Neo - the NST graphical editor

Helge Ritter, September 95 [1]

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

Abstract

The graphical editor *Neo* (**N**ST - **e**dit**o**r) allows the graphical construction of *NST*-programs, their interactive execution and the creation of C-source code. This document describes the basic features of *neo* and describes some introductory examples.[2]

Contents

[1]Compiled: March 28, 2002, Id: neo.tex,v 1.4 2001/02/21 14:17:24 walter Exp

[2]the latex- and ps-files `neo.tex` and `neo.ps` for this document are currently located in subdirectory `doc/tex` of the *NST* file tree. The *Neo* sources are under CVS-control in subdirectory `neosrc` of the *NST*-file tree. For remarks on the available examples, see the `readme`-file therein.

1 Overview

Neo allows graphical development of *NST* -programs by visualizing *NST* -units as icons that can be created, copied, connected and removed by simple mouse actions. *Neo* is currently running on Linux PCs, SGI Irix and Sun Solaris workstations. The Linux, Irix and Solaris versions can also be compiled to use OpenGl (under Linux available in the Mesa package) for 3d graphics.

Its main features are:

Interactive graphical construction of NST *-programs.* *NST* -units are visualized as icons. Programs are created by connecting icons into circuit diagrams. Subroutines are icons that contain subcircuits. More than 100 *NST* -units for neural networks, data visualization, mathematical transformations, image processing etc. are currently available as building blocks.

Interactive execution of NST *-programs.* Circuits, subcircuits and single *NST* -units may be executed and/or adapted interactively. Execution can be interrupted to check intermediate values or to try out modifications of the program. The current state of single units, subcircuits or of the entire program can be saved to file and restored later.

Visualization facilities. A number of *NST* -units allows to visualize data at any point within a program. Besides simple two-dimensional pixel images, histograms and graphs, a set of three-dimensional visualization primitives can be combined in a modular fashion to build customized and interactive 3D-data-viewers for on-line monitoring of experiments.

Interactive extensibility. By creating new subcircuits, the user can extend the predefined collection of *NST*-units. This can be done interactively, without programming. User-defined *NST*-units can have integer, float and string parameters and they are dynamically loadable from files.

Circuits as parameters. A special *NST*-unit allows circuits to use other circuits as parameters. This allows to define circuits at a high level of abstraction and facilitates the design of more modular, and more re-usable software parts. Numeric and string variables are supported as well.

Source code generation. The constructed *NST*-programs can be written as `C`-source code. Each subcircuit is translated into a `C`-subroutine with its parameters given by the parameters of the sub-circuit. Parameters of the top-level circuit can be translated into command line parameters of a created `main`-program.

Documentation of created programs. The created source code is in a format such that manual pages can be produced in HTML, Tex or nroff cat-format with the accompanying tool `mma`. The circuits themselves can be printed from within *Neo* as encapsulated postscript files.

2 The User Interface

2.1 The Folder Window

The available units are grouped into smaller subsets ("folders") comprising units that are functionally related. The left window (the "folder window") shows the contents of the currently selected folder as icons. Selecting the '?' mark on those icons that show it brings up a viewer with a for the corresponding NST-unit. Selecting the 'e' on those icons that show it loads an illustrating the use of the corresponding NST-unit.

The currently selected folder is indicated in the folder window title and by a highlighted button in the left menue bar. By activating a different button in the menue bar, the corresponding different folder may be selected and its contents will replace the current display in the folder window. Menu bar buttons with capital inscriptions in brackets or followed by a colon ':' show the title of a group of folders. The group can be "collapsed" into a single title button, or expanded from a "collapsed button" ("collapsed" buttons have their inscription in brackets).

2.2 The Circuit Window

The adjacent, central window (the "circuit window") is the window where the construction of circuit diagrams takes place. It shows the current circuit or a subcircuit thereof. To add a new unit to the circuit diagram in this window, simply drag its icon from the folder window into the circuit window. When the icon is released in the circuit window, one or two query windows will pop up that ask for parameter values for the new unit. When the entered parameter values were legal, the unit will be created and will appear as a new icon in the circuit window, otherwise an error message will appear and the current circuit will remain unchanged.

The *Circuit*-button in the menue bar allows you to switch between up to five circuits that can be held simultaneously in *Neo*'s memory. Switching to a different circuit deactivates the current circuit (you can return to it any time later, with its current state fully restored) and activates the new circuit (i.e., all subsequent commands apply to the new circuit, which will appear in the circuit window).

The title bar of the circuit window shows the name of the currently displayed (sub)circuit, plus some current settings (the number of local variables that are "visible" for the displayed circuit, and in brackets the number of steps for the `Run`command, and whether adaptation is activated (`"+a"`) or not, cf. below). By pressing the 'A' key, the display can be toggled to a second mode in which the title bar identifies the shown circuit, and the icons show their NST unit type and number ids.

You can select all or a (sub)set of icons and/or annotations in the circuit window (e.g., for operations such as copying, deleting or moving them) by delineating with the mouse (left mouse button down) a rectangular area within the circuit window. The selected icons and/or annotations will become highlighted. The current selection can also be modified by toggling the select status of single items with the left mouse button, while simultaneously holding the left shift-key down.

If the windows take up too much space, choose a smaller font by calling *neo* with the `-Font i` option. `i` can be one of `0,1,2` or `3`, with `0` selecting the smallest, `3` the largest font. There are further command line options that are displayed by issueing `neo -h` (short listing) or `neo -H` (full listing with current values) on the command line.

3 Connectors

The icon of a newly created unit differs from the icon in the folder window by the appearance of one or several colored "connectors". Each connector on the left (right) side of the unit indicates an input (output) field, with field number zero for the topmost connector of the icon. The color indicates the field type: yellow indicates a field of one or several "scalar" pins (each scalar pin holds a single float value), red indicates a field of one or several float vector pins (a float vector pin holds an array of float values; it takes up less space as an equivalent number of scalar float pin, but the individual array elements are only accessible at the C programming level)[†3]. Green indicates a field with of integer vector pins (analogous to a float vector pins, but holding values of typ `short int`), and blue indicates a field of byte vector pins. Each such pin holds a vector of byte (i.e. `unsigned char`) values ("byte vector pin"). Any of the three vector type pins may be surrounded by a little brown square. This indicates that the vector is of "dynamic dimension" and makes its number of elements fit as required. Finally, there are some units that have fields with magenta "text pins" . Such pins hold a null terminated, variable length string (*Neo* cares about any issues of changing string sizes etc. automatically). A field can also have pins of different types. In this case, its connector is drawn white.

When the mouse pointer is on a connector, the middle mouse button can toggle the display between three modes: (i) display nothing, (ii) display connector data and (iii) display connector info. The mode setting is sticky and remains in force also when visiting other connectors (even between *Neo* sessions), until it is reset again.

3.1 Connecting Units into Circuits

Several units can be connected into a *dataflow diagram* by drawing wires between their connectors (move the mouse pointer to the first connector, then to the second connector while holding the middle mouse button down; release the mouse button). The new connection is indicated by a highlighted wire (the shape of the wire can also be adjusted by the mouse). For fields with more than one pin, the single wire may represent a whole bundle of pin to pin connections. The highlighting

[3]a field with a single float vector pin of `n` float values is equivalent (and fully backward compatible) with a "packed field" of earlier *NST*-versions. Often, specification of a negative dimension `-n` for a field is interpreted as a shorthand to make a field with a single float vector pin of `n` float values

indicates that the wire is “selected” and that its detailed pin-to-pin connections are displayed in the adjacent, narrow “wire edit window” and can be changed there (a different wire may be selected for editing with the mouse pointer).

Drawing a wire is non-symmetric: if the source connector is visited first, *Neo* will try to insert as many “wire fibers” as possible, possibly substituting already existing connections to the destination connector. When instead the destination connector is visited first, *Neo* will not insert any “fibers” automatically; this must then be done with the wire edit window.

The wire edit window shows the individual pins of the source and destination fields of the selected wire as the left and right columns of the window. It also allows to view the types of the pins of that make up the white connectors. The detailed pin-to-pin connections are shown as black lines (the connections of a wire between two packed fields are shown as a single line between two single pins). Pins filled with a little rectangle are destination pins that receive already a connection from a different wire.

There are some restrictions on which pin/field types may be connected. Within the vector type (i.e., float, int and byte vectors (red,green,blue)), pins of equal dimension may always be connected (*NST* inserts automatic type conversion calls between connected pins of different types [4] Pins of text type may only be connected to pins of the same type (without any dimension constraints for text pins). Scalar pins cannot directly be connected to vector pins. An exception occurs if a scalar field (i.e., yellow connector) has the same number of pins as the dimension of a vector pin. In this case, *Neo* allows to make a connection between both, treating the scalar field as if it were a vector of the appropriate dimension. In all other cases, transferring data between scalar and vector pins requires the insertion of a suitably dimensioned splitter or merger unit between the two pins.

To delete a wire, either select it and remove all its connections in the wire edit window, then return into the circuits window, as a result of which the wire will disappear. Or, more simply, select it while holding the D-Key down. This deletes the wire in one step.

In addition to its connectors, each icon displays one small area at its bottom, and four larger areas (labelled `-` or `=`, `m`, `x` and `c`) at its top. The small area at the bottom is for deleting the unit that is associated with that icon (it also indicates whether the icon uses variables or not, cf. below). The four areas at the top will be explained later.

3.2 Execution of Circuits

The `Step`-command executes and adapts all visible icons (and their subicons recursively) once, the `Run`-command as many times as specified with the prefs-command (this allows also to choose certain settings, such as to disable adaptation for the run-command). Since only the visible icons are executed, opening of any icon on the screen will have the effect of restricting execution/adaptation to the subcircuit that is contained in the opened icon, since this is then displayed in the circuit window.

When the `Run`command is used, pressing the `Esc`-key allows to interrupt execution earlier (this still has to wait for the current step, which may involve long loops, to be finished; to allow also interruptions within loops, activate the corresponding option in the `Prefs`submenu. Note, however, that this will slow things somewhat down). Execution order is determined by the *horizontal positioning* of the icons that make up the circuit. Execution order of units is from left to right (bottom

[4] Note that conversion from float to int or byte truncates, and conversion to byte is modulo 256. The latter may give rise to dark patches in image data.

to top, if two units have precisely the same horizontal offset). Therefore, execution order can be easily changed simply by suitable rearrangement of the icon's horizontal positions. An execution step may be followed by an adaptation step. Adaptation adaptation order is the opposite, i.e., from right to left. The number of execution steps and whether each execution step is followed by an adaptation step, can be set with the `Prefs`-command.

As an extremely simple example, load the `xydemo`-example. It just consists of a wave generator unit connected to an `xyplot`-unit. Each click at the Step-Button executes the wave generator and the `xyplot`-unit once and the (x, y)-pair produced by the `wavegen`erator is displayed in the window of the `xyplot`-unit. Note that the (x, y)-pairs produced by the `wavegen`erator are released at two fields with one pin each, while the `xyplot`-unit expects its input in one field of two pins. Use this example for trying out various features, such as initializing, resetting, modifying or copying units. You also can delete any part of the circuit and later reload it.

During an experiment, it is frequently desireable to reconfigure some or all icons by issuing certain control calls (using the `ctrl_unit`-routine for them. To this end, the "ctrl"-command is provided. With this command, it is possible to issue the standard *NST*-control calls for all units, the units currently visible in the circuits window, or the selected unit(s) in the control window. Another possibility to change the setting(s) of single units is the parametrization window of the respective units (see Sec. "Viewing and Modifying Parameters of Units").

3.3 Usage of the Keyboard

For a listing of the *Neo* editor keys, press the F1 key.

A few commands use the keyboard: holding the SHIFT-key down while using the mouse allows to select (will be highlighted) several units for Copy's and Move's. Holding the D-Key down allows to delete a wire by clicking at it. Pressing the ESC-key allows to interrupt the execution of a circuit. There are also three occasions when the right mouse button is used: the first is to move from the current circuit nesting level to the next outer level. The second is to escape from an interactive `view3d`-window. The third is to open the variable window of the `ROOT`-circuit or to create a variable window for a compiled unit (see Sec. 7).

3.4 Encapsulation of Circuits

If a circuit diagram becomes too complex, the entire diagram or a subcircuit of it may be repackaged into a new icon (in *NST* , this corresponds to make the units of the circuit subunits of a new host). To this end, you can use the Move-command in conjunction with a new `void_unit`: first create a `void_unit` or a `container_unit` from the standard folder. Both types of unit can accommodate the (sub)circuit that is to be repackaged and should therefore be created with a suitable set of connectors for communication with its future contents (however, connectors may be later added or removed, as need arises†[5] Then move the (sub)circuit that is to be repackaged into the newly created `void/container_unit` by taking the following steps: 1. select all icons that belong to the (sub)circuit to be moved (selection of icons requires to delineate a rectangle with the mouse and the left mouse button, or to toggle the select state of single icons with the left mouse button while holding the SHIFT-key down). 2. toggle the Move-Button. As a result, you will be requested to "open" the destination icon. This requires to 3. click at the =-area in

[5] actually, the `container_unit` is also a `void_unit`, but with a better interface window for making later changes

the top row of the void/container_unit icon. This will “open” its icon, i.e., make it the new host the content of which is displayed by the circuits window (initially, the circuits window displays the content of the host ROOT). As a result, the circuit window will go blank (the newly created void/container_unit does not yet contain any subunits). 4. toggling the move-button again will make the subcircuit appear (with all moved objects selected) in the circuit window and finishes the entire operation. All that remains is to set up the necessary connections between the moved subcircuit and the connectors of its new host (these are visible at the left and right borders of the circuits window). Pressing the right mouse button will then return to the outside (previous) circuit level (to get back into the void_unit, simply click =-area of its icon again). After encapsulation, the entire circuit can be manipulated as a single item. In particular, multiple copies of it can be made with the “copy”-command (the copying is a fully recursive “deep copy”, i.e., it includes all subunits to arbitrary nesting depths of the copied unit, and all current parameter and pin value settings), and it can be saved and retrieved as a single item to/from disk. In this way, one can easily build up collections of specialized circuits that can be held as files and serve as customized building blocks for the modular and hierarchical construction of larger circuits (see also Section “Loadable Folders”).

The simple encapsulation of a subcircuit by a container allows only the use of the subcircuit as a whole. However, *NST* 's use_named-unit allows to implement a more structured access to the resources within a container: if the container has been given a name (say, e.g., box) with the *Name*-command, any of its subunits that also carries a name (say, e.g., dothis) can be invoked by the composite name box:dothis from any use_named unit that shares with box the same container (i.e., is itself at the same level as box or is a subunit of a unit that is at the same level with box[6]). A similar operation is provided by the use_method-unit, however, here the interface need not be specified “by hand” but instead comes automatic, and there are several “data transport modes” that can be set.

In this way, several related units (say, for initializing, training, testing, using, inspecting a particular neural net) can be encapsulated as a single module; still their services can be used separately from outside. Such collections of related methods are the NST counterpart of a class and their containers are thus referred to as *class containers*. *Class containers* can also be created (i) interactively in the form of prog_units (when these contain *public function* that then appear as method subunits), or (ii) be obtained by suitably wrapping C or C++ classes as NST units (cf. the manual mkudoc for a detailed description).

For containers with such collections of related units, it is usually no longer desired that execution of the container executes all its subunits. This can be achieved by inserting a no_op or return_unit unit, which protects its operands from execution/adaptation [but not from control calls!].

Access across more than one level of encapsulation is also possible; e.g, if subunit dothis itself contains another named subunit part1). this unit can be called with box:dothis:part1 from the outermost level, and with dothis:part1 from within box.

The same scheme allows to embed var_def units within named containers and to reference them from var_set/get/info-units from outside. However, the var units, although using the same visibility rules, use a separate name space (i.e., the var_def units have a separate access scheme and don’t respond to the use_named unit).

[6] of course, if the named_unit is within box itself, it can invoke dothis directly by its name, not needing the “path-prefix” box: in this case

3.5 Viewing and Modifying Parameters of Units

The parameters of a *NST*-unit fall into two categories:

1. parameters that must be specified when the unit is created. The values of these parameters will remain fixed during the entire lifetime of the created unit. In *neo* , these parameters are requested with the first dialog window that appears when a new unit is created for the circuits window. Later, the values of these parameters can be displayed by clicking on the `m`-area in the top row of an icon. If they are modified, the entire unit is remade.

2. parameters that may be changed during the lifetime of a unit without remaking it. Typically, such parameters include values at the control field of a unit, or learning rates etc. In *neo* , these parameters are requested with the second dialog window that appears when a new unit is created for the circuits window. Later, these values can be displayed and/or modified by clicking on the =-area in the top row of an icon.

However, frequently one would like also to change parameter values from the first group. This is always possible by deleting the unit and creating a properly parametrized new instance; however, this will also disrupt all connections that were set up with the respective unit so far.

For all those reparametrizations that do not change the number of pins and connected fields of a unit (there is an exception to this, cf. below), the "modify"- command provides a means to make the desired changes without having to delete the unit and thereby lose its current connections (in fact, the unit is internally deleted and replaced by a reparametrized copy, but all wires are properly rebuilt, since the interface of the newly created unit is the same). To invoke the modify-command, select the `m`-mark on the unit's icon. This will bring up the dialog window showing the current creation parameters that can then be changed. If the changes are accepted, the unit will be modified accordingly. If the changes cannot be made (this is always the case when the interface for the new values would differ) the unit will remain unchanged and an error message will be issued.

An exception to this is the `container` unit that is provided to accomodate subcircuits. Here, the interface can be altered. Newly inserted fields or pins simply move existing connectors or pins to shifted places, carrying their wires with them. To remove a connector, however, first requires to cut off all its connections (use the wire-edit window for this).

Sometimes one may wish to modify a unit under program control. To this end, *Neo* provides the `remake_unit`, which acts like an "operator" that modifies its successor operand (see the man page for details).

Additionally, there are values at the connectors of a unit. To display the number of pins of a connector and (some of) their values, move the mouse pointer onto the pin and hold the middle button down. This will display information about the pins and (some of) their values in the title bar of the circuit window. If more information is desired, press the middle mouse button one or two times (while the pointer is still on the connector). This will toggle *Neo* 's connector display mode to show additional information each time the pointer touches a connector (as already explained above). Finally, if the connector is an output field, you may consider to connect a `print_vec` (for the display of a vector), `print_mat` (for the display of a matrix) or `print_str`-unit (for the display of a text string) to it. If the field is an input field and it is desired to set a number of pin values, you may consider to connect a suitably parametrized `const_vec`-unit.

3.6 Loading/Saving/Printing of Circuits

When circuit is finished, it may be saved to file (save command and save_as command in the menue bar) and/or executed. To execute a single unit, click its `x`-field.

The command “load” will load the circuit from a file into the circuit window (any previous circuit in the circuit window will become deactivated, if less than five circuits are are already in memory; otherwise *neo* will ask whether to delete the current circuit). To change to a different directory, enter the new directory name, ended by a `'/'` instead of the file name and type OK. This will replace the current selection with the new directory contents. To re-filter the current listing (after a change to the filter pattern), select the topmost line of the directory listing (this line should display the name of the current directory). To go one directory up, select the `..` entry.

With `neo -i file` the file can be specified on the command line. The full functionality of the loaded circuit will only be available, if *Neo* “knows” all its units. If you try to load a circuit that contains units that are not compiled into your version of *Neo* , *Neo* will complain and replace the unknown units by “dummy” units. These still allow normal viewing and editing of the loaded circuit to take place, but when executing, the dummy units will perform no operation. However, the edited circuit may be saved again and can be used with its full functionality under a sufficiently extended version of *Neo* . This feature allows, e.g., to use circuits that contain 3D-graphics even with a 2D-version of *Neo* and to even make modifications that will be preserved when later run on a more extensive version of *Neo* .

The command “save” saves the circuit in the circuit window to a file.

Regarding the format of the saved circuit, two different cases must be distinguished: 1. If the circuit window is the top level window, the saved circuit will be saved as “open circuit”, i.e., when reloaded, it will reappear as a circuit without any enclosing “box”. 2. If the circuit window is not the top level window (i.e., it displays the content of some subunit), the circuit will be saved as “encapsulated” by its box. When such a circuit is reloaded, it will not replace the current circuit (as an open circuit would) but instead be added into the current circuit, appearing as an additional single icon that contains the saved circuit and that has the same interface as the box that contained the circuit when it was saved.

You can save the circuit information and the data in the units into a single file or you can save the data into a separate file. Which data are written/read can be controlled by a file suffix. Usually, one should specify file names without suffixes. In this case, *neo* will read or write one or two files with default suffixes `.NST` and `.nst`. The first file (`xxx.NST`) contains the information about the structure of the circuit (“circuit file”, specifying the types of units, their positioning on the screen, and their wiring). The remaining data, namely parameters that pertain to the units of the circuit, can be either included (appended) in the `.NST` file, or, alternatively, they can be written to a separate, second file (`xxx.nst`, “data file”). In both cases, the write format for these data is the same as that used by the *NST*-`load/save_unit` routines. If one specifies a suffix that contains at least one lower case letter, *neo* will generate an upper-case suffix for the circuit file, and use the original suffix for the data file (however, this is not recommended practice and this feature may be withdrawn in a future version). If an all-upper case suffix is specified, writing or reading of the data file will be suppressed, and only a circuit file will be accessed.

To load or save the data of a single unit, click its `c`-area . A menu pops up and you can select the *Load* or *Save* operation from it. NOTE: the load command is somewhat dangerous: if the specified file does not match with the unit, of *neo* may hang or CRASH (after a crash you frequently can recover with `neo -r`, but in unfortunate cases this may fail). In case of doubt therefore better save your data before.

Finally, the “print” command will write an encapsulated postscript file for the layout of the circuit. This may be useful, e.g., for preparing written documentation of experiments.

4 Generation of Source Code

The "MakeSource"-command will prompt for two file names, the name `Name` for the output file which shall contain the generated source code, and a name for a template file (see below). If no template file is specified, a default template will be used.

neo will the create source code for the circuit that is displayed in the Circuits-window. If this is the ROOT-window, a main program plus procedure definitions for all subunits is written. Otherwise, the main-portion is absent and the file is assumed to be compiled into an object file. In addition, a corresponding make file `Name.mak` is created as well (using a template file .NEO_TEMPLATE.mak).

Each predefined *NST*-unit is translated into a call of its suitably parametrized constructor function, followed (if necessary) by initialization statements.

Any other unit is necessarily of type container. Normally, it will be translated into a call of an equivalent *NST*-constructor routine for which the necessary code is also generated by *Neo* . Only if it is empty and neither has any input nor any output fields, *Neo* will generate a call for an ordinary C-routine (this convention allows to indicate *Neo* the use of C-subroutines from other object files by simply inserting suitably named `void_units`).

In both cases, for each called subroutine, a declaration is written in addition. For a container unit that is labelled `my_special`, *Neo* will generate the declaration

```
extern unitptr MySpecial([...arguments...,] unitptr uHost);
```

In the second case (that of an empty container unit), *Neo* will instead generate the declaration

```
extern int MySpecial([...arguments...,]);
```

I.e., in both cases the label of the container unit provides the name for the created routine, and underscores and |-characters are removed with the next letter changed to uppercase.

The part `[...arguments...]` will only be present, if the container unit has any variables defined (the definition of variables uses the `Define`-command and is explained in Sec. 12). The defined variables can be of type `int`, `float` or `char*` and will appear as identifiers `iArgk`, `fArgk` and `tArgk` in the parameter list `..arguments..` of the created subroutine call/declaration/definition.

To have some control on the form of the resulting program, one can specify a template file. The generated program will then be a copy of the content in this template file, but with the following substitutions made for the special tokens `$t, $h, $p, $n, ${ and $}`:

`$t` will be replaced by a number of comment lines with title information

`$a` If source code is generated with the .Preprocessor option on, `$a` will be replaced by a number of lines suitable as the body of a .ARGUMENTS - section for use by `mmp`. Otherwise, `$a` will be replaced by the empty string. Note that the text occuring for `$a` is expected to be within a comment (to allow more control, the delimiters of this comment are not written by .neo).

`$h` will be replaced by a header section consisting of a declaration for each procedure written by *neo* .

`${` if the circuit for which the code is generated is the top level circuit in the ROOT-window, this is replaced by the beginning part of a `main()`-program (containing declarations, creation of units and their connections). Otherwise, this token will be replaced by the empty string.

`$}` if the circuit for which the code is generated is the top level circuit in the ROOT-window, this is replaced by the ending part of a `main()`-program. The beginning and ending parts are defined such that between the `${...$}`-pair one can place arbitrary C-code that uses the units built in the previous part of the main-program.

`$p` will be replaced by the code for all non-main procedures created by *neo* .

`$u` will be replaced by the name of the top-level unit created by *neo* .

`$n` will be replaced by the base name of the created source file.

Thus, the simplest template file might look like this:

```
$h
${ exec_unit($u); $}
```

If no explicit template file is specified, *neo* will use a default file (`/vol/nst/neosrc/NEO_SRCTEMPLATE.c`) that can serve as an example of a somewhat more sophisticated template file. With it, the generated program has the following predefined command line features:

`-n number` specifies the number of iteration steps for which the entire circuit shall be executed (0=indefinitely). The default is one iteration.

`-i inputfile` if specified, this file is expected to hold data for initializing the circuit (i.e., a `load_unit`-call will try to access the data in this file).

`-o outputfile` if specified, the data in the circuit will be saved to this file (by issuing a `save_unit`-call for the entire circuit) when the program terminates.

`-noadapt` omit adaptation

`-t` switch trace on.

5 Control Flow within Circuits

Frequently, it is desired to execute a subcircuit repeatedly, before further units are executed. This can be achieved with the `for_loop`, the `multi_for_loop` or with the `iterator`-unit (the use of the latter is now deprecated, since many of its functions are better implemented using a `prog_unit`, which, among other things, can also be turned into an iterator type unit). While the `for_op` units offer the flexibility to dynamically change the number of loop iterations at an input pin, the latter offers some handy options for saving iteration results into one or several vectors, and for sequentially loading elements from one or several vector inputs. A single execution call of an `iterator` or `for_loop` unit has the effect to iterate execution of its successors a fixed number of times. The number of successors that are affected by the iteration is specified when the `iterator`-unit

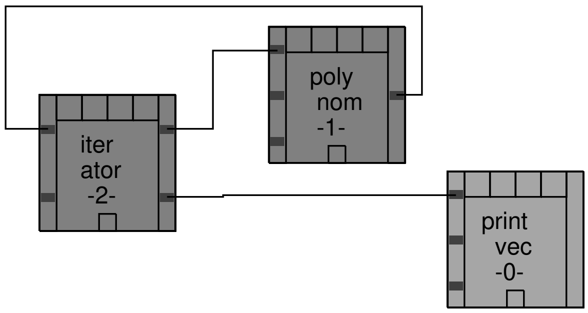


Figure 1: Iterating the polynomial $y = x^2$ to make a vector of square numbers.

is created. The `iterator` unit can be viewed as an *operator* that acts on a number of `operands`. *Neo* indicates operators by surrounding them with a red frame while delineating their *scope* (i.e., the range of affected operand units) with a dotted rectangle whose vertical borders can be shifted with the mouse in order to include new or to exclude old operands from the current scope[7].

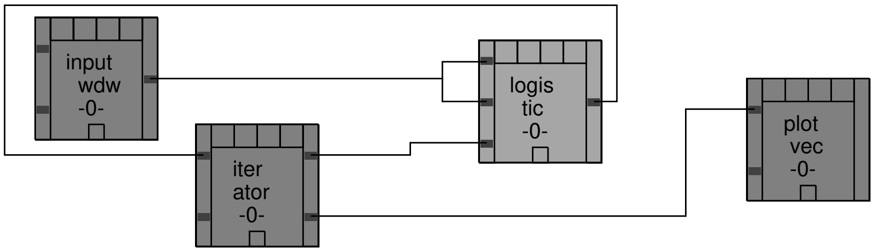
Unlike the `for_loop`, the `iterator`-unit offers the additional option to accumulate results of these iterations in its *saving fields*. Likewise, it can offer the iterated units at each iteration a "time slice" of the values at its *feeding fields*. Whether an `iterator`-unit has saving- or feeding fields is also determined when it is created. For more details, see the manual page[8]s

Fig. 1 uses the `iterator`-unit to prepare a 10-dimensional vector of a sequence of function values $(f(0), f(1), \ldots f(9))$ (the file for this example has the name `sqlist`). This is achieved by iterating the function $f(x)$ (in the present example, $f(x) = x^2$ is computed by the `polynomial`-unit) for 10 times. To cycle through the required argument values $x = 0, 1, \ldots 9$, the input of the `polynomial`-unit is connected to `out_0` of the `iterator`-unit. This output holds for each iteration $i = 0, 1, \ldots$ of the iterat*ed* unit the value `i·CTL_in[0]`, where `CTL_in[0]` is the value at the single pin of the `iterator` unit's control input field (default=1). Each iteration's result value $f(i)$ is fed to `in_0` of the `iterator`-unit. The `iterator`-unit assembles the resulting sequence of values at this field into a single 10-dimensional vector that is then ready at its saving field `out_1` when the last iteration has been completed. This mechanism illustrates how to make a sequence of intermediate results from an iteration available in a single vector for the use by subsequent units.

Another frequently useful operation is the switching of connection between units. This can be achieved with the `switch_input` and the `switch_output`-units (the `switch_str`-unit offers the same capability for text strings). The first one allows to switch the source of a common output among a fixed number of inputs, the second allows to switch a fixed input to one of a fixed number of outputs. 2 shows an example (the file for this example has the name `logistic`). It is very similar to the previous example: a function is iterated 10 times and the resulting sequence is displayed, this time graphically using a `plot_vec`-unit. Now, the function is computed by a subcircuit, hidden in the icon labeled `logistic`. If you open this icon, you will find a subcircuit of three unit that computes the logistic function $x(t+1) = r * x(t) * (1 - x(t))$ (the `polynomial`-unit does the

[7]some operators, such as the `switch_op` and the `array_of`-unit, have a scope that cannot be changed in this fashion.

[8]these conveniences require that the number or iteration steps of the `iterator` is specified when the `iterator` is created and is kept fixed thereafter. A simpler `for_loop`-unit that allows to change the number of iterations dynamically is likely to be added soon.



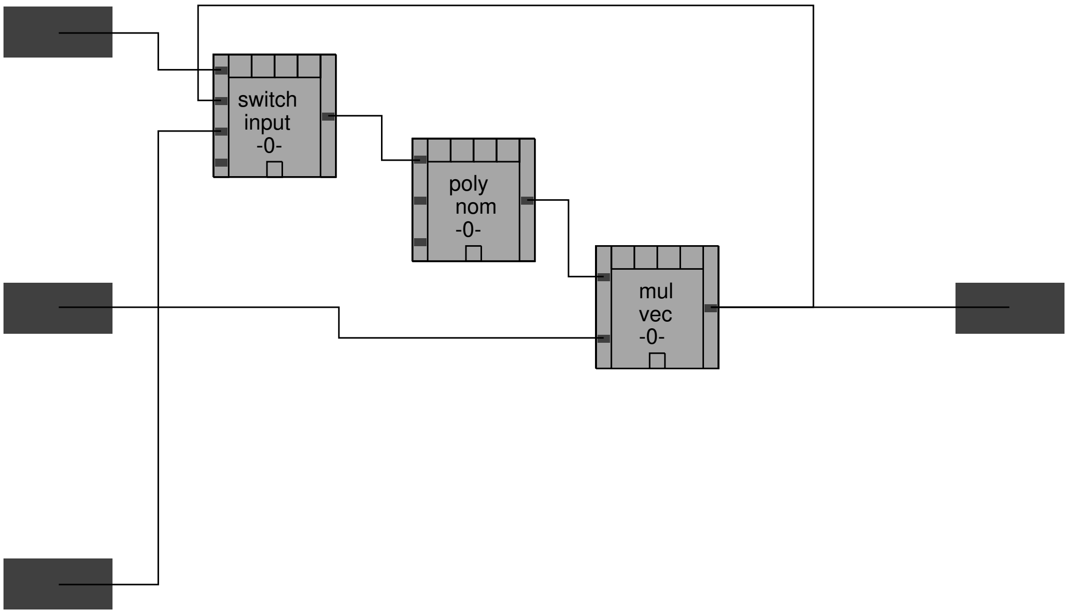


Figure 2: Circuit for computing and displaying the first 10 iterates of the logistic map. The upper diagram shows the top level circuit, the lower diagram the subcircuit in the `logistic`-unit. Note the usage of a `switch_input`-unit to create a feedback loop.

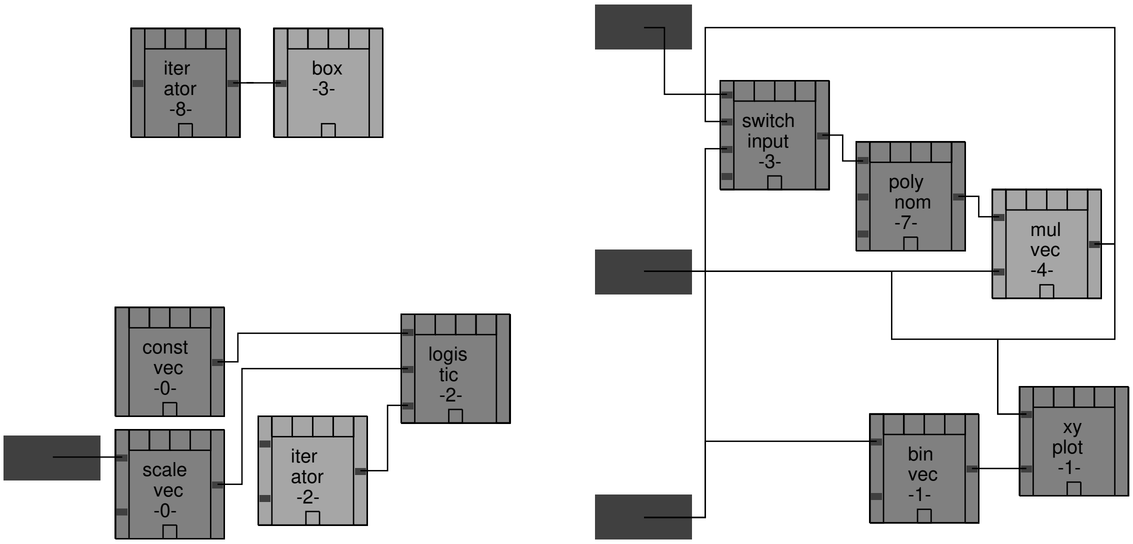


Figure 3: Circuits for generating the Feigenbaum tree. The top level circuit *(upper left)* just iterates the subcircuit in the `box`-unit *(lower left)*. The innermost subcircuit is the already familiar `logistic`-feedback loop, but with two units added for plotting

$x(t) * (1 - x(t))$-part and the `mul_vec`-unit premultiplies the factor r (taken from the second input of the enclosing container unit). The special thing here is the presence of a *feedback*-connection that feeds the result $x(t)$ of the previous computation back to the input as a new starting value for the next computation step. However, for the first step, the starting value shall be taken from outside (via the first input of the enclosing container unit). This is achieved with the `switch_input`-unit: its selector pin (at its next-to-last field) receives the current iteration count (via the third input terminal of the enclosing container unit). At iteration 0, this will route the value at `in_0` of the `switch_input`-unit to its output from where it is used as starting value for the further computations. At the next iteration $i = 1$, and now the value at `in_1` of the `switch_input`-unit is switched through. However, this is precisely the result from the previous iteration, so the desired effect is achieved. At all further iterations, $i > 1$, i.e., beyond the highest input channel of the `switch_input`-unit. This makes `switch_input`-unit select its highest channel, i.e., for all the remaining iterations the feedback loop also works as desired.

If you run this example, be sure that you are on the top level (otherwise, only the displayed subcircuit will be executed, with correspondingly boring results). The input window allows to continuously modify the values of $x(0)$ and the gain factor r. To return to the command level, press the `ESC`-key (NOTE: make sure that the mouse cursor is outside the parameter input window. Otherwise, *neo* doesn't notice the `ESC`-key, since it waits for parameter input).

A last example combines the previous ingredients to produce the Feigenbaum-tree (the file for this example has the name `feigenbaum`). The top level circuit of this example consists just of an iterator unit (this time in forward mode) and a container unit `box` that is iterated 200 times. Each iteration computes 100 points with a fixed r-value for the Feigenbaum tree. This is accomplished by the subcircuit within `box`. First it scales the current iteration count at its input linearly to the range of [2.8,4] and uses this as r-parameter for the subcircuit that iterates the logistic map for a fixed number of times (here 200). A constant starting value of $x(0) = 0.5$ for each iteration sequence is provided by the const_vec-unit.

The logistic-subcircuit itself has also been extended. It now contains a new xyplot-unit to plot each computed $(r, x(t))$-pair. However, an additional bin_vec-unit disables plotting for those pairs which have not yet sufficiently converged. This is achieved by requiring that $t > 100$: the bin_vec-unit binarizes t, yielding 0 below, and 1 above the required threshold. The result is fed to the control input of the xyplot-unit (using the property that a control value of 0 switches off all operations of a unit).

Among further useful units not discussed here is the switch_op. This unit does a similar thing as the familiar switch statement in C (for the switch_op, numeric or string case labels are possible). Likewise, return_unit returns from a subcircuit whenever it is executed while its single input pin has a non-zero value. Another frequently useful unit is once_only, which executes its operands only the first time it is called and skips them thereafter, unit it is reset with a NST_INIT or a NST_RESET.

Note that in this somewhat more complicated circuit several important constants (iteration counts, range limits etc.) are scattered throughout the various subcircuits. Moreover, some of these constants are related, such as, e.g., the scaling limits of the scale_vec-unit and the x-range limits of the xyplot-unit. Later we shall see how the assignment of values to such parameters can be coordinated and centralized by the use of *variables*. Note also that the above is a “pedagogical example” to illustration various elements of *Neo* . Using the possibilities that come with the (more recent) prog_unit would allow many simplifications.

6 Importing Data

In the preceding examples, all data were self-generated. However, frequently external data from files have to be used. This section gives a brief overview of the facilities to import data from files into *neo* -circuits. Before using any of the described units, one should first check whether the task can be solved with the prog_unit, using e.g., a function from the file #import library (these functions are modelled after the stdio-C-functions).

To access numeric data from a text file, one approach is to use the scanf_unit which is modeled after the familiar C-function and which allows arbitrary reads under the control of a format string. The NST-scanf-unit understands a subset of the conversion specifiers of the scanf-function from the standard C-library (included are the cdefs and the scanset []-specifiers, as well as assignment supression with an asterisk)[9]. Each specified assignment produces a pin of corresponding type in output field 0 of the unit. In addition, the scanf_unit recognizes the non-standard format specifier %# that skips a comment portion of the file (the comment portion is empty, if the next nonwhitespace character is not a '#'. Otherwise, it starts at the '#' and ends at the first line that has not a '#' as its first non-whitespace character). Another important property is that different scanf_units that are associated with the same file name can access the file also with a shared file pointer. As a consequence, it is possible to, e.g., use two units with different format strings to first read, e.g., a header portion of the file, and then to make repeated reads for the file body.

[9] in addition, there are several non-standard conversion specifiers for reading vectors or image data as single tokens, cf. the manual page of the scanf-unit.

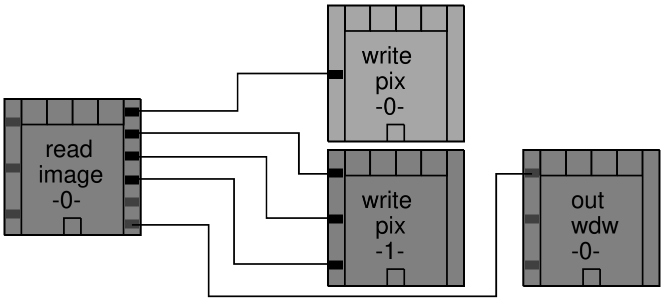


Figure 4: Reading and displaying image data

Demo4-1 provides an example how to use the scanf_unit to read data from a file.

If the files adhere to some more regular structure, you can use the read_file-unit. It offers the capability to skip a header portion of the file and to read for each exec_unit-call one line of the remaining file content. Alternatively, an initial portion or the entire file content can be read into a buffer and then be accessed at higher speed in either sequential or random order. If in sequential mode the end of file has been reached, reading continues with the first record. A simple example is contained in file readfile.

To access image data from pnm-files, you can use the read_image-unit. It is intended to read a sequence of image data files with names basename#.ext, where # denotes an integer index, and the remaining portions basename and ext are shared by all image files read. The index #-index may be automatically incremented or decremented between successive exec_unit-calls of the read_image-unit. The read image data are made available at one (plus three in case of color) packed output field(s) of the read_image-unit. The circuit of 4 shows a simple example, using two write_pix-units to display the mono and the color channels of the images read, and a output_window to display the current image number #. Since the reset limit for the index # is set to 10, reading continues with image 0, after the 10th image was read.

Often, it is desired to read images together with numeric data. This is possible with the read_data-unit. The file format expected by a read_data-unit allows to mix on each data lines numeric tokens with file names. The numeric tokens are made available at normal output fields, the file names are interpreted as image data files the contents of which is read and made available at a packed output field that is created for each such item. A format string determines, which items are read and which are ignored, and various possibilities exist to specify path prefixes for the image file names (see man page for details).

The circuit of 5 shows a simple example. Here, the read_data-unit reads cyclically a file with an image followed by a number of marker positions on each line. For each pair of marker coordinates, there is a mouse_xy-unit that is initialized such that it draws the marker in the image displayed by the write_pix-unit (the file for this example has the name readdata).

A last data source are the various frame grabbers. Fig.4 shows an example where the androx_grab-unit is used to grab a continuous sequence of images from one of the androx boards. NOTE: before starting this circuit, make sure that the server servand is running on host druide, otherwise the program will crash.

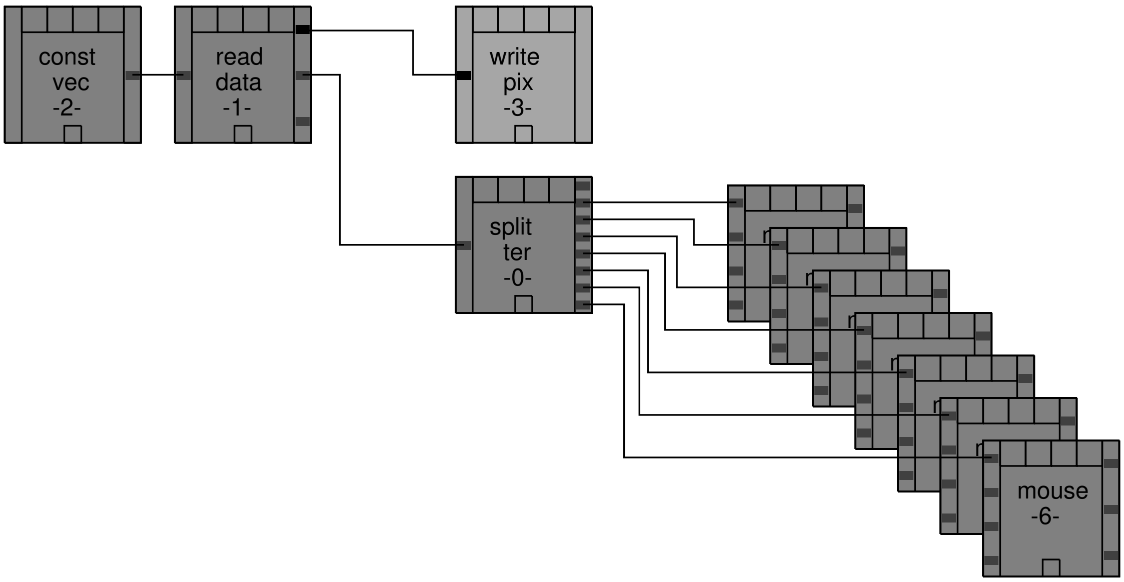


Figure 5: Reading and displaying image data together with marker information

7 Exporting Data

Usually, one would like to write results back to disk. The most flexible way to do this uses the `printf_unit` or the `fprintf()` `prog_unit` function (must be imported from the `file` library). Both are modeled after the familiar C-function and allow printing of a number of result values under the control of a format string, either to `stdout`, to a file or into another text string. `Demo4-1` provides a simple example of how to use the `printf_unit`.

If the results are the weight values of adaptive units, found by some training procedure, their storage to file can be achieved with the `save`-command, either for all units, or for individual units selectively. The save command can also be applied from within a circuit, using the `fileio_op` operator unit.

Otherwise, if vector or image data produced as output of some unit(s) are to be written to file, you can use the `write_file`, `write_image` and `write_data`-units. The first appends on each `exec_unit`-call one line with the current values at its input pins to its output file. The second writes on each `exec_call` a separate `ppm`- or `pgm`-image data file. The written image data files have names `basename#.ext`, where `#` is a running index that is automatically incremented after each call of the `write_image`-unit (for details, see the man page).

Likewise, there is a corresponding `write_data`-unit, which allows to write data records of arbitrary (with fixed structure) sequences of images and numeric vectors. The written file is in a format that can later be read by the `read_data`-unit. It contains the vector data directly, but the image data only by their file names. Whether the associated image data files are also newly written or not (in case they already exist on disk) can be selected by the user.

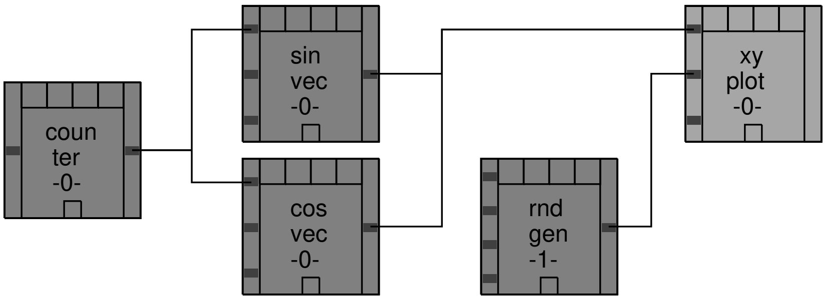


Figure 6: Using the xy_plot-unit to plotting a lissajous graph superimposed with a cloud of random dots.

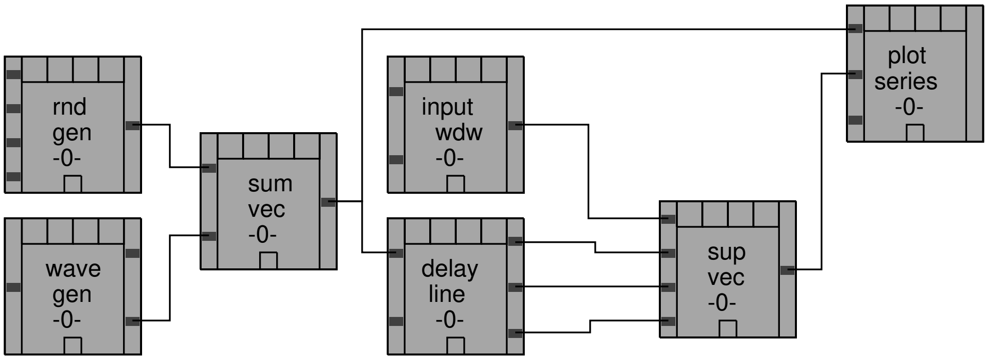


Figure 7: Autoregressive filter for a noisy time series.

8 2D-Visualization

We already saw examples of the print_vec, write_pix, plot_vec and xyplot-units for the display of data vectors in numeric, pixel and plotted form. The plot_vec-unit is actually a conveniently specialized version of the more general xyplot-unit which allows to create 2D-data plots in which single points, histograms, connected graphs and error bars and crosses may be mixed. 6 shows an example in which the xyplot-unit is used to draw a lissajous pattern superimposed on a cloud of random dots with gaussian distribution.

To display continuous time series, you can use the plot_series-unit. 7 shows as an example a simple linear filter for a time series that is produced by a wave_gen-unit. The original time series is artificially degraded by adding some small amount of random noise (rnd_gen-unit). A delay_unit creates three time delayed versions of this signal. These are then added with filter coefficients that can be adjusted via the sliders in the input window. The original (red) and the derived (cyan) signal are then continuously displayed in the same graph window.

Finally, to display markers in pixel images, you can use the mouse_xy-unit (this is now depre-

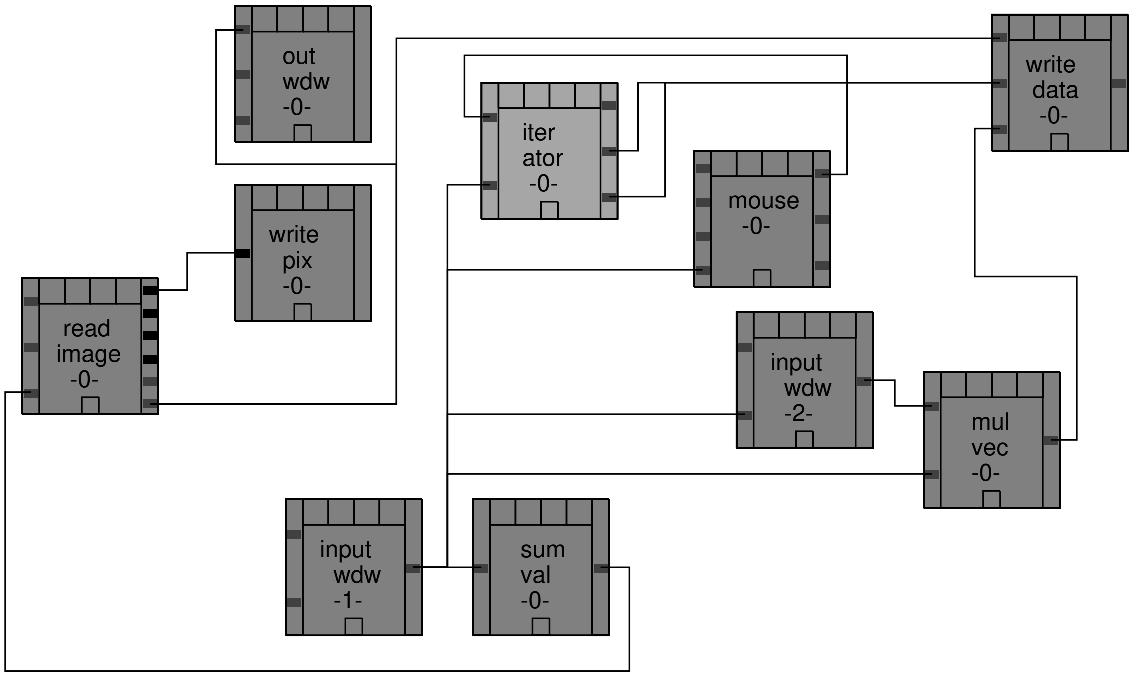


Figure 8: A simple "marker editor" for interactive placement of markers in a sequence of images read from file. The marker information is saved in an output file.

cated. Better use the `draw_sym` unit for that, or the `draw_sym` class from the `draw` `prog_unit` library).

To draw into a particular window, place the `mouse_xy`-unit after the unit that has this window (when created, a `write_pix`-unit "looks leftward" and seeks the nearest predecessor unit that manages a window in which the `mouse_xy`-unit can draw its markers. If it finds such unit, it selects the associated window as its target window. If it fails to find such unit in the current subcircuit, the search continues in the container one level up and so on; if no predecessor unit with a window is found, an error message is issued). Each `exec_unit`-call of the `mouse_xy`-unit will then draw a marker in the selected window. Depending on the mode the `mouse_xy`-unit is in, the location of the drawn marker is the position of the mouse cursor, or the coordinate values entered at the input fields of the `mouse_xy`-unit. Thus, the former mode allows interactive marking, the latter mode positioning of markers according to coordinate values from some other data source.

8 shows a circuit that combines some of the facilities discussed so far to build a little application for putting markers in images and saving the marker positions together with the image data (the file for this example has the name `markim`). When the circuit is run, it allows the user to step through a number of images that are read by the `read_image`-unit. After the `write_pix`-unit has displayed the new image, the `input_window`-unit queries the user to choose a new image or to choose to put markers into the current image In the first case, the `input_window` feeds a value of zero to the control inputs of the circuitry on its right and thereby disables their operation. The other outputs of the `input_window` are combined to obtain the index increment of ± 1 for the next cycle of the `read_image`-unit. In the second case, the circuitry on the right is not disabled, and the iterator iterates the `mouse_xy`-unit 3 times, thereby allowing to set three markers in the displayed image.

The `iterator`-unit also assembles the three coordinate pairs into a 6-dimensional vector (as in the logistics-example), and this vector, together with the index of the currently fetched image, is written by `write_data`-unit (provided it is allowed to do so by the next `input_window` that gives the user a choice to dismiss the marker positions that he has just selected). When operating this circuit, note that the output file that is associated with the `write_pix`-unit must be explicitly opened and closed via the unit's parameter window (this is to avoid accidental overwrites of existing files). Note also that the `write_data`-unit is configured such that the image data themselves are not written as new files, instead, only the file names of the source files are written into the output file.

9 3D-Visualization

One motivation behind *NST* was to provide some simple means for tayloring 3D-data visualizations with little effort. Currently, the basis for this is the `view3d`-unit (additionally, you can use the `plot3d_unit`, which can do only line graphics and does not require OpenGl). It allows to view a three-dimensional volume from different directions that can be changed with the mouse. The objects that appear in the volume seen through the `view3d`-window are created, positioned and oriented with the aid of other *NST* -units (all 3D-related units are currently in the folder `solid`). Basically, there are three categories of units: (i) those that create instances of three-dimensional objects (such as `make_pyramid, make_cone, make_arrow, make_mesh, make_link, make_object`), (ii) those that apply translations and rotations to instances of created objects (`rotate` and `translate`) and (iii) those for auxiliary functions, such as illumination, selecting materials of which objects are made, and coordinate transforms.

The basic idea is simple: when a container unit contains a `view3d`-unit, the window of this `view3d`-unit will show all those objects that have been created by units of the first category (i) that appear in the circuit to the right of the `view3d`-unit (i.e., those that come later in the `exec-`queue)[10]. Units from the second category (ii) can be interspersed to change the relative position between objects. These changes accumulate from left to right. Units from the third category can be used to switch to a different material, or to query the coordinates of points on particular objects.

To achieve this, the `view3d`-unit is, like the `iterator`-unit, an operator unit that takes control over the execution of all (or a specified number of) units to its right (these units become the operands of the `view3d`-unit).

As an introductory example, consider the circuit in **??** (the file for this example has the name `arrows`). The rightmost three units create three orthogonal arrows with a common origin. In the absence of the coordinate transform units left to them, these arrows would be centered within the cube and be aligned parallel to its edges. The presence of the `translate`-unit introduces a translation, before the arrows are rendered. This translation is determined by the three values at the input field `in_0` of the `translate`-unit. Likewise, the three `rotate`-units introduce rotations about the x-, y- and z-axis of the current coordinate system (note that each transformation always refers to the axes orientations that result from the accumulated effects of all coordinate transform units that are more leftwards, therefore the order of these units is important). Again, the rotation angles can be adjusted via an input field. The input values of these units taken from an `input_window`-unit that allows their adjustment via six sliders. In this way, the arrows can be translated and rotated interactively. Finally, the `use_material`-unit is to be credited for the nice brass-like appearance of the arrows.

[10] of course, that an object is visible requires in addition that its position is in the volume viewed by the `view3d`-unit

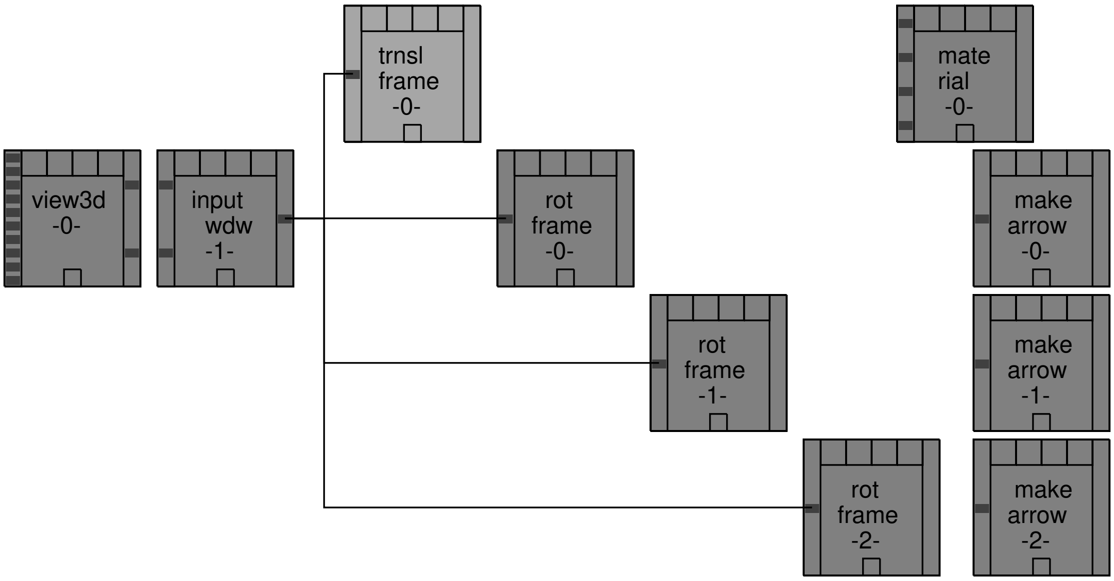


Figure 9: The “arrow”-example illustrates how objects can be visualized and moved.

This example essentially illustrates all important aspects behind the 3D-visualization within *NST* . More complex shapes are built from simpler parts with suitable coordinate transforms and material selections between them. These are then encapsulated into a container, with the size and positioning parameters of interest fed to a suitable number of input pins. In this way on can construct a kit of increasingly complex shapes for rendering robot scenarios or customized data plots.

An example of this kind is shown in **??**. It shows the circuit that renders the three-link robot of **??** (the file for this example has the name `robot3`). The rendered image can be accessed as pixel data with the use of the `read_pix`-unit.

This is illustrated with the example `robotview`. `read_pix` unit that “sees” the image in the window of the box left to it (the `read_pix` unit searches for the next suitable window that is to its left in the queue). This makes the window content available as a pixel array on which other units can operate. In the present case, the output of the `read_pix` unit is then processed by the subsequent unit named “prepro”. Inside the prepro-unit is a gabor-unit that transforms the output from the `read_pix` unit into a feature vector. The feature vector is retransformed into an image and displayed by a `write_pix` unit. When executed, one can move the robot, rotate it and observe the changes in the displayed gabor feature vector.

Besides solid parts, another important type of parts are three-dimensional meshes (including polygons as a special case) and “clouds” of points, vectors or error crosses. These types of objects can all be created with the aid of the `make_mesh`-unit. It is rather versatile, and for a description of its capabilities the reader must be referred to its man-page. Here, we only want to illustrate its capabilities with two examples of its use.

The first example (12) is a circuit in which a self-organizing map unfolds driven by a random sequence of points clustering on the two-dimensional hyperboloid $z(x, y) = x * y$. The self-organizing

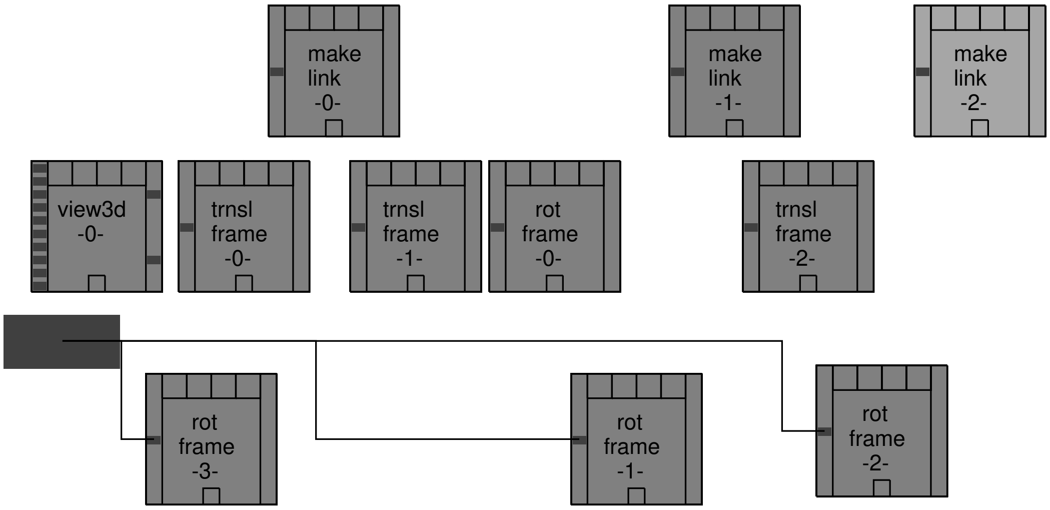


Figure 10: A container unit that contains a simple three-link robot. The upper line of units makes the parts, the middle line is responsible for the fixed transformations, and the lower line for the joint rotations. Their angles can be controlled from outside via input to the leftmost input connector.

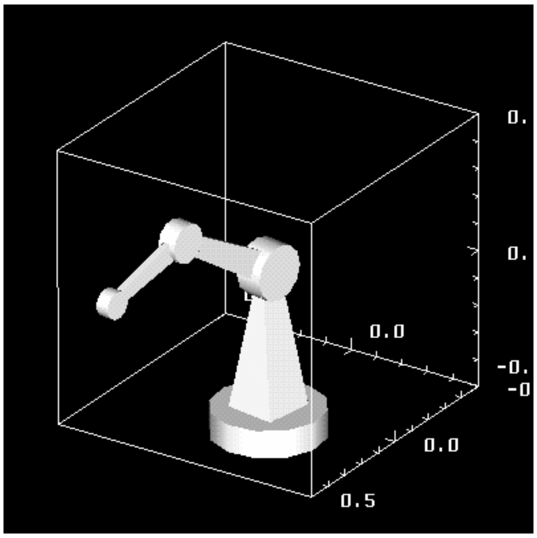


Figure 11: neo:robot3pic
The robot created by the circuit of **??**.

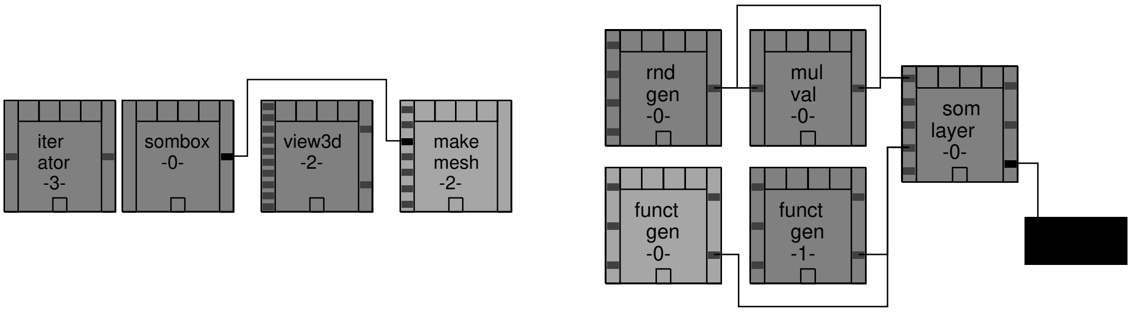


Figure 12: -unit to visualize the evolution of a self-organizing map.

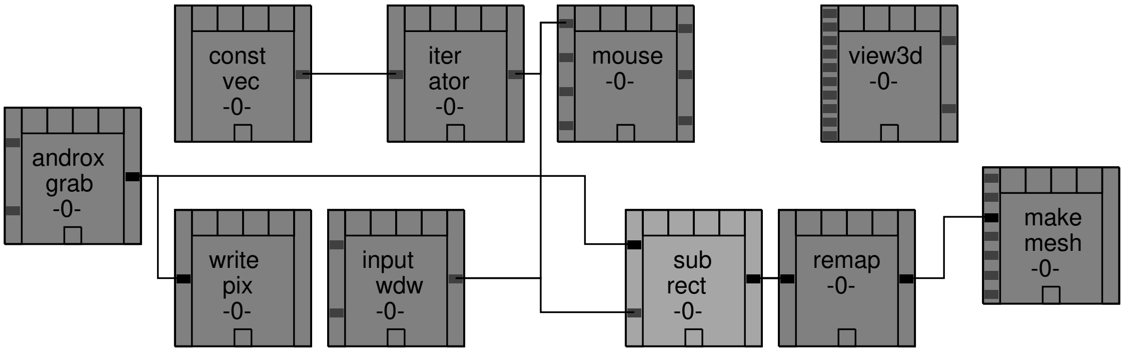


Figure 13: Visualizing 2D-data as an evolving mesh surface with the `make_mesh`-unit

map is created with the `som_unit`. It is encapsulated in a container unit `som_box` that contains in addition the necessary units for taking care of the proper learning step size control (exponentially decaying values for the learning step size and the neighborhood radius, both expected at `in_2` of the `som_unit`). The mesh formed by the reference vectors of the self-organizing map is a subset of the internal weight values of the `som_unit`. These weight values are available in the form of a single packed field at `out_3` of the `som_unit`. A suitable mask for the `make_mesh`-unit (see its creation window) picks out the proper subset of weight values (every 4th weight must be omitted). This subset then is visible through the `view3d`-unit.

The second example (13) again uses a `make_mesh`-unit, but this time configured differently. In the previous example, the `make_mesh`-unit displayed an entire mesh surface in one step, this time it is in "incremental mode". In this mode, each `exec_unit`-call adds one strip of data points (given as a vector of corresponding smaller dimension at input field `in_0`) to a mesh surface that thereby gradually grows until it has reached its maximal size. Then, each new strip that is added at one side shifts out a corresponding strip at the opposite side of the mesh. In the present example, the data are taken as the intensities of a horizontal scan line in a pixel image continuously grabbed from the Androx boards. This scan line is cut out with the help of the `subrect`-unit and subsampled to 40 pixels width by the remap-unit. The vertical position of the line can be adjusted with the input_window-unit. The mousexy-unit indicates the vertical position of the current scan line by placing two markers in the display of the write_pix-unit. This requires the iterator (providing the two calls to the mousexy-unit). The horizontal position difference of 80 pixels is achieved by having a counter increment of this amount. This is achieved by the const_vec-unit that feeds the required increment value to the control field of the iterator-unit. The dynamically unfolding mesh-plot illustrates how the time history of a seqence of 2D-data may be visualized.

Another useful 3D-rendering unit is the make_object unit. It allows to define a complex object (that even may include rotary joints) via a description in a file. In addition, one may specify a number of points on the object for which then values in world coordinates are computed and updated under each move of the object. These values are made available at output fields of the created object unit. The world coordinate frame into which the transformation is made can be "anchored" to any intermediate coordinate frame by inserting a define_world unit in the proper place. Finally, the world_to_screen unit allows to transform the computed values into screen coordinates, e.g., for attaching markers with a mouse unit etc.

10 Operators

In many cases, it would be useful to have circuits in which some units play the role of parameters that can be specified later. Of course, one can always use void_unit for this purpose, however, substitution of the actual units then has to be done by hand, including the necessary rewiring. The virtual_unit provides a more elegant solution. Like a void_unit, it must be created with the interface that is required for its later function. However, in contrast to a void_unit it assumes its function not by being filled with subunits. Instead it inherits the function of *a selectable successor of its container unit*. This successor will be called the "operand" of the virtual_unit. Several virtual_units can reference the same operand. In this way, one and the same unit can be called in several places in a circuit (the referenced operands can even be virtual_units, or container units with virtual_units; in these cases, the chain of references is simply followed until it terminates at a "normal" operand unit).

This mechanism allows the implementation of very general and powerful methods in encapsulated circuits that bind to their operands only at run-time. We will call such circuits "operators". *Neo* recognizes circuits that are operators or operands of some operator and indicates this property by a red surrounding square for an operator, by a yellow one for an operand (note however, that in the current version there are some restrictions when either a mouse_xy or a read_pix-unit are used as operands. These units "search" their predecessors for a suitable source window that they will use for their operation. If these units are operands, they logically should search among the predecessors of the corresponding virtual_unit. However, currently this is not implemented, and the search always uses the direct predecessors, as if there were no referencing virtual_unit).

One example of an operator that we have met already is the iterator. Other implemented examples are the the ctrl_op, the spatial_operator and the view3d-unit. Another interesting example is provided by the make_mesh-unit. Its format string allows to specify that it uses an operand to modify or extend its drawing of mesh points (not yet implemented). For each mesh point, the operand will be called once. When it is called, it receives a selectable subset of the data for the current mesh point. By implementing a suitable drawing routine in the operand unit, the user thus can very flexibly embed additional graphical information in the various types of plots that can be made with the make_mesh-unit (for details, see the man-page).

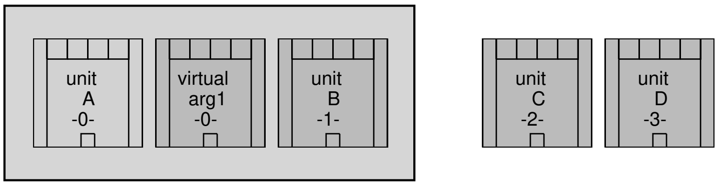


Figure 14: The `virtual_unit` binds to the first (indicated by the `arg1`-inscription) subunit `C` that follows the operator box. As a result, the units become executed in the order `(A C B) D` (brackets indicate the enclosing container).

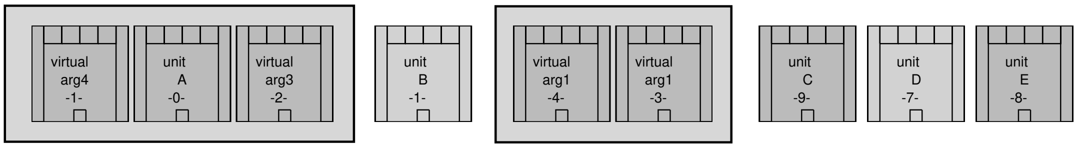


Figure 15: This example illustrates how several `virtual_unit`s can be used to change the normal execution order and to execute particular units repeatedly (the execution order in this example is `(D A C) B (C C) E`).

Further examples will be considered in the next section, where the use of the `virtual_unit` in conjunction with variables is exploited to create interactively new and flexibly usable units.

When a unit is used as operand of an operator, it should be entirely under the control of its operator. To achieve this, a unit that is bound by an operator loses its function under direct `exec`-, `ctrl`- or `adapt`-calls (the other methods are not affected by the operator/operand referencing mechanism, i.e., `load`'s, `save`'s and `remove`'s always work unchanged).

For instance, in 14 the unit `C` is referenced by the `virtual_unit` with the preceding container. When the entire circuit is executed, the units thus become executed in the order `(A C B) D` (*not* `(A C B) C D`, since `C` as an operand of the `virtual_unit` is not executed at the the top level). A more involved example is shown in 15. Here the execution order is `(D A C) B (C C) E`, illustrating how instances of `virtual_unit`s can be used to change the normal execution order and to execute particular units repeatedly. Several `virtual_units` can be nested as shown in 16: here, execution of the circuit results in the calling sequence `((C A) B) D`, since the innermost `virtual_unit` references another `virtual_unit` on the middle level which in turn has `C` as its operand (Note the suppression of the execution of the `virtual_unit` at the middle level, since it is the operand of another (the innermost) `virtual_unit`!). For adapt-calls, the execution order is always inverse to that of an exec-call.

How is the operand of an `virtual_unit` specified? When the `virtual_unit` is created, the positional offset of its (future) operand unit, relative to the container of the `virtual_unit`, must be given. (a value of 1 selects as operand the immediate successor of the container of the `virtual_unit`). Subsequently, the `virtual_unit` will behave under `exec_unit`- and `adapt_unit`-calls as if the referenced operand unit were in its place. That is, at each call of the `virtual_unit`, its input values are internally transported to its operand unit, then the operand unit is called and the result values are transported back. The binding to the operand unit is dynamically established each time the `virtual_unit` is called (fixed is only the positional offset of the bound operand behind the container).

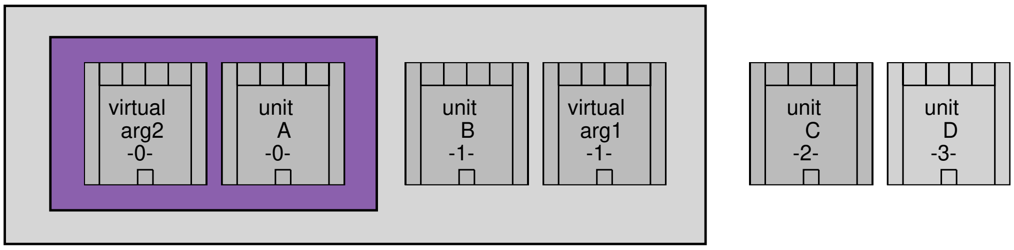


Figure 16: An example of nested use of `virtual_unit`s.

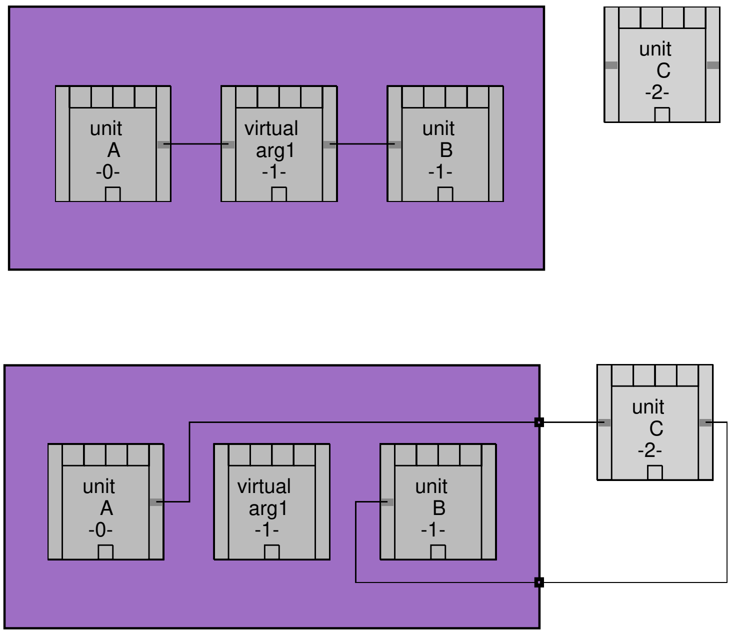


Figure 17: Communication with an operand unit: *Top:* data transfer can proceed via fields created with the `virtual_unit`. At execution, data are copied between corresponding fields of the operand unit `C` and the `virtual_unit`. *Bottom:* Alternatively, a `virtual_unit` without fields may be used and communication achieved via direct connections with the operand unit `C`.

How can an operator circuit exchange data with an operand unit that is referenced through an virtual_unit? There are two alternatives (they may be used in combination): For the first alternative, for each field of the referenced operand that shall be accessed the created virtual_unit is equipped with a field of same type and dimension, and this field is specified to correspond to the target field of the operand unit (the necessary specifications must be made when the virtual_unit is created). Then, at each call of the virtual_unit, the current input values at the input fields of the virtual_unit are copied to the corresponding input fields of the referenced operand, then the operand unit is called, and result values at its output fields are copied to the corresponding output fields of the referencing virtual_unit. This is shown in 17 (Top), where an virtual_unit references unit C. C has a single input- and output-field, each of which has a counterpart on the referencing virtual_unit (Note that the field positions of corresponding fields need not be the same; as long as corresponding fields are of identical dimension and type, arbitrary correspondendences may be defined). While this way of communication is very convenient (no rewiring is necessary, when an operand unit is replaced), the copying costs time. Such copying can be avoided with the second method: Here, the virtual_unit is created without input and output fields. Its presence just serves to define when its operand is to be called. Any communication with the operand proceeds as usual by suitable connections. Thus, the equivalent circuit of 17 (Top) would now look as shown in refneo:insert4 (Bottom). If desired, both methods may be combined, i.e., some fields of an operand may be accessed via corresponding fields of the referencing virtual_unit, while others use the usual, direct way of communication [11]

As a final remark we mention that the exec_opnd() function in the prog_unit allows to make endow the prog unit with some of the features of the virtual_unit.

11 Structuring Circuits with the Virtual unit

Frequently, a program allows a natural subdivision into a number of separate stages, such as initialization, training, testing, visualization etc. In NST, each such stage would be implemented in the form of a suitable circuit diagram. In many cases the different stages must be able to share some of their data. E.g., initialization, training, testing and visualization may all need to access the same set of neural network units, while testing and visualization might share some units for displaying error curves etc. From the previous section we see that such sharing of units can be achieved with the virtual_unit: each stage is implemented as a separate subcircuit in a container that contains the necessary number of virtual_units to access the units that are shared.

Demo7-4 (Fig.) provides an example of this technique. It demonstrates how a PSOM-net learns to mark the position of the index finger in a set of images of a rotating wooden hand replica. The entire program is subdivided into four logically separate units: training (train , testing test, computation of features comp-feat, and the transformation effected by the PSOM-network psom2. These units are arranged in such an order that units that use other units (via virtual_units in their subcircuit) as operands are positioned left of these operands. For instance, the train-unit uses both the the comp-feat unit and the psom2-unit and is, therefore, positioned to the left of both.

[11] It is even possible to use both mechanisms with the same field. E.g., an input field of an operand unit can be directly connected to a data source A, while it has several corresponding input fields at several referencing virtual_units that are connected to further data sources B, C, ... In this case, each call of one of the referencing virtual_units has the effect that *its* data source overwrites the current values at the inputs of the operand unit before it is executed.

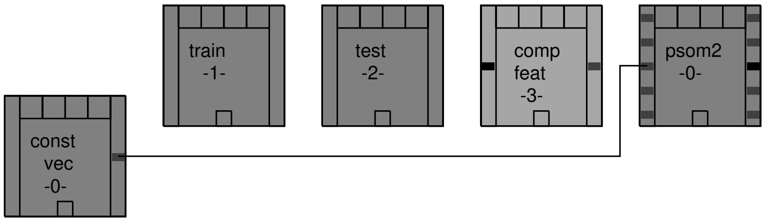


Figure 18: Top level circuit

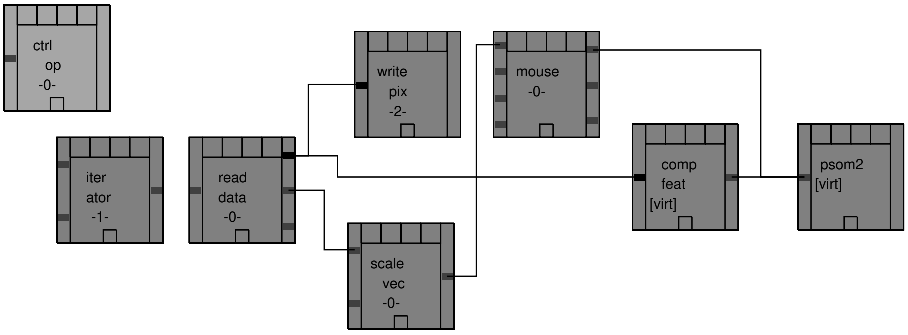


Figure 19: The subcircuit `train` of the top level circuit of the previous Figure.

The lower part shows the subcircuit that is contained in the `train`-unit (the subcircuit of the other units are not shown). It consists of a `ctrl_op`-unit that initializes the remaining units, followed by an `iterator` loop that reads 16 training data pairs from a file (using a `read_data` unit). Each training pair consists of an image (at the packed output field of `read_data`) and a 2d-vector (at the next output connector) that specifies the position of the index finger in the image.

The image is displayed with a `write_pix unit` and fed to the `comp-feat` unit. Access to this unit is through a `virtual_unit` that references the actual `comp-feat`, which is outside the subcircuit at the top level. The `comp-feat` unit computes a feature vector from the pixel data and sends it to the `psom2`-unit, which is again accessed by another `virtual_unit`.

The finger position vector is rescaled to pixel coordinates (`scale_vec`), the position is marked with the `mouse-unit` (it draws into the preceding `write_pix` window) and sent as two additional target values to the (virtual) `psom2`-unit (for more details, see the `demo7-4.HLP`-file).

This approach leads to a very modular structure: as long as the interfaces are kept unchanged, any of the shared units can be replaced interactively by a different one. In the present example, for instance, one could replace the `comp-feat`-unit by a different version that computes the image features in a different way, or one could replace the `psom2` by another network type. The only requirement is that the interfaces are kept unchanged.

12 Variables

So far, all parameters of the units of a circuit had to be provided as numerical or string constants when the unit was created or modified. This is not very flexible, in particular if subcircuits shall be reused. It is also very inconvenient, when the value of a parameter that is shared by many units must be changed in many places. It is particularly cumbersome, if the parameters of several units are related in some less obvious way. To overcome this problem, *neo* provides some means for the use of variables.

The basic idea is simple. For each container unit, the user can define an input window (the container's "variable window") with a number of "slots" for data values such as integers, floats, or text strings. Additional text in the window layout may be used to indicate and comment the meaning of the individual slots. The slots are then available as "local variables" for the direct subunits of the container unit. That means that any in any creation or parameter window of a direct subunit entries of the form `@0, @1, @2 ...` (input slots for numbers may also contain arithmetic expressions such as `@2*(1-@0)`) may be used as abbreviations for the explicit data values that are currently present in the corresponding slots 0, 1, 2 ... of the user-defined variable window (the data values may contain entries of the form `@0, @1, @2 ...` in turn). In this way, a single value in a slot of a variable window can be used in multiple subunits. Furthermore, if source code is generated, *Neo* will write for each container unit a constructor function in which the variables that were defined for the container unit appear as function parameters of the proper type (`int`, `float` or `char*`) (cf. Sec. 4).

A variable window can be defined for any container unit. When a value in a variable window is changed, all subunits of the container unit are remade. Consequently, a container unit with a variable window appears much like a primitive unit, with the variable window taking the role of the creation window of the container unit. However, unlike primitive units, which are provided through a compiled wrapper routine , container units with variable windows can be defined, stored and modified interactively and at runtime, without any compilation. Thus they offer a very convenient means for extending the functionality of *neo* in a modular way.

Before we describe how to define a variable window for a unit, we consider an example of the use of variables, together with operators. To this end, load the `plotfun`-circuit. It contains the `plot_fun`-unit, which is an operator that takes one operand unit. The `plot_fun`-unit assumes that its operand has an input field and an output field of 1 pin each, both at field position 0. When executed, the `plot_fun`-unit will create a 2D-plot of the transformation from input to output that is effected by its operand unit. The parameters of the plot (size, location, range, nr of points) can be set by opening the variable window of the `plot_fun`-unit by clicking the `M`-area . The values in this window are used by the subunits of the `plot_fun`-unit. One of the subunits is an `virtual_unit` that accesses the operand and makes the unit into an operator. To create the plot, the operand must be called repeatedly, therefore the `virtual_unit` is itself the operand of an `iterator`-unit. The number of iterations equals the number of plotted points and is also set via one of the the variables of the variable window.

The `plotfun`-unit provides a model for an entire class of interesting operators the effect of which is to somehow *inspect and visualize* the contents or state of their current operand. The basic scheme for such "inspector units" is always the same: the inspector unit always contains some `virtual_unit` that must "fit" to the to-be-inpected unit that is the operand. Some additional "machinery" (e.g., suitable plotting or viewing units) inside the inspector unit is then suitably connected with the `virtual_unit` to access the data in the to-be-inspected operand unit and to visualize it as desired. The `InspectSom2d`, `InspectPSOM2d` and `InspectPSOM3d` units (they are loadable under these names, but appear with shorter labels in the circuit window) provide further examples of this technique. Simply moving them in front of an operand unit of corresponding type (and modifying their creation parameters to fit the characteristics of the operand unit) is all that needs to be done to "look into" the corresponding operand unit. In addition, since the circuit within these inspector units is fully accessible, it can be easily modified to customize the inspection operation to the requirements at hand.

Another frequently useful class of operator units are “scheduler units” for learning parameters. Their construction follows an analogous scheme as in the case of the inspector units. However, in this case, they do not retrieve output parameters of their operand for visualisation. Instead, they writes values into those inputs of their operand that expect learning step sizes, and they update these values between successive execution calls in a user-defineable (through their creation window) manner. Examples of such units are provided by the `SchedLLM` and `SchedSom2d`-units.

These examples may suffice to illustrate how the combined use of variables and `virtual_units` allows to configure many frequently needed combinations of operations into readily usable and easily combineable operator circuits. In particular, this scheme is also rather effective for structuring more complex *NST* -programs along the lines described in the previous section: Define a number of operator units for the main operations used in the program, such as `Init`, `Train`, `Test`, `DisplayThis`, `DisplayThat` etc. To carry out their function, these units may act on one or a few operand units (some of which may be shared between several of these operators) in which the variable data of the program (such as neural network units etc) are maintained. This “programming style” is exemplified with the example program `psom_hand`. It uses a `psom` to map a number of marker locations into a hand posture. This is done on a 5x5x5-testing grid, when the unit `Test` is executed (click its x-field). However, this only works properly, when the `psom` has been properly trained before. This is done for a 3x3x3-grid of postures, when the `Train`-unit is clicked once (IMPORTANT: enter the `Control`-command and issue a `NST_INIT`-command before. This is required to set the `psom2`-unit into learning mode. Ideally, this should be done by the `Train`-unit also, but a suitable control-command-issuing unit still awaits its implementation).

We conclude this section with a brief description of the necessary steps for the definition of the variable windows for the containers of the corresponding circuits.

To define a variable window for a container unit, open the unit so that its contents appears in the circuits window. Then invoke the `Define`-command. This will open a form requiring you define the format of the variable window (specifying the usual `input_window`-format string). For each variable that you desire, provide one input item of corresponding type in the layout of this window. Additional text may be arbitrary, but should usually help to indicate the significance of each input item. Test your layout with the test-button until it is as desired. If you accept the layout, the input items of the newly made window define the variables that are available for the subunits of the container unit. The value in the i-th slot of the newly defined window is referenced by `i` ($i = 0$ references the first position), i.e., tokens of the form `@i` can now stand for the corresponding variable values in the input forms of the subunits of the container unit (to make this process easy, choose for the newly defined variables initial values that are compatible with the to-be-replaced current values of the units that shall be “variablized”). In case of numeric variables, arithmetic expressions, such as `@2*(1-@0)`, will also be evaluated properly. However, this is NOT SUPPORTED FOR STRING VALUES (note also that the possibility to use strings of the form `@i` is lost (a terrible sacrifice!).

As soon as a variable window is defined, the container unit reacts differently when its ‘M’-field is selected: it opens its newly defined variable window, displaying the current values of the newly defined variable and offering the possibility to change them. If this is accepted, all subunits in the container unit will be updated correspondingly (irrespective of whether they reference any variables or not). An exception is the `ROOT`-container: since it cannot be accessed from outside via an `'M'`-field, accessing its variable window is effected by pressing the right mouse button at the top level.

If later on for a container its set of defined variables shall be changed, use the `Define`-

command again and modify the variable window layout accordingly. All current variable references of subunits to variables that lie behind the first modification point in the window layout will be lost and replaced by the values that these variables had before the modification was made. All earlier variable references remain intact (only layout modifications that change something essential with the input fields are counted; mere reformatting of the text portions of the window [without changing the order of the input fields!] does not disturb the current variable references). To revert the container to the state without any variables, specify an empty string for the format of its variable window.

A second occasion when variable references are replaced by their current values is when a unit is moved into a different container (since this container may not have the same variables defined, such replacement cannot be avoided).

A third occasion is when loading a circuit into a container. When the new host container has no variable window defined, it will inherit any variable window of the loaded circuit (should there be any). Otherwise, the variable window of the host container prevails and the load operation is treated like `move` above [12], i.e., the variable references of the loaded units are lost and replaced by their values.

Note however, that in all these cases only the variable references at the top level are affected. More deeply nested variable references remain intact, since their environment remains untouched by the move or load operation.

There are some limitations: currently, variables cannot be used to specify a variable number of subunits, or to specify the topology of the wiring. If the dimensionality of two connected fields changes due to a change (and the subsequent update) of variable values, *neo* will generalize the connectivity to the new dimension only in the simplest case when the two dimensions are equal and the fields were fully connected. Otherwise, *neo* complains and refuses to accept the new variable values (the old circuit remaining in place) (the precise rules how the existing connectivity is generalized for new variable values will be given in the appendix). Therefore, variables should be used with some care and preferably in places where their effect is easy to oversee, such as dimensions for packed fields, file names, or vector dimensions, to name a few.

13 Running *neo* -Circuits from the command line

There is an additional important property that allows Neo circuits to be run in batch mode: Assignments of the form `"@i=value"`, `i=0,1,2..`, can be used on the command line to override initial values of variables that are defined in the top-level dialog window of a circuit loaded with the `"-i filename"` option.

If the additional option `"-x iSteps"` is specified, execution will automatically start after loading, execute iSteps `exec_unit` calls of the entire circuit, and exit (to avoid the exit, specify iSteps as negative). If each `exec_unit` call shall be followed by and `adapt_unit` call, specify the flag `-a` (no arguments) together with `-x iSteps`.

Example: to start the demo `demo5-4` with initial shrink factors of 3 and 4 and a window title of `"MyWindow"`, enter

```
neo -i demo5-4 @0=3 @2=MyWindow @1=4 -x -1
```

[12] this suggests to liberalize somewhen the move-command if the target container has no variable window

The order of the command line parameters is arbitrary, and other options (such as the use of a startup-file with `-@startupfile`) may be interspersed. However, in the `"@i="`-assignments, the assigned value may not be separated by spaces from the `"@i="`-part!

Some error checking for legal ranges and argument types is done. The empty string `""` has a special status and cannot be used as an assigned value.

Note that the escape of a `-x iSteps` option only works when the program does not have any interactive loops that permit an escape in the absence of suitable user input.

To execute a *Neo* circuit without opening any *Neo* -windows invoke *Neo* with the command line options `-hide -x iSteps`.

14 Extension by new Units

The set of built-in units of *neo* can be easily extended by user-defined new units. The simplest way of extension uses the `prog_unit` or the `matrix_algebra`-unit. The `prog_unit` will make a unit that executes a C-program that can be typed in when the `prog_unit` is created (It uses a simple interpreter for a restricted subset of `C`-language). The `matrix_algebra`-unit is similar, however, it offers a Mathcad-like language with primitives for many linear algebra operations, such as matrix multiplication, inversion and operations such as eigen- or singular value decomposition. This allows a very compact notation; moreover, matrix and vector operations are carried out by compiled subroutines so that almost the efficiency of compiled C-code is achieved.

A very powerful extension possibility is via dynamically loadable `prog_unit` extensions: the prog unit allows to import additional, user-specified functions or classes from compiled libraries. Such libraries can be obtained from standard C or C++ libraries by writing a very simple and very small wrapper. The possibility of using function overloading and default arguments allows to provide convenient interfaces for the imported objects. For details, please consult the `prog_unit` manual page.

At the next higher level one can use `NST-`units that have been built from *neo* - units and been saved as encapsulated units by dragging them into a loadable folder (cf. next subsection) or by saving them directly to a file. These can be simply loaded with the “load” command (see **??**) or fetching them from the loadable folder where they were placed before.

Finally, there is a tool pair `mku` and `mku++` to write skeleton code for the rapid wrapping of existing C or C++ classes in the form of compiled *NST* class containers.

In both cases, the newly added units can be grouped into folders that then become available in the same way as the predefined folders are. Folders with units loadable from file will be called “loadable folders”, folders containing units that are written in C are called “compiled folders”.

14.1 Loadable Folders

The folders that contain units defined in files are *loadable folders*. The addition of such folders to *neo* can be made via the command line. To this end, specify simply `-f folderdeffile`, where `folderdeffile` must specify a file with a listing of the desired loadable folders that shall be added (for the format of this specification, see below). Another way to add a number of loadable folders is to specify them with the `-i` option, concatenated by plus marks. If the first name after the `-i` has no plus sign in front, it is taken as a circuit file, otherwise as the first loadable folder to load. Example:

```
neo -i mycircuit+folder1+folder2+folder3
```

Technically, a loadable folder is simply a directory collecting a set of *Neo* circuit files and/or dynamically loadable objects.

In the simplest case, the directory contains a single subdirectory with the reserved name `circuits` containing a set of `.NST` files of encapsulated circuits (usually, one should have no `*.nst` data files in this case, since loadable units are in most cases reparametrizeable, and any stored data will be lost anyway after reparametrization of a unit). The `.neofolders` file in the home directory and, optionally a list of names prefixed by a plus mark in the `-i` - argument specifies which folders will be loaded when *Neo* comes up. Each entry in `.neofolders` must specify (on a separate line) a pair of tokens: first the folder name (it will appear on the folder's button and should not exceed 10 characters), followed by the name of the folder directory (full absolute path) that has a subdirectory `circuits` which contains the units (as `.NST`- and optionally `.nst`-files that were saved as "encapsulated circuits", i.e., with a enclosing container box, cf. Sec.2.8) of the loadable folder. An example of a `.neofolders` file that defines two folders named `myfolder` and `yourfolder` would be

```
myfolder /homes/me/projects/myfolderdir
yourfolder /homes/metoo/morestuff/anotherdir
```

Additionally, there may be interspersed lines

```
FOLDERGROUP xxxxx
```

where `xxxx` is a folder group name. Such lines partition the listed folders into groups which can then be "collapsed" into single buttons (they will come up in the "collapsed" state, when `xxxxx` is given in brackets (i.e. as `[xxxxx]`) The units in these folders are then assumed to be in directories `/homes/me/projects/myfolderdir/circuits` and `/homes/metoo/morestuff/anotherdir/circuits`. All `.NST` and `.nst`-files in these directories are assumed to be units for these folders. Other files may be present, but are ignored. Therefore, such loadable folders can be flexibly modified, e.g., by using *neo* 's Save command for writing new or updated versions of circuits directly into them. As an example, the directory `demofolder` in `neosrc/demos` contains the `.NST` and `.nst`-files for some of the units described in the previous examples (to indicate that these files are encapsulated circuits, capital initials were chosen for the file names; however, the file names may be arbitrary). The file `DemoFolder` specifies the directory `demofolder` to be used as a folder named `example`. By calling *Neo* with

```
neo -f DemoFolder
```

from within `neosrc`, where the file `DemoFolder` is located, *Neo* will come up with an additional folder button named `example`. With this button, the included loadable folder can be opened, and its units become available by simple drag-and-drop actions.

This is also the case for the reverse direction: newly created units (both primitive and container units) in the circuit window may simply be dragged into the module window, when it shows a loadable folder (this is indicated in its title bar). This installs a copy of the dragged icon in the loadable folder from which then arbitrarily many copies can be dragged back into the circuit window. (technically, dragging an icon into loadable folder will save the icon in the directory that represents the loadable folder; thus, the icon becomes a permanent member of the loadable folder and will be available also in later invocations of *neo* , when the folder is loaded again).

There is a versioning scheme both for individually saved loadable units as well as for entire folders. For a description, see the manual `versioning.tex`.

14.2 Compiled Folders

Compiled folders contain units that are implemented in C or C++. *Neo* comes with a number of predefined compiled folders, and one of them, named `private` is provided as a convenient starting point for adding user-specific units to *Neo* . To this end, the folder `private` contains 10 /vol/nst/man/html3void_units (named *example0..9*) that can be replaced by other, user-specific units by editing the file `neo_private.c` suitably. This file contains for each of the 10 units a simple wrapper routine. By having a look at these wrappers, you may be able to figure out how to customize them to make your special units accessible from *Neo* (this basically requires to replace the `void_units` by the creation call of the to-be-added unit, to modify the format strings for the two dialog windows suitably, and to add the name of the object file containing the new unit to the list `NEO_PRIVATE` of objects in `Makefile.shared`). If you have made your modifications, the command `make neo` (issued from directory `o.arch` (`arch` being one of the architectures `linux`, `irix`, `sun`, `rs6000`) with the `neo-`sources in `neosrc`) will create a new, extended version of *Neo* . More details on this are given below.

Additional compiled folders may be added to *Neo* . Their addition requires writing of a file of wrappers. This is supported by a tool `mku` (or `mku+` for C++). For details, see the document `mkudoc.tex`.

To understand in some more detail the addition of compiled folders and units written in C into *neo* requires a brief sketch of the internal organization of *Neo* .

For each unit of a compiled folder that is available from within *neo* , there must exist (i) an object file or a folder in which the unit is defined as a *NST* -unit in the usual way (the units provided when *neo* comes up are mostly taken from `libnst.a`). However, to do its job *neo* needs for each unit some additional interfacing information that tells *neo* how to create the unit, how to change its parameters and where (i.e., in which folder) to accommodate it. For the units that are provided by default, this interfacing information is stored in the files `neo_xxxx.c` in directory `neosrc`, where `xxxx` stands for one of `math, funs, file, ... etc.` (the endings indicate the set of *NST* -units for which the file contains the interfacing information; however, the naming may be arbitrary).

The necessary details of these procedures are explained below.

14.3 How to add a new Unit to an existing Folder

First choose a name for the new unit, say `myunit` (the chosen name must be less than 15 characters long and it must qualify as a legal C-identifier; it is recommended to use the name of the C-creation routine that makes an instance of the *NST* -unit). Then choose a folder file to accomodate the new unit (e.g., `neo_private.c`) and add the following two entries:

```
extern NstIconDef_t ICONDEF_myunit;
```

in the part with the heading “declarations”, and

```
ICONENTRY(myunit),
```

in the definitions following the `BEGINFOLDER`-macro (don’t forget the comma!). This accomplishes the necessary declaration stuff. What still is missing, is the interfacing information required by *neo* to make suitable dialog windows for the unit, and to map their data entries onto the parameters used by the *NST* -creation and control methods for `myunit`. This is best explained by an example, assuming that `myunit` is a unit whose constructor function needs three parameters, one integer, one float value and one text string. Values of these parameters must be requested with the first dialog window, that appears when the icon for `myunit` is dragged into the circuit window. In the simplest case (illustrated by this example), the three required parameters are directly requested by the window (in more complicated cases, the window may request other parameters from which the constructor function parameters first must be computed). The layout of this window is defined by the entry `ICONFORM1` below:

```
ICONFORM1(myunit)
   " myunit - a special unit for this and that. %n"
   " first argument is :  %1i %n"
   " second argument is:  %0f %n"
   " and now a text string: %{hello world%}%30t %n"
ICONFORM2(myunit)
   " %d%lReset %n"
ICONLABEL("my|label")  /* the "|" makes a line break */
ICONMANPAGE("mymanpg")
ICONMAKE(myunit) {
   /* additional computations/checks etc. might come here */
   SRCLINE("$ = #($0,#1,@2,@);\n");  /* specifies source line */
   return myunit(IVAL(0),FVAL(1),TVAL(2),HOST);
}
ICONPARS(myunit) {
   /* issue a init-call, if reset button is pressed: */
   if (IVAL(0)) ctrl_unit(NST_INIT,u);
   CTL_in(UNIT,0) = 1.0;  /* set a pin value */
   SRCLINE("CTL_in($,0)=1.0;\n");
   /* note that u and UNIT denote the same handle */
   return 1;  /* if there is no ICONFORM1,
                 this may be the only statement */
}
```

The format of this layout specification follows the format specifications for the `input_window`-unit (i.e., %1i and %0f make an entry field for an integer or float value, initialized to values of 1 and 0, resp., while %t creates a field for text field of width 30 chars with the initializing value given by the text string between the %..% brackets; for more details, see the manual page for the `input_window` unit). The next entry, `ICONFORM2`, is analogous and defines the query window for the parameters of `myunit` that can be changed during runtime (in the present case, only the issueing of a reset-operation triggered by a button, is provided). As a general rule, the layouts of both windows should provide some assistance to the user for filling in suitable parameter values (such as, e.g., default values and limit checks, using the control fields available for the `input_window`-unit). In particular, the first line in both windows should provide the same info as the synopsis line in the unit's manual page. Note also that the confirmation buttons for the query window need not be (and must not be) defined by the user; they will be added by *neo* automatically. If the created unit needs no `ICONFORM2`-window, specify the empty string `""` for its format. Note, however, that a non-empty string MUST be specified for the `ICONFORM1`-window.

The third entry defines a label for the displayed icon (it should not be longer than 12 chars to fit into the available space). The fourth entry specifies the (base) name of a file containing the text of a manual page for the unit (*Neo* will search in the directory `NeoHome/NeoManPath` for the file `"mymanpg.3"`, [the suffix ..3 is appended], where `NeoHome` and `NeoManPath` can be set on the command line or in the startup file). The same name that has been specified for the manual page will also be used to look up an example file in the directory `NeoHome/NeoExmplPath` when the 'e' option is selected. If there is no manual page, specify `NULL` instead of the string (here: `"mymanpg"`).

Finally, the `ICONMAKE` and `ICONPARS`-entries define the bodies of two C-routines that handle the communication between *neo* and instances of `myunit`. `ICONMAKE` must contain all necessary statements for the creation of an instance of the new unit, and must return a `unitptr` (or NULL, if the creation failed). `ICONPARS` must contain all necessary statements to set parameters or issue control calls for the new unit (the return value is an `int` but not used). For both routines, access to the values entered in the associated `ICONFORMi` (i=1 for `ICONMAKE`, i=2 for `ICONPARS`) is provided by the predefined macros `IVAL(k)` (returns k-th numeric value of ICONFORM, cast to an int), `FVAL(k)` (as before, but cast to a float), `TVAL(j)` (returns j-th text value of ICONFORM; counting of text values includes numeric entries for i & f -fields [see the manual page for the `input_window`-unit]). All counts start from zero. It is also possible to access values from `ICONFORM1` from within `ICONPARS`, using the analogous macros `IVAL1..TVAL1`. Further predefined variables are: `HOST` (valid in `ICONMAKE`) denotes the host unit of the newly created instance (if a different host is specified, it will be changed automatically). `UNIT` in `ICONPARS` is the `unitptr` returned by the associated `ICONMAKE`-call (however, the shorter `u` is also defined).

The final elements to be explained are the two `SRCLINE`-macros that occur in `ICONMAKE` and `ICONPARS`. They define a piece of C-source code that is later used by *neo* when creating source code for the specified circuit. The occurrence of `SRCLINE` in `ICONMAKE` defines the source code that is inserted for the creation of the unit. It is specified as a string constant. The characters `$, # and` have special meaning. If followed by an integer value i, the sequences `$i`, `#i` or `@i` will be substituted by the value of `IVAL(i)`, `FVAL(i)` or `TVAL(i)`, respectively. A `$` without a following integer will be substituted by the identifier that *neo* uses in the source code for the created unit. Finally, the bigraphs `#(` and `@)` will be substituted by `NameOfRoutine(` (i.e., `myunit` in the present example)[13] and `NameOfHost`, respectively. A last set of substitutions is useful for window placement: if a unit has a window, and its `SRCLINE` contains one of the tokens `$x, $y, $w, $h`, these will be replaced by the position of the windows origin and its size at the time when the source code is generated.

More than one `SRCLINE` - statement may occur (their argument strings will be simply concatenated). If more complex lines must be built, the string constant may contain conversion specifiers in the format expected by the `printf`-function. For each such token, an additional argument must be provided that is then inserted at the position of the corresponding `printf`-conversion specifier in the manner known from the `printf`-routine. E.g., the first `SRCLINE`-statement could also have been written as

```
SCRLINE("$ = myunit(%d,%f,\"%s\",@);\n",IVAL(0),FVAL(1),TVAL(2));
```

(note here the need to explicitly specify the apostrophes to have a quoted string appear within the argument string of `SRCLINE`).

The entire sequences of these definitions must be placed in the folder file that shall accomodate `myunit`. No additional entries (except for C-comments) between the definitions of a block are

[13] for this, *neo* assumes that the name of the *NST*-creation routine is given by the label string defined in the `unitptr`-struct of the created unit.

allowed. More examples of such wrappers (including, e.g., how to handle variable numbers of input parameters) can be found in the default folder files `neo_xxxx.c`.

14.4 Adding new parameters to existing units

Sometimes, it is desireable to add new parameters to an existing unit. Simply inserting additional entry fields or buttons into its dialog windows would sacrifice compatibilty with previous circuit files. However, the special format token `%n'` (a nonnegative integer with an apostrophe as command letter) in front of a newly inserted entry field or button token marks this for *neo* as newly inserted and allows to use both old and new circuit files after the change. The number `n` specifies a "version level" and must be larger than any version level specified for a previous change in the same dialog window (several changes made simultanously can use the same value of `n`). Internally, each circuit file contains for each dialog window also its current version level (the maximum value of the version levels in all its `%'` tokens). When loading a circuit file, all dialog window items with a version level that is higher than in the file are initialized with their default values and no data for them are expected in the file. In this way, the version level stored in the file limits the set of dialog window items for which the file must provide values to the set that existed when the file was created, irrespective of later extensions, which, however, can fully be used by later files with higher version numbers.

14.5 How to add an entire new Folder

The file `neo_template.c` provides a template of an empty folder file. By following the steps outlined in the previous section add the definitions and wrappers of the units to be included in the new folder. In addition, choose a name (an arbitrary string, say "myfolder") and an identifier (a legal C-identifier, say `MYFLDR`) for the new folder. The name goes into an additional `# define` statement that must be placed before the `BEGINFOLDER() ..  ENDFOLDER`-pair. The icons for that folder are then placed with `ICONENTRY()` - definitions between these brackets.

```
#define FOLDER "myfolder"
```

```
The identifier completes the BEGINFOLDER-definition:
```

```
BEGINFOLDER(myfolder)
  ICONENTRY(myicon), ICONENTRY(youricon), ...
ENDFOLDER;
```

This completes the new folder file.

15 Configuring *Neo*

All that remains to be done is to make the new folder file known to *neo* . This requires to reconfigure *Neo* suitably. This is done in the central file `neo_defs.h`. It contains a sequence of entries (described below) that specify which folders are compiled into *Neo* (dynamically loadable folders can be added by specifying entries in the `.neofolders` file). `neo_defs.h` contains those entries that are intended to provide a base configuration of folders and units that should always be present. To add a folder (that itself must be defined in a separate file, as explained above), add a line

```
...
ADD_FOLDER(myfolder);
...
```

in neo_defs.h. If only a single unit shall be added to any already included folder (either in the base configuration, or included by a preceding USE_FOLDER()-entry in neo_defs.h, add a line

```
...
ADD_ICON(Myicon,FolderWhereToPlaceIt);
...
```

(a single unit is defined like a folder, but omitting the #define FOLDER and the part BEGIN-FOLDER..ENDFOLDER) With a suitable group of such entries in neo_defs.h you can specify a customized configuration of *Neo* , and several different configurations can be conveniently maintained, e.g., by including their sets of definition lines between #ifdef VersionFlag ... #endif directives. With the macro HIDE_ICON(IconName) icon IconName can be made "invisible" in its folder, i.e., instances from it can no longer be made interactively, however, it still remains available, e.g., when loading files of circuits that contain it[14].

16 Rebuilding *Neo*

When the necessary steps for extending *Neo* by new units or folders have been done, it is still necessary to build the new version. Assuming the familiar subdirectory structure of the nst-subtree (subdirectories o.arch, and the sources neo_xxxx.[ch] in a parallel directory neosrc), this is most simply done by issuing the command

```
make neo
```

from within o.arch. However, this requires that any specific information about your customized version of *neo* must have before been placed in the file nst_defs.h, and all required objects have been specified for the linker.

17 Bugs and workarounds:

1. A crash has occurred. In this case *Neo* may have been able to save its state to a recovery file, from which the session can be recovered with neo -r. However, this may not succeed in all cases.

2. neo -i inputfile crashes, although the circuit in inputfile was already working: in this case, *Neo* is unable to build all units successfully. To find out the unit that may cause the trouble, retry with the -TL ("trace loading") option. A likely reason is that the saved circuit contains units that reference data files, and these data files have changed or are no longer accessible under the names that were valid when the circuit was stored in inputfile (a particular trap is the use of the tilde in paths: since this is expanded to /homes/username, a different user may be unable to run the circuit!). Workaround: ensure that all needed files are present. If this is not possible, try to edit the inputfile.NST (remove the entries for the offending units).

[14] this is useful, e.g., to "phase out" the use of an obsolete unit, without sacrificing backward compatibility.

3. `neo -i inputfile` complains about illegal tokens. This occurs, when the data file `inputfile.nst` contains `NaN` or `Infty`-items. It does no real harm, just annoys. If no parameters are needed, specify `inputfile.NST` instead of `inputfile`. This lets *Neo* ignore the parameter file `inputfile.nst`. If the data file is integrated in the `*.NST` file, you can also specify the command line flag `-ignoredata`.

4. When the problems persist, you may wish to edit the `*.NST` file. It is a simple text file and allows corrections to parameters etc. The format is such that you also can view it with a HTML browser for easier navigation (but don't edit it with a HTML editor!!).

5. When running a circuit, the escape key seems to be without any effect: The esc-key is only polled when an input window is executed, or when an entire execution cycle of the current circuit is completed.

6. When a variable name `@i` is entered, *Neo* complains until the name (or an expression) is properly completed (does no harm).

7. Currently, the mechanism by which the `mouse_xy`-unit and the `read_pix`-unit reference their source window ignores if the `mouse_xy`- or the `read_pix`-unit is operand of a `virtual_unit`. Take this into account when using one of these units as operands.

18 Additional Information

You can find additional information in the following postscript files:

Neo Tutorial: A HTML document explaining *Neo* in about 20 lessons, with numerous up-to-date examples. Located in the `doc/tutorial` directory. **The best place to start.**

Neo in 20+ Pages: short and more up-to-date documentation of *Neo* . File `neo_primer.tex` in `man/tex`.

NST in 8+ Pages: short documentation of the underlying *NST* system. File `nst_primer.tex` in `man/tex`.

The NST Programming guide: detailed description of *NST* programming File `nstdoc.tex` in `man/tex`.

Neo/NST Technical Synopsis: an overview of the Neo/NST system architecture. File `techdoc.tex` in `man/tex`.

The tools Mku+ and Mku++: a description how to wrap C/C++ classes to make them available as *NST* units. File `mkudoc.tex` in `man/tex`.

The Nst loader library libnstld.a: a description of a library to instantiate Neo circuits in C/C++ programs from *Neo* their circuit files File `nstloader.tex` in `man/tex`.

Versioning for Loadable Units and Folders: description of the *NST* versioning scheme for loadable units and folders. File `versioning.tex` in `man/tex`.

19 Neo editor keys

(Available with F1)

```
Ctrl-A,E  goto begin/end of line
Ctrl-B,F  one letter backward/forward
Ctrl-C    copy selection into paste buffer
Ctrl-D,U  one letter downward/upward
Ctrl-G    goto token or find a token
Ctrl-J    invoke external editor $MYEDITOR
Ctrl-K    kill line (into paste buffer)
Ctrl-L    cut all
Ctrl-N    find next occurrence of selected text
Ctrl-H    backspace erase
Ctrl-P    find previous occurrence of selected text
Ctrl-Q    query man page of edited icon
Ctrl-R    delete current char
Ctrl-S    substitute selected text by contents of buffer and progress to
next/previous occurrence
Ctrl-T,Z  goto top/bottom of text
Ctrl-V    insert (and empty) paste buffer at cursor
Ctrl-W    if cursor is on bracket: goto matching bracket (includ-
ing comment
                delimiters)
Ctrl-X    kill selection (into paste buffer)
Ctrl-Y    yank paste buffer
```

Index