Automatic Generation of User Interfaces from Data Structure Specifications and Object-Oriented Application Models

Vadim Engelson[1], Dag Fritzson[2] and Peter Fritzson[1]

[1] Dept. of Computing and Information Science, Linköping University, S-58183, Linköping, Sweden, {vaden,petfr}@ida.liu.se
[2] SKF ERC B.V., Postbus 2350, 3430 DT Nieuwegein, The Netherlands, adsdtf@skferc.nl

Abstract. Applications in scientific computing operate with data of complex structure and graphical tools for data editing, browsing and visualization are necessary.
Most approaches to generating user interfaces provide some interactive layout facility together with a specialized language for describing user interaction. Realistic automated generation approaches are largely lacking, especially for applications in the area of scientific computing.
This paper presents two approaches to automatically generating user interfaces (that include forms, pull-down menus and pop-up windows) from specifications.
The first is a semi-automatic approach, that uses information from object-oriented mathematical models, together with a set of predefined elementary types and manually supplied layout and grouping information. This system is currently in industrial use. A disadvantage is that some manual changes need to be made after each update of the model.
Within the second approach we have designed a tool, PDGen (Persistence and Display Generator) that automatically creates a graphical user interface and persistence routines from the declarations of data structures used in the application (e.g., C++ class declarations). This largely eliminates the manual update problem. The attributes of the generated graphical user interface can be altered.
Now structuring and grouping information is automatically extracted from the object-oriented mathematical model and transferred to PDGen. This is one of very few existing practical systems for automatically generating user interfaces from type declarations and related object-oriented structure information.

1 Introduction

Almost all applications include some kind of user interface. Graphical user interfaces (GUI) provide the opportunity to control an application's execution, to modify the input data and to inspect the results of computations.

Application programs have different data structures. Each application domain puts special requirements on visual presentation of data. Therefore, graphical interfaces are traditionally designed individually for each application.

The following properties are expected from applications with graphical user interfaces:

- The data must be presented to the user in a well-structured way. The graphical user interface should be consistent with the computational part of the application (for example, elements of the graphical user interface for data input should correspond to components of the application data).
- The user interface should satisfy style guidelines, conventions and standards. The compromise between large amounts of information and limited screen space can be achieved if the graphical user interface allows the user to choose only interesting information and ignore all else.
- The user interface software should be portable and not be dependent on a specific operating system or compiler.
- Entered data should be persistent: it should be possible to store entered data outside the program memory and reload it again later.

For realistic applications the design and implementation of a graphical user interface often become rather laborious, expensive and error-prone. Currently available toolkits are very powerful. Unfortunately, they are also very complicated and not user-friendly enough. In order to obtain some result the programmer often has to take too many implementation details into account. The high cost of implementing user interfaces can be partly reduced by the use of *user interface generation tools.* Such tools usually include a WYSIWYG layout definition tool that helps the programmer to design the layout of windows, menus, buttons and other user interface items. The graphical user interface code is generated automatically.

However, every time the application code is updated or the layout is changed, the interface code between them has to be updated manually.

1.1 User interface generation based on data declarations

In this paper we propose a different approach, based on the automated generation of user interfaces from data structure information. As a preliminary we present some terminology.

Data structures in traditional languages (such as Pascal or C) are described by variable and type declarations. In object-oriented languages (e.g. C++) data structures are defined using classes, objects and relations (inheritance, part-of) between objects.

The *structure of a graphical user interface* can be described in terms of graphical elements such as windows, menus, dialog boxes, frames, text editing boxes, help texts etc., and layouts that define how these elements are placed on the screen.

In which *context* the interface is used ? The main purpose of a graphical user interface is to let the user inspect and modify some data. The input data can be edited by a stand-alone graphical tool, saved in file and then loaded by the computing application. The output data can be saved by the application and inspected by a separate tool. The application may suspend computations, initiate graphical interface in order to allow data editing, and then resume the computations again. *We consider graphical interfaces that can be used in all these cases.*

The data have some structure and it is used to control the application functionality. Therefore the structure of the graphical user interface should be similar to the structure of the application data.

Typically there is an implicit or explicit correspondence between the structure of a program and the structure of data. On the other hand, there is a correspondence between the structure of the interface and data structures of the program. This means that the way the programmer perceives the structure of the implementation is close enough to the way the end-user perceives the structure of the application area.

The basic idea of our approach is to *generate the graphical user interface automatically from the application data structures.*

The similarity between the data structures and the structure of the graphical user interface is characteristic for a wide spectrum of applications, including simulation tools and information systems.

Data persistence is a generic property that includes saving data structures on permanent storage such as a file system and being able to restore these data next time the application is executed. To implement persistence, we need routines that can save or load all the application data (or some part of the data). *Such persistence routines can be generated automatically from the application data structures.*

Automatic support for data persistence as well as generation of graphical user interfaces will allow designers to concentrate on the main goals of the applications rather than on mundane tasks such as implementing a graphical user interface and input/output.

We applied the method of generating user interfaces from data structure declarations to two object-oriented languages: ObjectMath (an object-oriented extension of Mathematica [Wolfram91]) and C++.

In Section 1.1 we have discussed some reasons and motivation for the design of a user interface generator according to these principles.

The rest of the paper is organized as follows:

First we describe relevant features of ObjectMath, an environment for scientific computing (Sect. 2.1) and consider a *semi-automatic* approach to the creation of user interfaces from application data structures, which also has been

tested in industrial applications. A new *automatic* graphical user interface generation is based on our PDGen tool (Sect. 3) that automatically generates persistence and graphical user interface code from given data-type declarations. This tool is applied to ObjectMath models.

In Sect. 4 we describe how the PDGen tool can be applied to ObjectMath models.

Sect. 5 discusses related work on persistence and display generation and we conclude with proposals for future work. More details can be found in [PDGen96, E96].

2 The Semi-automatic GUI Generating System

2.1 The ObjectMath Environment

Applications in scientific computing are often characterized by heavy numerical computations, as well as large amounts of numerical data for input and output.

The data often have a complicated structure including objects with fields of various types, vectors and multidimensional arrays. This structure often changes during the course of program design.

An important application area in scientific computing is the simulation of various mechanical, chemical and electrical systems. These applications can be described by mathematical models of the physical systems to be simulated. Additionally, routines for numerical solution systems of equations are needed, as well as routines for input/output and routines and tools for user interfaces.

The process of manually translating mathematical models to numerical simulation programs in C or Fortran is both time-consuming and error prone. Therefore, a high level programming environment for scientific computing, ObjectMath [Fritzson95, Viklund95, Fritzson93] , has been developed that supports the semi-automatic generation of application code from object-oriented mathematical models.

The ObjectMath programming environment has been applied to realistic problems in mechanical analysis. ObjectMath class libraries describing coordinate transformations and contact forces have been developed. They are used for mathematical modeling of rolling bearings by our industrial partner, SKF Engineering and Research Center.

In ObjectMath formulae and equations can be written in notation that is very similar to conventional mathematics. The ObjectMath language is an object-oriented extension of the Mathematica computer algebra language, in a similar way as C++ is an extension to C.

The ObjectMath language includes object-oriented structuring facilities such as *classes*, *instances*, single and multiple *inheritance* (for reuse), and the *part-of* relation (to compose new classes from existing ones).

2.2 The simulation environment for ObjectMath models

First, an *ObjectMath model* is specified with the help of a class relationship editor and class text editor. The *ObjectMath code generator* generates parallel or

sequential programs for systems of equations expressed in ObjectMath. Typically a system of ordinary differential equations is considered.

The generated code is linked with model-independent run-time libraries. The *executable code* requires a large number of input values (such as start values, limitations, model geometry and conditions, solver parameters) in order to start a simulation.

The *input data editor* is designed for input data inspection and update. It has a window-based graphical interface for ObjectMath variable editing and can load and save a file with variable values. In this paper we present two graphical user interface generation systems that can create an input data editor from model specifications. The first system is described here, the second in Sect. 4.

The simulation program reads the data prepared by the input data editor and computes a large amount of output data for every simulated time step.

This data can be explored with the help of an *output data browser* . This browser can create graphs that illustrate how the variables change during the simulation.

The *animation tool* reads the output data step by step and shows the model geometry in motion.

2.3 An ObjectMath example: a Bike model

In this section we present an ObjectMath model example, a mechanical model of bicycle (Fig. 1) in order to explain the relations between classes and instances in ObjectMath[3].

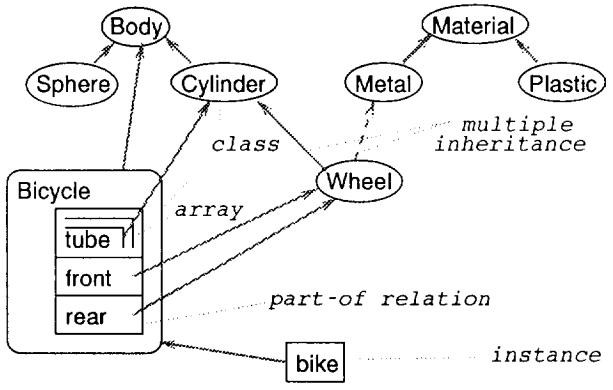

Fig. 1. *An ObjectMath class diagram for a bike model. Arrows denote single or multiple inheritance. The* **bike** *instance contains the parts* **tube** *(array of tubes),* **front** *and* **rear**, *which inherit from the classes* **Cylinder** *and* **Wheel**, *respectively. Such diagrams are editable with graphical class relationship editor.*

[3] We discuss the constructs relevant for graphical user interface generation only.

Every model specification consists of *classes* and *instances*. The textual part of classes and instances contain variable declarations, formulae and equations. This is the way the formulae and equations related to the same phenomenon are grouped. Classes and instances inherit variables, formulae and equations from one or several (multiple inheritance) classes. The classes serve as templates for instances.

The class `Bicycle` on Fig. 1 inherits all variables, equations and formulae from the class `Body`. The instance `bike` inherits everything from the class `Bicycle`. An instance or a class can also contain its *own* variable declarations and formulae. Every instance is created *statically* and its variables (both its own and inherited ones) can be referenced in formulae and equations of other classes and instances.

A `Bicycle` consists of three *parts* in this model: the `front` wheel, the `rear` wheel and the `frame` consisting of several `tubes`. The number of `tubes` is equal to the value of some variable, in our example it is `framesize` (this is not shown in the picture).

2.4 Variables and built-in data types

Let us assume that there are several ObjectMath variables[4] declared in the class definitions:

```
In Body:     Declare [angle, "doubleVec3", "rad", uII]
In Cylinder: Declare [radius, "double", "m", uII]
In Wheel:    Declare [pressure, "double", "H/m^2",uII]
In Bicycle:  Declare [framesize," int", "-", uII]
```

In the general form variable name, type, physical unit and persistence status are specified:

`Declare[`*name*`,` *type-name*`,` *"unit"*`,(uII|u00|uL)]`.

Types. In ObjectMath there is a fixed set of twenty primitive data types that can be used for variables in the model. Some of the types have complex structure and may contain up to 100 double precision real numbers, integers and strings.

A variable of type `double` has a double precision floating value. The type `doubleVec3` is a 3-element vector of `double`.

Units. A string such as `"H/m^2"` contains the name of the physical unit of the value this variable represents. This unit name is used as part of the prompting information for the relevant input field in the input data editor that is generated from the declarations above.

[4] This declaration syntax is for ObjectMath version 3.0. The latest version 4.0, fall 1995, has a different declaration syntax.

Persistence status. The variable declarations provide the persistence information: whether a variable should be initialized by the input data editor, should be output and stored as a computed result, or is simply a local variable for intermediate results.

Here `uII` means "to input from the input file", `uOO` means "to output to the output file", and `uL` means "local variable, neither input, nor output".

From this information a window with text input boxes (Fig. 2) is generated. The variable instance identifier, text input area for value editing and "units" are shown for each variable component.

2.5 Generation of input data editor

The *basic idea* of this approach is that part of the code necessary for graphical user interface creation is *automatically* generated from ObjectMath variable declarations. Then the *layout* information is manually inserted into this code.

In order to create the input data editor we have to create the hierarchy of windows and variables; this hierarchy is created half-automatically and combines display routines provided that create widgets for every ObjectMath variable type.

When model specification code is analyzed, the names of variables such as `angle`, `radius` and `pressure`, are converted to unique names (within the model) by adding prefixes (part and object names). This is the list of *all* the variables available in this model, where notation *name*`[`*n*`]` denotes an array with *n* elements:

```
bike'front'radius      bike'front'angle     bike'front'pressure
bike'rear'radius       bike'rear'angle      bike'rear'pressure
bike'tube'radius[bike'framesize]
bike'tube'angle[bike'framesize]             bike'framesize
```

A specially designed filter reads the model specification and generates a comma-separated list of function calls:

```
var_double_array("bike'tube'radius","m"),
var_doubleVec3_array("bike'tube'angle","rad"),
var_double("bike'front'radius","m"),
... ...
```

The calls of `var_...()` functions above register the variables as members of the list of relevant variables and return a frame handle which is used for constructing corresponding windows. The rest of the code needed in order to display these variables in a separate window (see Fig. 2) is inserted *manually* (manual part is shown in *italic* font):

```
make_dialog(
    layout_vertical(
      layout_horizontal(
        layout_vertical(
          var_double("bike'front'radius","m"),
          var_doubleVec3("bike'front'angle","rad"),
          var_double("bike'front'pressure","H/m^2")),
        layout_vertical(
          var_double("bike'rear'radius","m"),
          var_doubleVec3("bike'rear'angle","rad"),
          var_double("bike'rear'pressure","H/m^2"))
        )
      layout_frame(
        layout_vertical(
          var_int("framesize","-"),
          array("framesize",
            layout_vertical(
              var_doubleVec3_array("bike'tube'angle","rad"),
              var_double_array("bike'tube'radius","m")
))))));
```

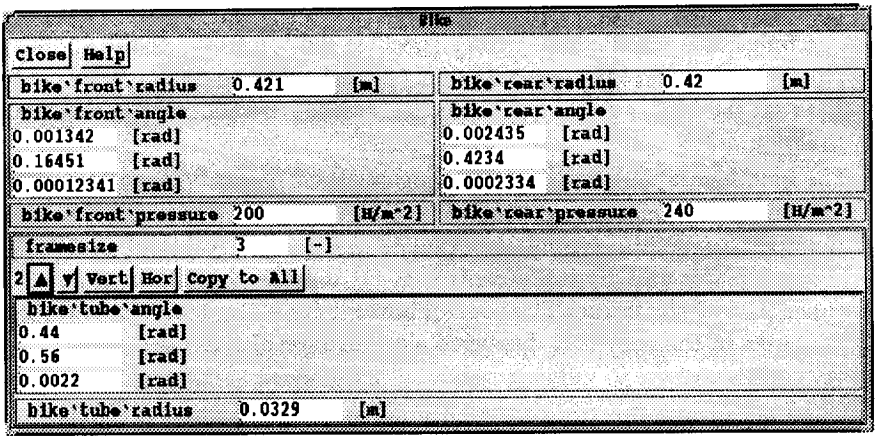

Fig. 2. *Example of variable display in the input data editor.*

2.6 Presentation of arrays.

Two variables (`bike'tube'angle` and `bike'tube'radius`) represented in the example (see Fig. 2) are arrays of `doubleVec3` and `double`. They have the same length; therefore they may be grouped together. These arrays share common

control buttons (`Vert`, `Hor`, `Copy to all` etc.) in the upper part of their frame. (It is also possible to build displays where every array has a separate control panel.)

As shown in Fig. 2 only one element (currently the 2nd element) of each array is visible, as indicated by the label "2" in the upper left corner. This is the compact presentation of the array. The buttons with the triangles ("Up" and "Down") switch the current element to the previous or the next, respectively. Then the label may change to "3" ("Up") or "1" ("Down"). The button "`Copy to all`" copies all the values from the visible element of the arrays to all other elements.

The buttons "`Vert`" and "`Hor`" change the presentation of the array: they spread its elements vertically or horizontally, respectively. Then the button "`Collapse`" appears that changes the presentation back to compact form.

2.7 Frame hierarchy definition functions

Every application window contains a hierarchy of *frames*. Each frame is a rectangular area that contains graphical user interface elements (widgets) such as labels, text input boxes, buttons, as well as compositions of other frames in the vertical or horizontal direction. A number of functions are needed in order to specify this hierarchy of frames:

- The function `make_dialog`(*frame*) specifies the top frame of the window.
- The function *frame*=`layout_vertical`($frame_1, frame_2, \ldots, frame_n$) specifies that the frames $frame_1, frame_2, ..., frame_n$ are allocated in the vertical direction.
- The function `layout_horizontal` allocates them in the horizontal direction.
- The function *frame*=`array`("*bound-variable*", $frame_1$) specifies that length of all arrays within $frame_1$ is equal to the current value of the variable *bound-variable* and they are controlled all together by the buttons in the upper part of the frame. The control buttons for array variables can change the index of the currently displayed element (see Figure 2).

There are several functions for additional help texts and decorations.

- The function *frame*=`layout_frame`($frame_1$) draws a rectangle around $frame_1$;
- The function *frame*=`layout_label`("*text*") specifies a label containing the *text* string.

The description this hierarchy is stored as a tree. When necessary, the tree is traversed, relevant Motif API functions are invoked, and the windows are displayed on the screen.

2.8 Description of variables

For every displayed variable a function for a corresponding data type should be called. These calls are automatically generated from the list of model variables. There is a separate function for each data type used in the ObjectMath language: `var_double`, `var_int`, `var_doubleVec3` and all others (totally, twenty) ObjectMath basic data types.

For example, *frame*=`var_double("bike'front'radius","m")` specifies that a text input box is constructed for the variable `bike'front'radius` of type `double` and that the physical unit is `"m"`.

Every such call registers a variable and arranges for the value of this variable to be displayed at an appropriate place in the layout. For every type a certain specific layout has been designed and hard-coded. For example, for the type `doubleVec3` (a structure with three double values) the layout is three vertically aligned text input boxes. Arbitrary `double` values can be entered here. For `integer`, `double` and `string` variables the layout is a single input box with the variable name to the left and the unit to the right (see Fig. 2). Arbitrary `integer`, `double` and `string` expressions can be entered into the input boxes respectively. .

Persistence. When the button `Save` is pressed, the persistence function is called and all the registered variables are written to the input data file. When the button `Load` is pressed all the variables in the list receive their values from the input data file. Both the input data editor and the application program should register the variables with the same name and type.

2.9 Evaluation of the first generation system

If the ObjectMath model changes, the graphical user interface programmer has two ways to solve the update problem. If many changes are introduced, the graphical user interface code should be generated again and the programmer has to insert the layout functions manually. If the changes are small and local (such as renaming some variables), the variable registration function calls (`var_...(...)`) should be manually updated.

The first generation system described so far in this paper has several *disadvantages*:

The update problem. If the model is changed, the new code that is automatically generated from the variable list must be manually merged with the layout description. Every small change in the list of variables from the ObjectMath model may lead to inconsistency between the generated application and the input data editor. Therefore the inherent flexibility of the ObjectMath environment cannot be used to full advantage.

Insufficiency of the basic type set. Only a limited number of basic data types are supported. These data types are either primitive ones or are specially designed for a particular application domain. There is no way to specify other types than these and there are no new type declaration constructs. The persistence routines and the layout routines are designed for the fixed set of types only.

Variable grouping. There is no automatic graphical user interface generation for distributing variables between different windows. Moreover, there is no automatic generation of the menu structure. However the structure of the model (i.e. the names of classes, objects and parts) can be used for this purpose.

Practical application of the system has also shown its *positive* features.

The first generation system has proved quite effective in producing practical user interfaces for specialized application domains such as bearing simulation. Recently SKF ERC researchers used the system to produce user interfaces for 6 new variants of similar bearing models using only 3-4 days of work. The difference between the variable sets in these models were rather limited and all the adjustments of the graphical interface for the input data editor were done manually.

3 The Persistence and Display Generating tool (PDGen)

The basic idea of PDGen is that display layout for every data item exactly corresponds (by default) to the type structure of this item.

Through the *display* for a given variable the user can inspect and update all the data items that can be reached from the variable by recursively traversing its structure. In the same way, the *persistence* routines save and load all the data items that can be reached from a given variable by recursively traversing its structure.

Traversing all of a complex data structure is a non-trivial task if we want to provide this automatically. Special complications arise in languages with pointers and dynamic data structures. Code necessary for this purpose can be automatically generated from data type declarations of the variables we are going to traverse.

We primarily orient PDGen to handling C++ data types. This tool can analyze almost any C++ data type and class declarations and add graphical user interface and persistence routines to an *arbitrary* C++ program. The manual efforts necessary for this are minimal.

The creation of the PDGen tool has been inspired by the PGen (see Section 5.1) approach from which we cite Walter Tichy et al.:

> *The class and type declarations can be used to generate browsers and editors. For instance, a class variable can be presented as a dialog box that contains sub-windows for all members to be inspected or edited. Pointers could be drawn as arrows to other variables. [...] The browsers and*

editors could be used to inspect or modify persistent data on files. More importantly, they could become the default graphical interfaces for all applications. The difference with other interface construction tools is that they require absolutely no programming. Debuggers are another application area. [Tichy94]

In Sect. 3.1 a graphical user interface generation example is given and in Sect. 3.2 display generation for every C++ data type is presented . In Sect. 3.4 we discuss window display issues; the generation process is analyzed step by step in Sect. 3.5 and the use of the generated code is discussed in Sects. 3.6 and 3.7.

The tool is based on the C++ language (Section 5.3) and the Tcl/Tk toolkit [Ou94].

3.1 Example of graphical user interface generation

Let us consider some type declarations that can appear in a header file (see Fig. 3(a)) of some application.

The PDGen tool analyzes these data type declarations, recognizes the C++ class hierarchy, and generates necessary code for creation of a graphical interface. If the application calls the function `show(bike)` (see Fig. 3(b)) then the dialog window shown in Fig. 3(c) appears on the screen. The specification of physical units (`rad`, `H/m^2`, `m`) is performed with the help of an attribute specification script (see Section 3.10).

All the data items that belong to `bike` are shown and they are available for editing. This example illustrates the display for *classes* and *arrays.* and elementary data items of types `int` and `double`.

The array `tube` is shown at the bottom part of the window. The user can press the buttons `"+"` and `"-"` in order to change the index of the currently displayed element of the array `bike.tube`. In the window shown in the picture the index is equal to 2, i.e. `bike.tube[2]` is displayed.

3.2 Graphical presentation of variables

Every window may contain one or several variables. The graphical presentation of every variable depends on its type and it is combined from graphical presentations of its components.

Types `char`, `char*` *and* `char[`n`]`. The types `char*` and `char[`n`]` are typically used for 0-terminated strings. A text input box is constructed for such variables and the string can be edited. Scrolling of the text is always provided so that character strings longer than the text box can be inspected and edited.

The display for variables of type `char` is similar to `char[1]`.

Types `integer`, `float` *and* `double`. Variables of these types are displayed as text editing boxes (see Fig. 3(c)). Only numbers or expressions (consisting of numbers and arithmetical operations) can be entered. The range of permitted values can be specified (see Section 3.10).

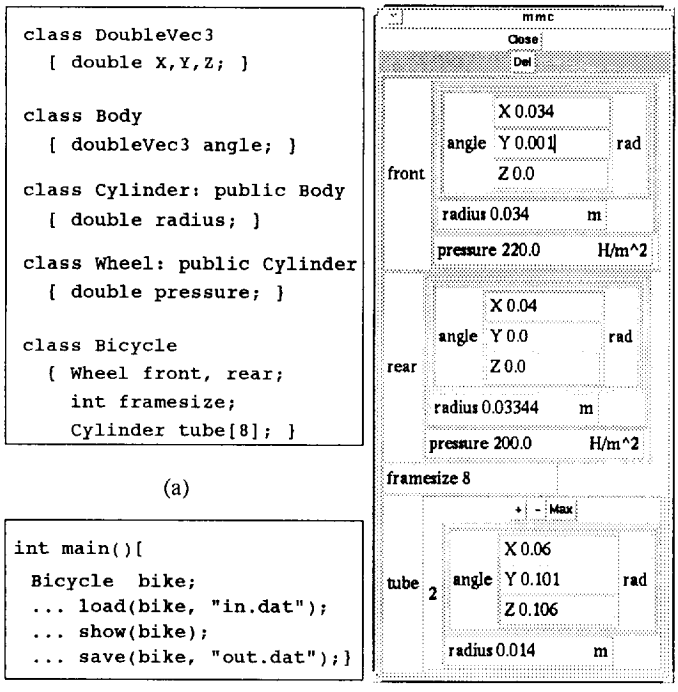

Fig. 3. *(a) Data type declarations. (b) Function call. (c) The window for editing the variable* `bike`.

Structures and classes. They are represented as horizontal or vertical[5] combinations of the components. The names of the data members of a structure or class are used as labels that appear to the left of corresponding components (see Fig. 3(c)).

Pointers. In our initial approach a pointer variable is represented by the referenced variable if the address is not `NULL`. There is a button `Delete` that deallocates the memory and sets the pointer to `NULL`. If the address is `NULL`, then there is a `New` button that creates a new variable in the dynamic memory, initializes it if it is an object and sets the correct value for the pointer variable.

Let us specify the class `Tree`:

```
class Tree
 {  Tree * right;
```

[5] In order to choose between horizontal and vertical combinations we use some heuristics. For example, we choose one that makes the resulting frame more similar to a square i.e. the ratio between the height and the width of the frame is closer to 1. With the help of customization options this default layout can be altered.

```
    int elem;
    Tree * left;
};
```

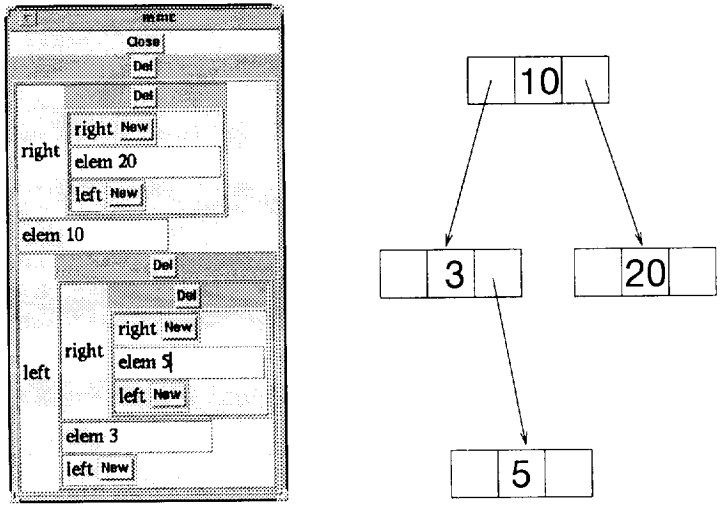

Fig. 4. *The window for editing the pointer structure of type* `Tree*` *and memory diagram of this structure.*

A variable of class `Tree` is displayed as a structure with three components (`right`, `elem` and `left`). We can also display a variable that contains a pointer to `Tree`. A variable of type `Tree*` is visualized as shown in Fig. 4.

This is a simple and quite straightforward approach if we are not concerned about the cases when two or more pointers refer to the same address.

In the *alternative* representation every dynamically allocated object (that has two or more references) is shown as a separate sub-window and arrows are drawn from the pointers to these objects in order to indicate the references.

Enumeration. The enumerations are represented as a group of radio buttons (or, as an alternative, as a pop-up menu). Enumerator names are written beside the buttons and only one of them may be selected at a time.

An object of class `Foo` is shown in Fig. 5(a).

```
enum weekday
  {  Mon, Tue, Wed, Thu, Fri, Sat, Sun  } ;
enum colors
  {  Red, Orange, Yellow, Green, Blue, DarkBlue, Violet  };
class Foo
   { weekday Days;
     colors  Colors;};
```

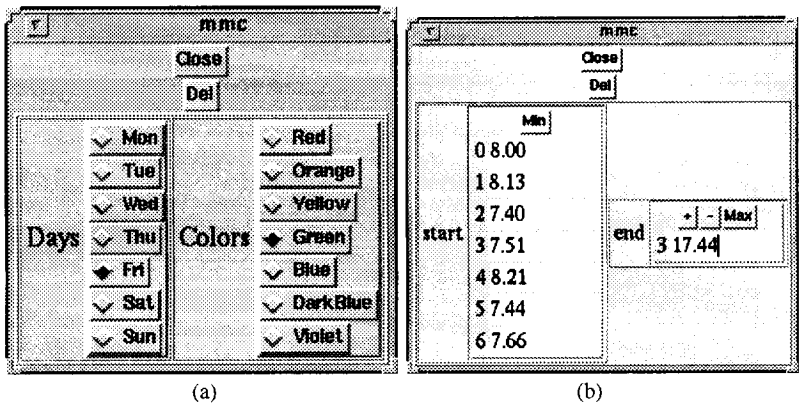

Fig. 5. *The windows for editing variable with (a) enumerators and (b) arrays.*

One-dimensional array. Fig. 5(b) shows how an object of class `Foo` is visualized.

```
class Foo
{ double start [7];
  double end[7];
};
```

The elements of `Foo::start` are shown in the *complete presentation*, i.e. all of them are available for browsing. In the *compact presentation* of the array `Foo::end` only one element (currently it is the element `end[3]`) is shown at a time. By using the buttons `+` and `-` we can increase or decrease the index of the currently visible element. The button `Max` switches the display to the *complete presentation*; the `Min` button changes the display back to the compact presentation.

An array of dimension larger than one is represented as a combination of one-dimensional arrays. This is not very convenient for browsing. A special browser [FWHSS96] is designed for two-dimensional arrays. We are working on an universal array browser for an unlimited number of dimensions.

A special interface is provided for dynamically allocated arrays. These can grow and shrink dynamically. For this purpose the buttons `Insert` (insert new element after the current one) and `Remove` (remove the current element) are added above the presentation of the array values. This option has some limitations and requires some additions in the description of data structures: the dynamic array (A) must belong to some class and an additional integer variable (A`_length`) should store its length:

```
class Test
 { Element * array_foo;
   int     array_foo_length; }
```

3.3 PDGen restrictions

There are some restrictions in the PDGen system that are partly caused by the restrictions of the PGen tool.

- References, constants, bit fields, unions, pointers to functions, pointers to members, `void*` pointers are skipped and ignored, because they cannot be persistent or are compiler-dependent, or there is no sense in keeping them persistent.
- Virtual base classes are supported for persistence only.
- Pointers to memory inside an object are supported for persistence only.

3.4 Hiding and detaching windows

The number of data items that can be displayed on the screen simultaneously is limited. Normally we cannot show a hierarchical layout of more than approximately one hundred text editing fields. We propose a window handling scheme where every displayed data item can be in three states (see Fig. 6):

- **hidden**: only a button with the data item identifier is shown in its default place;
- **normal**: the data item is displayed as usual in its default place;
- **detached**: the button with data item identifier is shown in its default place; the item is shown in a separate (top-level) window.

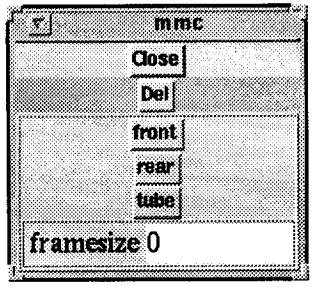

Fig. 6. *The variables* `front`, `rear` *and* `tube` *are hidden in the window for editing the variable* `bike`.

Switching between normal, hidden and detached state is performed by the mouse buttons. In each case the user has to click on the name of the item.

The *default* status is "hidden" for all non-elementary data elements and "normal" for elementary ones. The user can specify the default status with the help of attributes discussed in Section 3.10.

The buttons have the same function as pull-down menu items. The end-user has complete control over the information layout on the screen and there is no problem with the windows occupying all the display space. This way the

user can hide unnecessary information and select interesting data for display in separate windows. Since the buttons have almost the same behaviour as pull-down menu, this approach is rather close to graphical user interface standards and conventions.

3.5 Data type analysis and code generation.

This section discusses the generation process in detail, phase by phase.

The stages of graphical user interface generation are shown in Fig 7.

The PDGen tool reuses some ideas and essential code fragments from the PGen tool [Tichy94, PGen94, Paulisch90].

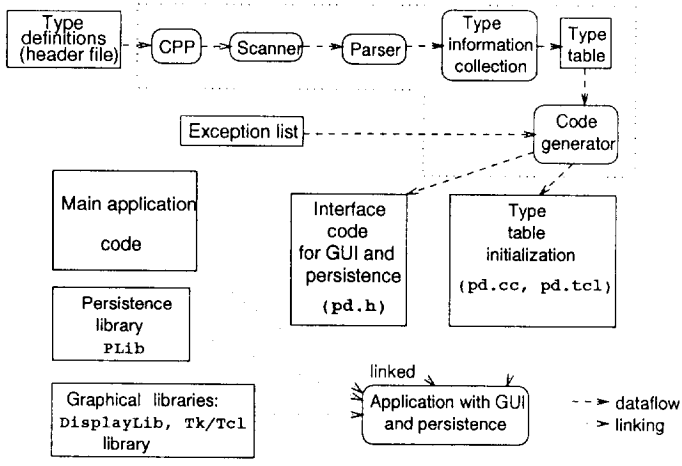

Fig. 7. *Phases of graphical user interface generation from C++ code and generation results.*

The basic source of the graphical user interface generation is a file with data type declarations. In C++ applications it involves one (or several) *header* files.

Parsing. The file is preprocessed by standard C preprocessor cpp and analyzed by the C++ scanner and parser.

Together with the PGen analyzer (see Section 5.1) we reuse the C++ grammar parser with semantic actions for syntax tree construction.

The syntax tree contains nodes of different kinds and references between them. The collection phase traverses all the nodes describing `typedef`, `enum`, `struct` and `class` declarations and produces the data-type table.

The syntax tree contains all the syntactical elements that appear in the input file. For our purposes we use `typedef`, `enum`, `struct` and `class` declarations only. In the `class` declarations we are interested in data members and constructors only.

Data type table. This table contains the names of all types, the names and types of class data members, inheritance information, the element type and the length of arrays, as well as information on elementary data types.

The analyzer assigns a unique type number (used for references); the numbers are assigned in increasing order: the first several numbers are reserved for fundamental types such as `int` and `double`.

The exception list. This list is optional and helps to prevent inclusion of unnecessary classes and data members to the type table.

Generation. The generation phase creates **type information** and **overloaded access routines.** All the generated code belongs to the class `PD` (**P**ersistence and **D**isplay) defined the header file `pd.h` , with member functions defined in the file `pd.cc`

Generation of data type information. The `PDGen` code generator writes to `pd.cc` a code fragment that initializes the data type table.

For the example given above (Fig. 3(a)) a fragment of relevant code is:

```
initSimpleType(3,"int",sizeof(int));
....
initClassType(33,"Wheel",sizeof(Wheel),0);
initBaseClass(33,32,0, "Cylinder");
initMember(33,12,offsetof(Wheel,pressure),   "pressure");

initClassType(34,"Bicycle",sizeof(Bicycle),0);
initMember(34,33,offsetof(Bicycle,front),    "front");
initMember(34,33,offsetof(Bicycle,rear),     "rear");
initMember(34, 3,offsetof(Bicycle,framesize),"framesize");
initMember(34,35,offsetof(Bicycle,tube),     "tube");

initArrayType(35,"Cylinder[8]",sizeof(Cylinder[8]),0,8);
```

The standard C macro `offsetof(Bicycle,rear)` calculates the offset (position) of data member `rear` within objects of class `Bicycle`[6].

This code fragment is later compiled and linked with the application. When the application starts, the data type table is initialized. This table is available to the persistence and display routines at the run time. When the data are saved, loaded, or shown, appropriate routines use this table in order to recursively traverse the data structures.

Generation of overloaded access routines. For every data type or class T that is defined in the header file the appropriate instances of the overloaded functions `PD::show(`T`&p)`, `PD::load(`T`&p)`, `PD::store(`T`&p)` are constructed.

These functions can take the variable of type T as an argument.

[6] This approach is more portable and safe than to sum up the sizes of every data member.

3.6 Input and output procedures

When the functions `load` and `store` are called , a variable of the corresponding data type is passed as a parameter.

The function `load` reads the variable value from the file and restores it in the memory. The function `store` saves the value on disk.

These functions traverse the application data that can be reached from the passed argument variable by recursively following data members, including pointer references. Every step of this process is controlled by the data type table. When data items of an elementary type are reached, the functions load/store the data in some format (textual or binary); see Section 3.8 where formats are discussed.

When the data is *loaded*, memory is dynamically allocated if a pointer variable is visited and its value is not *NULL*. When memory is allocated for class instances, the class constructor is called without parameters. It is assumed that every class has a constructor without parameters.

3.7 Data display procedure

The function `show` activated from the application program displays the required variable.

For data display we have designed a universal data browser which can show and edit the data when the data-type table is given. We use the `Tcl/Tk` graphical library [Ou94]. First, the C++ variables are associated with Tcl variables. It produces the following effect: if a Tcl variable changes, the C++ data change automatically. If a Tcl variable value is requested, then it is taken from the C++ variable.

We recursively traverse all data members, including pointer references. The algorithm that builds the window as a hierarchy of frames recursively traverses Tcl variables with the help of the data type table (Tcl script `pd.tcl`) which is generated automatically.

When some value is *updated* by the user, the corresponding Tcl and C++ variables are automatically updated; when it is updated by the application, its text presentation is changed, too.

If necessary (i.e. when the `New` button is pressed in the display of a pointer variable), memory is allocated and a class instance is initialized by the constructor.

3.8 Data storage formats

Complex data items are traversed recursively when loaded or stored. The original PLib library includes `load` and `store` routines for two machine-independent formats, ASCII and XDR [SUN90]. XDR is a data representation format used in remote procedure call. These formats are not self-describing formats, i.e., there is no possibility of discovering mismatches between the loaded data and the program data structures.

In the PDGen tool we extend this format by simply adding the type table information. When the `store` procedure writes some data, it also writes the type table. When the `load` procedure reads data from file, it also reads the type table and verifies that it is identical to the original one (for all data types that actually appear in the loaded file).

Difficulties can arise if old data are `loaded` by a program with new data structure. In our approach some basic data scheme correction is provided. If the old data contain classes with permuted order of data members, the correction works automatically. Extra members in the old data are ignored, the missing ones are not initialized. Finally, the user can explicitly specify the old and the new name for *renamed* class data members and renamed class names in the exception list.

3.9 Universal browser design

Since the data table is stored together with data (i.e. in a self-describing data format), a stand-alone universal browser can easily be designed. This is one of directions of our future work.

This browser automatically adapts the interface to the structure of loaded data. It works independently on underlying C++ data type declaration files and can browse and edit a file with arbitrary data structures if it is prepared by the PDGen persistence routines or by the universal browser.

It should be noted that there will be limitations for dynamic memory allocation during editing, because the C++ code (where necessary constructors without parameters are defined) is not available to the universal browser.

A semi-universal browser may contain some application-specific C++ classes and adapt itself to data structures constructed from these classes and elementary types.

3.10 Attributes

Attributes are used for additional control over class instances, type components and single data items in such cases when we want to alter the default behaviour of the PDGen tool when traversing the data elements.

The attribute information is *orthogonal* to the type structure declaration. Therefore it should normally be described outside the code containing the data types.

The graphical user interface designer (or generator) writes the attributes in a separate script file (the attribute definition file) which can be unspecified until the application program starts. It allows altering many preference settings and options without recompilation and even during the runtime.

Each attribute specification has syntax:

`set_attr` { *component1, component2, ...* } { *attribute1, attribute2, ...* }

Component is specified as *path* or *Class-name*::*path* where the *path* has the same syntax as C++ qualified names. This means that the data members are selected with dot (.), and array elements are specified in square brackets [*index*].

The use of patterns and regular expressions (within pair of "-s) is allowed instead of standard C++ path syntax. In this case the attribute specification applies to all paths that match the pattern.

The attributes are specified as *attribute=value.*

Example: The attribute specification script

```
set_attr { Bicycle::front.pressure  Bicycle::rear.pressure }
               { postfix = "H/m^2" }
```

states that for these variables the postfix (area normally used for physical units) should have the value `H/m^2`. The script is checked for correctness of the syntax; e.g. the system verifies that **pressure** is defined as a member of the class `Wheel`.

The same effect can be achieved by specifying a pattern:

```
set_attrp { "*pressure" } { postfix="H/m^2" }
```

The complete attribute specification necessary for the window in the Fig. 3(c) is:

```
set_attrp { "*angle" }  { postfix = "rad" }
set_attrp { "*radius*" } { postfix = "m" }
set_attrp { "*pressure*" } { postfix = "H/m^2" }
```

We just mention several other available attributes:

- `validate` specifies a Tcl function that will be called each time when the input text area is altered.
- `hidden` specifies how and whether the item value is shown at the beginning. It can be shown, hidden or detached (Sect. 3.4).
- `load` and `save` specify whether the value is loaded from disk file and saved. By default it is both loaded and saved. `required` specifies that the user *must* enter some value; `read-only` specifies that the user cannot update it.
- `layout` specifies whether the array or structure should be displayed in vertical or horizontal layout. By default a heuristic is applied.
- Finally, `hook` gives the designer "free hands"; it specifies a Tk/Tcl function that is responsible for complete graphical representation of the value. A Tcl variable name with the value and Tk window name is given. The function is written by the designer and it has to create a window with the given name.

4 Automatic Generation of GUI from ObjectMath Models

The basic idea behind the second generation system is to generate all components of the graphical user interface from the application model, and to avoid manual editing when the model is updated and the user interface code is re-generated.

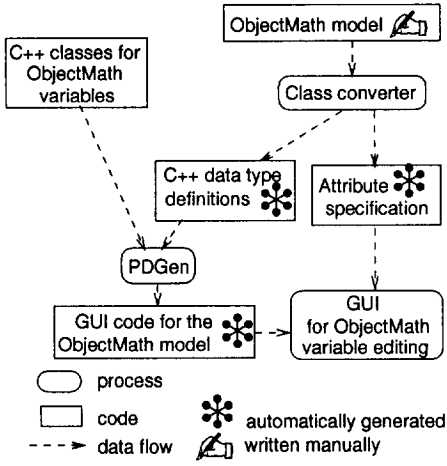

Fig. 8. *Second generation graphical user interface generating system.*

The phases of the generation process are depicted in Fig. 8. First the ObjectMath model is analyzed by the *class converter*. All the data necessary for the class converter are contained in the class hierarchy diagram and the ObjectMath variable declarations. The class converter translates the ObjectMath class hierarchy to the relevant C++ class hierarchy.

The ObjectMath variables can be of twenty different predefined types which are implemented as C++ classes. For example, the ObjectMath type `DoubleVec3` (contains three double precision real numbers and can serve as operand in various ObjectMath arithmetic expressions) corresponds (in the simulation code) to the C++ class `DoubleVec3`. The data members of these C++ classes can represent the corresponding ObjectMath variables in the graphical interface. For example, the class `DoubleVec3` is defined as

```
class DoubleVec3
 { double X,Y,Z; // data members
   ....          // member functions, friend functions etc.
 }
```

and the `DoubleVec3` type in ObjectMath can be presented in graphical user interface as three text input boxes marked with `X`, `Y` and `Z`.

The C++ class hierarchy (generated from the model) and C++ classes (that correspond to ObjectMath variable data types) are merged together and passed as the input for the PDGen tool. This tool generates code for graphical user interface for the corresponding ObjectMath model.

Attribute specifications are necessary for the application with graphical interface at run-time. They contain some information absent in the C++ class declarations, such as physical units and persistence status. These attribute specifications are generated separately for the members of every class by the class converter.

4.1 Translation of an ObjectMath model to a C++ class hierarchy

Every single ObjectMath class gives rise to a C++ class. The ObjectMath inheritance means that all variables are inherited by the subclass. The same happens in the C++ class inheritance.

ObjectMath instances correspond to C++ classes, too. Such ObjectMath instances inherit all variables and formulae from the superclasses, and, in addition, they may declare their own variables. ObjectMath *parts* serve for aggregation of class instantiations. They cannot specify their own variables. The parts can be modeled by C++ class data members.

ObjectMath variables become C++ data members.

The ObjectMath model (as a whole) corresponds to a single C++ class that includes one data member for every instance in the model.

4.2 Translation example

For the purpose of illustration we take our basic example (Fig. 1), a bicycle model. When the conversion described above is applied, the C++ type declarations shown in Fig. 3(a) are generated.

The attribute information is generated from the parameters in `Declare` statements. The attribute information is created according to the syntax rules described in the Section 3.10:

```
set_attr {Body::angle       } {postfix="rad"}
set_attr {Cylinder::radius  } {postfix="m"}
set_attr {Wheel::pressure   } {postfix="H/m^2"}
set_attr {Bicycle::framesize} {postfix="int"}
```

The graphical interface generated for this example is identical to the Fig. 3(c).

4.3 Advantages of the second generation approach.

The new approach successfully solves the problems arising in the first generation approach (Section 2.9). There are no *update problems* because the application and its graphical user interface are generated simultaneously. The set of supported types can include arbitrarily complex types because we analyze all type declarations and derive the graphical user interface from them. The *variable grouping* and menu structure are automatically derived from the class structure of the ObjectMath model.

An additional advantage of the new approach is the automatic generation of persistence routines for arbitrarily complex data types.

5 Related work

5.1 Persistence generation systems

There exist various ways to make objects persistent in object-oriented database management systems. In [BB88] such objects must be instances of special classes. In [Deux91] and [LLOW91] objects are assigned to "persistent variables" or "placed into persistent sets". In the *OBST* system [OBST94, CRSTZ92, ACSST94] application developers can program in an object-oriented extension of C. There are no pointers; unique object identifiers are used for references instead. The language includes an option to create objects as persistent data. Primitive data components are not lightweight, therefore high performance necessary for scientific computations and fitting memory constraints is hard to achieve.

Typically OODBMS provide automatically persistence for specific language with the help of OODB language compiler.

The *PGen* system [Tichy94, PGen94, Paulisch90] analyses C++ header files and generates C++ code for reading and writing variables of arbitrary types and classes defined there. This way persistence of data can be easily achieved. Traditionally, to make C++ objects persistent the user has to write the `Store()` and `Load()` functions for every class in the application. The PGen tool generates appropriate functions automatically. In most cases almost no modifications are needed in the C++ header files.

One of the difficult problems with persistent data is how the data should be converted if the type declarations change. It is difficult to do automatically with C++ header files. A special type declaration editor can be designed to trace down all the changes in the type definitions. Then a conversion program can be generated.

Our system reuses the ideas and the code of PGen and extends it for variable display generation.

5.2 Display generation systems

In the *OBST* system a graphic shell visualizes the OBST objects and is used for debugging the data.

The systems *DOST* and *SUITE* [Dewan87, Dewan91, Dewan90, Suite91] generate variable displays from C header files. The translator analyzes specially annotated header files and generates C code that controls message-based communication between the application and a universal display manager. The generated code is linked with the original application. A variable of arbitrarily complex type can be displayed. The display manager can show various C data structures (including pointers and dynamic arrays coded as pointers). The layout can be customized by a large number of attributes (such as colors, help texts, constraints and validation functions) that can be adjusted interactively (with the help of special preference setting dialog) or in the code. In these systems a single description of data types can specify the internal data, the data used for communication between the processes and the data for structure displays. Despite

the large number of attributes associated with every type, variable or variable element, there is no possibility for the programmer to construct new graphical elements when necessary. Persistence can be implemented outside the system.

The *SmallTalk visualization system* [Dewan90A] uses the fact that all the objects in the program have an ultimate ancestor, `Object` that has access to meta-information about the objects, e.g. description of its structure. The display of any object is based on this meta-information. The user can change the attributes of object display by adding some extra SmallTalk code. Persistence can be supported by other SmallTalk methods and it is not part of the visualization system.

Some modern *debuggers* [Debug92] show displays with selected C, C++ or Pascal data structures for data inspection. They are based on symbol tables and dynamically change during program execution. The displays appear automatically in a manner similar to our approach, but they *cannot* be customized by the designer. The user can modify the values, but the validation procedures cannot be specified.

5.3 The C++ language and access to meta-information

C++ is a high level object-oriented language. Nowadays it is widely used in industry for scientific software design, including scientific computing.

C++ supports many ways to simplify the work of the application programmer and to avoid writing unnecessary code. Macro definitions, templates, operator overloading, class inheritance and standard class libraries cover almost all typical needs of application designers. They allow code to be written at a very high level and its size is close to the minimal possible if accurate design is applied. Therefore C++ code analysis and generation is not applied very often. One case where this is necessary is automatic code generation for persistence and displaying data for arbitrary C++ data types.

C++ is a hybrid language in the sense that it operates both with objects and non-objects. This creates difficulties when applying a uniform approach to all values. C++ is a strongly typed language. Therefore when we create code for universal operations that could be applied to many types, code for every type should be written.

It is possible to design persistence and display routines for C++ manually. The problem is that *for every data type* separate routines should be written. Unlike a SmallTalk object, a C++ object cannot automatically provide (or inherit) meta-information about its structure (declarations of types, component names, sizes and types) during run-time. Therefore it does not know how to read, write or display itself.

Unlike SmallTalk, in C++ we cannot tell simply that the variable `foo` should be stored, loaded or shown. For this purpose a relevant function must be declared and defined. The argument type for this function must be the same as `foo` has, and this function must access the internal structure of `foo`. Obviously, the code and control information for such a function should be created in advance. Such

information can be extracted from C++ data type declarations. This is what the PDGen tool does.

6 Conclusions and Future Work

There is a substantial need of automatic generation of graphical user interfaces for many applications. The first generation system for generating user interfaces described in this paper has been in industrial use during more than two years. Experience shows that the model changes tend to require a number of manual changes to the user interface. We have provided a more flexible system that can automatically cope with model changes. Therefore, the more universal second generation system has been designed. The user interface constructed in a partly manual way using the first generation tool can now be generated completely automatically.

The PDGen tool is applied to ObjectMath, C and C++ programs, but it can also be adapted to other languages. Type definitions can be extracted in several ways:

- the source code is parsed and analyzed (in our tool we reuse the analyzer from PGen (see Section 5.1) and apply it to the C++ code),
- analysis of the symbol tables generated by some standard compiler (this is the approach implemented in Suite, see Section 5.2),
- extracting type definitions from the model description, if the application is generated from this model (we apply this to the ObjectMath models),
- creation of the type table with the help of a special data-type definition editor.

The last approach can be combined with the customization tool. In this way both data-type definitions and information about interface details (attribute values) will be integrated under the strict control of one tool. This reduces the possibility of data type mismatches and update problems.

In some languages the type information can be available at run-time with the help of built-in functions; in this case there is no need in code analysis and generation at all.

We are currently working on several extensions to the basic idea implemented in the automatic graphical user interface generator. Some interesting questions that could be considered include:

- the application of the PDGen tool to programs in other languages;
- integration of the tool with (extensible) symbolic debuggers;
- automatic generation of graphical user interface for member functions (not only for data members). For example, if a member function has no arguments it is displayed as a button. When the button is pressed, the function is invoked.
- a more general array browser implementation;

- integration of ObjectMath with tools for data visualization, as it is implemented in the output data browser;
- implementation of the universal browser (Section 3.9) that adapts the graphical user interface according to the type tables given together with the input data;
- the design of a meta-editor that can edit data type definitions.

Remote data editing with the help of widely distributed WWW browsers is another application area. The data resides on the server and can be updated by the clients with the help of HTML interactive forms. One of our future research topics is automatic generation of HTML-based editing tools from data structure specifications.

Finally we would like to mention that a WWW site devoted to the PDGen tool has been organized [PDGen96]. More details about the systems discussed in this paper are available in [E96].

Acknowledgments

Lars Viklund and other members of PELAB group contributed to the design of the ObjectMath language and its implementation. Ivan Rankin improved the English in this paper.

This project is supported by the Swedish Board for Industrial and Technical Development, the Swedish Board for Technical Research and SKF Engineering and Research Centre and the PREPARE Esprit-3 project.

References

[ACSST94] J.Alt, E. Casais, B. Schiefer, S. Sirdeshpande, D. Theobald. *The OBST Tutorial*, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, FZI.049.2, 15nd December, 1994

[BB88] A. Björnerstedt, S. Britts, "AVANCE: An Object Management System", *SIGPLAN Notices*, vol. 23, pp. 206-221. Nov. 1988. In *Proceedings of the OOPSLA'88 Conference*, San Diego, CA, 25-30 September 1988.

[CRSTZ92] E. Casais, M. Ranft, B. Schiefer, D. Theobald, W. Zimmer. *OBST - An Overview*, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, FZI.039.1, June 1992

[Debug92] Debugging a Program. *SparcWorks documentation*. SunPro, October 1992

[Deux91] O. Deux et al., "The O2 System", *Communications of the ACM*, vol. 34, pp.34-48, Oct. 1991

[Dewan87] P.Dewan, M.Solomon. "Dost: An Environment to Support Automatic Generation of User Interfaces". In *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* Vol. 22, No. 1, January 1987, pp. 150-159.

[Dewan90] P. Dewan, M. Solomon. "An Approach to Support Automatic Generation of User Interfaces". *ACM TOPLAS*, Vol. 12, No. 4, pp. 566–609 (October 1990)

[Dewan90A] P. Dewan. "Object-Oriented Editor Generation". *Journal of Object-Oriented Programming*, vol. 3, 2 (July/Aug 1990), pp. 35-49

[Dewan91] P. Dewan. "An Inheritance Model for Supporting Flexible Displays of Data Structures". *Software - Practice and Experience*, vol. 21(7), 719-738 (July 1991)

[Dewan91A] P. Dewan. *A Tour of the Suite User Interface Software.* Included in [Suite91]

[E96] V. Engelson. *An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing.* Linköping Studies in Science and Technology. Licentiate thesis No 545. Department of Computer and Information Science, Linköping University, 1996.

[Fritzson93] P.Fritzson, V. Engelson, L.Viklund. "Variant Handling, Inheritance and Composition in the ObjectMath Computer Algebra Environment". In *Proc. Of DISCO'93 - Conference On Design and Implementation of Symbolic Computation Systems*, LNCS 722, Sept. 1993

[Fritzson95] P. Fritzson, L. Viklund, J. Herber and D. Fritzson. "High-level Mathematical Modeling and Programming". *IEEE Software*, July 1995, pp. 77-86.

[FWHSS96] P. Fritzson, R. Wismüller, Olav Hansen, Jonas Sala, Peter Skov. "A Parallel Debugger with Support for Distributed Arrays, Multiple Executables and Dynamic Processes". In *Proceedings of International Conference on Compiler Construction*, Linköping University, Linköping, Sweden, 22-26 April, 1996, LNCS 1061, Springer Verlag.

[LLOW91] Ch. Lambs, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, vol. 34, pp. 50-63, Oct. 1991

[OBST94] OBST, a persistent object management system. Available as `ftp://ftp.ask.uni-karlsruhe.de/pub/education/computer_science/OBST`, see also `http://www.fzi.de/divisions/dbs/projects/OBST.html`

[Ou94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994

[Paulisch90] F.N.Paulisch, S. Manke, W.F.Tichy. "Persistence for Arbitrary C++ Data Structures". In *Proc. of Int. Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, FRG, May 1990, pp. 378–391.

[PDGen96] V. Engelson. *The PDGen tool.* Information available as `http://www.ida.liu.se/~vaden/pdgen`.

[PGen94] *PGen, a persistence facility.* This software is available as `ftp://ftp.ira.uka.de/systems/general`.

[Tichy94] W.F.Tichy, J.Heilig, F.N.Paulisch. "A Generative and Generic Approach to Persistence". *C++ report*, January 1994. Also included in [PGen94].

[Wolfram91] S. Wolfram, *Mathematica - A System for Doing Mathematics by Computer* (second edition), Addison-Wesley, Reading, Mass., 1991.

[Viklund95] L. Viklund and P.Fritzson, "ObjectMath: An Object-Oriented Language for Symbolic and Numeric Processing in Scientific Computing", *Scientific Programming*, Vol. 4, pp. 229-250, 1995.

[Suite91] SUITE, user interface software. This software is available as `ftp://ftp.cs.unc.edu/pub/users/dewan/suite`. Some information available as `http://www.cs.unc.edu/~dewan/`

[SUN90] SUN Microsystems Inc. *Network Programming Guide (Ch. 5,6)*, 1990