Contents

1 Introduction **1**

2 Syntactic Rules of ObjectMath and Mapping between Textual and Graphical Representation **3**

3 The Textual Syntax of ObjectMath **3**

4 Graphical Representation of ObjectMath Models **5**
4.1 The ”Violation Impossible” Rules 6
4.2 Permanently Checked Rules 6
4.3 Rules Checked when an Editor Session is Finished 7

5 Operations of ObjectMath diagram editor **7**
5.1 Adding Objects and Relations 7
5.2 Deleting objects 8
5.3 Moving Objects 8
5.4 Other Operations 8
5.5 Completeness and Correctness 9

6 Layout **9**

7 Conclusions **10**

1 Introduction

ObjectMath is a mathematical modeling language and an integrated environment for programming in this language [1]. This language combines computer algebra capabilities available in Mathematica[2] with object oriented constructs. One of the reasons to introduce object-orientation into Mathematica is to facilitate *reuse* of functions and expressions. Knowledge about abstract and real world objects and their numerical properties can be expressed in variables, functions and equations. However, in order to reuse them, these variables and equations should be encapsulated into classes, which in turn can be gathered into reusable class libraries.

The use of inheritance facilitates reuse of equations and formulae. It appears that in the case of ObjectMath, like in many other object-oriented languages, an application might include many levels of inheritance. In addition, some classes are composed of objects which instantiate other classes. Multiple inheritance allows constructing classes from other classes combining different orthogonal kinds of functionality.

Typical ObjectMath models contain from 10 to 30 classes and instances. It is hard to get an overview of the whole system without graphical presentation of relations between classes and instances. This is the reason why a graphical editor is necessary. Initially an class diagram editor was developed for ObjectMath 3.0 by Lars Viklund and Rickard Westman. Figure 1 of [1] shows this graphical editor in use. This editor was able to display directed graphs without loops only and therefore could not be used for multiple inheritance and for part-of relations between classes. We developed a new editor for ObjectMath, version 4.0, in 1993.

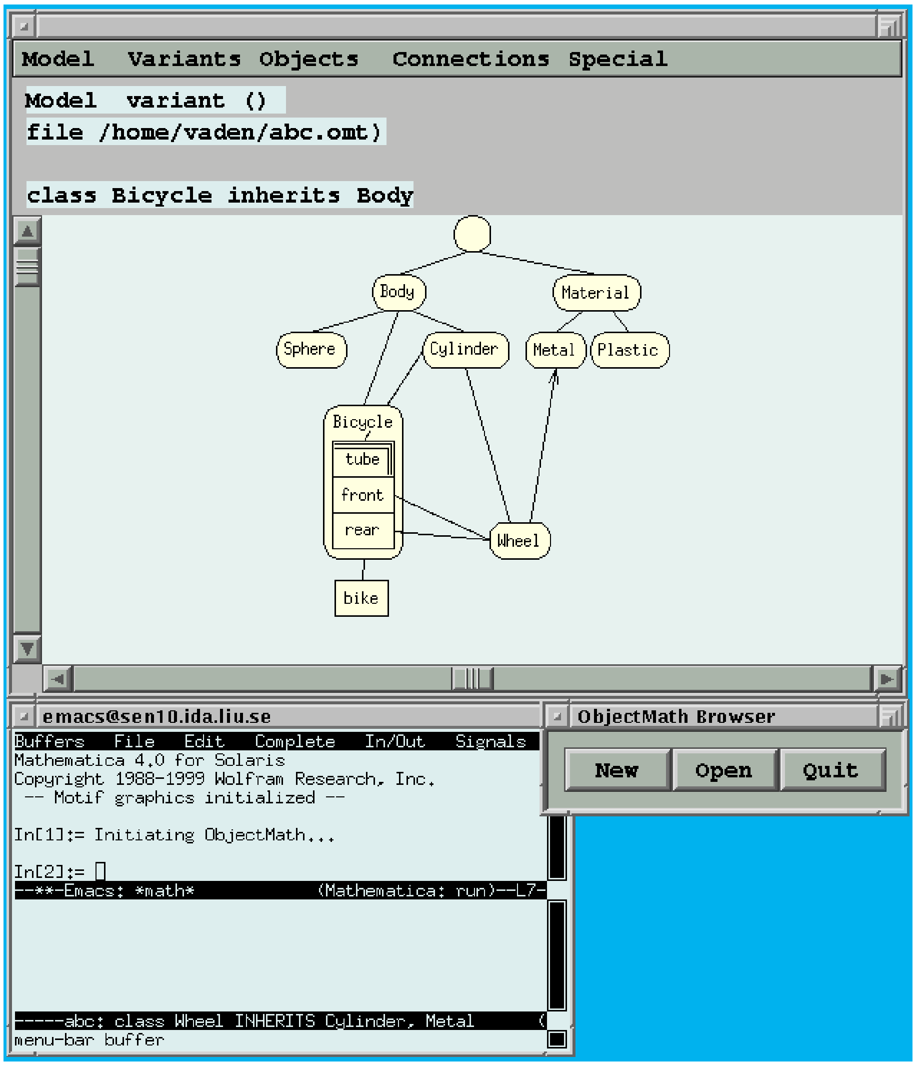

Figure 1: *The ObjectMath environment consisting of a diagram editor window, a program text window and the start window.*

This report contains:

- formal definition of the syntactic rules for relations between object-oriented constructs in ObjectMath;
- definition of a mapping between models in ObjectMath and class relationship (i.e. inheritance and composition) diagrams;
- definition of operations on relationships between object oriented constructs that can be performed by the user;
- definition and implementation of the above mentioned mapping and operations in a graphical class diagram editor.

2 Syntactic Rules of ObjectMath and Mapping between Textual and Graphical Representation

In this section we give an example of textual representation of an ObjectMath model. There are grammar rules (context-free and context-dependent) defining the ObjectMath syntax (Section 3).

The graphical representation has been designed in order to reflect the structure of ObjectMath models written according to the rules. In Section 4 we introduce the ObjectMath graphical notation and explain the mapping of syntax rules into operations of the diagram editor.

3 The Textual Syntax of ObjectMath

We consider an ObjectMath model of a bicycle consisting of two wheels (front and rear) and five tubes (cylinders). Currently we ignore all class variables and all equations since they do not affect graphical user interface and are not shown in the diagram. The ellipsis (`...`) mark the places where variable declarations, functions and equations can be inserted.

The Figure 2 contains an ObjectMath model of a bicycle in textual notation.

```
model abc;
                                        ...
class Body      ...    end;
class Material ...     end;
class Metal          inherits Material ... end;
class Plastic(b,b)   inherits Material ... end;
class Sphere         inherits Body     ... end;
class Cylinder       inherits Body     ... end;
class Wheel inherits Cylinder, Metal ... end;
class Bicycle inherits Body
    part tube[five] inherits Cylinder;
    part front      inherits Wheel(t);
    part rear       inherits Wheel;
    ...
end;
instance bike inherits Bicycle  ...  end;
```

Figure 2: *An ObjectMath example (bicycle model) in textual notation.*

To explain the components of the ObjectMath textual representation further we describe the syntax of ObjectMath using extended BNF notation. The following syntactic meta symbols are used in this description:

- `[ ]` – optional grammar elements.
- { } – grammar elements repeat zero or more times.
- | – alternative elements.
- Keywords are written in **bold** face.

This description (Figure 3) contains 11 formal rules of the ObjectMath grammar, and 5 rules in natural language defining context dependencies. There are many more rules which specify how *variables and equations* are written in ObjectMath. These rules are ignored here since they do not affect graphical user interface and are not shown in the diagram.

Rule 8 should be explained in more detail using mathematical notation.

Assume that a graph consists of k nodes $\{n_1, ..., n_k\}$ and m directed edges $\{(n_{1_1} \rightarrow n_{1_2}), ..., (n_{m_1} \rightarrow n_{m_2})\}$. A directed loop is a sequence of edges $((n_{j_1} \rightarrow n_{j_2}), (n_{j_2} \rightarrow n_{j_3}), ..., (n_{j_p} \rightarrow n_{j_1}))$, where $p > 0$. In Rule 8 we represent all classes, parts and instances as nodes. An edge $(n_a \rightarrow n_b)$ exists if n_b inherits class n_a or n_b contains n_a. The rule states that if an ObjectMath model is correct there *should be no directed loops* in the graph.

```
1 program   ::= model  model-name ";"
              equations
             { definition }
2 definition ::= ( class  class-name formal-parameter-list
              | instance instance-name  [ array-definition ] )
              inheritance
              { part-definition }
              equations
              end ";"
3  Rule: class-names and instance-names are unique within
the model.
4  part-definition    ::= part  part-name [ array-definition ]
                        inheritance  ";"
5  Rule: part-names   are unique within the container,
    i.e. with the class or instance  definition.
6  inheritance     ::= [ inherits
                    class-name actual-parameter-list
                    { "," class-name actual-parameter-list}]
7  Rule: Inherited class name should be defined somewhere else
   in the hierarchy. This class is called a superclass.
8  Rule: If inheritance and composition relations are
   represented as directed edges of a graph they cannot
   form loops (i.e.  directed loops)
9  Rule: number of actual class parameters in the inherits
   clause matches the number of formal class parameters in the
   class definition.
10  array-definition      ::=    "[" name "]"
11  actual-parameter-list ::=    name {"," name}
12  formal-parameter-list ::=    name {"," name}
13  class-name            ::=    name
14  instance-name         ::=    name
15  part-name             ::=    name
16  equations             ::=    ...
A name is an arbitrary identifier.
```

Figure 3: *The grammar and syntax rules of the ObjectMath textual notation.*

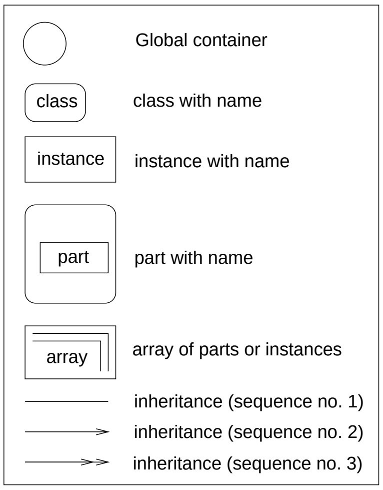

Figure 4: *Graphical notation for ObjectMath class diagrams. The inheritance relations are numbered because the order of classes in case of multiple inheritance affects the program semantics. The container for global objects (*Global container*) is used for two purposes. First it contains global variables, functions and equations which do not belong to any particular instance. Second, the icon of* Global container *is connected to all classes and instances that have no superclasses.*

4 Graphical Representation of ObjectMath Models

We suggest a notation for ObjectMath diagrams, shown in Figure 4. The example in textual notation (Figure 2) corresponds to the diagram depicted in Figure 1. This notation is sufficient for static (non-changeable) graphical representation. However, in order to work with such diagrams in an interactive environment a mapping between syntactical rules and operation of the editor should be designed.

The practice of development of interactive environments for diagram editing suggests that there can be three different strategies defining how the syntactic rules are enforced:

Violation impossible: All manipulations with a diagram that can lead to erroneous diagrams are not available in the tool.

Permanent check: The tool includes a special feature for checking diagram correctness at any time during editing. All editing operations that lead to incorrect diagram are canceled, and therefore cannot be applied.

Commit check: The tool checks correctness when editing of a diagram is finished and the diagram is saved or exported. Some rules might be violated unless the diagram is used by some other tool, e.g. a compiler.

All three these types of rules occur in the ObjectMath diagram editor.

4.1 The ”Violation Impossible” Rules

The violation of the following rules (the rule numbers refer to Figure 3) is not possible in the ObjectMath graphical notation used in the diagram editor:

Rule 1: The diagram consists of icons. The order of the definitions does not affect ObjectMath semantics. Therefore the icons can be placed in any order.

Rules 2 and 10: there are class and instance icons; there is an instance array icon.

Rule 4 and 10: there are part icons placed within a class or instance icon. There is a part array icon (same as the instance array icon).

Rule 6: Relations between icons are shown as connection lines. Some lines have arrow heads. The order in which classes are inherited (in case of multiple inheritance) affects the semantics. In the case of conflicts between definitions inherited via two different superclasses, the definitions inherited via the first inheritance override the definitions inherited via the second inheritance relations. A line without arrows is used for single (or the first) inheritance relation. The direction of this relation is not shown in the diagram, but it is obvious in most cases. A line with n arrows is used for the $(n + 1)$-th inheritance relation.

Rule 7: All connection lines represent inheritance relations and such lines must go from some icon to a superclass icon.

Rules 13, 14, 15: The names are shown in the icons.

4.2 Permanently Checked Rules

The operation of the editor includes some checks which enforce the following rules:

Rules 3, 5: Names cannot be repeated in the diagram. Each time a new class, instance, or part is created (or renamed), its name is checked.

Rule 8: No directed loops can be created when diagrams are modified. This rule cannot be easily enforced by graphical notation. Non-directed loops are allowed since multiple inheritance exists. Directed loops are not allowed since inheritance cannot form a loop[1].

Rules 11, 12: The parameters are not shown in the graphical notation. However, when an icon or a connection line is created or selected for editing, the parameters are displayed in the dialog window.

[1]One way to obey this rule in the graphical interface is to guarantee that all the icons with inheritance relations should be placed so that the inherited class is located *above* a subclass or an instance. Editing diagrams using such editor is, however, quite difficult and is not convenient.

Before each diagram modification that can potentially lead to violation of rule 8, the new model is checked and the modification is not allowed if the rule is violated. Rules 3, 5, 11, 12 are checked each time the corresponding icon or connection is created or modified. If these rules are violated, the operation is canceled.

4.3 Rules Checked when an Editor Session is Finished

The following rule is checked each time the model is read, saved or sent for compilation:

Rule 9: Comparison of number of actual and formal parameters passed at inheritance. The semantics of class parameter passing in ObjectMath is explained in [1], Section 3.3.

This rule is violated, for instance in the case of the part `front` (see Figure 2). The **`inherits`** clause contains the class `Wheel` with *one* parameter. The number of the parameter names can be changed if the line representing an inheritance relation is selected and the relation is edited. The class `Wheel`, however, has *zero* parameters. This can be changed when the class `Wheel` icon is selected and its parameters are modified. These two places (class `Wheel` properties and the properties of inheritance relation between `Wheel` and `front`) can be edited by the user independently and in arbitrary order. Therefore the violation of this rule cannot be forbidden. Instead, when the model is read, saved, or sent for compilation, a message about this error appears.

5 Operations of ObjectMath diagram editor

The operations that can be performed with the diagram editor are presented in Figure 5.

The major operations that affect the diagram are adding, deleting and moving objects.

5.1 Adding Objects and Relations

If the currently selected object is a class, then another class (a subclass), or an instance (of this class) can be created. An inheritance relation is established. Also, a part can be added to this class.

If the currently selected object is an instance, a part can be added to this instance.

When a class is created, its formal parameters must be specified. When an instance, a part or a class is created the actual parameters are specified.

To create a new inheritance relation the user selects a subclass or a part, chooses the corresponding menu item, and then selects the parent class.

These operations allow construction of an arbitrary syntactically correct class relationship diagram from the empty diagram.

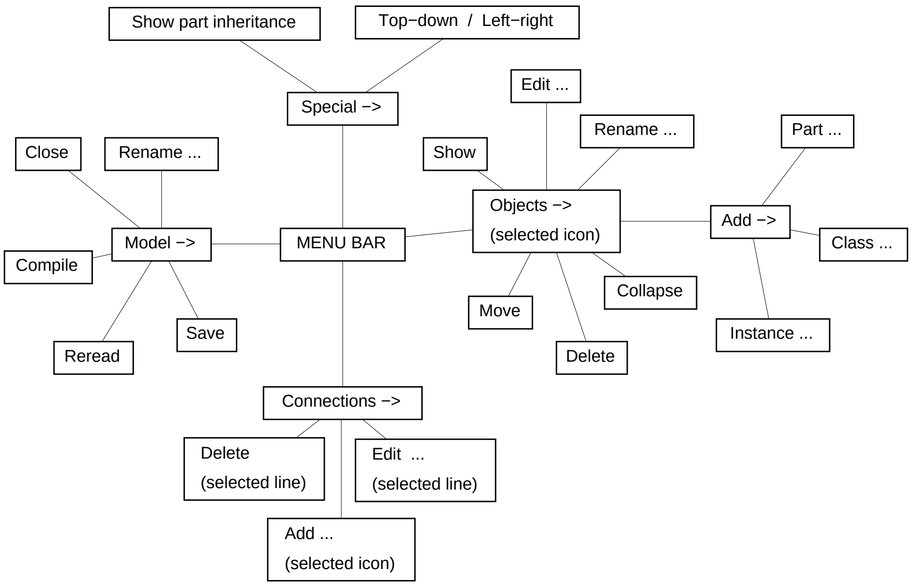

Figure 5: *Menu choices of the ObjectMath class diagram editor. The alternatives leading to new dialogs are marked with ellipsis (* **. . .** *)*

5.2 Deleting objects

If a class is deleted, all descendants of the class are deleted, too.

When the last left inheritance relation to the superclass is deleted, the subclass is moved to the top level of the hierarchy (i.e. becomes attached to the *Global container*).

When a class or an instance is deleted all its components are deleted, too.

This way an empty diagram (i.e. a diagram with a single *Global container* icon) can be created from an arbitrary diagram.

5.3 Moving Objects

A class or an instance can be moved. First the icon of a class or an instance is selected, then the menu item is chosen, and finally a new parent class is selected.

Also a part can be moved. First the icon of a part is selected, then the menu item is chosen, and finally a new container class is selected.

5.4 Other Operations

There are several other operations in the editor:

Edit A class, instance, part, or connection can be selected and its attributes (parameters, inheritance number, array index) can be edited.

Rename Classes, instances and parts can be renamed.

Show Classes and instances have textual part (i.e. variables, functions, equations) can be displayed in the text editor and modified there.

Collapse When a class is collapsed all its descendants are temporarily hidden.

5.5 Completeness and Correctness

The operations mentioned above have two important properties:

Completeness: For any two syntactically valid class relationship diagrams A and B there exist a sequence of operations that transforms A to B.

This can be done in many different ways, for instance:

- all nodes of A can first be deleted;
- only the *Global container* is left;
- all the class nodes of diagram B can be added in order from the top classes to the bottom classes of the inheritance hierarchy;
- all instances are added;
- all parts are added;
- all additional inheritance relations in case of multiple inheritance are added.

Correctness: If a sequence of editing operations is applied to a syntactically correct diagram, the resulting diagram will be correct.

This is performed by checking the diagrams for violation of the rules as described in section 2.

6 Layout

Each diagram editor should have facility for automatic layout of the icons in a diagram. In this section we discuss the layout of a directed acyclic graph. The ObjectMath diagram is not acyclic, but it has an acyclic subgraph. We also discuss how users modify the default layout.

The editor uses an automatic layout algorithm for acyclic directed graphs. In this algorithm the root of the tree is placed at the middle on the top border of the diagram window. The area reserved for the parent node (`P` in Figure 6) has the same width as the sum of all the widths of the child nodes (`C1`, `C2`, `C3`) and the gaps of width `G1` between those. The child nodes are placed below the parent node at a distance of `G2` from each other. The parent node is placed in the middle of the space above the child nodes so that `D1=D2`.

The lines from the parent node `P` to the child nodes start at evenly spread points (the width `D3` is divided by 4).

The graph of relations in ObjectMath diagrams is not acyclic since multiple inheritance may occur and since parts inherit some classes. In the diagram (see Figure 1) a cycle consisting of six nodes (*Global Container*, `Body`, `Cylinder`, `Wheel`, `Metal`, `Material`) occurs. For layout computation we find an acyclic subgraph. For each class or instance the algorithm chooses the deepest superclass. Only this inheritance relation is chosen. The depth of the class is the depth of its deepest superclass plus one.

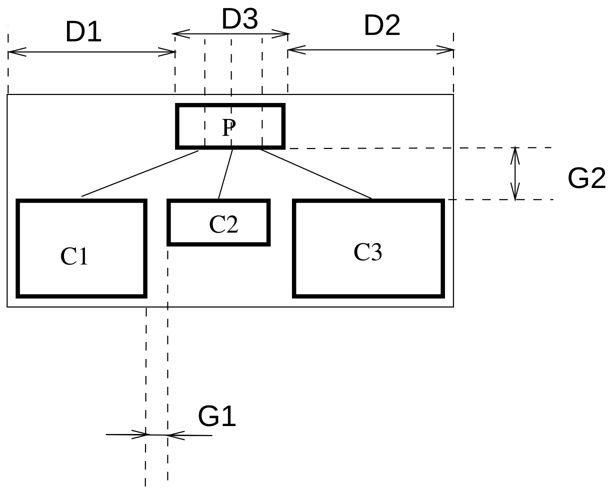

Figure 6: *Layout algorithm for directed acyclic graphs*

The depth of the *Global container* is zero. The acyclic subgraph contains all the nodes of the original diagram, but not all connections are included.

When the layout of nodes for an acyclic graph is computed, all the connections of the diagram are placed between the nodes. The inheritance almost always corresponds to connectors placed in top-down direction.

The default layout can be used as a convenient starting point for the user to modify. Any node can be shifted, together (or without) its descendants. The displacement between the default and modified node position is stored in a file (as comments) and this information is preserved from one editing session to the next.

7 Conclusions

Given a grammar for textual representation of class relationships and syntax rules we have developed a mapping from textual to graphical, representation. A more complicated task has been to design rules for diagram editing consistent with the syntactic rules. Three types of rules were identified, and they are explained in the report. A well designed mapping will have most rules belonging to the first group that does not require additional checks during the operation of the diagram editor. The rules of the second group may lead to difficulties for the user. The rules of the third group make difficulties both for the user and for the integrated environment (e.g. the diagram cannot be saved unless the errors are fixed).

Based on this mapping, a class diagram editor with facilities for multiple inheritance and composition relationship was developed. These two features differ this editor from a number of tree diagram editors which are used traditionally. A simple but efficient and user-friendly automatic diagram layout facility is implemented in the editor.

References

[1] Peter Fritzson, Vadim Engelson, Lars Viklund. Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment, In *Proceedings of the Conference on Design and Implementation of Symbolic Computation Systems (DISCO 93)*, vol. 722 of Lecture Notes in Computer Science, pp. 145–160. Springer-Verlag, 1993.

[2] *Mathematica*, Wolfram Research Inc., http://www.wolfram.com