GML: Graph Modelling Language

DRAFT VERSION

Michael Himsolt[1]

December 19, 1996

[1]This research is partially supported by the Deutsche Forschungsgemeinschaft, Grant Br 835/6-2, research cluster "Efficient Algorithms for Discrete Problems and Their Applications"

Acknowledgements. I would like to thank all people on the file format mailing list (`fileformat@fmi.uni-passau.de`), for their helpful comments in providing this paper. Special thanks to Martin Ramsch (`ramsch@fmi.uni-passau.de`) for his ISO 8859 character set tables.

Contents

Chapter 1

Introduction

1.1 File Formats

1.1.1 User vs developer perspective

File formats do often provide tough problems both for the software engineers who write programs and for the people who are using them. Software engineers want formats that store data in an efficient manner, and are easy to read and write. Users want way to save their data in a convenient and fast manner, and don't want to be bothered with the choice of a specific format.

1.1.2 Converters

The consequence is that almost every graphics or desktop publishing system has its own file format, optimized for the needs of that product. This means that direct data exchange between different products is impossible, since the file formats will be mutually exclusive. So, most programs contain lots of converters that transform data between different formats.

1.1.3 The User's Perspective

Having lots of converters this is inconvenient for the user. First, it means that $O(n^2)$ converters are ideally needed to exchange data between n programs. However, it is unlikely that each program can read and write each other format. If programs cannot exchange data since they don't understand each others format, but both understand a third program, there is a still a way to exchange data with that format. However, this is inconvenient because the user has to find out which programs are candidates for such an operation. Furthermore, data may be lost in a format conversion. This may be because the other format is simply too complicated, or even secret information. There is obviously no way to avoid both issues, but they should not be as bothering as they are.

1.1.4 The Developer's Perspective

From an engineering point of view, it is conceivable that most programs need their own format to represent their data in an efficient way. However, it is not so obvious that the user needs to be bothered with this. First, consider converters. One easy way to get rid of them is to provide one powerful format that has a core part which is understood by all participating applications, and can be extended to meet a particular format's needs. The RTF and SGML formats for marking text are such approaches, The PICT graphics format is a successful example for a graphics format. Second, consider efficiency. It is often the case that a more generalized format is less efficient in terms of storage space or loading time. However, one can provide the choice of saving data either in a native, efficient format, or in a exchange format. Many desktop publishing programs use this approach.

1.2 Intermezzo: What is a graph ?

Definition 1 (Graph) *A graph is a tuple*

$$
\begin{array}{lll}
G & = & (V, E), \text{where} \\
V & & \text{is the set of } \textit{nodes}, \text{ and} \\
E & \subseteq & V \times V \text{is the set of edges}
\end{array}
$$

We also define a mapping label *which assigns information to nodes, edges and labels:*

$$label : G \cup V \cup E \mapsto \Sigma^*$$

NOTE 1. In graph theory, Σ is usually a fixed alphabet; GML needs a more general approach and allows to attach arbitrary attributes to graphs, nodes and edges.

1.3 A Simple GML Example

Example 1.1 shows a simple graph which consists of three nodes. The more complex example 1.2 demonstrates how to attach text to graphs, nodes and edges, and how to specify coordinates and images for nodes and edges. These example illustrates the key elements of GML:

- A GML file is made up of pairs of a key and a value. Examples for keys are `graph`, `node` and `edge`.
- The key idea behind GML is that there are some standard keys like graph, node and edge, and anybody is free to add its keys to add specific information.
- Values can be integers, floating point numbers, strings and lists, where the latter must be enclosed in square brackets.
- The graph in Example 1.2 did not specify how to place the labels. They are arranged in a convenient manner in Figure 2, but an application might also ignore graph labels and print node labels over the images. There is no way to prevent a program from not drawing labels, but we will show in sections A.3.2 and A.4.1 how the placement of labels is specified.

```
graph [                     Defines a new graph
    node [                  Defines a new node
        id 1                This node has the id 1
    ]
    node [                  Defines a new node
        id 2                This node has the id 2
    ]
    node [                  Defines a new node
        id 3                This node has the id 3
    ]
    edge [                  Defines a new edge
        source 1            Source is the node with the id 1
        target 2            Target is the node with the id 2
    ]
    edge [                  Defines a new edge
        source 2            Source is the node with the id 2
        target 3            Target is the node with the id 3
    ]
    edge [                  Defines a new edge
        source 3            Source is the node with the id 3
        target 1            Target is the node with the id 1
    ]
]
```

Figure 1.1: A simple graph in GML (circle of three nodes)

```
# This file is in version 1 of GML
Version 1

graph [

  # This graph has been created by the program "demo"
  Vendor "demo"

  # directed determines whether a graph is directed (1) or not (0).
  # In a directed graph, edges are have arrows that indicate the direction.
  directed 1

  # A label is a text attached to an object
  label "The principles of space travel"

  node [
    id 1
    label "Earth"
    graphics [
      x 0.1
      y 0.0
      w 0.1
      h 0.1
      image "earth.gif"
    ]
  ]

  node [
    id 2
    label "Mars"
    graphics [
      x 0.9
      y 0.0
      w 0.055
      h 0.055
      image "Mars.gif"
    ]
  ]

  edge [
    source 1
    target 2
  ]
]
```

Figure 1.2: A complex GML example

1.4 GML Design Issues

1.4.1 Syntax: Simple or Complex ?

A complex syntax – like a programming language – gives the designer the freedom to express facts in a efficient and easy-to-read manner. The `dot` format is a good example for this practice. However, the price to be paid is a less simple implementation. A simple syntax has the advantage that the implementation is easier, but the format is less efficient in terms of storage space and runtime. However, there is no reason why the format should be less powerful; everything that can be expressed with a complex format can be expressed in a simple format.

Answer: Simple We chose a simple format over a complex one. We loose some efficiency, but we gain a much easier implementation and thus a wider distribution. However, simplicity is limited: since the format needs to be universal, some details will be slightly complex. For example, strings are always terminated by " characters. Therefore, we need a mechanism to deal with a quote that appears inside a string. Other issues are maximum line length and non ASCII characters, such as German umlauts, ä.

1.4.2 Data types

Which types of values do we need to represent? The answer is: all data types present in programming languages. This includes numbers (both integers and floating point), boolean values, characters, strings and composite data types such as record, array, set and list structures.

1.4.3 Constraints

There are several external constraints which have to be considered:

Maximum line length Some systems cannot handle arbitrary long lines without problems. Therefore, we need to restrict line lengths to a size that fits all systems.

Character Set Internationalization is an important issue these days, and the ASCII characters are no longer sufficient for a real application. Therefore, we will use the ISO 8859 character set, which is a common way to code non ASCII character sets within ASCII. ISO 8859 is also used in HTML, so we are in good company.

Range of values The range of numbers is another sensitive point. We will assume 32 bit signed integers and double precision floating point values. These should be supported by all current systems.

This rules out other data types like unsigned integers and 64 bit integers, but those could still be stores as strings and converted afterwards. We will not assume a maximum length for strings, as this would be a restrictions for applications that store long texts in strings.

1.5 Notes on our Notation

We use a Pascal like notation with some object oriented extensions[1]. A GML file is composed of key-value pairs, which we call objects. An object is a parametrized data type:

```
type Object(Type) = record
    key:   Key;
    value: Type;
end;
```

In the following, we will use the types `Object(Integer)`, `Object(Real)`, `Object(String)` and `Object(List(Object))`. We will also assume that the type of an object is available at runtime, as in the following example:

```
case type(o) of
    Object(Integer):       Action for type Integer;
    Object(Real):          Action for type Real;
    Object(String):        Action for type String;
    Object(List(Object)):  Action for type List(Object);
end
```

1.5.1 Paths

The data structure Object forms a tree, where elements of type `Object(Integer)`, `Object(Real)` and `Object(String)` are leaves, and elements of type `Object(List(Object))` are inner nodes. We will frequently use paths in this tree to describe the location of an object.

Definition 2 (Path) *Let $k_1, k_2, \ldots, k_n, n \geq 1$ be keys. The path*

$$.k_1.k_2.\ldots.k_n$$

denotes all sequences of objects $o_1, o_2, \ldots, o_n$ where

$$\begin{array}{rcl} 1 \leq i \leq n & : & o_i.\mathit{key} = k_i \\ 1 \leq i \leq n-1 & : & \mathit{type}(o_i.\mathit{value}) = \mathit{Object(List(Object))} \\ 2 \leq i \leq n & : & o_i \in o_{i-1}.\mathit{value} \end{array}$$

Examples of paths are `.id`, `.graph.label`, `.node.label` and `.node.graphics`. A path describes a class of sequences of objects which can start anywhere in the object tree. We can also define paths that start at a specific object:

Definition 3 (Path Starting at an Object) *Let $k_1.k_2.\ldots.k_n, n \geq 1$ be a path and o be an object. Then*

$$o.k_1.k_2.\ldots.k_n$$

[1] Graphlet's implemention of GML is in C++, but we feel that Pascal is better for for aesthetic reasons, and more people are familiar with Pascal than with the fine prints of C++.

denotes the path starting at o, that is all sequences of objects $o_1, o_2, \ldots, o_n$ *where*

$$\begin{array}{rcl} 1 \leq i \leq n & : & o_i.key = k_i \\ 1 \leq i \leq n-1 & : & type(o_i.value) = List(Object) \\ 2 \leq i \leq n & : & o_i \in o_{i-1}.value \\ & & o_1 \in o.value \end{array}$$

Finally, we omit the leading point on a path if it starts at the root object:

Definition 4 (Path Starting at the Root) *Let* $k_1, k_2, \ldots, k_n, n \geq 1$ *be keys.*

$$k_1.k_2.\ldots.k_n$$

denotes the path

$$R.k_1.k_2.\ldots.k_n$$

where R *is the root object of the tree.*

Chapter 2

Other Graph File Formats

NOTE: This chapter is not complete yet.

2.1 Simple adjacency lists

Many systems use simple adjacency lists, perhaps enriched with labels or coordinates. Often, an adjacency list is terminated by the end of the line. While this format type is convenient and easy to use in these systems, it has several disadvantages for our purpose. First, it is not expandible. Second, labels are usually restricted to one character or a single word. Further, the degree of a node is limited on systems which do not support arbitrary line lengths.

2.2 GraphEd

GraphEd had a format which is in spirit very simpliar to the one which is presented in this paper. However, its syntax is more complex than necessary in several aspects.

```
GRAPH "" =
1 {$ NS 32 32 $} ""
 2 ""
;
2 {$ NS 32 32 $} ""
 3 ""
;
3 {$ NS 32 32 $} ""
 1 ""
;
END
```

This format contains several complications

- There are several ways to represent lists, e.g `"[ ]"`, `"{$  $}"` and `GRAPH...END`.
- Adjacency lists start with a node and end with `";"`, whereas the list of nodes in a graph is terminated by `"END"`.

- Some syntax elements are superficial, like the "=" after the keyword GRAPH.

On the other hand, the format supported generic attributes (inside "{$ $}") which were very similar to the one we propose. The main difference was that GraphEd's attributes had a key and a list of values, where we will only have one value per key. Of course GraphEd's approach made it easier to write files, but the data structures behind that were more complex, since two list structures where needed. GraphEd needed a list of attributes and a list of values, whereas we need only a list of key-value pairs. Another difference is that GraphEd put the graph structure and labels into a syntax outside the attributes, where we will combine them, once again to have only one data structure.

2.3 TEI (SGML) Format

The format used in TEI [reference] is actually a SGML DTD. It has the advantage that SGML provides a powerful standardized framework for it. However, the SGML syntax is more complex than ours, so parsing is more difficult. Our goal is a format that can easily be converted into any other format, so a easy parser is essential. Also, we do not need a format with the power of SGML here.

2.4 VRML Format

VRML is a format who's syntax is quite similar to the one defined here. Basically, it uses a key-value structure. The main difference is that a key can have a list values. The format is extensible.

2.5 dot Format

To be done

2.6 Tom Sawyer Software Format

Tom Sawyer Software, Berkeley makes commercial graph layout and graph editor toolkits. Their file format uses keys in a line started by // , followed by a list of values, each on its line:

```
// Graph Layout Toolkit
Hierarchical Layout
// minimumSlopePercent
20
// Nodes
2
42
```

There is no hierarchical structure, although they can be modelled with dummy begin/end keys. The format is extensible; new elements can be added through a C or C++ interface.

Chapter 3

The GML Format

This section describes the syntax of a GML file, and how scanner and parser for the object tree are constructed. Graphs are of no relevance here; the next chapter will show how to extract a graph from a GML file.

3.1 Syntax

Figure 3.1 shows the syntax of GML in BNF notation. In this format, x^+ denotes a sequence of one or more x items, and x^* denotes a sequence of zero ore more x items. Characters in ‘quotes’ denote terminal characters, and words in $<$*angle brackets*$>$ denote nonterminals.

$<\mathit{GML}>$	$::=$	$<\mathit{List}>$
$<\mathit{List}>$	$::=$	$\lambda \mid <\mathit{KeyValue}>\ (<\mathit{WhiteSpace}>^{+}<\mathit{KeyValue}>)^{*}$
$<\mathit{KeyValue}>$	$::=$	$<\mathit{Key}>\ <\mathit{WhiteSpace}>^{+}\ <\mathit{Value}>$
$<\mathit{Value}>$	$::=$	$<\mathit{Integer}> \mid <\mathit{Real}> \mid <\mathit{String}> \mid \text{‘[’}\ <\mathit{List}>\ \text{‘]’}$
$<\mathit{Key}>$	$::=$	$[\text{‘a’-‘z’‘A’-‘Z’}][\text{‘a’-‘z’‘A’-‘Z’‘0’-‘9’}]^{*}$
$<\mathit{Integer}>$	$::=$	$<\mathit{Sign}>\ <\mathit{Digit}>^{+}$
$<\mathit{Real}>$	$::=$	$<\mathit{Sign}>\ <\mathit{Digit}>^{+}\ \text{‘.’}\ <\mathit{Digit}>^{+}\ <\mathit{Mantissa}>$
$<\mathit{String}>$	$::=$	$\text{‘"’}\ <\mathit{Instring}>\ \text{‘"’}$
$<\mathit{Sign}>$	$::=$	$\lambda \mid \text{‘+’} \mid \text{‘-’}$
$<\mathit{Digit}>$	$::=$	$[\text{‘0’-‘9’}]$
$<\mathit{Mantissa}>$	$::=$	$\lambda \mid \text{‘E’}\ <\mathit{Sign}>\ \text{Digit}^{+} \mid \text{‘e’}\ <\mathit{Sign}>\ \text{Digit}^{+}$
$<\mathit{Instring}>$	$::=$	$ASCII - \{\text{‘\&'}, \text{‘"'}\} \mid \text{‘\&’}\ <\mathit{character}>^{*}\ \text{‘;’}$
$<\mathit{Whitespace}>$	$::=$	$<\mathit{space}> \mid <\mathit{tabulator}> \mid <\mathit{newline}>$

Figure 3.1: The GML Grammar in BNF Format.

3.2 Further specifications

3.2.1 ISO 8859 Character Set

In Figure 3.1, *Instring* excludes the characters `"` and `&` characters from a string. This is necessary because a `"` inside a string would terminate the string prematurely. The `&` character is used by the ISO 8859 character set to introduce a special character. This special character starts with an ampersand, is followed by a name and is terminated by a semicolon. For example, `"` inserts `"`, `&` inserts `&`, and `ä` inserts a German ‘ä’. For a complete list of characters, see tables B.1, B.2, B.3 and B.4 in section B.

We do not allow ISO 8859 characters outside strings, especially not in keywords. Thus, a sloppy parser might not know the ISO 8859 character set, but just ASCII, and can safely read and write GML as 7-bit ASCII files. Many applications do in fact not need to look into the labels, or use only simple labels.

3.2.2 Line Length

The maximum line length in the file format must not exceed **254** characters. This ensures that even systems with a more restrictive line length can cope with a GML file.

3.2.3 Key Syntax

We use a very restricted format for keys, which does not allow characters such as '_', '`$`' or '`:`'. This is because they might not be legal characters for variables in some interpreted languages. With this restriction, keys may be used as identifiers in interpreted languages. It also simplifies the syntax a lot.

3.2.4 Key Size

Because of the maximum line length, keys must less than **254** characters. However, it is quite convenient to have a key and a data item on one line, so it is a good idea to have a key size of less then than 126.

3.2.5 Line breaks

Line breaks may occur anywhere in the file format where white space is allowed. Line breaks inside strings are line breaks in the string[1].

3.2.6 # Comments

Any line which starting with `#` (whitespace *before* the `#` is allowed) is ignored by the parser. This is a standard treatment in most UNIX programs. For example, using the following as a first line

```
#!/usr/local/bin/gmlview
```

[1] This limits strings to a maximum *line width* of 253 characters, which seems reasonable.

specifies that the program `gmlview` interprets the file. It is also common to include foreign data (e.g. a Postscript representation of the graph) through that mechanism. Many drawing programs use the reverse mechanism to insert their data into a Postscript file.

GML includes also a `comment` key which adds comments to a file. However, a parser will read and store these comments, so they should be reserved for *small* comments. Any comments inserted with # should be ignored by the parser.

3.2.7 Order of duplicate keys

It is perfectly legal to have duplicate keys within the same list. For example, an array might be represented as follows:

```
array [
    element [ ... ]
    element [ ... ]
    element [ ... ]
]
```

To avoid problems, we require that the order of *duplicate* keys is *preserved* by the parser. The order of not duplicate keys does need not be preserved. This is because programs might not be able to record the exact order of the attributes in the file. If would also make it more difficult to add more attribues as the file format grows[2].

3.2.8 Unknown Keys

Any parser which encounters an unknown key should preserve it and its value, and write them back when the graph is saved into a file. Exceptions to this policy are changes in the structure, e.g. deletion of the parent of the unknown objects, and consistency problems (see 5).

3.2.9 Default values

One important requirement for GML is that an application which writes a file may omit all "not interesting" keys. However, the following default values should be be assumed for missing key-value pairs:

- **0** for *Integer* values.
- **0.0** for *Real* values.
- `""` for *String* values.
- [] for *List* values.

[2]Of course, one could require that the new attributes are just appended to the old list. However, if a program is written in a modular fashion, the attributes will be written by procedures `p1`, `p2`, ..., `pn` in that order. If `p1` is extended, the new attributes would have to be written *after* `pn`, which would break the modularization.

This makes sure that files with missing keys are treated equally by different programs. For example, one could define that a missing object width is substituted by some default value. But, what should be used ? Common values are 1, 2, 16, 32, 42 and 64, or the *current default setting of the program.* However, especially the last variant is highly dependend on the current state of the program, and might lead to overlapping objects and therefore hard to read drawings.

Nevertheless, a program may (and should) implement an option "substitute defaults for missing values". Such a clean up operation[3] which is available by request is less confusing than the substitution of system state dependend values.

3.3 Graphlet's Parser Implementation

It remains to show how to construct an Object tree from a GML file. This is done while parsing the second and third rule in the syntax definition.

List `::=` *KeyValue* (*WhiteSpace*$^+$ *KeyValue*)*
  var *l*: List(Object);
  l := emptyList;
  foreach *KeyValue* in the list **do**
    o := new Object(*KeyValue.Key*);
    o.value := *KeyValue.Value*;
    append *o* to *l*;
  done
  return l;

Value `::=` *Integer* |
  return *Integer*;
 Real |
  return *Real*;
 String |
  return *String*;
 `[` *List* `]`
  return *List*;

3.4 How to represent common data structures

As earlier said, GML is by no means restricted to graphs. In fact, all common data types can be represented in GML. This section is a cookbook for designing data structure representations in GML.

3.4.1 Boolean

Boolean values can be represented by Integers. `false` is represented by 0, `true` is represented by any other number.

NOTE 1. We decided not to implement a separate datatype for boolean values because this would only complicate the parser.

[3]Graphlet provides such operations in the "Tool" menu under the keyword "Clean up".

3.4.2 Large numbers

Large integer or floating point data types can be represented as Strings. The string is the standard ASCII representation of the value.

NOTE 1. A more compact representation for large integer values could be obtained by coding the bits of the number in the same fashion as the `uuencode` program which is available on UNIX systems. However, this would assume that the layout of the bits in memory is fixed (which is *not* the case), and would also complicate the parser. It is also not clear how many applications would need this.

NOTE 2. If space is an issue, An application may encode very large integer numbers as a list if integers in a 2^{15}- or 2^{31}-adic number system.

3.4.3 Bitset

A bitset should be represented as a string of 0 and 1 characters.

NOTE 1. The same as for large numbers (section 3.4.2) applies here, too.

3.4.4 Records

A record data type

```
name: record
    a: type1
    b: type2
    ...
end
```

can be represented in GML as follows:

```
name [
    a value_of_a
    b value_of_b
    ...
]
```

In place of *value_of_field*, insert the value of the corresponding field of the record. This can either be an integer, a real or a string value or a list.

3.4.5 Lists, Sets, Arrays

A list of data type

```
name: List(SomeType)
```

can be represented in GML as follows:

```
name [
  obj value_of_first_element
  obj value_of_second_element
  ...
]
```

where `obj` should be replaced by a suitable name for the elements of the list. Section 3.2.7 specifies that a parser must preserve the order of the `obj` attributes when it reads the program.

Sets and arrays are represented with exactly the same schema. In the case of arrays, it might be neccessary to use a list for *value_of_field* and add an `id` key to the value. This `id` represents the index of the array element. Associative arrays might use a `name` key or some more complex structure.

3.4.6 Name clashes

The freedom of adding new keys bears the problem of name clashes. To avoid this problem as much as possible, any application should put its private information in a object of type list who's private key represents the name of the application. For example, the Graphlet system will insert all private information into a an object with key named `Graphlet`[4].

[4] Of course, this is still not 100% save, but as close as we can get with a reasonable effort. HTML uses the same approach and works still fine after all these years.

Chapter 4

Extracting Graphs from GML files

Up to this point, GML was not related to graphs in any way. In fact, GML is designed so that it can map any data structure onto an ASCII file. To extract a graph from a GML file, we parse the file and extract the list of Object structures. Then, we run through the objects and extract the graph structure from them:

var *objects*: List(Object);

objects := parse (file);
foreach o **in** *objects* **where** o.key = "Graph" **do**
    g := **new** graph;
    foreach $\bar{o}$ **in** o.value **where** $\bar{o}$.key = "Node" **do**
        n := **new** node (g);
        n.attributes := $\bar{o}$.value;
        remove $\bar{o}$ **from** o.value;
    done
    foreach $\bar{o}$ **in** o.value **where** $\bar{o}$.key = "Edge" **do**
        e := **new** edge ($\bar{o}$.source.value, $\bar{o}$.target.value);
        e.attributes := $\bar{o}$.value;
        remove $\bar{o}$ **from** o;
    done
    g.attributes := o.value;
    remove o **from** *objects*;
done

To write a graph to a GML file, use the following schema:

procedure print (g: Graph);
begin
    print "`graph [`";
    print (g.attributes);
    foreach n **in** g.nodes **do**
        print "`node [`";

```
        print (n.attributes);
        print ]";
    done
    foreach e in g.edges do
        print "edge [";
        print e.attributes;
        print "]";
    done
    print "]";
end

procedure print (objects: List(Object))
begin
    foreach o in objects do
        print o.key;
        case type(o) of
            Integer :
                print o.value;
            Real :
                print o.value;
            String :
                print o.value; must ensure line length < 255
            List :
                print "["
                print o.value;
                print "]";
        end
    done
end
```

In the above program, we assume that a statement **print** s writes s with the following properties:

- s is surrounded by quotes.
- The output is adjusted to a maximum line length of 254 characters, inclusive quotes.
- All characters are properly translated into the ISO 8859 character set. Especially, the `&` and `"` characters are replaced by `&` and `"`.

It is a good idea to save all objects in the file and print them out again, unless they are unsafe and changes to the graph have been made (see also chapter 5). This will make sure that valuable information supplied by other programs wont get lost.

Chapter 5

Consistency: Safe and Unsafe Objects

With GML, we can specify objects that depend on other objects. That is, changing one object forces other objects to be updated. There are many examples for such behaviour:

Edge coordinates Edge coordinates must be updated when the endnodes of the edge move. Technically, if a node is removed, all adjacent edges must be removed.

Label coordinates The same as above applies to label coordinates.

Graph Theory Information on graph theoretical properties, such as planarity, is often needed by applications and should therefore be added to a graph. This may save costly recomputation (if an appropriate algorithm is available at all). This information must be updated whenever the structure of the graph changes.

Maximum and Minimum Coordinates Some applications need information on the maximum or minimum x and y coordinates. They must be updated whenever a node or edge changes *position* or *size*.

Size of the graph Information on the size of the graph, that is the number of nodes and/or edges is also an often found information in graph file formats. However, this information must be updated whenever the structure of the graph changes.

In some cases, it is quite obvious how to handle updates. On the other hand, in the case of Graph theoretic properties, it might be difficult and time costly to update, if at all feasible.

There are many solutions to this problem. Besides omitting all potentially dangerous keys, a simple solution would be to define consistency conditions for a few known objects and forbid any other consistency violating objects. This would especially mean that graph theoretical properties, which might be the result of several hours computing, are ruled out.

The most complex solution would be to give a precise specification for the dependencies of each object, and force updates. However, this would lead to a very complex file formant, and would be difficult to implement.

Therefore, we settle for the following solution: It is legal to add objects which become inconsistent after changes, but a a hint must be given that the key bears a potential consistency problem.

Definition 5 (Safe and Unsafe Objects) *We call any object which must be updated or removed upon a change unsafe. We discriminate safe and unsafe objects by their key:*

- *An object who's key starts with a* ***lower case*** *letter starts is* safe*.*
- *An object who's key starts with a* ***capital letter*** *starts is* unsafe*.*

Any program that changes (a) the topoligical structure of the graph or (b) an attribute must delete all objects it cannot update properly.

Appendix A

Predefined Keys

This section lists predefined keys. The design of GML leaves applications any freedom to define their own keys. However, in order to maintain compatibility with other applications, standard keys should be used as much as possible. The entries in the lists below will be as follows:

`Key` *Type* `Context` **Save/Unsafe**

where

`Key` is the (name of the) GML Key.

Type is the type of the value. We use the following types:

- `Integer` is an integer value.
- `Real` is a floating point value.
- `String` is a string value.
- `List` is a list value. *true*.

`Context` specifies a path whichmust be the prefix for the key:

- If the context is "`.a.b.c`" and the key is "`x`", then this key must be on a path "`.a.b.c.x`". If the type is not *List*, then `x` is obviously a leaf in the object tree.
- If the context is "`.`", then the key may occur everywhere.
- If the context is empty, then this key is defined only if it occurs at the top level

Safe, Unsafe specifies whether this object should generally be regarded as safe (see Chapter 5).

A.1 Global Attributes

The following attributes can be used with all objects.

`id` *Integer* `.` **Safe**
An id is an identifier for an object. The values of id objects do not need to be unique throughout a file; this is defined by the application.

`label` *String* . **Safe**
Defines the label of an object. A graphical browser should use this attribute to annotate the object with a text.

`comment` *String* . **Safe**
Defines a comment. Note that such a comment will be read *and* saved. This attribute is **safe**, so handle with care. There is no unsafe version of `comment` since this would not be helpful as comment is too general.

`Creator` *String* . **Unsafe**
The program which created this object. This attribute is unsafe since every program should register its own name here, or forget about it.

`name` *String* . **Safe**
Defines the name of an object. The name is a textual alternative to the `id` attribute.

A.2 Top Level Attributes

The following attributes are only defined at the top level of a file, that is they may have no prefix.

`graph` *List* **Safe**
Defines a graph. In the current version of GML, the top object is almost always a graph.

`Version` *Integer* **Unsafe**
Describes the GML version of this file. Current value is 1. Files with an unknown version number should be rejected.

`Creator` *String* **Unsafe**
The name of the program which created this file.

The fact that these attributes are only defined at the top level does not mean that they can not be defined at other locations. For example, given a proper semantics, graphs can be defined inside graphs.

A.3 Graphs

A.3.1 Graph Attributes

`directed` *Boolean* `.graph` **Safe**
Defines whether a graph is directed (0) or undirected (1). Default is undirected.

`node` *List* `.graph` **Safe**
Defines a node. Each node should have an id attribute, which must be unique within all node.id attributes of a graph object.

`edge` *List* `.graph` **Safe**
Defines a new edge..edge.source and.edge.target reference the endnodes of an edge

A.3.2 Node Attributes

A node is usually identified by its id attribute, and often has a label attributes Its graphical properties are described by the graphics attribute.

`id` *Integer* `.graph.node` **Safe**
Defines an identification number for the node. All `id` numbers within a graph must be unique.

`edgeAnchor` *String* `.graph.node` **Safe**
Defines how the edges are attached to the node.

A.4 Graphics Attributes

Generally, the graphics attributes in GML are modelled after the graphics in Tk.

`x` *Real* `.graphics` **Safe**
Defines the x coordinate of the center of the bounding rectangle of the object.

`y` *Real* `.graphics` **Safe**
Defines the y coordinate of the center of the bounding rectangle of the object.

`z` *Real* `.graphics.center` **Safe**
Defines the z coordinate[1] of the center of the bounding rectangle of the object.

`w` *Real* `.graphics` **Safe**
Define the width of the bounding box of the node. If omitted, the width is 0.

`h` *Real* `.graphics` **Safe**
Define the height of the bounding box of the node. If omitted, the height is 0.

`d` *Real* `.graphics` **Safe**
Define the depth[2] of the bounding box of the node. If omitted, the depth is 0.

`type` *String* `.graphics` **Safe**
Defines the graphical object. Values for `type` are `arc`, `bitmap`, `image`, `line`, `oval`, `polygon`, `rectangle` and `text`.

NOTE 1. The current implementation of Graphlet assumes that edges always have the type `line`.

NOTE 2. The current implementation of Graphlet assumes that labels always have the type `text`.

[1] Not yet implemented in Graphlet.
[2] Not yet implemented in Graphlet.

`image` *String* `.graphics` **Safe**

The value of `image` is the name of an image file which is used to draw an object. If no `w`, `h` and `d` attributes are given, the application my use the images dimensions to scale the graphics. Otherwise, the image is scaled to fit into the given dimensions.

NOTE 1. The actual image capabilities may depend on the graphics subsystem used by the program[3]. For example, not all systems can display pictures in JPEG format, or scale them. Programs should replace the image by a dummy object if sufficient graphics capabilities are not available.

NOTE 2. The corresponding`image` attribute in GraphScript and GraphScript's C++ interface is slightly different as this attribute needs a Tcl/Tk `image` object instead of a filename. We are using a different approach here to be more portable.

NOTE 3. Images are always stored in separate files. This is because application programmer interfaces for graphics file formats almost always support reading from a file, while converting from a string in a file is usually not supported.

`bitmap` *String* `.graphics` **Safe**

The value of `bitmap` is the name of a bitmap file which is used to draw an object. A `bitmap` differs from an `image` in that the bitmap has only a foreground and a background color, while images may have their own arbitrary color table.

`point` *List* `.graphics.Line` **Safe**

Defines a point of the polyline that is used to draw the edge. A straight line edge does not need to specify points. If points are specified, they must include the end points of the edge.

The following example shows how to use the `Line` attribute:

```
graphics [
    Line [
        point [
            x 10.0
            y 10.0
        ]
        point [
            x 100
            y 20.0
        ]
        point [
            x 20.0
            y 20.0
        ]
    ]
]
```

The example specifies a line with three points. In reality, this corresponds to an edge with one bend (the first and the last `point` entries are the endpoints of the edge).

[3] Graphlet can read GIF and JPEG files through Tk, but cannot scale them in the current version.

NOTE 1. If the `Line` attribute of an edge is omitted, an application should substitute a straigt line between the endpoints of the edge.

NOTE 2. `Line` us *unsafe* because the coordinates become invalid as soon as the endpoints are moved.

`point` *List* `.graphics.Line` **Safe**
: Defines a point of the polyline that is used to draw the edge. A straight line edge does not need to specify points. If points are specified, they *must* include the end points of the edge.

`point.x` *Real* `.graphics.Line` **Safe**
: Define the z coordinate of a point.

`point.y` *Real* `.graphics.Line` **Safe**
: Define the z coordinate of a point.

`point.z` *Real* `.graphics.Line` **Safe**
: Define the z coordinate of a point.

`width` *Real* `.edge.graphics` **Safe**
: Sets the line width (in pixels). If nothing is specified, 0.0 should be assumed.

`stipple` *String* `.edge.graphics` **Safe**
: Defines a stipple pattern to draw the line (if the drawing system supports that).

This section is subject to be extended.

A.4.1 Edge Attributes

Each edge must have attributes source and target to specify its endnodes.In an undirected graph, there is no graph theoretical distinction between source and target, but the coordinates of the drawing might impose a direction on the edge.

`source` *Integer* `.edge` **Safe**
: In a directed graph, `source` defines the source node of an edge. In an undirected graph, source defines one of the endpoints of an edge.

`target` *Integer* `.edge` **Safe**
: In a directed graph, `target` defines the source node of an edge. In an undirected graph, target defines one of the endpoints of an edge.

Appendix B

ISO 8859 Characters

Tables[1] B.1, B.2, B.3 and B.4 present an overview of the ISO 8859-1 standard character set.

Description	**Character**	`Code`	`Entity Name`
quotation mark	”	`"`	`&`
ampersand	&	`&`	`&`
less-than sign	$<$	`<`	`<`
greater-than sign	$>$	`>`	`>`

Table B.1: ISO 8859-1 Characters, basic

NOTE 1. Since LaTeX does not use the 8859-1 character set, the above table is only an approximation.

NOTE 2. Entries marked “???” in the above table will be filled in the final version of this document.

[1] This is an adapted version of Martin Ramsch's ISO 8859 tables, available on the world wide web at `http://www.uni-passau.de/ ramsch/iso8859-1.html`. The author has granted us the permission to modify and use his table

Description	**Character**	`Code`	`Entity Name`
non-breaking space	␣	` `	` `
inverted exclamation	¡	`¡`	`¡`
cent sign	???	`¢`	`¢`
pound sterling	£	`£`	`£`
general currency sign	???	`¤`	`¤`
yen sign	???	`¥`	`¥`
broken vertical bar	|	`¦`	`¦`
section sign	§	`§`	`§`
umlaut (dieresis)	¨	`¨`	`¨`
copyright	©	`©`	`©`
feminine ordinal	???	`ª`	`ª`
left angle quote, guillemotleft	???	`«`	`«`
not sign	???	`¬`	`¬`
soft hyphen	???	``	``
registered trademark	???	`®`	`®`
macron accent	???	`¯`	`¯`
degree sign	???	`°`	`°`
plus or minus	???	`±`	`±`
superscript two	???	`²`	`²`
superscript three	???	`³`	`³`
acute accent	???	`´`	`´`
micro sign	???	`µ`	`µ`
paragraph sign	???	`¶`	`¶`
middle dot	???	`·`	`·`
cedilla	???	`¸`	`¸`
superscript one	???	`¹`	`¹`
masculine ordinal	???	`º`	`º`
right angle quote, guillemotright	???	`»`	`»`
fraction one-fourth	???	`¼`	`¼`
fraction one-half	???	`½`	`½`
fraction three-fourths	???	`¾`	`¾`
inverted question mark	???	`¿`	`¿`
division sign	???	`÷`	`÷`

Table B.2: ISO 8859-1 Characters, Special and Mathematical Characters

Description	**Character**	Code	Entity Name
capital A, grave accent	???	`À`	`À`
capital A, acute accent	???	`Á`	`Á`
capital A, circumflex accent	???	`Â`	`Â`
capital A, tilde	???	`Ã`	`Ã`
capital A, dieresis or umlaut mark	???	`Ä`	`Ä`
capital A, ring	???	`Å`	`Å`
capital AE diphthong (ligature)	???	`Æ`	`Æ`
capital C, cedilla	???	`Ç`	`Ç`
capital E, grave accent	???	`È`	`È`
capital E, acute accent	???	`É`	`É`
capital E, circumflex accent	???	`Ê`	`Ê`
capital E, dieresis or umlaut mark	???	`Ë`	`Ë`
capital I, grave accent	???	`Ì`	`Ì`
capital I, acute accent	???	`Í`	`Í`
capital I, circumflex accent	???	`Î`	`Î`
capital I, dieresis or umlaut mark	???	`Ï`	`Ï`
capital Eth, Icelandic	???	`Ð`	`Ð`
capital N, tilde	???	`Ñ`	`Ñ`
capital O, grave accent	???	`Ò`	`Ò`
capital O, acute accent	???	`Ó`	`Ó`
capital O, circumflex accent	???	`Ô`	`Ô`
capital O, tilde	???	`Õ`	`Õ`
capital O, dieresis or umlaut mark	???	`Ö`	`Ö`
multiply sign	???	`×`	`×`
capital O, slash	???	`Ø`	`Ø`
capital U, grave accent	???	`Ù`	`Ù`
capital U, acute accent	???	`Ú`	`Ú`
capital U, circumflex accent	???	`Û`	`Û`
capital U, dieresis or umlaut mark	???	`Ü`	`Ü`
capital Y, acute accent	???	`Ý`	`Ý`
capital THORN, Icelandic	???	`Þ`	`Þ`

Table B.3: ISO 8859-1 Characters, Capital Letters

Description	**Character**	`Code`	`Entity Name`
small sharp s, German (sz ligature)	???	`ß`	`ß`
small a, grave accent	???	`à`	`à`
small a, acute accent	???	`á`	`á`
small a, circumflex accent	???	`â`	`â`
small a, tilde	???	`ã`	`ã`
small a, dieresis or umlaut mark	???	`ä`	`ä`
small a, ring	???	`å`	`å`
small ae diphthong (ligature)	???	`æ`	`æ`
small c, cedilla	???	`ç`	`ç`
small e, grave accent	???	`è`	`è`
small e, acute accent	???	`é`	`é`
small e, circumflex accent	???	`ê`	`ê`
small e, dieresis or umlaut mark	???	`ë`	`ë`
small i, grave accent	???	`ì`	`ì`
small i, acute accent	???	`í`	`í`
small i, circumflex accent	???	`î`	`î`
small i, dieresis or umlaut mark	???	`ï`	`ï`
small eth, Icelandic	???	`ð`	`ð`
small n, tilde	???	`ñ`	`ñ`
small o, grave accent	???	`ò`	`ò`
small o, acute accent	???	`ó`	`ó`
small o, circumflex accent	???	`ô`	`ô`
small o, tilde	???	`õ`	`õ`
small o, dieresis or umlaut mark	???	`ö`	`ö`
small o, slash	???	`ø`	`ø`
small u, grave accent	???	`ù`	`ù`
small u, acute accent	???	`ú`	`ú`
small u, circumflex accent	???	`û`	`û`
small u, dieresis or umlaut mark	???	`ü`	`ü`
small y, acute accent	???	`ý`	`ý`
small thorn, Icelandic	???	`þ`	`þ`
small y, dieresis or umlaut mark	???	`ÿ`	`ÿ`

Table B.4: ISO 8859-1 Characters, Lowercase Letters