Using C and C++ Objects from Neo/NST: the Tools Mku+ and Mku++

Helge Ritter, Nov 97 [1]

Department of Information Science
Bielefeld University email: helge@techfak.uni-bielefeld.de

Abstract

This manual provides the main documentation for using the source code generator tools `mku+` and `mku++`. `Mku+` generates the necessary interface code to make a collection of cooperating `C`-routines (a `C`-"class") accessible as a NST-unit in NST-programs or interactively under *Neo*. `Mku++` provides a similar service for a genuine `C++` class. Both tools are extensions of the earlier `mku` tool and allow to maintain a clean separation between the NST/Neo level and the level of the interfaced objects, which suffice to be available in compiled form.

[2]

Contents

1 Overview

This manual supplements the `NST programming guide` and provides the necessary information how to "wrap" existing `C` or `C++` objects such that they can be used within

[1] Compiled: March 28, 2002, Id: mkudoc.tex,v 1.2 2001/02/21 14:13:19 walter Exp

[2] the latex- and ps-files `mkudoc.tex` and `mkudoc.ps` for this document are currently located in `/homes/helge/nst5/doc/tex` and `/homes/helge/nst5/doc/ps`.

NST. The writing of the necessary wrapper code is supported by the two tools `mku+` (for the `C` language) and `mku++` (for `C++`). The emphasis is on re-use of already existing functions that have either been programmed in an object-oriented fashion in ANSI-C or that are available in the form of a `C++` class, keeping NST code and the actual class implementation entirely separate (this differs somewhat from the main focus of the `NST programming guide` that explains how to program new NST units, using `C` and `NST` in an integrated fashion).

In the following, we will use the term "class" both for `C` and `C++` to refer to a set of closely related functions ("methods") that (usually) operate on a collection of shared ("private") data that remains persistent between function calls.

! In `C`, implementation of a "class" with private persistent data requires each function to have in its argument list besides its primary function parameters an additional "handle" parameter to access a dynamically allocated instance of the shared data set that represents the "class instance" (cf. below).

Before considering the use `mku+` and `mku++`, a brief sketch of the basic general scheme how classes can be represented by NST "class containers" may be useful.

2 Making Classes available as NST units

Simple classes. For simple classes, whose methods have one main operation ("execution" of some main function associated with each instance), with a few auxiliary "control" operations plus methods to load and save the current state from/to a file, the most natural way will be to embody a class instance as single NST unit, with obvious choices for the NST `exec`, `ctrl`, `load` and `save`-functions. Additionally, NST offers a special `adapt` method for classes representing adaptive objects which have a second main "adaptation" operation besides their execution. The majority of existing standard NST units follow this scheme, and the tool `mku` can be used to support the writing of the necessary code (not any further considered here).

Larger classes. To represent a class with a larger collection of methods in NST, the recommended way is to represent each class instance as a *named container unit* (henceforth referred to as *"class container"*), that contains for each class method a separate subunit (henceforth denoted as *"class subunit"*). Each class subunit represents with its `exec` function the class method and, therefore, can have its input and output interface specifically tailored to the needs of its method. A method of a class instance is then invoked by executing the corresponding class subunit. To make this possible, both the class container and its class subunits must be *named* (in *Neo*, this is indicated by an underlined label), so that the class subunits can be invoked by means of NST's `method` and/or `use_named` units (using NST's "path names", such as `NameOfContainer:NameOfSubunit1` etc.). The sharing of a common set of "private" data is achieved by making all class subunits share the data set held by their container unit (for details, cf. below).

Additionally, all class subunits of a class container share common `ctrl`, `load` and `save` functions that can be considered as being associated with their class container and that, therefore, always pertain to the entire class instance.

To implement this very flexible scheme for a given class requires to write a suitable constructor function that instantiates a class container with the desired class subunits and that assigns as the `exec` functions of the class subunits the given class methods.

Additionally, we may wish wrapper code defining dialog windows to allow creating and using instances of the class container interactively under *Neo*. The tools `mku+` (for a class implemented in `C`) and `mku++` (for a class implemented in `C++`) generate most of the code that is required for both parts; the few additional steps required are described below.

Nested classes. NST class containers can be nested, as any other kind of NST units. This does not necessary imply that the wrapped class must have a similar structure (it may, but it need not); it can be freely used to impose a hierarchical ordering on a larger groups of methods, which is particularly convenient and recommendable for classes that have many methods and that shall be used interactively under *Neo*.

! It is in the freedom of the designer what behavior he defines for the `exec` and `adapt` method of the class container. The default for a “normal” container would be to execute (adapt) all its subunits, which obviously is not useful for a class container. Therefore, the tools `mku+(+)` generate code that disables this behavior (by prepending the subunits with a `return` unit) such that execution (adaptation) of a class container is without effect (this does not apply for the `ctrl`, `load` and `save` methods. As explained above, these shall indeed act on the entire class instance). This is the recommended standard practice; however, it is easily possible to bind additional class methods specifically to the `exec` and `adapt` functions of the class container itself. This may be useful, if there are one or two methods that are clearly privileged in the class.

3 Wrapping C Classes with `mku+`

We consider as an example a `C` class `vects` that offers methods to store and manipulate a set of `float` vectors of a given, fixed dimension. An instance of `vects` can hold zero or more vectors, and methods `add()`, `set()`, `get()` and `clear()` allow to add a further vector, replace the elements of an already stored vector, get the elements of an already stored vector and remove all stored vectors, respectively. A further method `forall` is provided to step through all vectors (cf. below). The interface to a given implementation might be defined by the following header file:

```
typedef struct vec_{
   /* this is an auxiliary struct */
   float *pfData;
   int iIndex;
   struct vec_ *pNext, *pPrev;
} vec_t;

typedef struct vects_{
   /* this implements an instance */
   int iDim;
   int iNum;
   vec_t *pFirst, *pLast;
   void (*add)(float*, struct vects_*);
   int  (*num)(struct vects_*);
   float *(*get)(int, struct vects_*);
   float *(*set)(int, float*, struct vects_*);
   void (*clear)(struct vects_*);
   void *(*forall)(float*, int*, void*, struct vects_*);
} vects;

extern vects *create_vects(int);
extern void   remove_vects(vects*);
```

Listing 1: definition of a C “class”

To create the necessary wrapper code for using this class from NST, invoke `mku+` with arguments specifying the desired name (say, `vcl`) for the constructor function of the NST class container, followed by the name `vects` of the `C-struct` implementing the to-be-wrapped class, followed by the names to be chosen for the class subunits that will wrap the methods. For example, to generate wrapper code for the first five element functions, we might issue the call

```
mku+ vcl vects add num get set clear
```

(we ignore the last method `forall` for a moment). This will produce the following three files:

`nst_vcl.c` A file containing a code template for a NST constructor function `vcl(..)` for a class container with subunits named `add`, `num`, `get`, `set` and `clear` with associated `exec` functions `exec_add .. exec_clear` to represent corresponding class methods. The symbol `CUSTOMIZE` indicates all places where code has to be filled in to finalize the interface and below we will illustrate this for the present example.

`neo_vcl.c` A file containing a code template to wrap the NST constructor function `vcl()` that is defined in `nst_vcl.c` for interactive use from within *Neo*. Finalizing this template mainly requires to specify the dialog windows querying the parameters for calling `vcl()` and to issue a call `vcl()` with parameters taken from these windows.

`vcl.mak` A `makefile` to compile the generated pair `nst_vcl.c` and `neo_vcl.c` of NST/Neo wrapper files into a shared library `vcl.so`. The `makefile` assumes that there exists also an object file `vects.o` (or that there are two source files `vects.[ch]` from which it can be made) that implements the wrapped class `vects`. This object will become included into `vcl.so`, so that `vcl.so` will contain everything that is needed to use the new class.

The following are mainly comments about the necessary steps to complete the generated wrapper file, using the example of `nst_vcl.c` as an illustration. The following steps need to be considered:

Import of the original C-class: This requires the availability of all definitions of the class structs (and any auxiliary structs appearing therein) plus definitions of any methods used. For a properly implemented class, all necessary definitions will be collected in a header file. `Mku+` expects this header file to be named after the class (i.e., `vects.h` in the present example) and will insert a corresponding `#include` directive in the generated `nst_xxx.c` file.

Definition of constructor function for NST class container: Most of the definition of the constructor function is already given in the generated code template. The name will be given by the first command line parameter (here: `vcl`). Usually, the to-be-wrapped class instance will depend on a number of parameters and the constructor function must fetch these via its argument parameters. In the present case, the only required information is the dimension of the vectors to be handled, i.e, we will have a single `int` parameter (in addition to the always present `uHost` argument). This information is then used to create a corresponding instance of the to-be-wrapped class. To make it accessible to `uNew` and all subunits, we use a `struct nst_vcl` that keeps the address of the to be used class instance as a local variable `pClass` for the created NST unit; i.e., this struct is assigned to the `nst_var_of()` pointer of the wrapping container unit `uNew`:

```
typedef struct nst_vcl_{
   vects *pClass;  /* ptr to instance of C class instance to be used */
} nst_vcl_t;

unitptr vcl( int iDim, unitptr uHost) {
   unitptr   uNew;  /* handle to unit to be returned */
   nst_vcl_t *pVar; /* handle to local variables of unit */
   ...

   /* make class container unit: */
   if ((uNew=primitive_unit(0,0,uHost))==NULL) {
      ... diagnostics ...
   }
   /* create vects instance of to-be wrapped class
      and assign address handle to uNew's pVar->pClass ptr:
   */
   nst_var_of(uNew) =
   pVar             = CALLOC(1,nst_vcl_t);  /* make pVar instance */
   pVar->pClass     = create_vects(iDim);
   return_unit(uNew);  /* to protect subunits below */
   nst_indicate_export_of_methods(uNew);
   ...
}
```

Listing 2: NST constructor to wrap C class

! In many cases, `pClass` will be the only element that is needed in the struct `nst_vcl_t`; then, this struct could in principle be omitted, and everywhere `pClass` be used directly in place of `pVar`, saving one level of indirection everywhere. However, care must then be taken in the `remove` routine when `pClass` is freed: As long as `pClass` coincides with the `nst_var_of()` pointer of the to-be-removed unit, NST will automatically apply the `free()` routine to this pointer. Usually, this will interfere with a previous class destructor routine call to the same pointer. This problem can be circumvented by setting `nst_var_of()=NULL` before NST can make its `free()` call (which does nothing, if applied to `NULL`):

```
int remove_vcl(unitptr u) {
   vects *pClass = nst_var_of(u);  /* pClass directly used */
   remove_vects(pClass); /* application of class destructor */
   /* now pClass already freed, but NST will try again! */
   nst_var_of(u) = NULL; /* now NST's free() cannot do harm! */
   return nst_remove0(u);
}
```

The remaining two statements protect against direct execution of the class containers subunits (cf. above) and set a flag that indicates to NST that the container serves the special function of representing with its subunits a collection of class methods.

Constructors for class subunits of class container: The generated code contains already one constructor call for each specified class subunit. However, the interface that each class subunit needs must be decided by the programmer, and suitable parameters for the `primitive_unit()`-constructor calls inserted. Usually, these will be computed from the parameters of the class container constructor function discussed in the preceding point. In the present case, choices for the class unit's interfaces might result in

```
{
   unitptr uSub_add   = primitive_unit(NST_INP, NST_FLOAT, iDim,
                          NST_OUT, NST_SCALAR,1, NST_HOST, uNew);
   unitptr uSub_get   = primitive_unit(NST_INP, NST_SCALAR, 1,
```

```
                              NST_OUT, NST_FLOAT, iDim,
                              NST_OUT, NST_SCALAR,1,NST_HOST,uNew);
   unitptr uSub_set   = primitive_unit(NST_INP, NST_SCALAR, 1,
                              NST_INP, NST_FLOAT, iDim,
                              NST_OUT, NST_SCALAR,1, NST_HOST,uNew);
   unitptr uSub_clear = primitive_unit(0,0,uNew);
   /* name subunits, inherit uNew's data ptr \& set up method: */
   name_unit("add",uSub_add);
   nst_var_of(uSub_add) = nst_var_of(uNew);
   uSub_add->exec = exec_add;
   ...
}
```

Listing 3: NST constructor cont'd

Most of this code is already in place, only the argument lists of the `primitive_unit()` calls had to be completed. The `nst_var_of()` statements allow each subunit to share with its host `uNew` access to the wrapped class instance. This will be used by the method functions explained in the next step.

Complete `exec`-functions: As a final step, each class subunit must (i) build from its current input data suitable calling parameters for its associated class method, (ii) invoke this class method, and (iii) write any result values back into the outputs of its interface. Steps $(i - iii)$ must be carried out for each call to the `exec`-function of the class subunit. The skeleton code for these `exec` function is already created by `mku+`, but code implementing steps $(i - iii)$ still must be inserted.

For example, subunit `uSub_get` shall implement method `get`, which fetches a stored vector by an integer index and returns the starting address of the vector data, if the vector could be found, or `NULL` if the specified index value was illegal.

To implement this, subunit `uSub_get` has a single scalar input pin to specify the index, and a `iDim`-dimensional output field to return the vector data plus a further single scalar pin to signal success or failure. Thus, a suitable `exec` function to implement steps $(i-iii)$ above would look like

```
int exec_get(unitptr u) {  float *pfData;
   nst_vcl_t *pVar = nst_var_of(u); /* get local var */
   vects *pClass   = pVar->pClass; /* now pClass=class handle */
   /* invoke class method with parameters fetched from input: */
   if ((pfData = (*pClass->get)((int) fNstInp(u,0,0),pClass))) {
      /* copy result data into outputs: */
      memcpy(pfNstOut(u,0,0),pfData,pClass->iDim*sizeof(float));
      fNstOut(u,1,0)     = 1;  /* indicate success */
   } else fNstOut(u,1,0) = 0;  /* indicate failure */
   return OK;
}
```

Listing 4: wrapping a method

Similar code has to be filled in for the other `exec` methods to complete the implementation.

Further Methods: Analogous remarks apply to the completion of the `ctrl`, `load` and `save` methods. The only difference is that these apply to the entire class instance, with obvious semantics for `load` and `save`. The `remove` method must be completed by a call of the class destructor function, after which the NST default constructor function is called:

```
int remove_vcl(unitptr u) {
   nst_vcl_t *pVar = nst_var_of(u); /* get local var */
   remove_vects(pVar->pClass);  /* << this line was inserted */
   /* NOTE: DON't remove pVar itself -- is done by nst_remove0! */
   return nst_remove0(u);
}
```

Listing 5: implementation of `remove` function

! Wrapping a class as a class container has the advantage that later further class subunits can be added without breaking the usability of the changed class container in previously generated .*.NST files under *Neo*(this applies to the circuit part of these files; the data part has to be discarded when class subunits are added).

! We did so far not discuss any `adapt` methods. When needed, these are handled entirely analogously as the `exec` methods, i.e., provide analogous definitions `adapt_get` etc., fill in code for steps analogous to $(i - iii)$ above and set the corresponding `adapt` function pointers of the NST class subunits.

The present scheme offers the additional flexibility that, when appearing convenient, any of the class methods may be bound to more than one class subunit. For instance, the `adapt` method of a `bpclass` might be bound as the `adapt` function to a subunit whose `exec` function already represents the execution method of `bpclass`. Additionally, the same method might be bound to the `exec` function of a second class subunit. This would allow invocation of the adapt operation also as part of some `exec` operation, provided the second subunit is used. Alternatively, the second unit might offer a different interface.

4 Exported Iterator methods

Many methods are conveniently formulated in terms of an iteration over a range of data elements held in a class. This calls for method subunits that work as operator units (similar to the `for_op` unit), but that, when referenced, act on the successor units *of the referencing unit* (and not on their own successor units, which would be within the class container). We call such methods *exported iterators* for obvious reasons.

Since exported iterators are useful in many contexts, NST offers a simple scheme for their implementation. It requires only to write a function, say `void *forall(..., void *pEl, ...)`, that on each call advances a pointer `pEl` to the next element that is to be iterated, applies the desired operation to this element, and returns the value of the advanced pointer. Calling the function with `pEl=NULL` must return `pEl` on the first element of the range; a return value `pEl=NULL` must signify that the range is exhausted.

To indicate to `mku+` (or `mku++`) that a particular method is to become an exported iterator requires to prefix the method name with a hash symbol. In the example class `vects` the last method, `forall`, is of this kind, so that for its inclusion we would call

```
mku+ vcl vects add num get set clear \#forall
```

This would add in the constructor function another `primitive_unit()` for a further subunit plus the following four lines for its configuration:

```
...
nst_operands_of(uSub_forall) = 1;
```

```
    name_unit("forall",uSub_forall);
    nst_var_of(uSub_forall) = nst_var_of(uNew);
    uSub_forall->exec = (exec_function_t) exec_forall;
    ...
```

Listing 6: an iterator subunit

The major difference is the first line, which tells NST that the subunit is to be treated as an operator unit. The necessary implementation of the `exec` method for these units now has to call the `forall`-function explained above. This requires to use an `exec` function that has an *an additional argument* to pass the iteration pointer `pEl` with each call of the `exec` method, and to return the advanced value.

This mechanism allows to hide all the “machinery” for iterating the successor units in the calling NST unit, which only needs to repeatedly call the `exec` method of the wrapper class subunit, starting with iterator pointer `NULL` for beginning with the first element, and executing the class subunit and its own successors until the iterator pointer again becomes `NULL`.

The implementation of the `exec_forall` routine might then look as follows:

```
void *exec_forall(void *pEl, unitptr u) {
   vects *pVar = nst_var_of(u);
   int iIndex=0;
   /* code for advance of pEl & single iteration on *pEl: */
   pEl = (*pVar->forall)(pfNstOut(u,0,0), &iIndex, pEl, pVar);
   fNstOut(u,1,0) = iIndex;  /* no data fetched */
   return pEl;
}
```

Listing 7: implementation of iterator method

When a `method` or `use_named` unit detects in the referenced method subunit an `exec` function of this kind, it will execute the following pseudocode:

```
   ...
   pEl = NULL;  /* "predecessor" of element 0 */
   while ((pEl = exec(pEl,u))) {
       /* pEl != NULL: not yet at end, thus: */
       execute successor units
   }
   ...
```

Listing 8: pseudocode description of iterator

This will produce the desired effect to traverse the full range of class elements, executing the successor units of the `use_named` or `method_unit` for each class element once.

! **Nested class containers**. The subunits of a class container may themselves again be made into class containers. This happens, when a class subunit is not made by a simple `primitive_unit` call, as in the present example, but by another class container constructor function. If this function wraps a different class, nothing else needs to be done (except perhaps a suitable nameing of the nested class unit), since exec function and variables have already been set by the constructor function. However, such unrelated nested class containers are not particularly useful. A more useful case occurs when the nested class container accesses the same class instance as its host class container. This can be easily achieved by setting its `nst_var_of()` and its exec functions suitable; however, this code is not automatically generated by the `mku` tools and has to be inserted manually.

! A very useful and straightforward way is to collect some class subunits in a common host unit (setting all `nst_var_of()`'s to the variables of the shared top-level class container) and thereby partition a large collection of methods into smaller subgroups of related methods. To make the common host unit into a class container requires an additional `nst_indicate_export_of_methods()` statement. The required code is analogous to that for the top level class container and has to occur in the place where otherwise the constructor function for a non-nested class subunit would sit:

```
-
...
uNested = primitive_unit(...,uNew);
/* uNested becomes the nested class container */
nst_var_of(uNested) = nst_var_of(uNew);
/* to share the already created class instance */
return_unit(uNested);  /* the protective return */
nst_indicate_export_of_methods(uNested);
/* this makes the class container complete.
   Now fill it with class subunits: */
/* first class subunit within uNested: */
uSub = primitive_unit(... uNested);
nst_var_of(uSub) = nst_var_of(uNested);
uSub->exec = whatever exec function is desired
nst_name_unit("innermost",uSub)
/* next class subunit within uNested: */
...
```

Listing 9: Making a nested class container

The analogous scheme can be used with `C++`. The file `nst_llm_net.C` presents an example of this technique.

5 Wrapping `C++` Classes with `mku++`

Most remarks above translate rather directly to the `C++` case, and `nst_vects.C` is the `C++` source file that implements the above example in `C++`, using header files `vects.H` and source file `vects.C`.

To separate the given implementation from the .NST interface, the wrapper defines an auxiliary class which uses the original class as its base class and extends it for each wrapped method by an element function that issues the method call, "sandwiched" between the necessary data transfers (for input and output) with its associated NST class subunit.

This, together with the following remarks and a look at the generated source code, perhaps might suffice to make clear what happens:

1. The header file is now assumed to contain the definition of a `C++` class

2. the generated source file defines a derived class whose name is obtained by suffixing the original class with `_aux`. The derived auxiliary class needs as additional element functions the interface functions for NST (i.e., one `exec` function for each method subunit plus the global `ctrl`, `load` and `save` functions. Skeletons for these functions are provided and must be completed. Additionally, we must define a class constructor in terms of the constructor of the base class methods.

3. similarly as before, we must insert a constructor call to make an instance of the (derived) `C++` class in the constructor function for the NST class container, and use its address to set the `var` pointer of the class container, as well as the `var` pointers of all its class subunits.

4. Since the `exec` methods are now defined within the derived auxiliary class, the necessary interface functions "outside" become very simple and are generated entirely by `mku++` (the formerly necessary code insertions [to implements steps referred to as $(i-iii)$ above] must now be made in the element functions added to the derived auxiliary class).

5. It might look simpler to avoid the apparent detour via the "outside" interface functions and to instead directly set the `exec` function pointers of the NST subunits to the adresses of the element functions in the derived auxiliary class. This, however, will fail, since usage of function pointers to .C++ class methods is somewhat more restricted as accustomed from .C: the compiler would be unable to set up the proper correspondence between the functions and their accomodating class instance's data.

6. Iterator methods are implemented in the same way, but note that the adopted convention for the iterator pointer (advancing before the current iteration) differs from the standard use of `C++` iterators (which are advanced *after* each iteration).

As before, this scheme needs the header file plus the object file implementing the wrapped C++ class, thus allowing NST level and implementation level of the class to be kept entirely separate.

6 Class Containers under *Neo*

Under *Neo*, class containers are marked with a red plus; they are handled in the same way as other units, and their construction is done in the same way as for the familiar, "simple" class units (dragging the class's icon into the circuit and specifying the necessary parameters for the constructor function). Unlike "simple class icons", the created icon can be opened to inspect the methods that it contains. Each method appears as "fixed subunit" that cannot be moved, duplicated or deleted individually, but that can be invoked by its name via the NST `method` or `use_named` units. Class icons of this kind are indicated by a red plus mark in the upper left corner of their icon.

The class subicons of a class container cannot be modified independently; they can only be remade together with their class container. However, it is possible to define for each class subicon a separate *Neo*dialog window by using the `ICONFORM_FOR_METHOD()` macro *within* the `ICONMAKE()` function (where class container's `uNew` is created):

```
...
ICONFORM_FOR_METHOD(nameofmethod_1)
   " ... textstring for method_1 dialog window ..."
ENDFORM;
...
ICONFORM_FOR_METHOD(nameofmethod_k)
   " ... textstring for method_k dialog window ..."
ENDFORM;
..
```

Listing 10: Definition of dialog windows for class subicons

Here, each `nameofmethod` is a name of one of the class subicons (without apostrophes), and all or just a subset of class subicons may have dialog window definitions. A class subicon may itself be a class container with its own nested class subicons. However, since there is only a single *Neo*wrapper for the top-level class container, any dialog window assignments for nested class subunits must also be made witin the `ICONMAKE` routine of the top level class unit, identifying the proper nested class subunit by its composite path name (e.g. `set:beta`) for a class subunit named `"beta"` nested in class subunit `"set"` within top level class container.

If any of the dialog window definitions contain input widgets, these will appear, but remain unused, since class subunits cannot be remade individually; their dialog windows are, therefore, just for displaying some help text. An exception are "placeholder" class subunits (cf. below). Their dialog window requests a single name to specify another NST object which for which they then act as placeholder (cf. below).

To invoke the methods in a class container, use the `method` unit, specifying the name of the method in the desired container. If the referenced method instance exists, the `method` unit will inherit the interface of the method unit and its execution will invoke the referenced method instance. A similar function is provided by the `use_named` unit, but the call by `method` unit is more efficient and is to be preferred.

! There are two differences in the semantics of the `method` and the `use_named` unit: (i) the `method` unit's outputs provide access to the output values of the referenced unit by-reference instead of by-copy (as the `use_named` unit does). (ii) the method unit leaves no side-effects on the input values of the referenced unit (again, this differs from the `use_named` unit that leaves the referenced unit with new input values copied into its inputs).

With this scheme, even complex `C` or `C++` objects can be broken down into small and independently callable parts with interfaces that can be sufficiently simple and clear for interactive use. It also provides a convenient base for the implementation of objects whose contents can dynamically grow or shrink, since exportable iterators provide the necessary flexibility to process each element of a dynamical class object through an arbitrary *Neo*circuit.

7 Adding Diagnostics

Usually, the class objects will need some diagnostic output. When the class is used from NEO/NST, we would like to redirect this output suitably into NEO/NST.

A proper way to approach the task is to issue error messages always through a function that is exchangeable [3] In `C`, the suggested way is to use a function pointer; in `C++` a virtual function can be used. When the object is created from NST, the function pointer can be redirected to a suitable NST error output routine; in `C++` the virtual function is redefined for the derived auxiliary class.

To simplify interfacing, we suggest to equip each class object for its diagnostic output with a virtual function or function pointer. As the type for this function we suggest `int (*)(FILE*,const char,...)`, i.e., the type of the `fprintf()` function. This makes it easy to issue error messages with a variable number of embedded parameters.

[3] A poorly designed class object may use a fixed set of diagnostic output routines. Then, a solution most likely will require to modify the code of the class object. Another poor solution would be to call NST's diagnostic routines from the object; this would make a standalone usage of the object (without NST) impossible.

When using a C class object from NST, we can redirect the diagnostics function pointer to the NST function `int nst_eprintf(FILE*,const char*,...)`, which is of the same type as the `fprintf` function, but ignores its `FILE*` argument and instead sends its output through NST's error output routines. This requires the additional line

```
...
/* replace diagnostics with NST diagnostics print function: */
pVar->pClass->ERROR_OUTPUT = nst_eprintf;
...
```

in the NST constructor function, with `ERROR_OUTPUT` `#define`d as the name of the diagnostics function pointer of the used C class.

In C++, we must instead of the function pointer `ERROR_OUTPUT` redefine a virtual function of the same name. Since this function has a variable argument parameter list, we must use the `va_list` mechanism to access the parameters and pass them to NST's `nst_veprintf()` function, which does the same as `nst_eprintf`, but with a `va_list` parameter for the arguments:

```
...
int ERROR_OUTPUT(FILE *fp, char *pcFmt, ...) {
   va_list vArg;
   va_start(vArg,pcFmt);
   (void) nst_veprintf(fp,pcFmt,vArg);
   va_end(vArg);
   return 1;
}
...
```

The `mku+` and `mku++` tools generate the necessary code for both cases, requiring only to `#define ERROR_OUTPUT` to the name of the function pointer/virtual function via which the imported class does its error output.

$\boxed{!}$ If the class objects wants to distinguish between errors and warnings, it should use an additional, second function/function pointer, with an analogous set of the above wrapper statements, but now using the functions `nst_wprintf()` and `nst_vwprintf()`, resp.

Note also that in C++ we need to declare the destructor as virtual, when some element functions are virtual.

8 Wrapping Class Families:

Frequently, a complex concept is better broken down into a family of several, closely cooperating classes. This can give considerable gains in modularity and flexibility. Consider, for example, the concept of a graph with nodes and edges. Instead of a single class with methods for the construction and deletion of nodes and edges between nodes, we might have a class T1 for the edges that cooperates with a class T2 for the nodes. If properly designed, this would allow to endow a common set of nodes (instance t2 from T2) with several types of edges (separate, differently parametrized instances from T1 with their own constructor and destructor functions). Another example is a class T1 for the methods of a neural networks that might wish to use instances of a class T2 of data sets as its parameters.

The general feature of these cases is that instances t1 of a class of type T1 have methods that for their operation need access to instances t2 of a second class of type T2, i.e., class T1 has among its parameters objects of another class T2 (with obvious generalization to more than a single other class).

In order to allow wrapping of such class families, NST allows to provide a NST class container with "class placeholders" that can be dynamically bound to suitable instances of other classes (such as T2). These class placeholders translate the referencing of one class instance by another into a corresponding referencing between two named NST units. Classes that use other class instances as their parameters thus can appear under Neo as class containers with class placeholder icons that can be bound interactively to other class containers that represent the parameters.

The translation is based on the availability of a class T1 method (say, `T1::select(T2 *t2)` for designating a particular class T2 instance `t2` as the parameter to which subsequent actions of a particular class T1 instance will refer (with the semantics that `T1::select(NULL)` is legal and indicates that the parameter has become unavailable (i.e., subsequent T1 actions that require the parameter will produce an error message). If no such method exists (e.g., because the given class T1 uses T2 everywhere as an explicit argument in its methods), it is usually easy to derive from T1 a new class with new element functions in which all T2 parameters have disappeared and the T2 parameter is designated by calls to a select function instead.

The class placeholder for binding a T2 instance to T1 is a special NST class subunit that basically wraps the `T1::select(T2 *t2)` call suitably. It is first created in the same way as a "normal" class subunit (i.e., by a `primitive_unit` call, specifying a suitable interface). This normal class subunit, say `uSub`, is then "converted" into the class placeholder by issuing a call to the special constructor function `virtual_method()`:

```
uSub = virtual_method("name", Accept, Notify, pcOptions, uSub);
nst_name_unit("var",uSub);
```

Here, `name` can specify the name of a class container that is the first binding candidate, `int (*Accept)(unitptr)` is a to-be-supplied function that must specify whether a binding candidate (a NST class container) is of acceptable type (return value 1) or not (return value 0). `Void (*Notify)(unitptr, unitptr)` is second, to-be-supplied function that essentially is responsible for the `T1::select()` call (explained below), `pcOptions` specifies options and may be NULL.

The `Notify(uSub,vNew)` function will be called from the class placeholder subunit (with its first argument `uSub` set to the class placeholders `unitptr`) whenever the referenced class container has to be changed from the current instance to a new instance `vNew` (these calls are issued automatically whenever NST receives a request (e.g. from NEO) to bind a different instance). The particular value `vNew=NULL` is used to indicate that the current instance has become inaccessible or deleted. The `Notify(uSub,vNew)` call is assumed to retrieve the `C++` element function `T1::select` from `uSub` and to call it with the class address of the `T2` instance retrieved from `vNew`. Therefore, a suitable implementation of `Notify` is, e.g.,

```
void Notify(unitptr u, unitptr v) {
   // will be called with v = NST-class-container-of(t2)
   void *t2 = v ? nst_var_of(v) : NULL;  // t2 from class container
   ((T1*) u)->select((T2*) t2);     // desired C++ function call
}
```

Listing 11: example of Notify function

(of course, with the function name `select` replaced by the actual name of the corresponding function of class T2)

When a call to `Notify` occurs, the `Accept` function was already applied to `vNew` and has returned 1. This can be used to prevent binding of unsuitable classes. The `select` function is assumed to do (at the `C++` level) further type checking of the to-be-bound class instance and to accept it only if these checks are positive).

For further details, see the MANPAGE of the `virtual_method` unit.

9 Example:

The files `llm.C`, `llm.h`, `data_set.C` `nst_llm_net.C`, `neo_llm_net.c` provide an example of this technique.

10 Sources and Makefiles:

The source files for the examples in this manual are:

```
nst_vects.c        NST wrapper for C-example
neo_vects.c        NEO wrapper for C-example
vects.c            implementation of C class
vects.h            headers of C class
vects.mak          makefile for C example

nst_vects.C        NST wrapper for C++ example
neo_vects.c        NEO wrapper for C++ example
vects.C            implementation of C++ class
vects.H            headers of C++ class
vects++.mak        makefile for C++ example
```

Calling `make -f vects.mak` or `make -f vects++.mak` will compile the corresponding example sources and produce a shared library named `vects.so`. Since we have stubbornly used the same base name for the `C` and the `C++` implementation, the resulting library will in both cases be named `vects.so`, but will be a different binary (however, indistinguishable in its behavior at the NST level; in fact, already the *Neo* wrapper is the same source file for both implementations). The makefiles may need some editing in the paths used.
To use the wrapped `C` or `C++` class, call `Neo` with the line

```
neo -i +vects.so
```

If the usual naming conventions are obeyed and `C++` files are placed in `xxxsrc` directories and are suffixed `*.C`, the normal `nst5` makefiles should automatically do the right thing.