I2G

INTERGEO

# Deliverable Nº: D3.10

# i2g Common File Format Final Version

# The Intergeo Consortium

# June 2010

**Version:** version of August 24, 2010 1:38 P.M.


**Main Authors:**

Santiago Egido (Maths for More & WIRIS)
Maxim Hendriks (Technische Universiteit Eindhoven)
Yves Kreis (Université du Luxembourg & GeoGebra)
Ulrich Kortenkamp (PH Karlsruhe & Cinderella)
Daniel Marquès (Maths for More & WIRIS)



**Project co-funded by the European Community under the eContentplus Programme**

| | |
|---|---|
| **Project ref.no.** | ECP 2006 EDU 410016 |
| **Project title** | Intergeo - Interoperable Interactive Geometry for Europe |

| | |
|---|---|
| **Deliverable status** | submitted |
| **Contractual date of delivery** | M33 – June 2010 |
| **Actual date of delivery** | June $30^{th}$ 2010 |
| **Deliverable title** | i2g Common File Format Final Version |
| **Type** | Presentation |
| **Status & version** | submited version of August 24, 2010 1:38 P.M. |
| **Number of pages** | 44 |
| **WP contributing to the deliverable** | WP3 |
| **WP/Task responsible** | Daniel Marquès |
| **Authors** | Santiago Egido (Maths for More & WIRIS), Maxim Hendriks (Technische Universiteit Eindhoven), Markus Hohenwarter (Johannes Kepler Universität Linz & GeoGebra), Ulrich Kortenkamp (PH Karlsruhe & Cinderella), Yves Kreis (Université du Luxembourg & GeoGebra), Jean-Marie Laborde (Cabrilog & Cabri), Paul Libbrecht (DFKI GmbH), Daniel Marquès (Maths for More & WIRIS) |
| **EC Project Officer** | Krister Olson |
| **Keywords** | dynamic geometric system, file format, openmath, standard |
| **License** | This document is available under the license *Creative Commons Attributions Sharealike Germany 2.5* [Cre08] |

# Contents

# 1 Executive Summary

The present document is the specification of the Intergeo File Format, Intergeo File Format v1, as of June 2010.

It is emphasized that the Intergeo File Format will continue to grow; in order to maintain it, the Intergeo ASBL (Association Sans But Lucratif - non-profit organization) has been constituted. See Section 7 for a list of future improvements. Note that all expected changes are additions; with several implementations already working, it is felt that the basic structure of the file format is sound, and no essential modifications are planned. At the time of this writing, the file format is rich enough that it can represent almost all high-school level geometrical constructions.

This specification is the result of intensive collaboration between Dynamic Geometric System (DGS) software developers and experts, and aims at creating a file format that could serve as a standard in the DGS industry. Hence the interest in describing existing implementations and providing some help to new implementors.

The format is split into three specifications: the container, the file intergeo.xml and the symbol list. The container specification explains how all necessary files used to define a construction are bundled into a single ZIP file. It also contains the important file intergeo.xml, which is the core of the format and is explained in detail. It comprises three differentiated parts. The elements part declares all geometric objects. The constraints part provides the relationships of the objects and, thus, defines the dynamic (or interactive) behaviour. The display part describes the styles and how the geometric objects are drawn.

The XML schema that should validate any intergeo.xml file is presented with the skeleton of the elements, constraints and display part. The children of the elements, constraints and display part can be specified with their respective XML schema fragments. However, these fragments are not written in this document because they can be generated dynamically from the list of symbols. Thus, different templates are introduced according to the elements, constraints and display parts. Some additional restrictions are described without using XML schema.

The separation of the possible geometric elements and constraints from the file format implementation is done through the so called list of symbols. The complete list of symbols is included in Appendix A. Such a list is important because choosing correctly its content is crucial to achieve the successful interoperability between all DGS's.

**Differences with the previous version** The main improvement with respect to the version, described in Deliverable D3.6 [HKKM09], is the amount

of new symbols introduced in this version. New elements have been added, such as polygons and vectors. The list of display symbols has grown, so that common visual styles can be specified. Some new constraints have been added, too, most noticeably symmetries and translations. A mechanism for polymorphism is provided: families. Additional options have been added to the intersection constraints, so that now it is possible to specify whether an element such as a line segment has to be extended into a line in order to find intersection points.

# 2 Introduction

The Intergeo file format is a file format designed to describe any construction created with a Dynamic Geometry System (DGS). As a first application, the format can be used to interchange content between geometric software. At present, the format is restricted to the geometry in the plane, although it does not seem difficult to extend it, in the future, to the space.

Dynamic Geometry Systems (DGS) is a kind of software used to experiment with geometry. A construction, a drawing with geometric elements, is displayed to the user. But the most exciting part of a DGS is that it is possible to move some of the elements with the mouse pointer and the whole construction is recomputed while keeping predefined geometric relationships, which are the object of study. Although the origin of DGS is geometry, they can be applied to the study of other areas of mathematics or even subjects like, for example, physics.

A wide variety of DGS exists. Before this project, each system used incompatible proprietary file formats to store its data. Thus, most of the DGS makers have joined to provide a common file format that will be adopted either in the core of the systems or just as a way to interchange content.

The file format proposed in this document aims to be the convergence of the common features of the current DGS together with the vision of future developments and the opinion of external experts. In addition, we state the following objectives:

- The resulting file format will be adopted by the manufacturers that are authors of this specification.
- The files that satisfy this specification should be interchangeable between the different DGS.
- The format will be based on modern technologies.
- The format will be extensible. The use of name spaces and additional files will allow for capturing the flavour of the different DGS's, since they will be able to use their most representative features. To mention two examples, Dynamic Styling (see Subsection 7.3) could be achieved by adding files to the container so that some DGS might retrieve from them the information it requires; and Number Representation (see Subsection 7.8) could be achieved by those DGS capable of symbolic manipulations by adding scripts. These additions would interfere with the workings of other DGS's.

Providing a file format for DGS is a task related, but not equal, to express geometric constructions in general. However, some additional issues arise. For example, styling (colours, point sizes, etc.) and the capability of moving

the objects with the mouse (interactivity) are very important for a DGS while they have not sense at all in geometry as a theoretical mathematical discipline. We conclude remarking that traditional techniques used to describe a geometry figure are not always suitable for a DGS.

## 2.1 The three layers of the file format specification

The Intergeo file format specification is split into three layers. Each layer is different in nature and can exist by itself.

The following diagram shows the structure of the layers

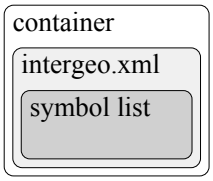


Figure 1: Specification structure in layers

**The container layer specification** describes the container as a ZIP file. This means that the container is a bundle of compressed files. It contains the important file `intergeo.xml` and other files like media (images, audio, video), data, text and a preview of the construction.

**The intergeo.xml layer specification** describes which rules must follow an XML file to be a valid DGS construction. Such a file is self-contained except for media files that are located in the container. An XML-schema is provided to help specifying and validating any construction. This layer specifies the general structure of the file, how the different statements are constructed from the symbols and how the most atomic values like (real or complex) numbers, variables, text and formulas have to be encoded. However, the list of symbols that is the origin of the statements is not part of this layer.

**The symbol list layer** will be used to allow for each DGS to create its own extensions to i2g, by possibly adding files to the container that can be read and handled in system dependent ways. The symbol list is a collection of Content Dictionaries describing symbols; in particular, some symbols have been chosen to be part of a DGS construction. See Appendix C for a more detailed description of its implementation.

## 2.2 Design decisions: Constraints versus Constructions

The general framework was clear from the outset: to design a semantically rich format, that could be interpreted by at least all DGS in the consortium. But also possibly by others, and maybe by other types of programs as well, for example computer algebra systems or proof assistants. One main design decision in this respect consists of the choice of constructions, as opposed to constraints. We will describe this in more detail now.

DGS deal with sets of geometrical objects that have certain relations. We call such a set of objects with given relations a *figure*. We conceive of these objects in some underlying space, for example the Euclidean plane. In principle, if nothing else is said about them, objects can move around freely in this space. The relations then specify *constraints* on the movement of these objects.

**Example 1** *Two points P and Q, together with a line l; there is the following constraints:*

*line l is incident to both points P and Q.*

**Example 2** *A circle* Γ*, a point P, a line l and the constraints*

*P is on l*
*l is tangent to* Γ
*the distance of P to the center of* Γ *is* 10*.*

We can make the simple observation that the constraints do not determine the positions of the objects uniquely. This causes multiple problems that lie at the heart of dynamic geometry.

First there is the problem of how to create an instance of the figure. (We say this has to do with the *static* aspect of the figure.) In example 1, the points *P* and *Q* could still lie anywhere on the line *l* as it stands. For any instance, we must specify where *l*, *P* and *Q* should be. But once we have specified one, the other two are not completely free anymore. This particular example is not hard. Example 2, although more difficult, is still doable. But in general, it is very difficult to give any particular solution for a set of constraints. There is not even a quick method to decide whether there are any instances: a set of constraints could be too restrictive and leave none.

Second, there is the dynamic behaviour of a figure, caused by the freedom still left by the constraints. In example 1, what should the user be able to move? May the line be picked up and translated or rotated in its entirety, the points being translated and rotated with it? Can the user only move one of the two points, the line being adjusted accordingly? Constraints of a

strictly classical geometrical nature, such as the ones stated above, do not say anything about this behaviour. For the approach of a DGS, this is not enough.

A natural way to shed light on both these problems is a more precise specification of how the objects depend on each other. We could stipulate first of all which objects are *free*, meaning that they can be varied over the whole range of possibilities in the underlying space (think of the plane) by the user. We would then proceed step by step saying which objects depend only on the free objects, which ones depend only on these new objects and the free objects, etcetera. Such a specification is called a *construction*. It allows for an algorithm to rapidly create instances or decide that there are none. It also enables a DGS to give more consistent dynamic behaviour: objects are only movable insofar as they still have some degrees of freedom left if the objects they depend on are kept fixed. The behaviour for all different cases (e.g. a line through a fixed point) can be decided in advance. Other objects dependent on the object being varied have to change as well, and this still leads to decision problems, but they are less severe. We could give a construction for example 1 as follows:

**Example 3** *Two points P and Q, together with a line l, and the following construction:*

*free_point(P)*
*line_through_point(l,P)*
*point_on_line(Q,l)*

The line *l* would then depend on where *P* is placed. That point could be varied freely. The line could then be rotated around *P* (and *Q* would most logically rotate with it), and while *P* and *l* are kept fixed, *Q* could still slide over the line. Note that such information could not be gleaned from the figure.

It thus seems like a figure might be too general to be practical, and we might be better off with a construction. We therefore decided to go with constructions. This decision implies less interoperability with constraint-based systems, since some of their resources will not be encodable into the format. But it ensures that construction-based DGS will be able to interpret the resources, which they might not if we used figures. Indeed, although there are systems like Geometry Expressions [Sal08] that take a constraint-based approach, and some systems like Geometer's Sketchpad [KCP08] support it, most systems only use constructions.

Another effect of the decision is the explosion of keywords. We have to distinguish between "line_through_point" and "point_on_line". This is in sharp contrast to figures, where one relation "incident" would suffice, as can be seen from the `plangeo` Content Dictionaries. In general, if there are *n* different types of objects, the construction approach now forces $n^2$ different

types of incidence on us. This means a more bloated specification of the file format. On the other hand, it is easier for software developers to parse constructions, so it saves trouble there.

## 2.3 Other design decisions

Several decisions have been made in teleconferences or in the mail list that deserve recording, even though they are not truly important. It is thought that describing them here will help understand why the Common File Format has been designed this way.

In constraints, the output element will keep the out=“true" mark. Even though strictly speaking it is redundant, it improves human reading.

It was also decided to recommend the use of free_point and free_line. Even though they may seem unnecessary, if no constraint is used the behaviour of the element is unspecified. They are also useful in writing the file and in doing topological sort, so please use them.

During the discussion on how to implement polygons, whose constructors have to handle a variable number of vertices, it was decided that constraints will have a fixed number of arguments, some of which may be auxiliary elements with a variable number of inner arguments. Hence, a polygon element has only one argument, a list_of_vertices_coordinates, and it is this list the one which has a variable number of arguments.

Symmetries will accept as input and output elements of any kind, not necessarily of the same type. While using atypical combinations of types in a reflection might be a door open to incompatibilities between DGS, it was felt that writing a list of allowed combinations would be awkward. For instance, reflecting arcs on circles may result in segments or rays. This decision will extend to other transformations, such as rotations, translations and dilations.

Building regular polygons given a side and a number of vertices is complicated in non-euclidean geometries. It was decided to leave this constraint for later.

Specifying the style of highlighted elements (those with the mouse focus) was considered and dismissed. Existing DGS indicate which element has the focus by using many visual aspects (colors, brightness, borders, sizes, opacities), and so too many new styles would be needed to specify all possible behaviours, which in addition would possibly result in some ugly effects. Hence, it was decided that each DGS would handle highlighting in its own way. Because of the same reasons, many default values for styles have not been specified; instead, they have been left system dependent (through the whole deliverable, some feature will be said to be “system dependent" when its availability or functionality depends on which DGS is being used).

# 3 The container

The container is the topmost structure of Intergeo file format. It is composed of a collection of files stored as a directory hierarchy in a ZIP file format.

The container comprises the construction description as well as other data, for example, media (images, sound, video, ...), previews (PDF, SVG or PNG), metadata or private data/legacy file formats that some software may keep, among other information. Thus, instead of encoding everything in a single XML file it is more natural to store all such information in a standard format like a ZIP package and, in addition, benefit of the compression.

The proposed directory structure is the following

| | |
|---|---|
| `construction/` | mandatory |
| `construction/intergeo.xml` | mandatory |
| `construction/preview.pdf` | optional |
| `construction/preview.svg` | optional |
| `construction/preview.png` | optional |
| `metadata/` | optional |
| `metadata/i2g-lom.xml` | optional |
| `resources/` | optional |
| `resources/<image-files>` | optional |
| `resources/<audio-files>` | optional |
| `resources/<video-files>` | optional |
| `resources/<data-files>` | optional |
| `resources/<text-files>` | optional |
| `private/` | optional |
| `private/<domain-name>/` | optional |
| `private/<domain-name>/<files>` | optional |

Figure 2: Containter structure

The `construction` folder is always mandatory and contains the important file `intergeo.xml`. The specification does not enforce any preview format, however all files should be named `preview.*`. Files with names other than intergeo.xml or `preview.*` are not allowed in this part. Thus, any auxiliary file should be placed in the resources folder.

The whole construction can contain optionally metadata. We do not impose any specific format but we recommend including a file named `i2g-lom.xml` with the metadata as specified in the document [HLCMD08].

The `resources` directory contains any media or data file needed by the `intergeo.xml` or preview files. Constructions must be possible to open with solely the `intergeo.xml` and the resource files. Inside the `resources` folder it is permitted to add any hierarchy of subdirectories. However, we recommend placing the images, audio, video, data and text under the folders

`images`, `audio`, `video`, `data` and `text`, respectively. The valid file formats for the resources depend on their usage. It is not the responsibility of the container specification to impose any specific format type for the media.

DGS's are free to store any file within their private directory and ignore completely the private directories belonging to other systems. Note that if a file is referred from the `intergeo.xml` it should be always placed in the `resources` directory. We strongly recommend placing all files inside a directory name based on the domain name of an organization. For example, if my organization has the domain `MyOrg.net`, the recommended directory name would be `net.myorg` (note the use of lowercase). This is done in order to distinguish my private directory from the private directories of other software developers.

# 4 intergeo.xml

This is an XML file that contains the full description of the construction. The file must be always named and placed at `construction/intergeo.xml` when it appears inside the container. Except for other auxiliary media files, this file is always self contained. This means that this file alone, when it does not depend on media files, should be readable by DGS's.

There are three main distinct parts:

1. The `elements` part is a static view of all objects. The description of the objects in this part is minimal; it only indicates how the objects are constructed (their coordinates) and displayed (without styling).
2. The `constraints` part explains the geometric relations between objects. Some relations are purely algorithmic (an object can be constructed directly from existing ones) while others are constraint based (an object should satisfy a given property but, at the same time, it keeps a certain degree of freedom). Thus, the objective of this part is describing how some figures are recomputed when points (or other objects) are moved by the mouse pointer.
3. The `display` part comprises information necessary to render the objects. The display contains the styles and non-mathematical behaviour of the elements.

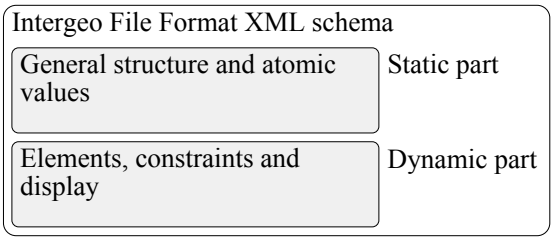


Figure 3: Intergeo file format XML schema

The splitting of data into these parts has some advantages. For example, it is possible to have more than one view or display for the same set of elements and constraints.

Because `intergeo.xml` is an XML file it is important to provide an XML schema (a file, also in XML format, that describes the structure of a family of XML files). There is not one single schema. Instead, there is one schema for each set of geometric symbols. The schema is composed of a static part which defines the general structure of the XML file and the atoms (leaves of the XML-tree). This static part is the same for all XML schemas. The dynamic part is generated automatically from the list of geometric symbols.

The current XML-schema that corresponds to the minimal set of symbols can be found at [Int09].

## 4.1 Construction

The construction is the root element of the intergeo.xml file. The schema fragment is the following

```
<xs:element name="construction">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="elements"/>
      <xs:element ref="constraints"/>
      <xs:element ref="display" minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

and an example is

```
<construction>
  <elements>
  ...
  </elements>
  <constraints>
  ...
  </constraints>
  <display>
  ...
  </display>
</construction>
```

## 4.2 Elements

The <elements> element comprises the enumeration of objects that will be part of the construction. They will be often geometric objects like points, lines, circles, ... But it can hold also any object that can be drawn like images, slider bars, buttons, etc.

```
<xs:element name="elements">
  <xs:complexType>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <!-- here goes the list of available elements -->
          <xs:element ref="point"/>
```

```
            ...
          </xs:choice>
  </xs:complexType>
</xs:element>
```

The previous schema fragment should be filled with all elements specified in the elements part of the symbols list.

The schema for each element has the form

```
<xs:element name=element–name>
  <xs:complexType>
    <xs:sequence>
      enumeration–of–arguments
    </xs:sequence>
  </xs:complexType>
  <xs:attribute name="id" type="xs:Name"
                use="required"/>
</xs:element>
```

For example, for the point with homogeneous or Euclidean coordinates:

```
<xs:element name="point">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="homogeneous_coordinates"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:Name"
                  use="required"/>
  </xs:complexType>
</xs:element>
```

And an extra definition for `homogeneous_coordinates` is needed:

```
<xs:element name="homogeneous_coordinates">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="scalar"/>
      <xs:group ref="scalar"/>
      <xs:group ref="scalar"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Note that both point and `homogeneous_coordinates` are specified with Content Dictionaries and the atomic elements scalar at the end of this chapter.

An example of the elements part is

```
<elements>
  <point id="A">
    <homogeneous_coordinates>
      <double>3.55</double>
      <double>−4</double>
      <double>0</double>
    </homogeneous_coordinates>
  </point>
</elements>
```

### 4.2.1 Additional elements restrictions

There are other restrictions that cannot be expressed with an XML schema and are described here.

**Unique identifiers restriction.** All values of the `id` attribute are used to identify the geometric objects and they must be used only once (unique identifiers).

**Only constants allowed restriction.** A declaration in this part is not allowed to refer to other objects. For example, each point needs coordinates and an XML coordinates element must appear as its child. Thus, scalars must be explicitly instantiated as real or complex floating numbers and cannot be replaced by variables or expressions pending to be evaluated.

**Elements symbols restriction.** Symbols are declared to be in the elements, the constraints or in the display part. Only symbols declared to be in the elements part can appear in the elements part.

## 4.3 Constraints

The <constraints> describes the relations between objects. Sometimes this relation is just a description of how an object is to be built and other times it explains how the object is constrained and partially free movable.

The schema is quite similar to the elements part:

```
<xs:element name="constraints">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <!-- here goes the list of available
           constraints -->
```

```
      <xs:element ref="line_through_two_points"/>
      ...
    </xs:choice>
  </xs:complexType>
</xs:element>
```

The list of available constraints is specified with Content Dictionaries. The schema for each constraint has the form:

```
<xs:element name=constraint-name>
  <xs:complexType>
    <xs:sequence>
      enumeration-of-arguments
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

For example, for the `line_trough_two_points` constraint the schema is:

```
<xs:element name="line_through_two_points">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="line"
                  type="output-reference"/>
      <xs:element name="point" type="reference"/>
      <xs:element name="point" type="reference"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

where `output-reference` and `reference` are defined below and are specified to act as references to objects using their id's. Example

```
<constraints>
  <line_through_two_points>
    <line out="true">l</line>
    <point>A</point>
    <point>B</point>
  </line_through_two_points>
</constraints>
```

### 4.3.1 Additional constraints restrictions

**Defined input references restriction.** All input and output references must be defined previously in the elements part.

**Composition of constraints forbidden restriction.** The arguments of the constraints cannot be other constraints. It is allowed only to refer to valid geometric objects, using the id's. Note that this rule still permits using constants in the arguments.

**Unique output reference usage restriction.** An identifier can be used only once as output reference except for some declarations. These declarations are based on symbols that explicitly declare that can coexist with other declarations with the same output reference.

**Constraints symbols restriction.** Symbols are declared to be in the elements, the constraints or in the display part. Only symbols declared to be in the constraints part can appear in the constraints part.

## 4.4 Display

The display part contains the styles and non-mathematical behaviour of the elements and the whole construction area.

Styling consists on specifying visual properties (like colours, line widths, point sizes, labels, etc.) and the interactive properties (fixed) of the element. While the elements and constraints parts are close to mathematical properties and behaviour, the display part contains visual properties and the behaviour that can not be formalized mathematically.

The current version of the styling system for Intergeo is inspired by the SVG and CSS standards. Thus, some symbols like *fill*, *stroke*, *stroke-width* are taken from the SVG and CSS standards but *point-size*, *label* and *fixed* are specific of the Intergeo File Format.

Each system will provide default values for the styles which may depend on the type of the element: points might be blue while lines might be red. Movable points might have a color different to fixed points. The point which currently has the mouse focus and is being moved might be distinguished with particular combinations of colors, borders, opacities, etc.

More than one display can be present and each one represents a possible view of the representation. A DGS could open two windows and changes in one would reflect in the other; each display would describe what is to be seen in each window.

Styles for elements will be specified individually in each display part. For instance, the same point could be movable in one window but not in another.

Each display can contain also styles applying to the whole construction, as opposed to one particular element; for instance, background-color, viewport, or axes.

The display contains a <style> for each element to be described.

```
<xs:element name="display">
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="style"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The style must refer to an existing element and comprises the different possible styles of the element.

```
<xs:element name="style">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="label"/>
      <xs:element ref="fill"/>
      ...
    </xs:choice>
    <xs:attribute name="ref" type="xs:Name" use="required"/>
  </xs:complexType>
</xs:element>
```

For example, the XML-Schema for <label> is:

```
<xs:element name="label" type="xs:string"/>
```

A complete example of a point with styles is

```
<construction>
  <elements>
    <point id="P">
      <homogeneous_coordinates>
        <double>4</double>
        <double>5</double>
        <double>1</double>
      </homogeneous_coordinates>
    </point>
  </elements>
  <constraints>
    <!-- -->
  </constraints>
  <display>
    <style ref="P">
      <label>The point P</label>
      <fill>#FFFFFF</fill>
      <stroke>#222222</stroke>
```

```
        <stroke-width>2</stroke-width>
        <point-size>5</point-size>
      </style>
    </display>
  </construction>
```

### 4.4.1 Additional restrictions

**Defined reference restriction** The reference must point to an existing element described in the elements part.

**Single style node by element restriction** All styles of an element must be specified in a single <style>.

## 4.5 Atomic values

Atomic values are the simplest ingredients of the Intergeo file format. They are specified also with an appropriate schema fragment.

**Note:** We might add more atoms in the future.

### 4.5.1 Scalars

Scalars represent either double or complex double numbers:

```
<xs:group name="scalar">
  <xs:choice>
    <xs:element ref="double"/>
    <xs:element ref="complex"/>
  </xs:choice>
</xs:group>
```

### 4.5.2 Double numbers

Doubles follow the [IEEE 754-1985] standard.

```
<xs:element name="double" type="xs:double"/>
```

Example

```
<double>4.37589837073</double>
```

### 4.5.3 Complex numbers

```
<xs:element name="complex">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="double" type="xs:double"/>
      <xs:element name="double" type="xs:double"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example

```
<complex>
  <double>2.4689</double>
  <double>−5.78231659</double>
</complex>
```

### 4.5.4 References to objects

Constraints accept as arguments references to objects.

```
<xs:complexType name="reference">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>
```

### 4.5.5 Output references to objects

Constraints usually have one argument that is the output. The output is the name of the object that is going to be computed from the other ones.

```
<xs:complexType name="output−reference">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="out" use="required"
                    fixed="true"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

## 4.6 intergeo.xml XML-Schema

As we have already said, the XML-schema is generated from the set of symbols and its signatures. The project that generates such schema can be get from the SVN repository `http://svn.activemath.org/intergeo/Format/xml`. Several examples of valid files can be found at `http://i2geo.net/xwiki/bin/view/I2GFormat/FileFormatSymbols`.

Although the original idea was using the OpenMath Content Dictionaries and their associated Small Type System files, we decided to write a very simple and single XML file with all symbols and generate the XML-Schema from it. Such XML file can be found at `http://svn.activemath.org/intergeo/Format/xml/symbols.xml`.

The symbols.xml is an XML file that contains essentially declarations like, for example,

```
<element name="point">
  <argument type="homogeneous_coordinates"/>
</element>
```

for declaring elements and

```
<constraint name="line_through_two_points">
  <argument type="line" out="true"/>
  <argument type="point"/>
  <argument type="point"/>
</constraint>
```

for the constraints.

The declarations in the symbols.xml file are much simpler than the corresponding ones at the XML-schema. Thus, it is possible to add more symbols to the schema without any advanced knowledge of XML. Another advantage of this approach is the possibility of generating different schemas addressing different purposes. For example, generate a schema that validates not only the official symbols of the format, but the proprietary ones. Another possibility is getting a schema to validate the file format for constrained based systems (as opposed to construction based which mainly targets the Intergeo File Format).

# 5 Symbol list

Symbols are the main ingredients used to describe a construction. They define how objects are built and their behaviour. Each icon in a DGS palette is roughly associated to one or more symbols; styles like colours, point sizes, line widths are also represented in the file format using symbols. All DGS's share a big set of common features that will be covered by the Intergeo official symbols. However, for additional features it is acceptable to use proprietary symbols that one day might become official.

The list of symbols is divided in three categories: elements, constraints and display depending on the part of intergeo.xml they appear in. Symbols for the elements and constraints categories are not primarily specified using a XML-Schema but with Content Dictionaries, which are part of the OpenMath standard. With some knowledge of how the atoms are expressed in XML, the description of the symbols with Content Dictionaries and their signature with the Small Type System (STS), the XML schema can be generated automatically.

The complete list of official symbols can be found in Appendix A or at `http://i2geo.net/xwiki/bin/view/I2GFormat/FileFormatSymbols`.

There will be a process for Content Dictionaries to become an official part of the file format. How this process will be structured and how new official Content Dictionaries can be added after the Intergeo project's lifetime is to be decided.

# 6 How to Identify Objects of the I2G Format

This section specifies ways of naming files that should be recognized as I2G files in the common file format. This part is normative, developers should follow them strictly so as to ensure that interoperability can be started.

**Format name:** Intergeo

**Media type name:** `application`

**Media subtype name:** `vnd.intergeo`

**Encoding:** binary
*(This media type may require encoding on transports not capable of handling binary.)*

**Security considerations:** There has been no examination of possible security risks associated with I2G files.

**Interoperability considerations:** This file format is cross-platform. Files are encoded in UTF-8 and compressed using zip.

**Published specification:** this specification (Intergeo's *Deliverable D3.10: i2g Common File Format Final Version*).

**Applications which use this media :** Cinderella, GeoGebra, GEONExT, JSXGraph, Cabri, WIRIS,...

**Additional information:**
1. Magic number(s) : NOT USED
2. File extension(s) : `i2g`
3. Apple Macintosh file type code : NOT USED
4. Object Identifiers: NOT USED
5. MicroSoft Windows Clipboard Names: `I2G`
6. Apple Macintosh Uniform Type Identifier: `eu.inter2geo` extends `public.archive` and `public.zip-archive`

**Person to contact for further information :**
1. Name : Intergeo ASBL
2. Email : asbl@inter2geo.eu

**Intended usage:** Common
This mime type shall be used to identify data files for the common file format for interactive geometry software.

**Author/Change controller:** Intergeo ASBL (asbl@inter2geo.eu)

**Procedures taken to ensure such names** A media type registration has been filed to the Internet Assigned Numbers Authority for the media-type Intergeo and granted from them as can be seen at: `http://www.iana.org/assignments/media-types/application/vnd.intergeo`.

The text above is based on this registration and extends it. It is the intention of the consortium to update that registration to match the above text.

This text contains recommendations for implementors of desktop applications to run on MicroSoft Windows and Apple Macintosh operating systems. Conforming to Apple's documentation (`http://developer.apple.com/mac/library/documentation/FileManagement/Conceptual/understanding_utis/`) about Uniform Type Identifiers, it is enough for applications to declare their support for this file format within their application descriptor. Conforming to MicroSoft's documentation (`http://msdn2.microsoft.com/en-us/library/ms649013.aspx`), it is enough for applications to call the appropriate functions at startup.

# 7 The Intergeo ASBL, Future Work and Request For Comments

The Intergeo Common File Format is not completely finished for two main reasons:

- Some new functionalities are wanted; see, for instance, the wish list page at `http://i2geo.net/xwiki/bin/view/I2GFormat/WishList`.
- Some problems common to DGS have not been fixed. Shortly we list some of them; note that they include hard research problems which would be difficult to solve even within one DGS. For a deeper discussion of these mathematical issues, we suggest to read [Kor99].

We explicitly invite those interested to join the discussion and propose solutions or give remarks.

In order for the Intergeo Common File Format to continue being developed after the Intergeo project ends, the Intergeo ASBL (Association Sans But Lucratif - non-profit organization) was constituted in 2010 under luxembourgish law. For more details, see `http://asbl.i2geo.net/`, which will be available soon with all informations.

The Intergeo ASBL has already reserved the application tree vnd.intergeo from IANA for the file format. The first general assembly - after the constitutional one - will be held during the I2GEO conference 2010 allowing all partners of the Intergeo project to join the non-profit organization and to continue the fruitful work together.

Despite the fact that we are listing a lot of yet to be solved issues, we are confident that the first version of the i2g file format is capable of handling any design decisions that result from the following discussion.

## 7.1 Handling of degenerate cases

Some objects can also be replaced by others in certain special cases, for example, a circle might degenerate to a line, or a conic might degenerate to two lines. Although currently most DGS do not use these degenerations (there are exceptions such as Cabri II plus and Cabri 3D), this could be desirable for the future. A DGS might construct a parallel line to a degenerate circle through three collinear points – with our current specification and typing mechanism it is not possible to capture this in the file format. However, we request advise from DGS developers on this issue.

## 7.2 Ambiguity Resolving

Ambiguity Resolving is crucial for finding the correct positions of elements in stored construction after loading, also known as the persistent naming problem [MP02] from parametric CAD. Assume that an intersection point of two circles is used in a construction. If the two circles are moved into a tangent position, and then the construction is stored, then both intersections have the same coordinates and thus cannot be distinguished. If the circles are moved into a position where both intersection points can be distinguished, then it is essential to pick the correct intersection point.

Most DGS solve this problem by having an implicit order of multiple outputs. This order is dependent on the implementation details of the algorithms and cannot be part of a specification. Also, a point might switch branches later due to homotopy-conserving implementations. This means that this approach cannot be used for a cross-software file format.

In the current version of the file format, the order of the inputs can be determined from the coordinates of the intersection points when they exist and differ from each other. For all other cases, the order is still unspecified. This will lead to problems when files are stored and exchanged.

## 7.3 Dynamic Styling Information

Some of the DGS support parametrized styles, for example a color that changes according to the position of an element. This cannot be specified in the file format currently. See the next section for a discussion of the problems that are specific to including parametrized values using functions.

## 7.4 Functions and Scripting

Almost all DGS support functions, used for plotting graphs, defining element dependencies, or changing the style of elements dynamically. All of these use a different language to specify the functions, though many aspects are shared. The conformance to standards varies wildly from OpenMath compliance to unspecified.

Right now it seems impossible to homogenize the various dialects. Actually, the translation from one language to the other can be done easily by humans if an automatic conversion fails, so we decided that all functions should be specified in the private sections of the file format. Each DGS may try to interpret the other function specification, of course, and store its own interpretation as well.

For this, we need a notion of “alternatives", which should be specified in an upcoming version of the i2g format.

Another difficulty in dealing with functions is that some DGS extend the notion of function to a general-purpose functional programming language. This proves that it is impossible to find equivalent functions algorithmically. Nevertheless, in many cases, mostly non recursive algebraic expressions, the translation is straightforward, and so it might be sufficient to use a heuristic approach.

## 7.5 Mathematical Typesetting

A de-facto standard for mathematical typesetting is TeX [Knu84], and the browser-compliant way is to use MathML [CIM08]. DGS software use both approaches, while the TeX implementation used is usually only a subset of the full TeX system as created by Knuth[1].

A logical solution to storing typesetting information would be to use the existing OpenMath syntax. It has not been decided yet how to handle the situation, and it is unclear whether it has practical relevance at all. We could not agree on a definitive way to typeset formulae. We suggest that a solution for the function specification problem above will be a solution for this problem as well.

## 7.6 Sending Commands to Constructions

Sending commands to constructions is now handled in the Intergeo API specification document [Kor09].

## 7.7 Macro Constructions

So far there is no notion of Macro constructions. We expect that macros are basically sub-constructions, and it is probably sufficient to add an additional `inmacro` attribute to the constraints and elements.

## 7.8 Number Representation

Currently, the specification of coordinates uses the IEEE standard for doubles (see Sec. 4.5.2 on p. 21). While this is probably sufficient for most purposes, it lacks the ability to describe *real* coordinates, for example the irrational numbers $\pi$ or $\sqrt{5}$. As there are constructions even in elementary geometry that require such numbers, it is desirable to be able to express them. The OpenMath standard supports this, however, the i2g format cannot be based

[1] Some use the hoteqn library, others use custom implementations

on the full OpenMath specification as most DGS developers cannot afford the necessary implementation work.

For the time being, we will restrict the number representation to the IEEE standard. This should not be the cause of severe problems, because the coordinates of dependent elements can be recalculated up to arbitrary precision by the DGS itself. If there is a need for other number representations, we will extend the mechanism as described in Sec. 4.5.2.

## 7.9 Acyclicity of Construction Dependencies

Most construction-based DGS require that the dependency graph of a construction be acyclic, but there are some systems (both constraint-based as well as construction-based DGS) that allow for certain circular dependencies. Therefore, we do not enforce this property, and we do not impose a special order for the constraints in the constraint part of the i2g format.

This implies that each DGS has to be able to handle cycles. The easiest resolution is to drop those constraints that close a cycle. Another solution is to add all constraints and break the cycles on-the-fly whenever an element moves. If the DGS can handle the additional constraint, then it should add it.

## 7.10 Library Support

In order to make it easier for new implementors to work with the i2g format, and hence improve the Intergeo project sustainability, it was considered creating a library to read and write i2g files, and a corresponding sourceforge project was applied for. However, it remain to be seen whether some general purpose software could be used for this, and so this effort has been suspended.

## 7.11 Higher Dimensional Geometry

With the availability of three-dimensional DGS it is necessary to extend the i2g format to 3D. The basic structure should be similar to what we have now, but more elements and constraint symbols are necessary.

# Appendix A : Intergeo file format symbols

The following is a list of symbols that we agreed on as of June 2010. It will be developed further by the Intergeo ASBL.

At any time, HTML documentation on the newest version can be found at `http://i2geo.net/xwiki/bin/view/I2GFormat/FileFormatSymbols`; even though not accessible to everybody, a second source, more convenient to download files, is the SVN server at `http://svn.activemath.org/intergeo/Format/` .

In the following list, the keyword is listed first, in boldface. On the next line, the types of its arguments are listed. The order of the arguments is fixed. Optional arguments are enclosed between [ and ] characters. Arguments that can be repeated any number of times are indicated with a *.

Observe that the word “undefined" is used with, at least, two different meanings. When the result of a computation is said to be undefined, it is meant a DGS dependent value that indicates error or voidness. When, under some circumstances, it is said that the behaviour of some operation is undefined, what is meant is that each DGS is free to provide its most sensible result; for example, two circles may or may not have intersections depending on whether the DGS handles complex coordinates.

## A.1 The elements part

**point**
*point_coordinates*
This represents a point in space. It will have coordinates for initialization.

**line**
*homogeneous_coordinates*
This represents a line. The coordinates are the homogeneous coordinates of the line; this is, $(2, 3, 5)$ represents the line $2x + 3y + 5 = 0$.

**line_segment**
*point_coordinates, point_coordinates [,point_coordinates]*
This represents a segment from a straight line; the name is due in consideration to possible future implementations of other kinds of segments. A line_segment is indicated by providing the coordinates of its two endpoints; additionally, a third optional point, called *via point*, may be used in projective geometry to indicate a point in the middle of the segment to specify which of the two possible segments is meant. For instance, line_segment $([0, 0], [0, 2], [0, 1])$ would represent the “normal" segment from $(0, 0)$ to $(0, 2)$, while line_segment $([0, 0], [0, 2], [0, 10])$ would represent the projective segment that starts at $(0, 0)$, goes to $(-\infty, 0)$, and then continues from $(\infty, 0)$ to $(0, 2)$.

**directed_line_segment**
*point_coordinates, point_coordinates [,point_coordinates]*

A line_segment with associated direction; instead of two endpoints it has a starting point and an endpoint. The same remarks for the via point apply as in the case of **line_segment**.

**ray**
*point_coordinates, direction | point_coordinates, point_coordinates*
This represents a ray (half line) which starts at the given point and proceeds to infinity in the direction provided, or via a second point (alternative option). Projective DGS may treat it as a special form of a **directed_line_segment**. If the given coordinates are at infinity the notion or ray does not make sense, but a DGS can choose to use the line at infinity instead.

**polygon**
*list_of_vertices_coordinates*
A polygon is constructed from a list of vertices. No restrictions are imposed on the vertices, so that all degenerate cases are possible: two consecutive vertices can be the same, three consecutive vertices can be collinear, sides can intersect, the border does not have to be a simple curve.

**vector**
*point_coordinates*
Constructs a vector going from the origin to the point with the specified coordinates.

**conic**
*matrix [,dualmatrix]*
This will represent a general conic. The coefficients of its associated quadratic form are provided in a 3 × 3 matrix of (possibly complex) numbers. An additional 3 × 3 matrix may be given, being the matrix of the dual conic. This can assist if the conic is a degenerate case (two lines, etc)

The following symbols are special kinds of conics. Their initial position is stored in the same format as a general conic. A difference with the conic symbol is that a DGS may opt to only accept reading a circle and request the intergeo java library to convert the matrix to a different form, whereas it might reject a general conic.

**circle**
*matrix [,dualmatrix]*
This will represent a circle, indicated in the same way as general conics.

**ellipse**
*matrix [,dualmatrix]*
This will represent an ellipse, indicated in the same way as general conics.

**parabola**
*matrix [,dualmatrix]*
This will represent a parabola, indicated in the same way as general conics.

**hyperbola**
*matrix [,dualmatrix]*
This will represent a hyperbola, indicated in the same way as general conics.

**locus**
*void* | *a set of points*
This will represent a locus. Its initial value can either not be provided (it will then have to calculated by the DGS) or a finite set of tracer points (specified together with the corresponding values of the mover point).

### A.1.1 Families

Four families have been defined. They are not elements, but sets of elements. They are used to specify the arguments that some constraints can have, and so they provide constraints with a form of polymorphism.

- line_family = {line, ray, line_segment, directed_line_segment}
- circle_family = {circle, arc}
- conic_family = {circle, circle arc, conic, parabola, ellipse, hyperbola}
- element_family = {all elements}

### A.1.2 Auxiliary symbols

**point_coordinates**
*homogeneous_coordinates* | *euclidean_coordinates* | *polar_coordinates*
The coordinates of a point, or the components of a vector, can be specified using any of the following three symbols:

**homogeneous_coordinates**
*scalar, scalar, scalar*
Provides the coordinates of a point in projective geometry; unlike the following two symbols, this one can be used also to provide for the coefficients of a line equation, see the symbol line above.

**euclidean_coordinates**
*scalar, scalar*
Provides the Euclidean coordinates of a point in the plane.

**polar_coordinates**
*scalar, scalar*
Provides the radius and argument of a point in the plane.

**direction**
*scalar, scalar* | *scalar, scalar, scalar*
Directions are auxiliary symbols, used to construct rays. They are specified either euclideanly as $(a, b)$ or projectively (homogeneously) as $(a, b, c)$.

**list_of_vertices_coordinates**
*point**
This auxiliary symbol is used only by the polygon element.

**list_of_vertices**
*point_reference**
This is not exactly an auxiliary symbol, but rather an auxiliary XML element (tag) used only within constraint polygon_by_vertices. It plays the same role as list_of_vertices_coordinates, only for constraints instead of elements, and it contains the references to the vertices as opposed to their coordinates.

## A.2 The construction part

The *new* or *output* arguments (that is, the dependent element of the construction) are indicated with the word “new". The first argument is always a new argument. The arguments which are not new are identifiers of existing objects upon which the new arguments depend. These objects must be declared in the elements part of the file.

*Remark*: Whenever distances or angles are mentioned in this document we refer to Euclidean measurements, unless otherwise stated.

**free_point**
*new point*
A completely unconstrained point.

**free_line**
*new line*
A completely unconstrained line. Note that the user interface of many DGS does not allow for controlling lines directly, but rather two points are moved to control their common line; hence, many DGS cannot implement this constraint.

**point_on_line**
*new point, line_family*
A new point restricted to lie on a given line.

**point_on_line_segment**
*new point, line_segment*
A new point restricted to lie on a given line_segment.

**point_on_circle**
*new point, circle_family*
A new point restricted to lie on a given circle.

**line_through_point**
*new line, point*
A new line that goes through a given point and can be rotated by the user, by moving the mouse. Note that the user interface of many DGS does not allow

directly for this operation, but rather an intermediate point which can be moved can be used to construct the line which passes through both points; hence, many DGS cannot implement directly this constraint.

**line_through_two_points**
*new line, point, point*
A new line that goes through two given points. If the two points are equal, the line is undefined.

**line_angular_bisector_of_three_points**
*new line, point, point, point*
A new line that is the angular bisector of three given points (in Euclidean measurements); it must pass through the second one. Degenerate cases: If the three points are the same, behaviour is undefined. If the first and third point are the same, then the line returned must be the line containing three points. If the second point is equal to the first or third, then the behaviour is undefined.

**line_angular_bisectors_of_two_lines**
*new line, new line, line_family, line_family*
Two lines that are the angular bisectors of the two given lines (in Euclidean measurements). If the lines are coincident, one bisector will be the common line and the other will be undefined. If they are parallel and not coincident, then one bisector is the line at infinity (for those DGS that support projective geometry) and the other bisector is undefined. A DGS can choose to use the line with the same Euclidean distance to both defining lines as the other bisector, as it is the continuous completion.

**line_segment_by_points**
*new line_segment, point, point [,point]*
A new line_segment passing through the first two points and, optionally, for those DGS that can handle projective geometry, also through the third. If the third point is equal to the first or the second, then the behaviour is undefined and each DGS can return an undefined value or what it thinks is the best line_segment.

**directed_line_segment_by_points**
*new line_segment, point, point [,point]*
A new directed_line_segment passing through the first two points and, optionally, for those DGS that can handle projective geometry, also through the third. If the third point is equal to the first or the second, then the behaviour is undefined and each DGS can return an undefined value or what it thinks is the best line_segment.

**ray_from_point_and_vector**
*new ray, point, vector*
A new ray that starts at the point and moves in the direction specified by the vector.

**ray_from_point_through_point**

*new ray, point, point*
A new ray that starts at the first point and goes through the second to infinity.

**line_parallel_to_line_through_point**
*new line, line_family, point*
A new line that is parallel to a given line and goes through a given point.

**line_perpendicular_to_line_through_point**
*new line, line_family, point*
A new line that is perpendicular to a given line and goes through a given point.

**point_intersection_of_two_lines**
*new point, line_family, line_family*
Elements in the line_family can be extended to lines; see Appendix B. A new point that is the intersection point of two given lines. In the degenerate case the point is undefined, or a DGS may use an continuous completion.

**midpoint_of_two_points**
*new point, point, point*
A new point that is the middle point (in Euclidean measurements) between two given points.

**midpoint_of_line_segment**
*new point, line_segment*
A new point that is the midpoint of a given line segment.

**endpoints_of_line_segment**
*new point, new point, line_segment*
A set of two points that form the end points of the given line_segment.

**carrying_line_of_line_segment**
*new line, line_segment*
The unique line that contains the given line_segment. Degenerate cases: if the segment is constructed to be always a point, the carrying line is undefined.

**starting_point_of_directed_line_segment**
*new point, directed_line_segment*
The starting point of a directed_line_segment.

**end_point_of_directed_line_segment**
*new point, directed_line_segment*
The end point of a directed_line_segment.

**line_segment_of_directed_line_segment**
*new line_segment, directed_line_segment*
The unique line_segment whose endpoints are the starting point and the end point of the given directed_line_segment.

**vector_of_ray**
*new vector, ray*

Returns the vector that indicates the direction of a ray; note that it is not uniquely defined.

**starting_point_of_ray**
*new point, ray*
The new point is the starting point of a ray.

**carrying_line_of_ray**
*new line, ray*
The new line is the unique line that contains the ray.

**circle_by_center_and_radius**
*new circle, point, line_segment*
A new circle with a given point as center and a radius as long as the given line_segment.

**circle_by_center_and_point**
*new circle, point, point*
A circle whose center is the first point and passes through the second.

**circle_by_three_points**
*new circle, point, point, point*
A circle that passes through the three provided points. Degenerate cases: if the three points are by construction collinear (but not equal), a DGS may give an error or return the line through the points (as a line or a degenerate conic). If the three points are by construction the same, a DGS may give an error or return the point (as a point or a degenerate conic).

**intersection_points_of_two_circles**
*new point, new point, circle_family, circle_family*
The intersection points of two circles. Elements in the circle_family can be extended to circles; see Appendix B. If there are fewer than two intersection points, a DGS may return the same point twice (in case of tangency), “undefined" or points with complex coordinates. If the circles are equal by construction, a DGS may give an error or return the common circle.

**other_intersection_point_of_two_circles**
*new point, point, circle_family, circle_family*
The new point is an intersection point of the two circles, and is different to the provided point. Elements in the circle_family can be extended to circles; see Appendix B.

**intersection_points_of_circle_and_line**
*new point, new point, circle_family, line_family*
The intersection points of a circle and a line. If there are less than two intersection points, each DGS is free to return repeated points, undefined values, or points with complex coordinates. Elements in the line_family and circle_family can be extended to lines and circles; see Appendix B.

**other_intersection_point_of_circle_and_line**
*new point, point, circle_family, line_family*

The new point is the intersection point of the circle and the line, and is different to the provided point. Elements in the line_family and circle_family can be extended to lines and circles; see Appendix B.

**intersection_points_of_two_conics**
*new point, new point, new point, new point, conic_family, conic_family*
The up to four intersection points between two conics. If there are less than four real intersection points, each DGS is free to return repeated points, undefined values, or points with complex coordinates. However, the algebraic multiplicity of the intersections has to be respected. If any of the elements intersected is an arch, it can be extended into a circle; see Appendix B.

**intersection_points_of_conic_and_line**
*new point, new point, conic_family, line_family*
The up to two intersection points between a conic and a line. If there are less than two intersection points, each DGS is free to return repeated points, undefined values, or points with complex coordinates. If the conic was degenerate and contained a line coincident with the intersecting line, a DGS may return “undefined" or the intersection line. If the element in the conic_family is an arc, it can be extended into a circle; and the element in the line_family can be extended into a line, see Appendix B.

**other_intersection_point_of_conic_and_line**
*new point, point, conic_family, line_family*
Returns the intersection point of a conic and line which is different to a provided point. The behaviour of this function must be consistent with that of intersection_points_of_conic_and_line. If the element in the conic_family is an arc, it can be extended into a circle; and the element in the line_family can be extended into a line, see Appendix B.

**circle_tangent_lines_by_point**
*new line, new line, circle_family, point*
The two lines which are tangent to a circle and pass through a point. If the point is on the circle and so there is only one tangent, a DGS may return the tangent (repeated), or the tangent and “undefined", or just “undefined". If the point was inside the circle, one or two times “undefined" may be returned. If the circle has radius zero and the point coincides with the center, two times “undefined" must be returned.

**foci_of_conic**
*new point, new point, conic_family*
The new points are the two foci of a conic. If the conic is degenerate and has only one focus, a DGS may return the focus (repeated), or the focus and an undefined value. One could also give circles, parabolas, ellipses and hyperbolas instead of conics, the former being subtypes of the latter.

**center_of_circle**
*new point, circle_family*
The new point is the center of the given circle (or arc).

**locus_defined_by_point_on_line**
*new locus, point, point, line*
Defines a locus as the trace of the first point when the second point moves over the line.

**locus_defined_by_point_on_line_segment**
*new locus, point, point, line_segment*
Defines a locus as the trace of the first point when the second point moves over the line_segment.

**locus_defined_by_point_on_circle**
*new locus, point, point, circle*
Defines a locus as the trace of the first point when the second point moves over the circle.

**locus_defined_by_point_on_locus**
*new locus, point, point, locus*
Defines a locus as the trace of the first point when the second point moves over the locus.

**locus_defined_by_line_through_point**
*new locus, point, line, point*
Defines a locus as the trace of the first point when the line moves around the second point.

**symmetry_by_point**
*new element_family, element_family, point*
Reflects an element about a point (central symmetry).

**symmetry_by_line**
*new element_family, element_family, line*
Reflects an element about a line (axial symmetry).

**symmetry_by_circle**
*new element_family, element_family, circle*
Reflects an element about a circle (circle inversion).

**translate**
*new element_family, element_family, vector*
Translates an element as indicated by the vector.

## A.3 The display part

Since the display part does not add *mathematical* information to the resource, this part will not have an OpenMath equivalent.

Most default values of visual aspects are system dependent, as explained in Subsection 2.3.

**stroke**
Stroke is used to specify whether a path has to be drawn and with which

color. Its default value is system dependent and, in particular, may depend on the type of element being painted. The value has to be given as a **paint**.

**stroke-width**
Stroke-width specifies the stroke width of the path. The value has to be given as a **length**. See also borderwidth. (The name of this style was changed, previously it had a _ instead of a –.)

**stroke-dasharray**
Can be either “none", the default value that indicates that the element is to be painted with solid strokes, or a list of lengths specifying in pixels the lengths of alternating dashes and gaps. Each length is a positive real number. If an odd number of lengths is provided, then the list of values is repeated to yield an even number of values. Thus, the length list 5,3,2 is equivalent to 5,3,2,5,3,2. By default, strokes are solid.

**borderwidth**
A real number, measured in pixels. Applies to elements that have a border, and so its applicability may be system dependent. It is used to indicate the width of the region painted with the “stroke" color as opposed to the “fill" color. The value has to be given as a **length**. See also stroke-width; as an example to illustrate the difference, some DGS allow lines to have borders, and so stroke-width would be the with of the line and borderwidth would be the width of the line border.

**border-opacity**
A number between 0 and 1, default value 1, indicating how opaque the border must be. Default value is 1. (The name of this style was changed, previously it had a _ instead of a –.)

**fill**
Fill is used to specify whether the interior of an element (such as a polygon) has to be filled and with which color. The value has to be given as a **paint**.

**fill-opacity**
This provides a number between 0 and 1 indicating how opaque is the interior of an element (such as a polygon). (The name of this style was changed, previously it had a _ instead of a –.)

**point-size**
Point-size specifies the size of the points. Points are usually displayed as circles with diameter equal to the value of this style. The value has to be given as a **length**. (The name of this style was changed, previously it had a _ instead of a –.)

**point-style**
A string indicating the shape of points; such as “circle", “star", etc. No particular values have been specified yet. (The name of this style was changed, previously it had a _ instead of a –.)

**label**
Label is the text that is displayed as the label of an element. This should

be viewed as the visible name of the object, handy for the user but not necessarily unique, as opposed to the identifier name, the unique identifier the software uses. Possible values are plain text (anything valid as XML text). The current format does not allow any mathematical specific format like TEX/LATEXor MathML.

**visible**
Indicates whether the element is visible (default) or must be hidden, maybe because it is just an uninteresting mid step in a construction. Possible values are “true" or “false".

**fixed**
Fixed specifies whether a given graphical object can not be moved by the user. Possible values are “true" and “false".

**background-color**
It specifies the background color of the drawing area. This is not a real style, since it does not affect a particular element.

### A.3.1 Auxiliary symbols

**length**
Length specifies a length. Lengths are decimal numbers (e.g. 12.345) that represent pixels. For simplicity, units are not allowed. Note that a given length measures the same independently of the scale of a construction.

**paint**
Paint can be none or a **color**.

**color**
Color has the form #RRGGBB where RR, GG and BB stands for the 0–255 hexadecimal values of the red, green and blue component of the color.

# Appendix B : Differences with the previous version

Two of the issues described in the Future Work section of Deliverable 3.6 [HKKM09] have been incorporated into the Intergeo File Format v1. These are:

- **Families**. A family is a set of element types that can be treated as an element type, and so it provides for the **polymorphism** that was needed to prevent the format from having an excessive number of constraints. For instance, the line_family consists of lines, rays, line_segments, and directed_line_segments. For a complete list of families, see `http://i2geo.net/xwiki/bin/view/I2GFormat/families`.
- **Styling**. Many symbols have been added to the display section; see section A.3.

In addition, other new features in Intergeo File Format v1 are:

- **Polygons and vectors** have been incorporated. Also, rays and directed_line_segments have really included, even though it had been decided earlier to add them.
- **Symmetries and translations** have been added. See constraints symmetry_by_line, symmetry_by_point, symmetry_by_circle and translate in the appendix A.2. Some other constraints have been added.
- **Extended intersections**. All the intersection constraints, including the other_intersection constraints, allow their arguments to belong in families; for instance, point_intersection_of_two_lines can be used also to intersect segments. The same happens with rays and lines, and arcs and circles (actually, arcs have not yet been formally included in the format). Segments which are not "parallel" might not intersect because they end before reaching the common point. It is sometimes useful to be able to intersect not the segments themselves, but the lines that contain them. Hence, the intersection constraints allow for intersecting not the elements provided, but their extending elements. The extending elements of line_segments and rays are lines; the extending elements of arcs are circles.

  One example: this is the intersection of a line and the line that contains a segment; note that within the line_segment tag it is specified that it has to be extended:

```
<point_intersection_of_two_lines>
    <point out="true">P</point>
    <line>l</line>
    <line_segment extended="true">s</line_segment>
</point_intersection_of_two_lines>
```

# Appendix C : OpenMath implementation of the symbol list

OpenMath is a standard used to express mathematical content through a series of functions, relations and constants called symbols which are specified within Content Dictionaries (CD). Each CD is an XML file that collects the description of a coherent set of symbols.

The symbol list will be implemented as a set of Content Dictionaries, which will provide an exhaustive enumeration of valid geometric elements, constraints and styles that can be used to generate a construction. These files may be inserted within the container layer and, if present, can overwrite default definitions. Although the symbols have been chosen to be part of a DGS construction, actually, they could live independently and be used for other purposes.

The advantage of using OpenMath as opposed to a self-chosen xml-format lies in the fact that the use of a Content Dictionary makes for a flexible, open, and reusable standard. First of all, the use of OpenMath enables Intergeo to use other Content Dictionaries already in existence, so it not only saves development time, but relies on already existing mathematical considerations for, e.g., algebraic expressions. Second, other kinds of software that want to use the format in the future can combine it with other Content Dictionaries to enrich its expressive power.

Unlike in the `intergeo.xml` specification, DGS's are free to create their own extensions (at the expense of losing interoperability with other systems).

# References

[CIM08] David Carlisle, Patrick Ion, and Robert Miner. Mathematical markup language (mathML) version 3.0. World Wide Web Consortium, Working Draft WD-MathML3-20080409, April 2008.

[Cre08] Creative Commons Inc. (CC). Namensnennung-Weitergabe unter gleichen Bedingungen 2.0 Deutschland. Available on the web, May 2008.

[HKKM09] Maxim Hendriks, Ulrich Kortenkamp, Yves Kreis, and Daniel Marquès. i2g common file format draft v2. http://svn.activemath.org/intergeo/Deliverables/WP3/D3.6/D3.6-Common-File-Format-v2.pdf, July 2009.

[HLCMD08] Maxim Hendriks, Paul Libbrecht, Albert Creus-Mir, and Michael Dietrich. Metadata specification. http://svn.activemath.org/intergeo/Deliverables/WP2/D2.4-Metadata/D2.4-Metadata-Spec.pdf, June 2008.

[Int09] Intergeo. Intergeo file format xml-schema for intergeo.xml. http://svn.activemath.org/intergeo/Format/xml/intergeo.xsd, 2009.

[KCP08] KCP Technologies Inc. Geometer's sketchpad v4, 2008.

[Knu84] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1984.

[Kor99] Ulrich Kortenkamp. *Foundations of Dynamic Geometry*. Dissertation, ETH Zürich, Institut für Theoretische Informatik, Zürich, 11 1999.

[Kor09] Ulrich Kortenkamp. I2g api specification. Deliverable D3.5, The Intergeo Consortium, 2009.

[MP02] David Marcheix and Guy Pierra. A survey of the persistent naming problem. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, pages 13–22, New York, NY, USA, 2002. ACM.

[Sal08] Saltire Software. Geometry expressions v1.1, 2008.