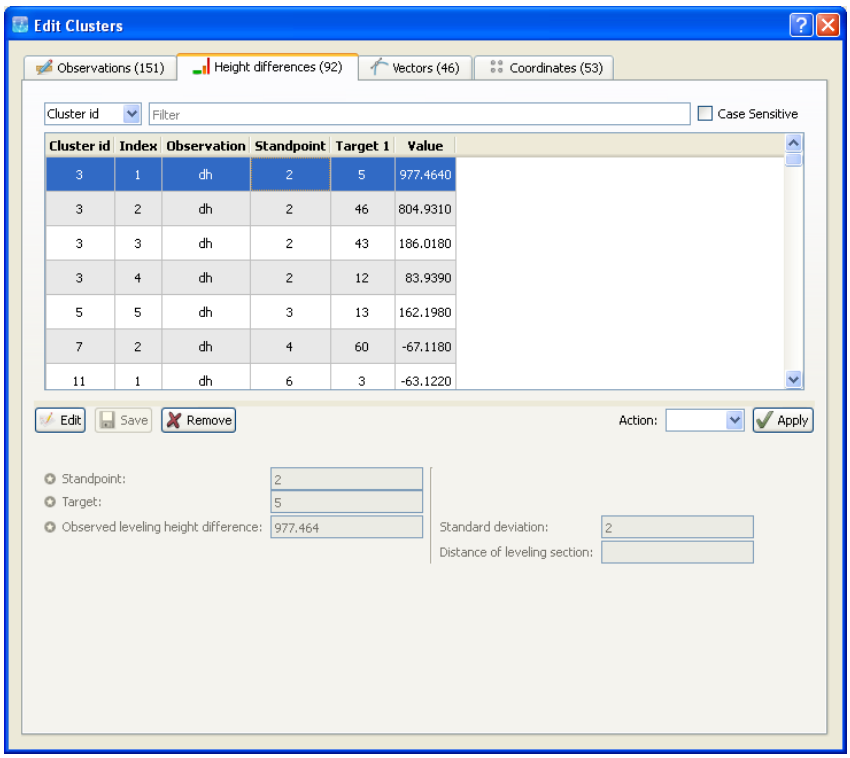

Figure A.31: *Edit clusters* dialog – *Height differences* tab.

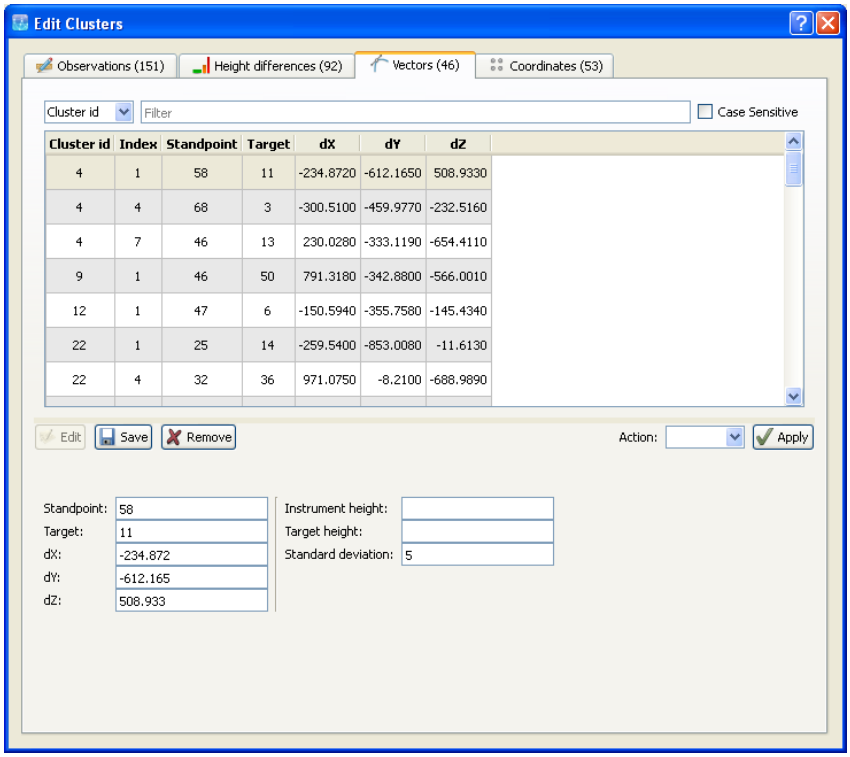

Figure A.32: *Edit clusters* dialog – *Vectors* tab.

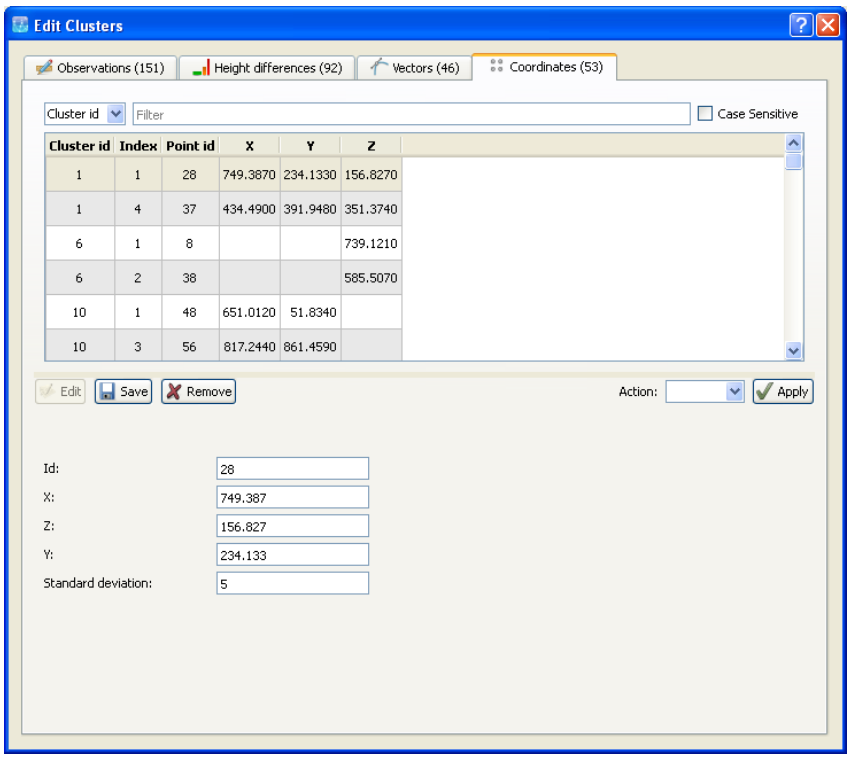

Figure A.33: *Edit clusters* dialog – *Coordinates* tab.

A.6 Adjusting the network

1. Start the network adjustment by clicking on the *solve* icon in the navigation panel.

2. Progress bar informing about the calculation progress appears.

A.7 Generating results in different formats

1. On adjustment success, a dialog for choosing a desired output format is displayed. On adjustment failure an error dialog informing of what went wrong is displayed.

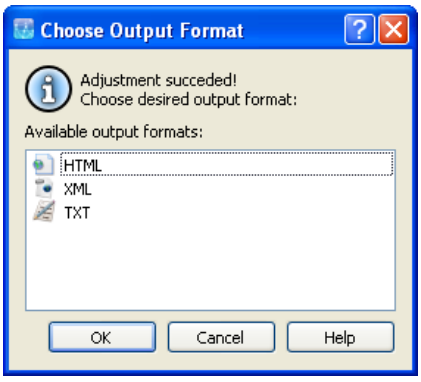

Figure A.34: *Choose output format* dialog.

2. Select which formats to generate and click on the `Ok` button.

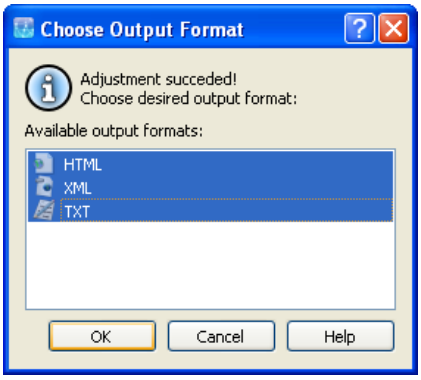

Figure A.35: *Choose output format* dialog – multiple selection.

3. Examples of adjustment results in HTML, XML and TXT formats follow.

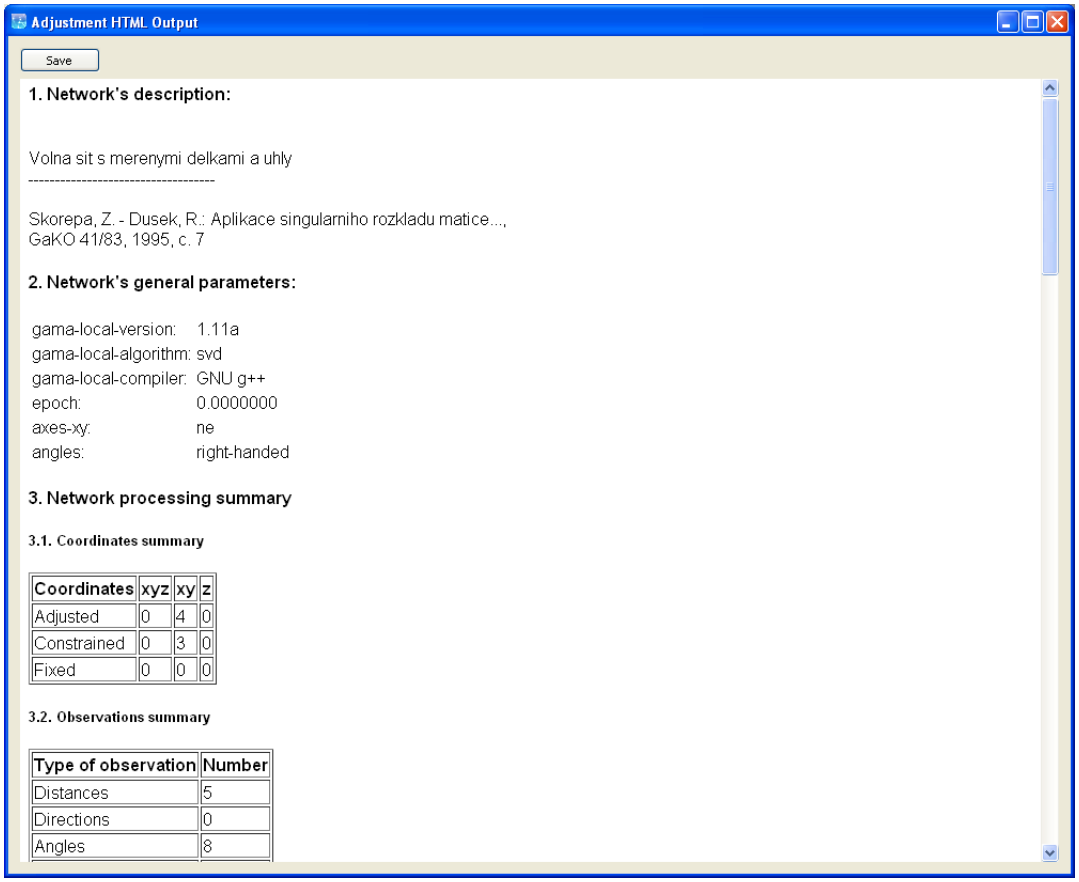

Figure A.36: Adjustment results – HTML output.

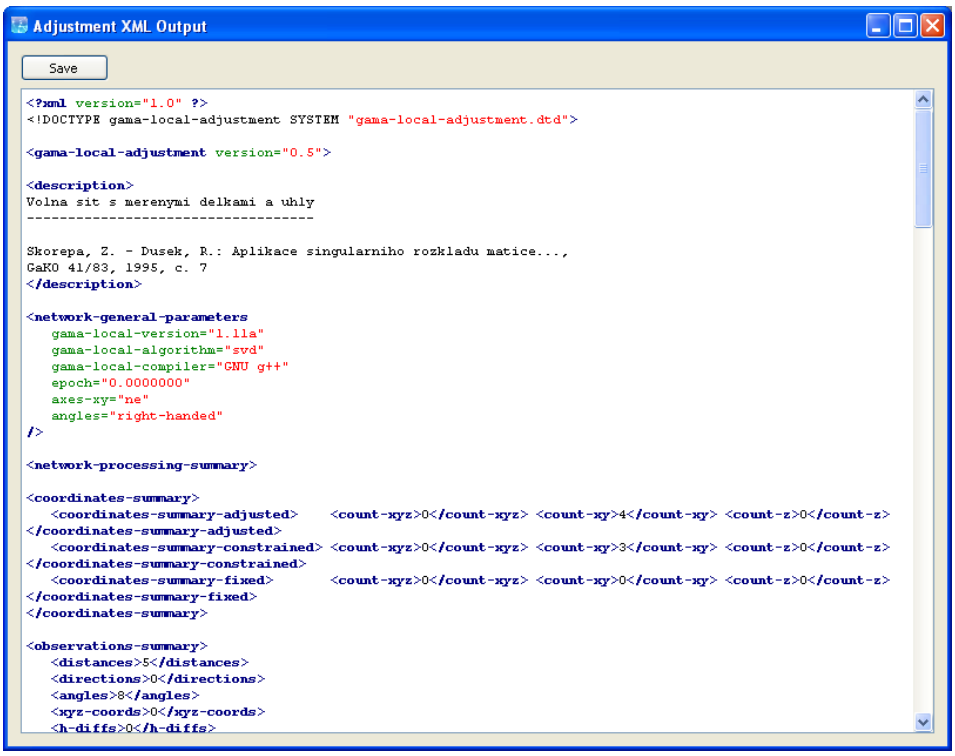

Figure A.37: Adjustment results – XML output.

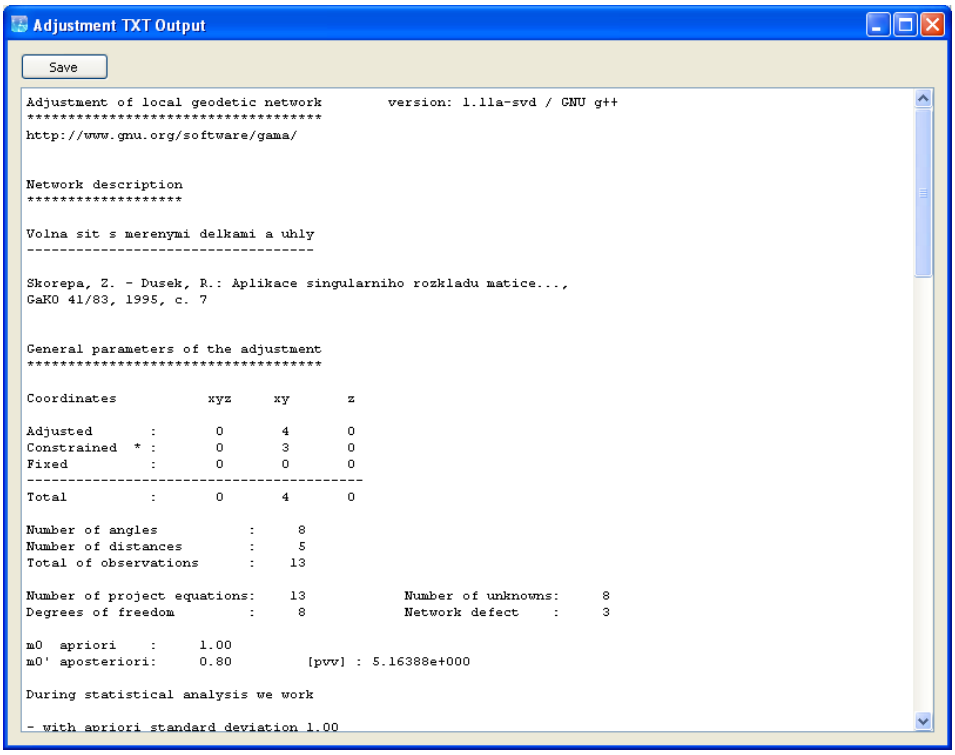

Figure A.38: Adjustment results – TXT output.

A.8 Graphical network overview

1. `NetworkOverview` plugin brings a trivial implementation of the graphical view on the network being adjusted. The view has zoom in / out features.

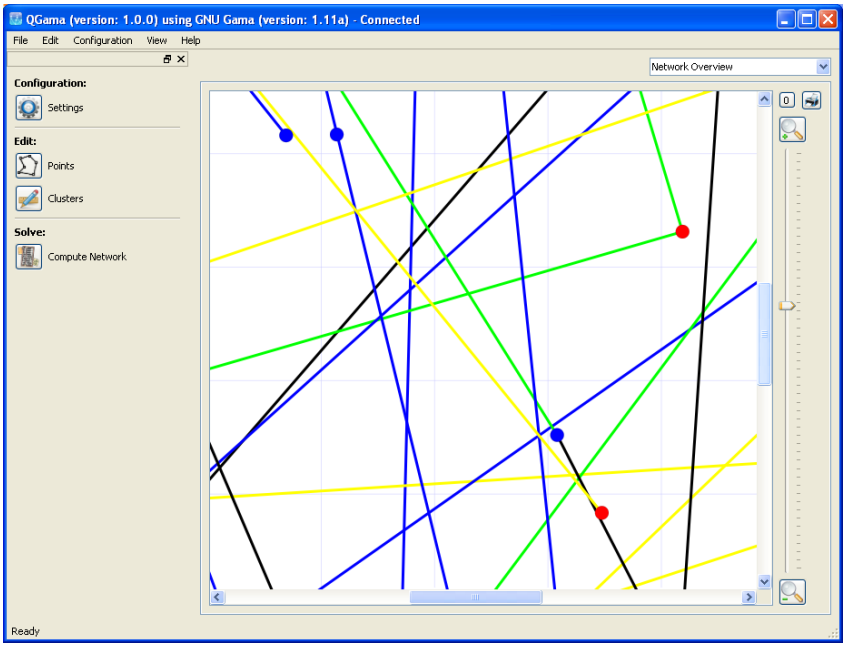

Figure A.39: *Network overview* – zoom in / out features.

2. View can be reset to its default extension (button `0`).

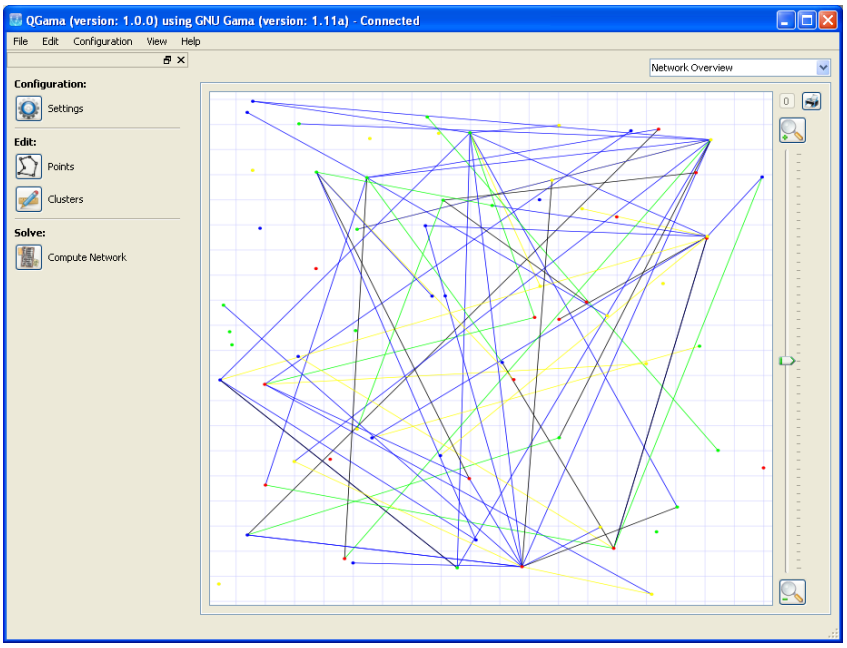

Figure A.40: *Network overview* – reset view.

3. Scene can be also printed (icon). An example of such printed PDF follows.

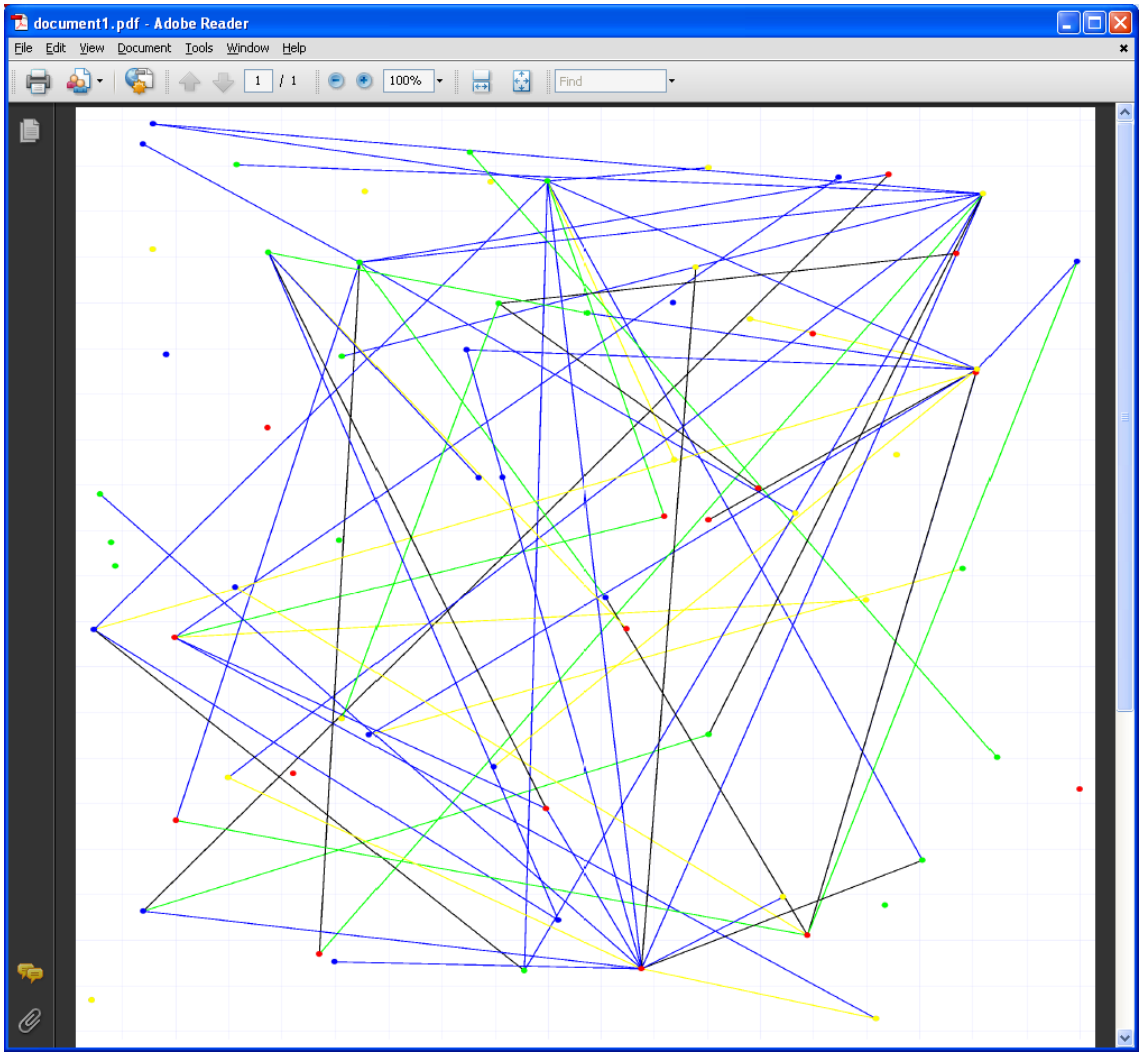

Figure A.41: *Network overview* – scene printed to PDF.

A.9 Uninstalation

1. Uninstallation process is very simple. Invoke the uninstaller from the *Start Menu*'s *QGama* folder.

Figure A.42: Uninstallation process – invocation of uninstaller.

2. Confirm the removal of all the `QGama`'s components.

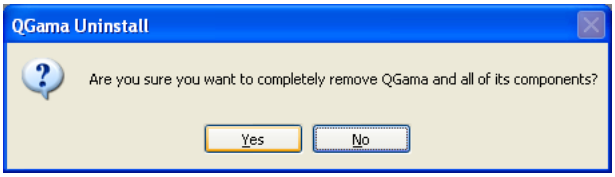

Figure A.43: Uninstallation process – uninstallation confirmation.

3. If everything goes well, a notification about the successful uninstallation is displayed.

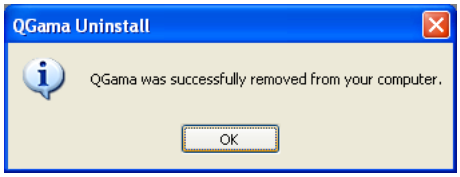

Figure A.44: Uninstallation process – successfully uninstalled.

Appendix B

GNU Gama SQL schema DDLs

```
/*
    GNU Gama -- adjustment of geodetic networks
    Copyright (C) 2010 Ales Cepek <cepek@gnu.org>, 2010 Jiri Novak
    <jiri.novak@petriny.net>, 2010 Vaclav Petras <vaclav.petras@fsv.cvut.cz>

    This file is part of the GNU Gama C++ library.

    This library is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 3 of the License, or
    (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 $
*/


create table gnu_gama_local_configurations (
   conf_id  integer primary key,
   conf_name varchar(60) not null unique,
   sigma_apr double precision default 10.0 not null check (sigma_apr > 0),
   conf_pr  double precision default 0.95 not null check (conf_pr > 0 and conf_pr
        <1),
   tol_abs  double precision default 1000 not null check (tol_abs > 0),
   sigma_act varchar(11) default 'aposteriori' not null check (sigma_act in ('
       apriori', 'aposteriori')),
   update_cc varchar(3) default 'no' not null check (update_cc in ('yes', 'no')),
   axes_xy  varchar(2) default 'ne' not null check (axes_xy in ('ne', 'sw', 'es',
        'wn', 'en', 'nw', 'se', 'ws')),
   angles   varchar(12) default 'right-handed' not null check (angles in ('left-
       handed', 'right-handed')),
   epoch    double precision default 0.0 not null,
```

```
35    algorithm varchar(12) default 'svd' not null check (algorithm in ('svd', 'gso'
          , 'cholesky', 'envelope')),
36    ang_units integer default 400 not null check (ang_units in (400, 360)),
37    latitude double precision default 50 not null,
38    ellipsoid varchar(20)
39 );
40
41 create table gnu_gama_local_descriptions (
42    conf_id  integer references gnu_gama_local_configurations (conf_id) on delete
          cascade,
43    indx     integer check (indx >= 1),
44    text     varchar(1000) not null,
45    primary key (conf_id, indx)
46 );
47
48 create table gnu_gama_local_points (
49    conf_id  integer references gnu_gama_local_configurations (conf_id) on delete
          cascade,
50    id       varchar(80),
51    x        double precision,
52    y        double precision,
53    z        double precision,
54    txy      varchar(11) check (txy in ('fixed', 'adjusted', 'constrained')),
55    tz       varchar(11) check (tz in ('fixed', 'adjusted', 'constrained')),
56    primary key (conf_id, id)
57 );
58
59 create table gnu_gama_local_clusters (
60    conf_id  integer references gnu_gama_local_configurations (conf_id) on delete
          cascade,
61    ccluster integer check (ccluster > 0),
62    dim      integer not null check (dim > 0),
63    band     integer not null,
64    tag      varchar(18) not null check (tag in ('obs', 'coordinates', 'vectors',
          'height-differences')),
65    check (band between 0 and dim-1),
66    primary key (conf_id, ccluster)
67 );
68 -- upper triangular variance-covariance band-matrix (0 <= bandwidth < dim)
69
70 create table gnu_gama_local_covmat (
71    conf_id  integer,
72    ccluster integer,
73    rind     integer check (rind > 0),
74    cind     integer check (cind > 0),
75    val      double precision not null,
76    primary key (conf_id, ccluster, rind, cind),
77    foreign key (conf_id, ccluster) references gnu_gama_local_clusters (conf_id,
          ccluster) on delete cascade
78 );
79
80 create table gnu_gama_local_obs (
81    conf_id  integer,
82    ccluster integer check (ccluster > 0),
```

```
83    indx      integer check (indx > 0),
84    tag       varchar(10) check (tag in ('direction', 'distance', 'angle', 's-
          distance', 'z-angle', 'dh')),
85    from_id   varchar(80) not null,
86    to_id     varchar(80) not null,
87    to_id2    varchar(80),
88    val       double precision not null,
89    stdev     double precision,
90    from_dh   double precision,
91    to_dh     double precision,
92    to_dh2    double precision,
93    dist      double precision, -- dh dist
94    rejected integer default 0 not null,
95    primary key (conf_id, ccluster, indx),
96    foreign key (conf_id, ccluster) references gnu_gama_local_clusters (conf_id,
          ccluster) on delete cascade,
97    check (tag <> 'angle' or to_id2 is not null),
98    check (tag = 'dh' or (tag <> 'dh' and dist is null))
99 );
100
101 create table gnu_gama_local_coordinates (
102    conf_id  integer,
103    ccluster integer check (ccluster > 0),
104    indx     integer check (indx > 0),
105    id       varchar(80),
106    x        double precision,
107    y        double precision,
108    z        double precision,
109    rejected integer default 0 not null,
110    primary key (conf_id, ccluster, indx),
111    foreign key (conf_id, ccluster) references gnu_gama_local_clusters (conf_id,
          ccluster) on delete cascade
112 );
113
114 create table gnu_gama_local_vectors (
115    conf_id  integer,
116    ccluster integer check (ccluster > 0),
117    indx     integer check (indx > 0),
118    from_id  varchar(80),
119    to_id    varchar(80),
120    dx       double precision,
121    dy       double precision,
122    dz       double precision,
123    from_dh  double precision,
124    to_dh    double precision,
125    rejected integer default 0 not null,
126    primary key (conf_id, ccluster, indx),
127    foreign key (conf_id, ccluster) references gnu_gama_local_clusters (conf_id,
          ccluster) on delete cascade
128 );
```