A CONCEPT FOR THE REPRESENTATION OF DATA AND ALGORITHMS

DIRK MÖLLER AND RUDOLF MÜLLER

ABSTRACT. We describe a tool to generate C++-classes which serve as a
normed general framework for the representation of data and algorithms in
a software library. The main idea is to represent in a generated class besides
structural aspects of a mathematical object also conditions defining feasible
instances, and to obtain by that a one to one correspondence between the
mathematical objects and their implementation. Together with libraries of
abstract data types, the system provides a comfortable environment for the
development and maintenance of mathematical software.

1. Introduction

In this report we describe a tool to generate C++-classes which serve as a normed
general framework for the representation of data structures and algorithms in a software library. The main idea is to represent in a generated class besides the structural aspects of a mathematical object also conditions defining feasible instances, and to obtain by that a one to one correspondence between the mathematical objects and their implementation. Together with libraries of abstract data types, as e.g. LEDA [8], the system provides a comfortable environment for the development and maintenance of easy reusable mathematical software.

The design of the classes incorporates ideas from the data modeling language ASCEND [12, 6]. Originally, ASCEND was designed to represent processes in chemical engineering as a modularized, hierarchical system of data structures, called atoms and models. Each model contains declarative statements describing constraints on the parameters of the represented chemical process. Analogously, we represent mathematical objects and algorithms as a modularized hierarchical system of C++-classes. The declarative statements, describing the set of feasible values for the

1991 Mathematics Subject Classification. 68N05,68P05.
The 2nd author is supp. by the Deutsche Forschungsgemeinschaft under grant Ma 446/1-3
This paper is in final form and no version of it will be submitted for publication elsewhere.

© 1994 American Mathematical Society
1053-7587/94 $1.00 + .25 per page

391
parameters in an ASCEND model, are replaced by test functions which can check the feasibility of an instance of a class.

Section 2 introduces with an example the difference between pure and safe versions of data structures and routines. Section 3 gives the formal definition of safe data structures. In Section 4 the structures of our classes are illustrated. Section 5 describes features supporting a comfortable usage of our classes and technical details of the implementation. We close with some final remarks in Section 6.

We assume that the reader is familiar with basic notions of graph theory (cf. [2]), complexity theory (cf. [3]), and object oriented programming (cf. [5]), especially the programming language C++ (cf. [15]).

2. An example

Let us start with an example. Suppose we need as a subroutine in a scheduling algorithm, say, an algorithm to compute a maximal weighted matching in a bipartite graph and suppose we have a library of C-functions, say, available in which we find the function

\[ M \text{ WeightedBipartiteMatching}(\text{graph } g). \]

To use this function, we have to know

- how the parameters have to be initialized with data,
- which conditions have to be fulfilled by the data,
- how the result can be extracted from the return value and the parameters.

After we extracted all that information from the documentation, we have to implement transfer routines which map the data from our data format to that of the parameters, and vice versa. Now, the function cannot be expected to check necessary conditions on the input data, for example, the condition that the variable of type graph is initialized with a bipartite graph. In this case, we have to ensure that our data always meets this requirement, or we have to implement a test routine.

In the following let us distinguish between the p-version (pure) and the s-version (safe) of a routine. The p-version does not check any precondition on the input. It works directly on a data structure guaranteeing a highly efficient implementation. The s-version is a p-version encapsulated in a box. The box contains everything to avoid that the p-version inside crashes, and, maybe, a lot of additional luxury as, for example, a user interface or an editor for the data.

The concept of p- and s-versions extends to data structures as well. For example, suppose we use for the representation of a weighted bipartite graph a graph data structure with an integer valued node attribute for the color class and a real valued edge attribute for the weight. This is a p-version, because it does not guarantee that a variable of this graph is initialized with correct data. The s-version has in addition methods to check, for example, that each node color is either 0 or 1, and that there are no edges between nodes of the same color.

The existing libraries of abstract data types strongly support the implementation of well understandable p-versions of data structures and algorithms. For example, with a tool like LEDA [8] it is no big issue to implement an efficient p-version of an algorithm for bipartite matching. Also, the p-version looks basically the same as its description in a textbook and needs, therefore, nearly no additional documentation.
We have been using LEDA over a year to develop a library of scheduling solvers. However, this development comes with additional requirements, which are not supported by LEDA. Since the scheduling library is going to provide data structures and algorithms for a large variety of scheduling problems (essentially everything described in [9, 10]), it has to contain a huge variety of data types and routines. For example, part of the library is a classification of machine scheduling problems (cf. [7]), which requires a combination of a variety of test functions, ranging from simple ones, like all activity durations are equal 1, to the classification of partially ordered sets. All data types have to come with additional services like I/O routines and editors. A lot of communication between different data types has to be organized, and everything has to be safe against unfeasible initializations.

Since similar problems occur in other software projects in our research group, for example, in the design of VLSI-algorithms, we decided to develop a tool to generate s-versions of data structures and algorithms in form of C++-classes. Before we describe the details of the implementation of our prototype, we give a precise notion of an s-version in the next section.

3. Safe data structures and algorithms

First we consider data structures. A data structure is designed to represent mathematical objects. The objects are the elements of a set \( M \). Variables of the data type can be initialized by instances. The set of all possible instances that a variable can attain is usually a superset of the feasible instances. An instance is feasible if the corresponding mathematical object is an element of \( M \). We say that a data structure is safe if there exists an efficient routine which checks whether an instance is feasible or not.

For example, a safe data structure for bipartite graphs is given by a data structure representing arbitrary undirected graphs without loops and multiple edges together with a routine testing whether a given initialization of this data structure is 2-colorable. Another possibility to make a data structure for bipartite graphs safe is to add node attributes containing a coloring and a test, which checks whether the coloring is a feasible 2-coloring.

The latter possibility uses a certificate to check feasibility. This is a well known technique in complexity theory: the class \( \mathcal{NP} \) consists of exactly those data structures (more precisely we should say formal languages) for which each feasible instance has a certificate whose size is polynomial in the coding length of the instance, and which can serve as the input for an algorithm which proves feasibility in polynomial time. So, in principle, we can represent each data structure in \( \mathcal{P} \) by a safe data structure and each data structure in \( \mathcal{NP} \) together with a certificate as a safe data structure. However, there are data structures in \( \mathcal{P} \) for which only the existence of a polynomial algorithm can be proven, and, on the other side, an exponential algorithm can be viewed as an efficient test, as long as we are only dealing with small instances of an \( \mathcal{NP} \)-complete structure. Therefore, we prefer the term efficient to polynomial in our context.

Turning to algorithms, we can use the definition of safe data structures to define safe algorithms. We require that each algorithm \( A \) transforms a value of an input data structure into a value of an output data structure, i.e., \( A \) realizes a relation in \( \text{input} \times \text{output} \). An algorithm is called safe, if this subset, viewed as a data
structure, is safe. In order to make a deterministic algorithm with a polynomial worst case runtime safe, we have only to ensure that it produces a feasible output for each feasible input. This is true, since the algorithm itself may be used as test routine.

As we said, we are heading for a tool to generate safe data structures in the form of C++-classes. From our definitions in this section we obtain a preliminary list of guidelines for the layout of such a tool:

- Each class should have a method to test feasibility (in our prototype it is called evaluate).
- Routines should be encapsulated in classes (they are called function classes).
- The tool should support the integration of predefined classes into new classes.

The last point is essential. To make everything safe, we need many test routines. These routines should be reusable in new classes. Also, they should be safe themselves, i.e., available as safe C++-classes. This requires support in integrating predefined classes into new classes. We will discuss that point again at the end of the next section, where we give some more details on the structure of data and function classes.

4. THE STRUCTURE OF THE CLASSES

First recall that our data classes are not designed to be the basis for p-versions of algorithms. So we need not provide all kinds of comfortable abstract data types, like, for example, stacks, queues or heaps. For our purposes it is sufficient to have the structures

- frame for the Cartesian product of sets,
- case for conditional structures,
- table for indexed families of elements,
- graph for binary relations.

Clearly, graphs could also be realized by tables. But, since most of our applications are related to graphs, it is convenient to support them directly.

To have these structures means that one can fill them with objects of classes that are already generated with our generation tool. Thus, it is possible to define frame, case, table classes that have already generated classes as components. The same holds for graph classes, for which the components are attributes of the nodes and of the edges. For table classes one can take integers, reals or strings as row indices.

For all classes the number of components and the types of the components are a part of the definition. This cannot be changed during run time.

Variables of frame and case classes are static in the sense that each component is initialized by a default (Internally, the default is a NULL pointer, but in the case of an access to the component an object is created and returned). Table and graph classes are dynamic in the sense that a table has initially no rows and a graph has initially no nodes and no edges. The number of rows, nodes and edges may become arbitrarily large.

An important point is that we allow recursion. Each dynamic class (tables and graphs) may contain components of its own type. For example, the type of a node
attribute of a graph may be the graph itself. Clearly, a recursion consisting only of static structures (frames and cases) is not possible.

Algorithms are represented by function classes. A function class has two data components, called input and output. It was shown in [14] (see also [11]) that this functional representation of algorithms may be a key issue in the design of an easy extendable software system. Examples for mathematical software systems with a functional design are Mathematica and its extension Combinatorica [16, 13].

Note that the discussion from section 3 already argued for a functional design. A more technical motivation for the splitting between input and output is external. The development of the scheduling algorithms is done in cooperation with the FAW in Ulm, Germany. The FAW is a research institute for applied artificial intelligence. They are going to integrate our library in a decision support system (DSS) supporting the solving of scheduling problems [1]. The kernel of the DSS is implemented in operational programming [4, 14], which requires a functional representation of algorithms.

From the standpoint of object oriented programming a functional design may look a little bit outside the main paradigm. There it is more usual to view an algorithm as a method to modify an instance of a data structure. However, note that this requires a tight binding of algorithms to data structures which may contradict an easy reusability.

Another argument against the splitting into input and output may be a loss in efficiency. For example, using a function class in a loop requires a generation of new input and output instances in each iteration. Now, this can be avoided in our prototype, because we allow that input and output share the same memory, i.e. that the input and output objects are links to the same representation (see section 5). Furthermore, in time critical applications, it is always possible to use the included p-version of a function instead of the s-version.

In order to make data and function classes safe, we give each class a method evaluate. This method tests the feasibility. The method evaluate should reuse predefined test functions in the same way as a mathematical definition uses previously defined notations. We have realized the evaluate in the following way. When defining a new class, the user includes in the description of this class some C++ source code. In this code all member functions of the new class can be used to check the feasibility of instances. Furthermore, all other function and data classes may be used. Our generation tool takes the edited code as source code for the member function evaluate.

5. ADDITIONAL LUXURY AND TECHNICAL DETAILS

We already mentioned that each class comes with a graphical user interface. With this interface the main methods of the class are invoked, which are

- I/O from/to a file system,
- a graphical editor,
- evaluate.

Every new class gets its own directory in the file system. With the I/O methods one can read from or write into this directory and delete stored instances in this directory.
Furthermore, every class automatically comes with a graphical editor. For the basic classes `Integer`, `Real` and `String`, this editor is simply a small window in which the value is written; for a graph, it is a graphical editor with pop-up menus for the edge and node attributes; for frames, it is a composed window with an editor for each component; etc.

For the integration into the DSS it was necessary to avoid pointer variables. We achieved that by realizing every variable in free storage, i.e., by using only pointer variables. But we hide them from the user. So if a user has defined a class A and declares a variable a of this class, a is a pointer to some space in the free memory, called the representation of A. In this way we have full control over the memory. This made it easy to implement a `link` operator `&=` for each of our classes. The statement

```
a &:= b
```

releases the memory to which a is referring (if no other variable refers to it) and makes a refer to the same memory as b. Beside the `link` operator, each class comes with an `assignment` operator. So the statement

```
a = b
```

releases also the memory to which a is referring, but then it generates a copy of b which has no memory with b in common and makes a referring to the copy.

The system requires a lot of partial data transfer between classes: integrated test functions have to be initialized by parts of the data class in which they are used, and the method `evaluate` of a function class combines several function classes using parts of the output of one class to initialize parts of the input of another class. This partial transfer is due to the fact that, in order to achieve a high reusability, every function is implemented for the most general case. For example, a test whether a graph is bipartite has simply a graph as input. If we need it for a weighted graph, we have to extract the graph structure from the weighted graph.

In object oriented programming, the concept of inheritance supports such a partial transfer of data. One can assign a variable of type `class A` with any variable of a class `B` derived from A. So, for example, one can derive a class `WeightedGraph` from the class `Graph`. A function class with input class `Graph` then accepts a variable of type `WeightedGraph` as input. However, when designing the classes for our scheduling library we found it nearly impossible to decide on an appropriate hierarchy of classes. The main problem is that new classes may be required that are between two classes in the existing hierarchy. Assume, a class `ABC` has been derived from classes A, B, and C. Now, if a new class AB is defined it is necessary to change the source code of class `ABC`. To avoid changing the source code after defining a new class, one can generate the full hierarchy over all base classes. However, this is impractical because of the very large number of classes that will be created.

Our conclusion regarding these problems was to leave an implementation of logical inheritance aside. In our prototype, each user defined class is an immediate subclass of a class `i.<class.type>`, for example each user defined frame is a subclass of `i.frame`. To support nevertheless an easy data transfer, we can assign, for example, variables of different frame types to each other. For example, suppose we are given two frame classes `AB` and `BC`. `AB` has components a of type A and b of type B, `BC` has components b of type B and c of type C. Let x resp. y be a
variable of type \( AB \) resp. \( BC \). Then
\[
x = y
\]
results in clearing \( x \), initializing the component \( b \) with a copy of \( b \) of \( y \) and initializing the component \( a \) with a default. Further, we also provide a method \texttt{update}, where
\[
x\texttt{.update}(y)
\]
leaves the component \( a \) of \( x \) unchanged, whereas the component \( b \) of \( x \) is assigned with \( b \) of \( y \).

The internal implementation that facilitates this behavior is as follows. The information of the components of a class — type and name — is stored in each instance explicitly in form of strings. Thus, the component information of a class is available in the assignment and \texttt{update} method, and the activated method looks for the components in both parameters and executes the appropriate assignments or updates. By this we simulate the idea of inheritance in a more general way.

From the mathematical point of view, one would desire that, whenever one realizes a subset \( N \) of a set \( M \), then the class for \( N \) should be derived from \( M \). This can hardly be realized in an hierarchical implementation, since instances of the class \( N \) may require a totally different representation than instances of \( M \). For example, interval \texttt{orders} are special acyclic graphs, but it is much more convenient to represent an interval \texttt{order} by a set of intervals, instead of representing it by a directed graph together with a test. This kind of logical inheritance has to be made explicit by so called bridge \texttt{algorithms} that transfer logically equivalent objects from one representation into another.

The generation of the source code for a new data class is done by functions of type
\[
\texttt{void generate.<class.type> (<class.type>.desc desc).}
\]
The parameter \texttt{desc} contains information about how the new class should look like. For example, if the new class is of type \texttt{table}, then the description contains the \texttt{index} type of the table, the names of the columns of the table and the type of each column. Further, it contains names of \texttt{.h} - files containing declarations of external routines, and the ascii-text of the evaluate method. The classes \texttt{<class.type>.desc} have also been generated with our tool. So they come, besides all other features discussed here, in particular with editing features. This makes it very comfortable to generate new classes.

The whole code has been developed with the Guu C++ compiler on Apollo and Sun workstations and is based on the LEDA library. It has also been successfully compiled with an AT&T C++ compiler. Graphics are implemented with X11/Release 4 and the MOTIF toolkit. The classes \texttt{<class.type>} already contain all the code needed for administrating a class. Only code which is specific to the types of the components of the new class, for example, the fields in a frame, is added. So, although the initial library is very large (appr. 2 MB), it grows only slowly with each new class added to it. A prototype of our system is already available. As part of the algorithm and data library ADLIPS (Algorithms and Data Structure Library for Project Scheduling), which was also developed at the Technical University in Berlin, it proved to be a valuable tool to generate and work with the several hundred classes needed in ADLIPS.
6. Conclusions

We have described a tool which helps to represent mathematical objects with
an efficient membership test one to one by C++ classes. Thus the mathematical
definition is completely computational active: the structural aspects are translated
into data structures, the constraints are represented by test functions.

This report describes work in progress, but our experiences with the prototype
are already very convincing. Within a relative short time we have been able to
build a comfortable environment to work with more than 200 algorithms and data
structures for scheduling. And everything is safe! There are still some drawbacks
(for example the relative complicated syntax which has to be used in the evaluate
scripts). But in total it seems to be rather easy to work with the system. The
next step will be to integrate software for VLSI design and partially ordered sets,
which was developed during the last years in our research group. Further, we plan
to integrate standard algorithms for combinatorial optimization, for example, those
coming with the LEDA library.

As we said in the introduction the design of our classes is similar to the hierarchy
of atoms and models in ASCEND. ASCEND comes with a lot more features to
support the modeling of chemical processes. We could not realize all of them in
our C++ classes, however these features could be rather helpful in representing
mathematical processes. Therefore we are currently investigating together with R.
Krishnan and P. Piela how ASCEND can be used directly for such a representation.

Abstract data types already support a good transparency of the structural aspects
of data structures and algorithms, but to reuse software and to build big libraries
one must also make the logical aspects transparent. We hope that our tool can be
a first contribution to that objective.

7. Acknowledgements

The development of the scheduling algorithms is done in a long term cooperation
with the FAW in Ulm. Especially, we want to thank D. Solte from the FAW for
introducing us, and sometimes also forcing us, into the strict software engineering
paradigm operational programming. Also, the work described here would not have
been possible without the financial support coming from the FAW.

We want to thank R. Krishnan and P. Piela from Carnegie Mellon University at
Pittsburgh for many fruitful discussions on high level representations of mathematical
algorithms and data structures.

Last but not least we thank D. Küh and J. Schulze who implemented the graphics
routines of the tool.

References

1. M. Bartusch, R. H. Möhring, and F. J. Radermacher, Design aspects of an advanced DSS for
5. S. N. Khoshafian and R. Abnous, Object-orientation - concepts, languages, databases, user


**Institut für theoretische Informatik, Humboldt-Universität Berlin, Unter den Linden 6, 10117 Berlin, Germany**

E-mail address: dmeeller@informatik.hu-berlin.de

**Fachbereich Mathematik, Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany**

E-mail address: mueller@math.tu-berlin.de