Neo in 20$+\epsilon$ Pages.

Helge Ritter, March 28, 2002[1]
Technical Faculty, Bielefeld University

Synopsis: *Neo* is an interactive environment for the graphical construction and execution of *"data flow diagrams"* (*"circuits"*) from NST-units. NST units appear as *"icons"* that can be manipulated with the mouse Execution of icons is in left-to-right (=reading-) order. There is a concept of *"operator icons"* that use one or several of their successor icons (the icons to their right) as *"operands"* for their own computation/action (simple, but important examples are various *iterators* for repeated execution of a group of units, or operators such as the `if_op` unit that execute their successor units only if a particular logical condition is met at their own input(s)). The usual left-to-right execution mode is complemented by an *adaptation mode* that uses the opposite (right-to-left) processing order (this mode is particularly useful for working with adaptive NST-units that require some form of *error backpropagation*, see NST-Manual). Created circuits and their state information can be saved to and loaded from files. Subcircuits intended for re-use can be collected in *"circuit folders"*. Additionally, circuits can be translated into C-source code that can be compiled and then executed without Neo.

1 Introduction

To start *Neo* , issue the command

```
lneo
```

This will start the loop wrapper for neo. Neo will bring up four windows, as shown in Fig.1 (which in addition shows a further window from a loaded application in the top middle). You can now create NST-units by selecting with the mouse (left button) the corresponding icons in the module window on the lower left (or in the lower part of the circuit window) and dragging them into the middle "circuit window". A dialog window will appear and require you to provide parameters needed to create an instance of the selected NST-unit. For some units, a second dialog window will query additional parameter settings. The newly created instance will appear as an icon in the middle circuit window. Icons in this window show colored *connectors*. Connectors on the left side of an icon always denote inputs, connectors on the right side always outputs. The little square at the bottom of each icon is not a connector; if it is selected with the left mouse button, the icon with the corresponding NST-unit will be deleted. The top row of each icon shows four little squares allowing to select a number of frequent operations for its NST-unit, cf. below.

Several units can be connected into a *dataflow diagram* by drawing wires between their connectors (Fig.1 shows an example). Connectors can be connected if they are of compatible type and if one of them is an output and the other an input. Each connector provides access to a set of one or several *pins*, each of which can hold a data element of a particular type. Yellow, red, green, blue and magenta connectors consist of pins that are all of the same type. Each pin of a yellow connector holds a single float value, each pin of a red, green or blue connector holds a vector of one or several float, integer or byte values. If the pin is surrounded by a brown rectangle, the vector has a variable number of elements (*"dynamic array" connector*). Finally, each pin of a magenta connector holds a single text string. The number of values in a pin is called its *dimension*, and only pins that are of the same dimension and are either both numeric or both of type text string can be connected (data transfer between connected pins of numeric but differing data types will tacitly involve a conversion operation that may lead to rounding and clipping when the destination pin is of a a less capacious type).

[1] Compiled: March 28, 2002

To connect two pins requires first to draw a wire between their connectors (select first connector, hold left mouse button down while moving to the second connector, release left mouse button).

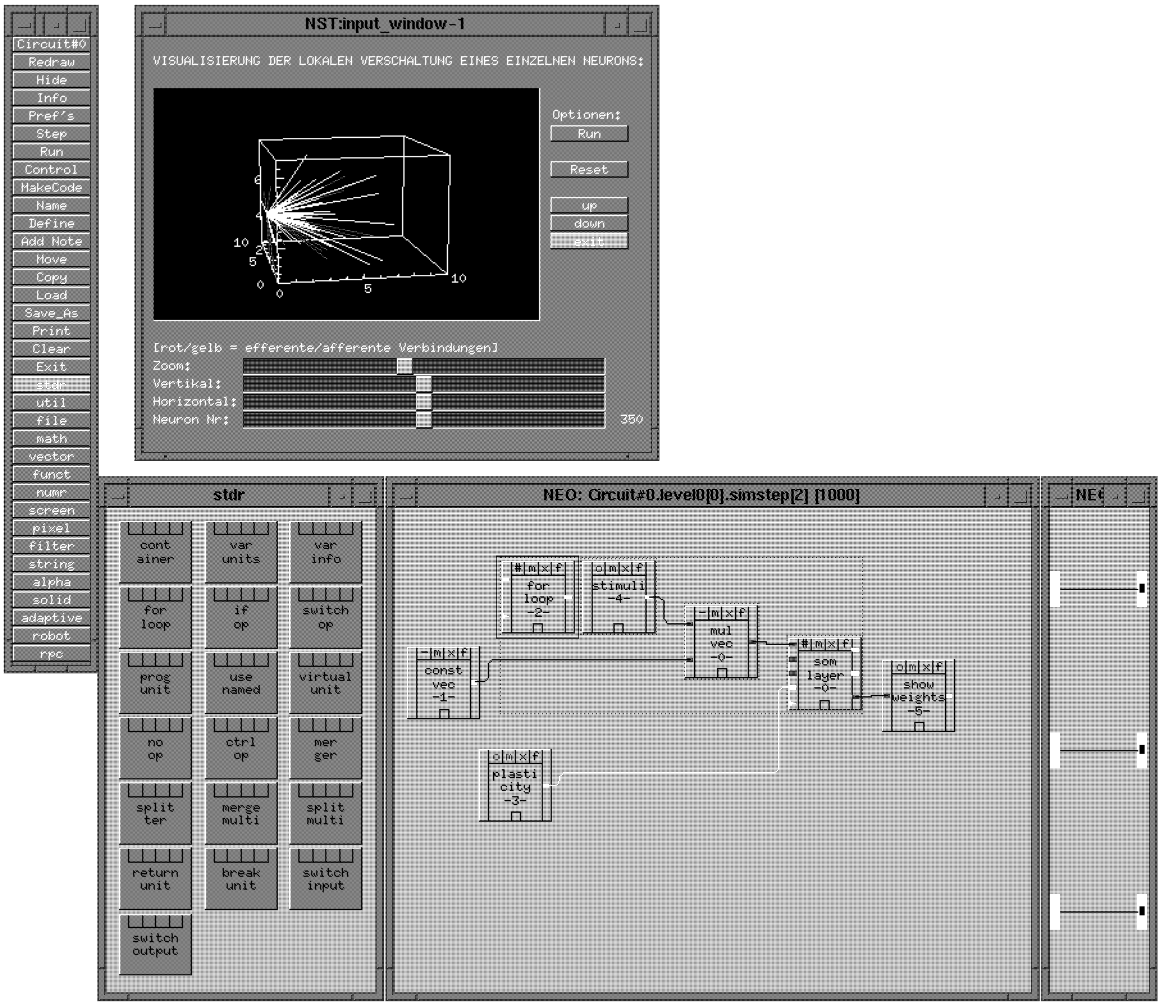

Fig.1: *Neo* user interface.

This will bring up in the rightmost *wire edit* window a detailed view of the pins of the source and of the destination connector (this time with colors indicating the types of single pins). Additionally, if both connectors have the same number of pins and all pins in corresponding positions are type compatible, *Neo* will suggest and display in the wire edit window a full set of connections between the pins at corresponding positions in both connectors (the wire edit window in Fig.1 shows this for a wire with three fibers). The desired detailed connection can then be made by inserting or deleting connections (select with the left mouse button) in the wire edit window, as needed. Returning into the circuit window with no connection left in wire edit window will make the corresponding empty wire disappear from the data flow diagram.

Most additional operations are issued with the left mouse button: selection of commands from the leftmost menue bar, moving, selecting or deleting icons, selecting connections for editing in the wire edit window and deleting connections, if the `'d'`-key is held down simultaneously (this is only required in the circuit window; in the wire edit window simply selecting a wire will also delete it). A brief summary of these operations is given below.

2 Neo Objects and Commands

Icons are dragged in from the folders that can be opened in the left module window (*New*: with the command line option `-ModuleWdwOn 0` the separate module window is absent and instead integrated into a larger circuit window. Use the **right** shift key to toggle the module window). Static or dynamic folders contain compiled objects and are thus fixed. Circuit folders can be changed at run-time by dragging into them additional icons from the circuit window (this will place a copy of the icon in the selected folder for later use) in or by using the `Load-command` (see below). Executing an icon transforms the data at its input connectors (at its left) into result values at its output connectors (at its right). Icons are executed from left to right with the *Step* command (executes all icons in the currently visible circuit), or, in arbitrary order, by clicking their `'x'`-marks individually. If an icon shows a `'='`, it has parameters that can be set interactively. If it shows a `'o'` or a red `'+'`, it contains subicons and can be "opened" (to return from an opened icon, press the right mouse button). In case of a red `'+'`, the icon is a "class container" (cf. below), otherwise, it is a normal container. The `'m'` makes a new, modified version that replaces the current icon (NOTE: *any state information of the previous icon is thereby lost!*). Selecting the bottom square deletes an icon. `'c'` stands for "commands" and allowsto invoke an (extensible) set of commands (cf. section "Defining Commands"), such as file-i/o of internal variables (allowing, e.g., to restore previously saved state from file) or setting a verbose level for error messages and warnings from this icon.

PREVIOUS

Class containers are identified by a red `'+'` and offer a number of methods in the form of named subunits that are fixed (cannot be moved or deleted). Each method (subunit) can be invoked via a `method` (or `use_named`) unit that must reference the to-be-invoked method unit via its path name, making it into a "proxy" for the referenced method unit. The `prog_unit` and the `tcl_prog` unit can declare some of their functions as *public*. This makes the so-declared functions callable as method subunits of the declaring `prog_unit` (or `tcl_prog` unit) (which thereby becomes a class container). Class containers are also the proper means for encapsulating `C++` classes as NST units (for a detailed description cf. the manual `mkudoc.ps`).

PREVIOUS

Selections Selection of a (sub)set of icons and/or annotations (e.g., for copying, deleting or moving them) is indicated by highlighting. The status (selected or unselected) of an icon or annotation can be toggled by clicking at it while simultaneously holding the shift key down. Selections of all objects in a rectangular region can be made by indicating diagonally opposite corners of a rectangle with the mouse (hold left button down while moving to the opposite corner).

Connectors receive input or offer access to output values. They consist of one or several pins. Each pin holds a numeric value or a text string, and the color of the connector indicates the type of its pins: yellow=all pins hold scalar float values, red, green, blue = all pins hold float, int16 or byte vectors, resp., magenta = all pins hold a variable length text string, white = a connector whose pins are not all of the same type. Selecting a white connector shows the types and dimensions of its pins in the title bar of the circuit window (with letters `scift` indicating the types scalar float; byte, integer, float vector; and text string. If a connector is surrounded by a brown frame, it represents a *dynamic array*with a variable number of elements.

PREVIOUS

Pins are the constituents of connectors. Each pin can hold a single *scalar value*, a numeric (`float`, `int16` or `byte`) *vector* or a *text string* (cf. above). Many units have a special input pin ("control pin", indicated by the shape of an arrow) that allows to disable execution and adaptation of the unit when given a zero value. The default value a this pin is 1 in most cases, other non-zero values may select particular control options and are unit-specific.

Inspecting/setting values For quick inspection of the first few and last few values in a connector (that has either only a single vector/text pin or that has only scalar float pins) select the pin with the left mouse button: this will make the values temporarily appear in the title bar (if the connector has several pins, instead of their values only their types will be displayed). A larger number of output values can be displayed by connecting a `print_vec`, `print_mat` or print_str unit. Input values can be set by connecting a const_vec unit or a suitable input_window unit.

Wires are used to connect icons into dataflow diagrams. This is done by dragging with the pointer while holding the left button down. Holding the d-key down while selecting a wire deletes it. A wire can consist of several "fibers" that will appear and can be edited in the right wire edit window when the wire becomes selected with the left mouse button. Arbitrary connections at the "fiber level" can be made between yellow connectors. Text string connectors can only be connected among themselves. Vector connectors can be connected, if their dimensions match.

c-Menue Each unit carries a c-symbol. Selecting it will bring up a little menu from which commands can be selected. The set of selectable commands can be interactively changed and new commands (which must be implemented in the form of named subunits) can be added. For a description, see Section 11.

PREVIOUS

Type conversions. If pins of differing types are connected, *Neo* automatically takes care of proper type conversions. This may lead to rounding and/or clipping if the destination pin is of a less capacious type. It is also somewhat slower than a connection between pins of the same type: each type conversion necessitates an elementwise copy operation[2] while otherwise data access is by reference, without any copying overhead.

Var units offer, in addition to wires, another means for transporting values from one place to another: A var_def unit defines and holds a named data set (consisting of one or more records of a user-defineable structure), and var_set (var_get) units can access the data set by its name from different places in the circuit diagram to write (read) selected records. To preserve the data in a var_def unit between save/loads, specify after the variable set name one of the options %S (for formatted save) or %B (for faster, but unformatted, binary save). For some further details, see Section *Encapsulation* below.

Nesting hierarchy. Icons can contain (indicated by an 'o') subcircuits. Executing an icon will execute the subcircuit that it contains. Selecting 'o' "zooms" into its subcircuit (makes it visible in the central circuit window for inspection or change); pressing the right mouse button leads back to the next higher level up. At the topmost level, the right mouse button will bring up the dialog window for the top-level circuit diagram, if there is one defined.

Graphics units such as write_pix, plot_xy and plot_vec open and maintain a graphics window that other units (such as the draw_sym or the mouse_xy unit) can use. To identify the destination window for the operations of such units (referred to as "window clients") requires to specify (in each window client's dialog window that appears when the unit is created or modified) either the destination window title (recommended practice), or an integer "destination window offset" (not recommended; $-n$ specifies the n-th window that is in execution order before the window client, n specifies the n-th successor window).

Operator units are marked by a red surrounding square. Such units, when executed, act on one or several of their successor units (their "operands"). A dotted "scope rectangle" encloses the operands that are affected by a particular operator [3]. For most operators, the vertical borders of their scope rectangle can be shifted with the left mouse button to change the current set of affected operands (the upper and lower border always adjusts itself). Most operator units iterate their operand(s) a number of times, often providing at their outputs some iteration-dependent set of input values for use by the iterated operand(s). Some operator units also

[2] however, if several wires can share the same conversion operation, they will do so, i.e., the conversion is then carried out only once.

[3] Operators that are created by embedding a virtual_unit in a container or void_unit currently have no scope rectangle displayed; their action on an operand is only indicated by a dotted frame around each operand.

expect some of their inputs to be connected with output(s) of their operand(s) to receive after each iteration some result value(s). Many high-level operations can be nicely formulated in this way.

%-Options. The constructor dialog window (in NST: the constructor function) of many units has a text string parameter that allows to set various options. This follows a similar scheme as in the format specification for the familiar `printf` and `scanf` routines of the `C`-language: Each option is activated by a "`%`-directive", which is a token[4] that begins with a `%`-character and ends with a command letter that identifies the desired option. The space between the initial `%` and the delimiting command letter can be taken up by zero or more (currently, maximally four) colon-separated numeric parameters (each parameter may also be an arithmetic expression; the expression may also contain `@i` or `$i`-variable identifiers, c.f. Section *Variables*). The set of `%`-tokens that is understood by a NST-unit and their associated command letters and parameters must be looked up in the documentation of the unit (often, the dialog window shows a synopsis).

by-reference units units fetched from a circuit folder can be made "by-reference". On later loads, instantiation of such units will require the availability of the folder from which they were fetched, and the created instances will automatically inherit any changes made in the mean time to the implementation in the folder. This can simplify updates; however, as with libraries in general, the responsibility for the changes not to conflict with the usage of the unit is entirely in the hands of the implementor.

PREVIOUS

By-reference units are marked with a red hat-symbol `'^'` in their top left corner (this can be switched on and off on icon symbols displayed in the folder window to select their instantiation mode). and their "innards" are unmodifyable under *Neo* .

initializer units The name `_init` is special: when this name is given to an icon, it will become an *initializer*[5]: whenever *Neo* loads a circuit, or instantiates, modifies or copies a unit, it will (after the load/instantiation/modification/copy of the entire object is complete) traverse all its units, look for initializers and execute these in the order in which they are encountered. This allows to embed initialization operations into `_init`-named containers, allowing, e.g., to build circuits such that they auto-start after loading.

PREVIOUS

3 Some important units:

There are some units that provide frequently required, general operations within circuits:

container. Recognized from the `'o'`-mark on an icon. Conceptually, container units are *Neo* 's counterpart of subroutines. They can accommodate a subcircuit that is executed (adapted, ctrl'ed) when the container unit is executed (adapted, ctrl'ed). Data can be passed via input or output connectors, additionaly the *Define*-command (cf. below) allows to add a user-defineable dialog window for remakeing the container's subcircuit with modified construction parameters for its units. The subcircuit inside a container can be inspected by "opening" the container (select its `'o'`-mark).

print_vec, const_vec to have a rapid look at some values or to provide some values to an input connector. Similar: `print_mat, const_mat, print_str, const_str` for matrices and text strings.

splitter, merger to split vector connectors into smaller parts or merge such parts again together.

[4] actually, the "token" need not be delimited by white space.

[5] this applies also to method subunits, such as, e.g., public functions named `_init()` in the `prog_unit`

switch_input, switch_output, switch_str allow to dynamically select among a set of identical inputs or to dynamically write to one of a set identical outputs. Not to be confused with the `switch_unit` that provides an analogon of the `C` `switch`-statement.

for_op. Iterates a (specifyable) number of successor units to its right. Important example of the concept of an "operator" that does something with a number of operands sitting to its right.

break_unit. To escape from a `for_op` loop.

if_op. Conditional execution of successor unit(s) to its right.

return_unit. To return from a container before executing all of its subunits.

prog_unit for quick help when something is missing: provides a little C-interpreter that allows to make a unit "on the fly" by entering a C-fragment. The `#loadlib` directive allows to make functions from self-written shared libraries callable from the `prog_unit`. In this way, functions written in `C` or `C++` can be made available in *Neo* with very little effort, using a `prog_unit` with a `loadlib` directive as a wrapper.

tcl_prog a similar unit for the Tcl/Tk language.

PREVIOUS

matrix_algebra. Very powerful math unit that allows to specify matrix/vector computations in a compact, C-like math notation. Conveniently combines the functionality of a large subset of other units. Alpha (because small syntax changes are still planned).

var_units allow to deposit data values in one location and retrieve them in another (for details, see Section *Encapsulation* below and the man page).

use_named allows to use (=execute, adapt, ctrl) the same unit in several places, using a name by which the unit is identified. To use a unit in this way requires to name it using *Neo* 's *Name* command (for details, see Section *Encapsulation* below and the man page).

use_method similar as `use_named`, but defines its interface automatically according to the referenced unit.

scanf, printf Units that implement the familiar C-statements. The `scanf`-unit contains extensions to load image data from portable pixmap or graymap (`ppm/pgm`-) Files.

window_to_eps make postscript output from 2D-graphics windows.

counter does what it says.

scale_vec handy for converting ranges, such as 0-1 to 0-255 etc.

serial_comp unit for carrying out frequently needed serial computations, such as forming the sum, the average, the max or the min of a sequence of (vector-) values.

input_window to make a window for data entry. Window layout is specified as plain text, widgets such as fields for numeric or text entry are specified by embedding suitable format specifiers in the form of `%-tokens` into the window text, much in the same way as familiar from C's `printf` function. This is also the syntax that is used when specifying dialog windows for user-defined units, or when programming a Neo-wrapper for a NST-unit.

remake_unit allows to write new parameters into the definition and control window of a successor unit and then to remake it according to the new specification.

array_of will instantiate an array of units, with each element modeled after the successor unit of the `array_of` unit. Execution of the `array_of` unit will take an index value at its input and "map" the array element (which is a unit) onto the successor instance (i.e., the successor acts as a "proxy" that can be moved over the array elements).

PREVIOUS

dummy when a circuit contains icons that are not known to *Neo* , *Neo* will complain and insert "dummy icons" as placeholders instead. These will have the same interface as the original icons, but they will do nothing, when executed, and they also cannot be modified (therefore, remaking circuits with dummy icons may lead to unpleasant results). However, circuits with dummy icons can be edited in the usual way (except that no parameter modifications to dummy icons are possible) and saved back to file. This will revert the dummies to their original definition. Therefore, when later *Neo* is provided with definitions for the unknown units and then loads the edited circuit, the circuit will appear as fully normal, with no trace of the former dummies left behind.

PREVIOUS

If a unit in the module window shows a letter 'e' in its top row, selecting it will load an example circuit for that unit. If it shows a '?', you can bring up a manual page by selecting the `'?'`-mark.

4 Neo Command Summary

Circuit Allows to switch between up to 5 circuits that can be held simultaneously in *Neo* . Switching to a different circuit deactivates the current circuit (but fully preserving its current state in memory) and activates the new circuit (becomes accessible for commands and appears in the circuit window).

Redraw does what it says.

Hide hides Neo's module, circuits and wire edit windows. To have Neo come up with its windows hidden, specify the `-hide` option on the command line.

Info looks for a file with the same base name as the currently loaded simulation, but suffix `.INF` and brings it up, if it exists. `.INF`-files are plain ascii text files that can be written with any editor. Useful for providing some online help with a simulation (since the introduction of annotations, this is almost obsolete and may disappear from future versions).

Search Allows to search for a unit (either in the circuit or in the module window) or to search and optionally replace strings in entry fields of dialog windows (create and parameter window or user defined dialog window) associated with a unit.

Prefs Set preferences for global parameters: number of steps done by the Run command, adaption yes/no, printing of trace information.

Step execute a single step of the currently visible circuit or subcircuit.

Run execute several steps (how many is set with `Prefs`) of the currently visible circuit. Can be interrupted with the `ESC`-key. Note, however, that the `ESC` key can only interrupt between successive steps. To allow interrupts also within loops, set the corresponding option in the *Prefs*-Command (this however, comes with some performance penalty).

Control send a control call to a single, to all visible or to all units.

MakeCode *(Currently unsupported)*generate source code and an accompanying Makefile for the loaded circuit diagram. The layout of the code can be customized to some extent by specifying a code template (see Neo-Manual for details).

Name replace the label of a unit by a *name*. This allows to execute the unit from within any place of the current container or any of its sub(-sub-sub..) units, using the `use_named-unit`. If a unit has a name, the name is displayed instead of the label. Note: only a name makes a unit callable from other places (labels and names are considered as different concepts; syntactically, the difference is that labels contain a #-char, while names do not). To un-name a unit (bring back its label), give it the blank name.

Define This command can serve three purposes: (i) to define a dialog window for parameters of a subcircuit, (ii) to change an existing definition of such a dialog window, or (iii) to make changes (using the AddField/DelField buttons) to the interface of a container for which a dialog window has been defined (so that the usual dialog window for modifying connectors is no longer directly accessible). For (i) and (ii), the subcircuit must be visible in the circuit window. `Define` then brings up a form in which the desired layout of to-be-defined/modified dialog window must be specified (in the form of a "format string" with text and widget tokens as understood by the `input_window` unit (see its man page for details)). Each entry "slot" that is defined in this way gives rise to a "`@`-variable" (the first entry slot to `@0`, the next to `@1` and so on). These variables (numeric and/or text strings are their possible types), or expressions of them, can then be used as entries in the dialog windows of the units of the subcircuit. Additionally, one may define for each `@`-variablea more readable name (e.g., `foo`, which is then used in the form `@foo` or `@{foo}` (the latter to avoid mergings with an adjacent string)). After a dialog window has been defined, it replaces the default dialog window of the container unit that holds the subcircuit. Thus, when now the `'m'`-option of the container unit is selected, the newly defined dialog window comes up and allows to set values for the parameters of the subcircuit, which then is remade accordingly. A blank definition removes the dialog window from the container and restores its default dialog window for modifying connectors (which is no longer directly accessible, but only via the `Define`-command, as long as a user-defined dialog window is defined). When the format of a previously defined dialog window is edited again, all original definitions for input fields have been replaced by tokens `@i`, indicating the variable names by which they are referenced. To add further input fields, however, requires always to specify the `%-`tokens required by the syntax for the `input_window` unit. For an example, see Section *Variables* below. **NEW**

Add Note. To add short comments into a circuit. They may contain additional text that can be displayed (and then edited) in a separate window by selecting the body of the annotation with the left mouse button. Those annotations that show a little "head" can be moved around by dragging it. To move an annotation that has no head requires to edit it and add a head temporarily. Selecting, moving and copying works as for icons. Annotations that start with the special tokens `"NAME: "` or `"MANPAGE:"` are interpreted and displayed in a special way: If the title starts with `MANPAGE`, the text will be used for the man page when the enclosing container unit is dragged into a circuit folder, for the significance of the `"NAME: "`-token, see Section *Variables*.

Move allows to move the currently selected icon or subcircuit to a different location (this may also be within a different circuit, activated with the *Circuit*-button). To select an icon/subcircuit use Shift-Select or Left-Alt-Select, or delineate a suitable bounding rectangle with the left mouse button.

Copy similar to move, but duplicates. The copied objects are automatically marked as selected (e.g., for a subsequent move).

Load Load a circuit or a folder.

Loading a circuit: If no suffix is specified, the circuit of the given name (suffix `.NST`) will be loaded (the old circuit will be deactivated and can be revisited with the *Circuit*-switch button, if there are less than five circuits in *Neo* 's memory; otherwise you will be queried to make space for the new circuit) and initialized with its accompanying data file (suffix `.nst`). To suppress using the data file, specify the `.NST` suffix. You can set up a list of to-be searched file paths in the unix environment variable `.NSTPATH` (must be defined in `.profile` or `.cshrc` etc.; Example: `setenv NSTPATH $HOME/mystuff:$HOME/furtherstuff`) NOTE: if there are both ungzipped and gzipped versions of a circuit around, the unzipped version will be loaded!

Loading a folder: If the specified name is prefixed with a plus `'+'`, Neo will interprete it (up to the next `'+'`-mark; several folder names, concatenated by `'+'`-marks, can be given simultaneously) as the name of a *loadable* or *dynamic folder*. It will add an accordingly labeled folder button to the main menu bar and bring up in the folder window a display of the specified folder's contents. If the folder had not existed before, Neo will have made a new (empty) folder directory, named after the specified folder; otherwise, Neo will locate the directory or shared library associated with the specified, existing folder and display its icons in the folder window. In the case of a circuit folder, new circuits can be added to the folder by simply dragging their icons into the folder window (this will save them permanently in the associated directory). The order in which the icons appear in the module window can be specified by placing a file named `.NEO` in each folder directory. This file must contain the to-be-ordered names in the desired sequence (icons not mentioned in `.NEO` are placed last. To use a folder directory or a shared library that is different from the folder name (which appears on the button), postfix the folder name by a `'='` sign, followed by the full path name of the to-be-used (or to-be-made, if non-existent) folder directory or shared library (example: `+myfolder=/homes/mystuff/folder1`). Note that for a shared library the folder name is compiled in and therefore fixed, however, the path may vary. To un-load a circuit folder, specify its name again, but prefixed with a minus (`'-'`) sign.

IMPORTANT: don't use names containing plus or minus signs, since these interfere with the use of these symbols to add/remove circuit folders!

Save_As Save the currently loaded or the currently visible circuit to a file. If it is a subcircuit, it is saved in an "encapsulated" form, such that it can be loaded as a single icon later on. The circuit diagram and data of a circuit can be saved into a single file, or into two separate files (this allows later to load circuit and data separately, i.e., to initialize the same circuit with different data).

Print (this command has become superceded the *Html* command below.

Html Generate (or updated previously generated) HTML documentation for the current circuit and its subcircuits. For details, see section 12 below.

Clear Delete the currently visible circuit or the set of selected objects.

Exit To do something else.

The remaining buttons belong to folders.

Many actions, such as `Copy` and `Move`, can be used for a single icon or for an entire group. The currently selected icon or group appears highlighted. To include further units in the current selection, use the left mouse button, holding the `SHIFT`-key down simultaneously, or delineate with the mouse a rectangular bounding area for the intended selection. Group moves in the current window are also possible by holding the `SHIFT`-key and dragging with the left mouse button down. Copies and moves can be across different circuits held simultaneously in *Neo* . With regard to selection, copying and moving, annotations are treated in the same way as icons are. Pressing the `'a'` key while in the circuit window will toggle the icon display between showing *Neo* 's icon labels and `NST`'s original unit names.

Further commands.A number of additional commands is available via the keyboard when the cursor is within a Neo window:

F1 Bring up help window

F12 collapse all Neo and application windows into a single button that allows to return to the original state.

PREVIOUS

NEW

'A' Alternate inscriptions: switch display between display of Neo labels/names and display of NST unit types. Note: the *Search* command always searches among the displayed strings.

Alt-'-' Enclose subset of selected icons in a newly created container.

Alt-'+' Undo previous *Alt-'-'*. The undo-possibility will only persist as long as there are no other changes are made to the circuit.

Alt-" (needs 2 selected units) Swap pair of selected units if the respective subsets of their pins that are connected to wires permit the swap.

5 Encapsulation

Container units are provided to encapsulate subcircuits and to provide a modular structure (use the *Alt-'-'* command to encapsulate a group of selected units, *Alt-'+'* to undo). In the simplest case, use of resources within a container is restricted to executing the entire container, thereby executing all its subunits. 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]). 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. 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` 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 Configuring *Neo* and command line options

Neo can be configured with two configuration files `.neofolders` and `.neo`. [7]. Both are expected in your home directory, but if you have files with the same name in your current directory, they will be used instead. To use instead of `.neo` a different initialization file, specify it with the option `-@filename` (again, first the current, then the home directory is searched, unless `filename` is an absolute path or starts with `./` or `../`). **NEW**

6.1 Loading folders

The `.neofolders` file allows to extend *Neo* by loading additional folders into *Neo* during startup. Each line in .neofolders specifies a to-be-loaded folder in the following form:

```
FolderShortName  FolderFileName  # optional comments
```

[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

[7]If you rename the executable to a new name `newname`, it will search for an initialization file `.newname` in place of `.neo`

This line would make *Neo* look for a folder that is stored as the file FolderFileName, load it into memory and add a selection button labeled FolderShortName to the menu bar. FolderFileName is either the name of a directory holding the units in the folder as a collection of *.NST/nst circuit files (*"circuit folder"*) or it is the name of a shared library containing the units of the folder (together with their *Neo* -wrappers) in the form of compiled objects (cf. section *Extending Neo* below). In this case, the FolderFileName should end with a suffix .so and *Neo* will dynamically link the library at startup (*"dynamic folder"*)[8]. Generally, access of shared libraries searches the path specified in the LD_LIBRARY_PATH environment variable. When the environment variable NST_LD_LIBRARY_PATHis defined, it will be used instead of LD_LIBRARY_PATH for all NST accesses to shared libraries. Note also that if the loaded shared library needs to access other shared libraries, their paths must be contained in your (NST_) LD_LIBRARY_PATH-environment variable! **PREVIOUS**

After a dynamic folder has been loaded, there is no difference to a static folder.

6.2 Customizing *Neo*

The .neo file can contain any subset of command line options accepted by *Neo* (for a listing with or without their default values, issue neo -H or neo -h). At least the two parameters -BorderWidth <Number> and -TitleBarWidth <Number>. <Number> should be set to the values used by the window manager in order to avoid undesireable "window crawling" when windows are closed and reopened. Example of a .neo-file:

```
# path prefixing all directory/files specified below:
-NeoHome /homes/myhomedir/myneodir
# template files for source generation:
-MakTemplate neosrc/NEO_MAKTEMPLATE.mak
-SrcTemplate neosrc/NEO_SRCTEMPLATE.c
# user interface settings:
-BorderWidth    5       # width of window manager's decoration
-TitleBarWidth 15       # likewise, for title bar
-Font 1                 # sizes 0 1 2 3 available
-CircuitScaleX  0.63    # vertical display shrink factor
-CircuitScaleY  0.63    # horizontal display shrink factor
# position (lower left corner) of Neo's main windows:
-CircuitWdwXll 275
-CircuitWdwYll 0
-ModuleWdwXll 78
-ModuleWdwYll 0
-WireEditWdwXll 790
-WireEditWdwYll 0
```

If a circuit has a definition window at the top level, its @-Variables can be set on the command line in the style @0=value @1=value...

Obtaining a long circuit window:add the command line option -ModuleWdwOn 0. This will make *Neo* come up with a "long" circuit window. With this option, the module window will only show up as a long rectangle in the lower part of the circuit window when the *right* shift key is pressed or when the cursor picks the small rectangular bar at the bottom at the circuit window. Circuits saved from a long circuit window can be reloaded into *Neo* started with the short circuit window and vice versa: in each case, the icon, wire and annotation positions will be rescaled appropriately. But note that *this is not perfect:* since the size of the icons themselves will not be subject to such scaling, there may occur small shifts of annotations relative to icons (has no effect on function) [9]. It is recommended to use the long circuit window for new work. **NEW**

[8]This feature requires that *Neo* was compiled with the options -rdynamic -DDL; it may not be available on some systems, such as SunOS.

[9]If icons are horizontally separated by only a single pixel in the long circuit version, reloading such a circuit in the short version may destroy this difference due to rounding to an integral pixel position and can in this seldom case affect the function of a circuit.

An additional file `.nstsyntaxcolors`can be used to specify the colors that *Neo* uses when doing syntax coloring[10]. NEW

To execute a *Neo* circuit without opening any *Neo* -windows (e.g., for running batch jobs) invoke *Neo* with the command line options `-hide -x <NumberOfExecSteps>` (however, this will not get rid of any graphics windows in your application!).

To bring up *Neo* with a circuit and add several folders from the command line, issue *Neo* with the command

```
neo -i NameOfCircuitFile+NameOfFolder1+NameOfFolder2
```

and similarly for more folders.

To display trace information while loading a circuit, use the `-TL` option (it can also be set in the *Load* command window). `-debug 1` switches on some diagnostics.

7 Editing

Edit commands for numeric and text fields are:

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 (including comment delimiters)
Ctrl-X	kill selection (into paste buffer)
Ctrl-Y	yank paste buffer

Select, cut and paste with the mouse within the same window work or with other X-windows works as usual. The command Ctrl-L has now been redefined to move the entire text into the cut buffer (was identical to Ctrl-K before).

The paste buffer keeps its contents also when a new window is used. *Important Note:* when an illegal entry is made (or when a partial entry looks as if illegal), editing via the mouse commands may be blocked. The above editing commands, however, continue to function and can be used to complete or correct the entry. PREVIOUS

8 Cursor-Triggered Execution

[10] For details, consult the description in file `nst_syntax.c` PREVIOUS

For windowed applications, here is a simple way of fakeing some of the conveniences of multitasking at the *Neo* level by letting pointer enter and leave events trigger execution and stopping of *Neo* circuits that are associated with the windows.

To this end, the `Define` command allows to specify for each container unit the name of a `input_window` instance to trigger execution of the container unit whenever the cursor enters the specified `input_window`. To allow also to stop execution again when the cursor leaves the specified `input_window`, a button of the specified `input_window` can be specified that will become pressed when the cursor leaves the window (it is then the task of the implementation to use this automatically generated button press suitably to achieve the desired stopping or whatsoever effect). The specification in the `Define` dialog takes the following form:

```
windowname %f:p:v
```

where `windowname` is the name of the `input_window`, `f` and `p` are the field and pin number of the selected button, and `v` the value to which the button shall be set (defaults are f=0, p=0, v=1, if no values are specified). `@i`-Variables can be used within the above specification; in this case, however, `@i` refers to the i-th parameter in the Creation dialog window of the container unit for which the `input_window` is specified (and not to the i-th parameter slot of the creation dialog window of the *enclosing* container, as usual (cf. Sec. 10).

9 2D Graphics

A number of graphics units (including `input_window`, `output_window`, `plot_vec`, `plot_xy`, `plot_series`, `write_pix`) open a graphics window when they are created. These units offer a range of basic graphical displays and are referred to as *window hosts*. Other graphics units (including `draw_sym`, `draw_pix`, `draw_str`, `draw_mesh`) don't have their own graphics window but can draw into the graphics window of one of the units from the first group. Since these units use the window of a window host, they are called *window clients*. When a window client unit is created, its host (=output) window is selected by either specifying the window title of the desired host window (this assumes that window titles are chosen uniquely), or its positional offset with respect to the created window client (thinking of all windows as sequentially ordered by their NST-execution order). The `mouse_xy` unit is similar to a window client, but it (in addition to drawing options) allows also to retrieve positional information of the mouse pointer with respect to the selected host window.

The `input_window` and `output_window` units enjoy a special status: Normally, they are not counted as window hosts (and are thus not useable as drawing areas for the window client units). However, the appearance of a format token such as `%400:300=mywin` reserves a (rectangular) *named area* (of name `mywin` and 400 pixel width and 300 pixel height in the example) in the window layout[11]. Each such named area (there can be several of them for a single `input` or `output_window`) can then accommodate the drawing output of any of the other 2D-drawing units[12], just by specifying the name of the named area as the window title when the unit whose graphics output shall be redirected is created (in the case of a window host, the named area will assume the role of the host window and the creation of a separate window will then be suppressed). Any options, such as autoscaling and also cooperation with the `mouse_xy`-unit are properly taken over by the named area within the `input/output_window` unit. Regarding window counting, each named area has to be treated as if there were an extra window host (Note: redirecting the output of more than one window host into the same named area may lead to undefined behavior. Redirection of one window host and several window clients is ok.)

[11] only supported under X11 and/or OpenGL graphics

[12] to display a RGB-window, e.g., for the `write_pix` and `draw_pix` in RGB-mode, you must replace the `=` by a `=:` instead, for a 3D-Graphics window by `=::`

10 Variables

Neo provides *variables* to simplify the creation of circuits whose parameters are related with each other, e.g, by some formula.

Variables are tokens of the form `@0,@1,..` or `$0,$1,..` (i.e., `@` or `$` followed by a number). `$0,$1,..` denote *global variables* that are visible everywhere; *Neo* allows to define a single set of global variables (see below). `@0,@1,..` denote *local variables* that “belong” to a particular container or void unit and that are only visible for the (direct) subunits of this container or void unit. Neo allows to define a separate set of local variables for each container unit. Units that access local variable indicate this by some yellow color in their deletion-field. Red color in this field indicates that the unit or some nested subunit accesses one or several global variable(s). For example circuits, there is a special global variable `$EXAMPLE_DIR` whose value is always the directory path from which the example circuit was loaded. A similar variable `$LOAD_DIR` is set whenever a new circuit is loaded (for the difference between `$EXAMPLE_DIR` and `$LOAD_DIR`, see below).

An Example. To see how variables can be used, step through the following example: 1. Create an empty container unit and open it. 2. Select the `Define`-command from the menue bar:[13] a dialog window appears. Enter the following:

```
value1: %5i  value2: %3.14f %n
```

You have now defined the layout of a dialog window for an integer value (initialized with `"5"`) and a float value (initialized with `"3.14"`). To display it, press the `"Test"`-button. Confirm with `"OK"` to accept your changes. The window disappears (becoming the current container's new dialog window, as you will see shortly) and you will be returned to the definition dialog window. You will now see that the format tokens `%5i` and `%3.14f` for the two values have become replaced by `@0` and `@1`, indicating two *variables* that can now be used by the subcircuits. Selecting the `Variables` button allows you to assign (optional) names to the newly defined two variables (omit this for the moment). You also could make further changes, test them again etc. Instead, Confirm with `"OK"` a second time to accept and finish your definition (`"Cancel"` would discard all your changes and restore the state before you entered the `"Define"` command). You will be returned to the circuit window, still showing the (empty) inside of the container. You can now use the tokens `@0` and `@1` as identifiers for the values 5 and 3.14, resp.: 3: create a `const_vec` unit (from folder `vector`) and enter `@0` and `@1` into its first two entry fields; next, create a `print_vec`-unit for two values and connect it (input field 0) with the `const_vec`-unit. Execute it to see the values from the `const_vec`-unit. 4. Leave the container (right mouse button) and modify it (`'m'`-mark) from the upper level. You will see that the newly defined dialog window pops up: it has replaced the `container` unit's default dialog window. Change the current values `"5"` and `"3.14"` to something else, confirm and execute the modified container to see that its subcircuit now has been remade with the new values. 5.

NEW

To use instead of `@0`, `@1` more readable names,select the `Define` command again (either from the container's c-menue, or from within the container, using the `Define` button in the main menu bar). Select the `Variable names` button. You can now enter for each variable `@0`, `@1` a textual name, e.g. `foo` and `foo1` (legal names must obey the rules for a C variable identifier). Leave `Define` with `OK` and inspect the creation dialog windows of the subunits inside the container: you will find that all occurrences of `@0` and `@1` have become replaced by `@foo` and `@foo1`, resp. From now on you can use both `@0,@1` or `@foo`, `@foo1` to denote your variables in the subunits. You also can always rename all instances of a variable by a single change in the Define command.

Finally, to bring the `container` unit back to its “virgin” state, enter the `Define` command again and then repeat step 2, but this time replace the current contents of the definition window (note that the format tokens `%5i` and `%3.14f` have been replaced by their abbreviations `@0` and `@1`) by the blank string, confirm with `"OK"` and then by `"Proceed"`: The dialog window will have disappeared and inspection of the `const_vec`-unit reveals that the variables `@0` and `@1` (or `@foo,@foo1`) have gone, and have become substituted by their most recent values.

To define global variables, repeat steps 2-4, but from the top level circuit (in this case, the inside of the container is the top level circuit and no `'m'` needs to be selected in step 4.). In this case, the variables `@0` and `@1` can also be referenced by the alternate, “global” names `$0` and `$1`. Moreover, `$0` and `$1` are visible in *all* subunits, not just inside the top level circuit.

Variables can be used in place of a numeric or character string constant in the entry fields of the creation or parameter dialog window of a unit (this requires that all variables that are used are also

[13] Equivalently you can also select the `Define` command from the c-command menu of the container icon (this saves you the opening of the container)

visible for that unit) or as parameters within the %-options used by many NST-units. If an entry field or a %-option expects a numeric value, it is also legal to specify an arithmetic expression, such as -2*(@0+@1). If a variable name is followed by a stringsuch that the variable name and the string name would merge (e.g., ${foo} followed by bar), protect the variable name with curly brackets (e.g. ${foo}bar). However, you need not worry about this when naming or renaming a variable from within the Define command, since Neo will insert protective brackets automatically, when needed. **NEW**

Variables can also be used in the text of annotations. In all cases, the number i in the identifier @i or $i of a variable refers to its (sequential) entry slot position in its dialog window.

Altering an existing variable definition. The Define command also allows to change an existing variable definition window. Previously defined variable entries appear as @i-tokens in the current definition (this is always; even when global $ variables or variable names are used). New entries may be inserted, using the appropriate %-tokens, as explained in the format syntax for the NST input_window-unit (these will appear converted to @i-tokens, when the definition window is revisited later). When an existing @i-token is removed, any current references to this variable will be replaced by their current values. When new variable entry fields are inserted, any existing references are renumbered appropriately. To remove a user-defined variable definition window of a unit entirely, replace its format specification by a blank string.

$LOAD_DIR **variable.** This is a predefined global variable whose value is the directory from which the current circuit was loaded ("." if no circuit was loaded). This allows to reference files needed by a circuit in a way that is independent of the circuit's actual directory.

$LOAD_DIR can also be used in loadable units. Since a loadable unit may be dragged into a circuit that already uses $LOAD_DIR to reference a different (namely the circuit's directory), making the newly inserted instance of the loadable unit is in this case also accompanied by a substitution of all its $LOAD_DIR references by their current value (which is the directory of the loadable unit). The $LOAD_DIR references, however, will "survive" unsubstituted if either (i) they reference the same directory name as the $LOAD_DIR of the circuit that is already present, or (ii) the current circuit is empty. In case (ii), the value "." of $LOAD_DIR for the current, empty circuit will be changed to the directory name of the newly loaded loadable unit (this allows to make changes to loadable units and write them back without losing all their $LOAD_DIR references). †[14]

Array index variables The unit that is the immediate successor of an array_of unit "sees" the special "array index variable" $#. Its occurrence will be expanded to the comma separated list of index values that belong to the array element that is currently mapped onto the successor unit[15] **PREVIOUS**

Local variables for primitive units.A dialog window that defines a set of variables can also be associated with each primitive unit. To this end, use the right mouse button while in the Modify or Create dialog window of the unit and specify the variable dialog window layout in the definition window that pops up. After an accept, the new definition window will supersede the original Create dialog. To bring the original Create dialog back, open the (superseding) variable dialog window and press the right mouse button. The variable dialog window can be removed by clearing its format definition. **PREVIOUS**

The variables of the newly defined dialog window can be used as parameters in the "underlying" (now hidden), "native" create dialog window of a primitive unit. Since also algebraic expressions are allowed, and since some of the parameters of the original window need not appear at all in the new window, the creation dialog of a unit can be completely re-parametrized (e.g, the specification of a long source text for a prog unit can become hidden, and only the specification of a few essential parameters is requested in the new window).

[14] If a circuit contains $LOAD_DIR references, Neo allows to save it only into directory $LOAD_DIR. If a different directory is specified, NEO will issue a warning and offer to replace all $LOAD_DIR references by their current values. Only then the circuit may be saved. This protects against making circuits that due to unresolvable $LOAD_DIR references cannot be reloaded later.

[15]this feature currently requires that the root circuit possesses a global variable window, even if it does not contain any variable definitions.

Variables and source code generation. Local variables of a subcircuit will be translated into a parameter list of its creation routine. Global variables will be translated into global variables of the accomodating C-translation unit. There is a restriction for variables that are embedded in text fields: for these, a full translation (into a parameter of the C-program) is only supported for those text fields that are in their dialog window preceded by a `"::"`. For all other text fields, embedded variables will be replaced by their values at the time when the source code is generated. (There is no such restriction for numeric fields).

Variables and annotations Variables `@k` or `$k` in the title text of an annotation appear in the circuit diagram expanded to their current value. This allows to make important parameter values visible at suitable locations of the circuit diagram. It also allows to make named units whose name was entered via a user-defined variable window: if a container has an annotation that starts as `NAME: something` and `something` qualifies for a name, the container will become a named unit with the name `something` (`"NAME: "` is, like `"MANPAGE:"`, a reserved keyword in the title of annotations that is interpreted in a special way).

Some important Notes. When units that use local (i.e., `@-`) variables are copied or moved into a different container in which these local variables are not visible, *Neo* silently replaces each `@`-variable by its current numeric or text value (note that global (i.e.,`$-`) variables are always visible and thus can always be kept!). This may lead to surprises when later trying to remake the circuit with different variable settings (Hint: copy the variable definition window first, using the *Define* command and cut-and-paste between NST windows).

When defining a variable for use in an entry field of a unit that is already filled with a constant value, it is recommendable to give the variable this constant value as its initial value (otherwise, you may not be able to replace the constant value by the variable; this always happens, when the different value of the variable would give rise to a re-dimensioning of connected inputs or outputs of the unit).

Note that the *evaluation of expressions* with variables (such as `2+@1`) occurs only within numeric fields. For text entries, such expressions are just particular text strings.

11 Defining Commands

PREVIOUS

Each icon can be bound to an extensible set of commands that can be displayed and invoked via a pulldown menu accessible from the `c`-field on the icon symbol. The set of available commands for an icon can be edited with the `Define` command[16]. Each command is a group of tokens, with different groups separated by commas. Only the first token of each group determines the associated command; the remaining tokens are just additional text to appear on the command's pulldown menu line. The command name can also be hidden by prefixing it with a tilde character; this allows to choose a command name that is different from what is visible in the pulldown menu.

Invoking a command in the menu starts the following three-stage search: if the command name identifies a named subicon of the command's icon, the effect of the command is to execute that subicon. If no such subicon exists and the command name is one of the NST control calls (with the optional prefix `NST_` omitted and either lower or upper case letters), the effect of the command is to apply the corresponding control operation to the icon. If also this fails, *Neo* finally checks if the command is among the following set of built-in commands: "define", "load", "save", "misc". If this fails, too, then an error message is given.

If the definition of a command contains the special directives `%i` or `%o`, these will be substituted by the current value of input or output pin 0 of field 0 of the named icon that belongs to the command[17] This allows to display in the pulldown menu also state information and parameters of the

[16] remember that for non-composite units the `Define` command is invoked with the right mouse button from within the modify (`'m'-`) dialog.

[17] To select a different pin or to control formatting, use the extended form `%a:b:c:di` and `%a:b:c:do`, resp. This will

current icon (including text strings).

New commands can added just by editing the list in the Define window. In most cases, this will be used to make the operations of particular subunits of an icon conveniently accessible from the `'c'`-menu (including, e.g., public routines of `prog_unit` instances), or to offer a (possibly hierarchical, cf. below) "mini-browser" of important internal parameters of the icon.

The macro `ICONCOMMANDS(commandlist)` can be used to endow an icon with an initial set of commands when it is created in the `ICONMAKE` wrapper routine.

A long menu with very many entries can be hierarchically organized by grouping commands with curly brackets (e.g., *ini,exe,*{*foo,sub1,sub2*}*,more,less*). Any such group will then appear only with its first entry (e.g., *foo*) in the top-level menu, the remaining entries (*sub1,sub2*) are "hidden" in a cascading submenu (this can be extended recursively to more than one nesting level). If an entry in the "leaf" of a chain of cascaded submenus is selected (e.g., *sub2*), the actual command is built by concatenating the selected command names of all the submenues in the chain, with colons in between (producing the command *foo:sub2* in the present example). In this way, it is possible to call named units deep down in the container hierarchy of an icon.

12 Generating HTML Documentation of circuits

NEW

The *Html* command menu button allows to generate a HTML documentation tree for the currently displayed circuit and its subcircuits. The documentation tree will be collected in a directory (to be specified with the command), e.g., `mydoc` and consist of several files distributed in three subdirectories `auto`, `html` and `obsolete`.

The first directory `auto` collects all automatically generated files (such as images of circuits and dialog windows, automatically generated texts from annotations, an index file and HTML-wrappers to combine everything into a nice frame set) for the documentation tree and its contents usually need not be touched by the user (its contents will be rewritten with each documentation update, thus any user-edited changes would be lost).

The second directory contains the part of the documentation that is intended to be editable for the user. When the documentation generator is run for the first time for a circuit, it will provide in this directory for each container or primitive unit `foo-x` an initial skeleton file `foo-x.html` that contains already most of the basic information about the circuit or unit `foo-x` (e.g., information about its interface and its *Neo* variables) as well as links to associated material in the auto-generated files. Any additional material about `foo-x` should then be collected in this file. It will remain preserved during later runs, since the documentation generator will not change the file `foo-x` anymore during later runs (as long as the unit `foo` remains part of the documented circuit, cf. below).

The third directory `obsolete` will only be created during later update runs of the documentation generator. When units or entire subcircuits of an ealier version of a circuit have become deleted, a later run of the documentation generator will identify the associated (now unused) files in the `auto` and `html` subdirectories. Instead of removing them automatically, it will deposit them in the `obsolete` subdirectory, leaving their final removal to the discretion of the user.

How can later renamings, interface changes or changes to its variables become automatically reflected in the user-edited files in `html`, although the documentation generator leaves these files unchanged during later updates? The truth is that these files themselves are never involved in the actual display of the documentation tree. Instead, each documentation update will create from each file in `html` (say, `foo-x.html` a corresponding counterpart `foo-x-main.html` in the `auto` directory. During this process, it will substitute a number of special symbols (each of the form `$[xxxx]`, for a fuller description, cf. below) that may be embedded in `foo-x.html` by actual

display the b-th pin of field a, using a display field width of c characters and a precision of d characters. The rear part of the specification can be omitted, in which case defaults are used.

values. By this substitution mechanism, static information in `foo-x.html` will appear in the generated `foo-x-main.html` as dynamically updated to the current version. It is then the newly generated `foo-x-main.html` that will become displayed when the documentation is viewed in a browser. Therefore, editing the viewed page for modifying the documentation is not a good idea (any changes will be lost on later updates); instead, changes must be applied to the page `foo-x.html`. This page is *not* displayed, but the displayed page offers a link labelled *(edit)* that opens the to-be-edited page in a separate viewer so that it can be conveniently modified with the browser's editor (note that the page will look a little bit different from the displayed `*-main.html` page since its symbolic tokens are not yet expanded (ignore this)). Note also that the changes will become visible in the main documentation only after the documentation generator has been re-run (for this purpose, you may wish to choose the much faster "only texts"-mode that omits the more time-consuming (re-)generation of the images).

The following symbolic tokens are defined for substition:

`$[NAME]` name of the current icon or circuit

`$[PATH]` nesting path of the current icon or circuit (each but the last path element is a link to bring up the circuit it stands for).

`$[DATE]` date of the generator run

`$[BYAUTHOR]` specifies the author, if given, empty otherwise.

`$[CLASS]` the name of the NST class of the unit with a link to the NST html manual page.

`$[HELP]` a help link labelled with the class name of the current icon or circuit

`$[PARAMETERS]` a link to the PARAMETERS section

`$[INTERFACE]` a link to the INTERFACE section

`$[DESCRIPTION]` a link to the DESCRIPTION section

`$[DIALOG]` a link to the unit's main dialog window

`$[DIALOG1]` a link to the unit's auxiliary dialog window

`$[DIALOG2]` a link to the unit's parameter dialog window

`$[EDIT]` a link to the editable version of the documentation page (i.e.,`foo-x.html` in the above example).

`$[SECTION:PARAMETERS]` a section with heading *PARAMETERS* followed by a table of the *Neo* variables that are visible for the current circuit.

`$[SECTION:INTERFACE]` a section with heading *INTERFACE* followed by a table of the input and output fields of the current circuit.

The tables that are generated for the last two tokens collect the main information in table of r rows (one for each variable or input/output field, resp.). Each row has two columns: the left column contains a "tag" (the name of the variable or the type of the input/output field). The right column contains a description text. If the documentation generator finds suitable information for these fields in the circuit, it will fill the corresponding table entries. These automatically provided table entries can be overwritten by the user with the following substitution tokens, that allow to address the rows and columns of the tables directly:

`$[LOCALx:TAG]` material following this token will become placed into the tag column of the table row for local variable `@x`.

$[LOCALx:TEXT] material following this token will become placed into the description text column of the table row for local variable @x.

$[INPx:TAG] material following this token will become placed into the tag column of the table row for input field x.

$[INPx:TEXT] material following this token will become placed into the description text column of the table row for input field x.

Analogously defined tokens exist for global variables (GLOBAL instead of LOCAL) and output fields (OUT instead of INP). The rows of global variables follow those of local variables, the rows of output fields follow those of input fields, and the final two tokens $[x:TAG] and $[x:TEXT] allow to directly address the cells of a row by its row number x (starting at x=0).

Delimiter for the material after any of these tokens is another such token or one of the tokens $[SECTION:PARAMETERS], $[SECTION:INTERFACE] or $[END].

The tokens themselves will not be visible in the generated (foo-x-main.html) file, only the material that they represent will appear. In particular, when a token does not apply (e.g., a container has no input or output fields), any material bound to it will be invisible, too. Therefore, any embedded specifications become automatically "silenced" when their associated referents (such as a field or a variable) has disappeared in a new version (and the specification "wakes up again" when the referent is introduced back).

The initially provided *.html files in html offer an example of the use of the above tokens.

13 Extending *Neo* by new units

Neo can be flexibly extended by loading collections of units in the form of *circuit* or *dynamic folders* into it.

Circuit folders are just directories with (encapsulated) circuit files collected for re-use (they can be made fully interactively, just by dragging circuit containers from the circuit window into the module window of an existing circuit folder). If *Neo* finds a file named NEO in a circuit folder directory, it inspects it for a list of unit names and orders the icons in the folder window accordingly.

Dynamic folders are compiled collections of units in the form of shared libraries. They also can be added at startup or during run-time and are then fully equivalent to static folders that are rigidly built in. Usually, a dynamic folder is built from a pair of source files conventionally named nst_somename.c and neo_somename.c. nst_somename.c should contain the implementations of all NST units that the folder offers. neo_somename.c should contain for each NST unit its associated NEO wrapper plus some header information. The format for neo_somename.c is the same as for the wrapper file of a normal static folder (to see examples, look at the neo_xxxx.c-files). The two files nst_somename.c and neo_somename.c should be placed in some directory named xyzsrc (a conventional choice is foldersrc, but only the trailing src is mandatory) that is at the same level as the nstsrc and neosrc directories. The NST makefiles will then automatically make a dynamic folder named xxxx.so from the two files and place it into the $ARCH directory [18]

Care must be paid to the name (defined by the argument arg of the BEGINFOLDER(arg)-macro in the folder source file) of a dynamic folder: it will become the "key" that must be known in order to later load the dynamic folder into *Neo* . Therefore, it should be easily recognizable from the basename nst_xxxx of the shared library. *Suggested convention:* if the basename contains an underscore, the foldername is the part delimited by the rightmost underscore before the leftmost suffix dot and this suffix dot (e.g., neo_filter.hr.so => filter). If there is no underscore before the leftmost suffix dot, the foldername is the entire basename. Note also that the units implemented by several shared libraries can be collected within a single folder of, say, name xyz, if all wrapper files specify the same xyz as the argument of their BEGINFOLDER() macro.

[18] of course, instead of the pair nst_xxxx.c and neo_xxxx.c the files can be named, partitioned and placed into directory locations in any different way, but then the necessary compilation and linking commands have to be issued by hand.

Using C++: NST matches very well with the object structure of C++. Existing C++ classes can be "wrapped" to make their methods accessible from NST. The necessary wrapper code can be generated with the template generator program mku++. A similar facility for C class objects is provided with the template generator program mku+.

Basically, each NST unit then represents one instance of the class, with its "main method" invoked with exec_unit(), and further methods via adapt_unit(), ctrl_unit(), save_unit() and load_unit(). The generated nst_*.C file must be compiled with g++, all other compilation and linking is done with gcc.

For a detailed documentation, see *Using C and C++ objects from Neo/NST: the Tools Mku+ and Mku++* (file mkudoc.tex).

#import directive[19]The prog_unit offers another way of extending *Neo* with self-written functions with very little effort. The #import directive at the beginning of a prog_unit program will import C-functions from shared libraries that then become callable inside the prog_unit. For details and examples, cf. the prog_unit manpage.

PREVIOUS

Documentation Each unit of a self-written folder can be assigned a nroff and html manual page and a example circuit. .Neo will allow direct access to this documentation (via the '?' and 'e' symbol on an icon) when the following conventions are followed: (i) the folder is a static folder or was loaded by a full path specification in .neofolders or with the load +folder=path/name command (if Neo had to resort to LD_LIBRARY_PATH to load a folder, it will currently not detect its documentation). (ii) The base name "xyz" of the manual page was specified by an entry ICONMANPAGE("xyz") after the ICONLABEL() macro in the wrapper code of the unit. (iii) The nroff (HTML) man page is named xyz.3 (xyz.html) and placed in a directory named cat3 (html3) that is at the same level as the directory that contains the shared library that implements the folder. Similarly, the example circuit must be named xyz.NST and placed in a directory named examples at the same level as cat3 (currently, Neo will locate an example circuit only for those units that have also a cat3 man page).

3d-Graphics. The presence of the flag -DOGL (to be specified both when libnst.a and libneo.a is built) leads to the inclusion of code for 3D-graphics under Mesa or OpenGL (without this flag, the produces executable will only allow 2D-graphics and the folder solid will be absent). -DOGL requires under Linux that libMesaGL.a and libMesaGLU.a are both linked with the executable and that their associated include-files/directories are in the include-search path. Under IRIX and SOLARIS, the corresponding proprietary libraries must be specified (libGL.a and libGLU.a for IRIX).

NEW

CORBA support. The prog_unit #import library corba allows the use of #import libaries that define proxy classes for CORBA objects so that remote objects can be accessed in a fully transparent manner. This requires to specify a CORBA name service with the -ORBInitRef command line option when Neo is started. For details, see the nstcorba manual.

For further details, consult the Neo- and/or the NST-manual.

14 Hints

To backup and save the contents of the current work send the SIGQUIT signal to neo (e.g. by typing the CTRL-
in the lneo terminal). Neo writes a NEORECOVER.NST file and can resume work with the "-r" option. One of the lneo features is to maintain a *stack* of those recovery files and to enable quick resume to (options "r,0,1,2,3...") and inspection of (options "v?,V?,c?,C?,d??,D??,DD??...") intermediate neo states previously saved. See also the tutorial section on lneo

To inspect the contents of (output) connectors, connect it to a suitable print_vec, print_mat or print_str unit (the value will show only after the attached unit was executed).

[19]the use of the older #loadlib directive is deprecated since it requires to specify the library name together with the (OS-specific) shared library suffix. However, apart from this, there is no difference

To set values of an input connector, connect it to a suitable `const_vec`, `const_mat` or `const_str` unit.
To single step through a `for_op` for testing, set the loop count to 1 and, if needed, get the iteration count from a (perhaps additional) counter unit.
Avoid complex circuits. Rule of thumb: when more than *seven to ten* units get involved, check if you can identify functional groups of units and repackage them into a subcircuit within a container (use the *Alt-'-'* command).
When copying units that use @-variables into a new container, first equip the container with definitions for these variables. (Otherwise, the variables will be lost and replaced by their current values, cf. above).

NEW

Batch operation of *Neo* circuits. To execute a *Neo* circuit 27 times without opening any *Neo* windows, issue the command

```
neo -i mycircuit -x 27 -hide
```

This can be used to operate *Neo* in batch mode or from non-graphic terminals.
Neo Test runs. To test Neo on a circuit, say `filename`, you can use the command

NEW

```
testneo filename
```

This will load `foo`, remake it two times and exit. The outcome of this operation is appended to a file `TESTNEO.log`. To test Neo on a whole set of circuits, `filename` may contain the usual wildcards, e.g.

```
testneo *.NST
```

does the test on all `*.NST` files in a directory and collects the results in `TESTNEO.log`. **Source code option**. For systematic exploration of parameter ranges, it may be convenient to use the source code option and set up a batch job for the generated program.
Entry slot in a dialog window is too short to accommodate an expression. The following workaround helps: Define for the unit a new variable window that is a copy of the old one, but with the offending entry slot made larger (reference the slot variables `$i` of the new window in the canonical one-to-one fashion from the slots of the old window).

PREVIOUS

15 Pitfalls

No manpages. If folders do not show the .'?' manpage symbol on their icons, check if the `-NeoHome` option has specified the right directory: it must have a .man/cat3 subdirectory with the man page files. If the folder is a dynamic folder, the manpages should be in a .cat3 subdirectory that is parallel to the directory holding the .so-library file of the folder. If an existing `cat3` manpage is not detected, check the base name of the man page against the entry in the `ICONMANPAGE()` field of the wrapper code. The correct name happens to coincide with the icon label in many cases, but the required correct name is *always* determined by the `ICONMANPAGE` argument and may differ from the label or the name of the icon!
No examples. In this case, check whether a .examples subdirectory is present in the `man` subdirectory of `NeoHome` or in a subdirectory parallel to directory holding the shared library object. If only some example circuits are not registered, check if their corresponding `cat3` manpages are present (this is a prerequisite!).
Problems with loading circuits. If a directory contains gzipped and uncompressed circuit files that have the same base name and are different versions of .NST and .nst files, .Neo will *silently* use the uncompressed versions. This may lead to surprises (and even crashes, when an uncompressed .NST-file belongs to the compressed .nst-file and vice versa).
Name clashes. Icons in folders are *only* distinguished by their print name! In the case of a name-conflict, the definition of a compiled icon always prevails over the definition of a loadable icon with

the same name. This protects the definitions of compiled units against loadable icons with the same name. The converse is not true: compiling into Neo units that use the same name as loadable icons will "shadow" the definitions of the loadable icons! To avoid such name clashes, it is suggested to choose for compiled icons always lower-case names. This allows then to protect the definition of a loadable icon by using a capital letter in its name.

Escaping from infinite loops. Neo allows you to build infinite or very long loops in which execution can become trapped. In this case, the only escape is to kill your process. This can be avoided, if you set the option `"Allow user-interrupt" in loops` in the `Prefs` Menue (this is not the default setting, since it comes with some performance penalty). You can then always escape with the Escape-key.

Apparently irreproducible behavior of a circuit. When using a circuit repeatedly, watch out for units that accumulate state information from previous runs (such as counters and file-reading units, in addition to the more obvious cases). To repeat the same run may require to reset such units first (in a properly designed circuit, this should be automated by suitable `ctrl_op`-units) Note also that the `remake`ing of a circuit sets the values at all unconnected input pins to their defaults. Attach `const_vec` or `const_str` units with suitable contents to "anchor" such pins at the desired non-default values.

Unexpected behavior of `Step` or `Run` commands. When the `Step` or `Run` command appears not to do the right thing, check if you are at the right level: both commands execute/adapt only the currently visible subcircuit! When a unit appears to do nothing, check if perhaps its control pin holds a value of 0.

Graphics output seems missing. When drawing with `draw_sym/str/mesh` into a `write_pix` unit that is executed *after* the drawing unit, the drawing may never appear, since the `write_pix` unit updates its image with its own current input values at each step, thereby overwriting any previous graphics. Remedy: either change the order of the units suitably, or set the control pin of the `write_pix`-unit to 0 (this will disable any drawing of the `write_pix`-unit on its own).

Unexpected parameter settings. When units that use local (i.e., `@-`) variables are copied or moved into a different container in which these local variables are not visible, *Neo* silently replaces each `@`-variable by its current numeric or text value.

Problems with adaptation. Adaptation of an adaptive unit requires to issue an `adapt_unit()` call after the usual `exec_unit()` call. This requires in Neo to activate adaptation mode (from the *Prefs* command) or to use the unit as an operand of a `for_op` or similar operator unit that has its adaptation mode activated. Therefore, if an adaptive unit seems not to adapt, first check these settings. Note also, that most non-adaptive units simply ignore the adaptation calls. Therefore, if during the adapt pass transformations from non-adaptive units are desired (e.g., to form the difference of two vectors), the simple presence of a corresponding non-adaptive unit (e.g., a `dif_vec` or `sum_vec`) will not help. In these cases it is necessary to insert a `ctrl_op` unit to translate during the adapt pass the adapt call into a `exec`-call and apply it to the non-adaptive units to make them perform the desired operation (use `ADAPT` as trigger and `EXEC` as control operation for the `ctrl_op`). Note also that adaptation proceeds in opposite (right to left) order than `exec`ution.

Source code does not compile properly. The current default template `NEO_SRCTEMPLATE.c` for source code generation assumes that the `Preprocessor` option is chosen. You need to make modifications if you use the other two options. A (simpler) template that can be used then is given in the Neo Manual. Since the transition to dynamic folders, `libnst.a` now only contains a small subset of all `NST` units. For using the source options, you usually must link the necessary objects extra (e.g., make a "large" `.  libNst.a` that contains everything and use this). You must also include the necessary headers (the simplest way is to make and include a `nst_all.h` that includes all NST include files).

16 Appendix: Circuit File Format

Circuit descriptions are saved in a simple format that can also be used to write circuit specifications in a text editor. Entries for units, wires and annotations are sequentially written in their normal "NST-nesting order".

16.1 Header part

For open circuits, a global header

```
#<HTML><BODY><PRE>
{header}
{folders}
  +foldername1  path1/folder1.so
  ...
  +foldernamek  pathk/folderk.so
{end}
{status NEO_VM.m NST_VM.m noadapt steps=1 }
------
```

provides path information for referenced folders, and some status and version information of the circuit. If the folders are specified without path (the recommended way), they will be looked up in `$LD_NST_LIBRARY_PATH`, if this is defined, or in `$LD_LIBRARY_PATH` otherwise.[20] The first line is a comment for Neo, but allows a HTML browser to read the remaining file (which will contain some HTML links to allow convenient navigation). The last line `------` is a delimiter that must always follow the header part for proper loading.

16.2 Entries for units:

Each unit of class .theClass has a minimal entry of the form

```
BEGIN theClass { %x,y #<A NAME=theClassi></A>
INTERFACE:   {input-spec}{output-spec}
LABEL:       theLabel
NAME:        theName    # absent for unnamed units
------
#
#  CREATE WITH:
#
@0    =  value0
@1    =  value1
...
@k    =  valuek
------
# if theClass is "container", entries for its subunits
# will begin here (recursively)
ENDOF theClass }
```

Here, all entries must be left-adjusted as shown. The part `%x,y` following the first { specifies the (unscaled) coordinates of the lower left corner of the icon symbol in the circuit window. The remaining comment is a HTML anchor (for container units, the file format allows the embedding of a list of HTML links after the `NAME:` (or `LABEL:`, if the unit is unnamed) specification. These links will then point to anchors such as the above that are provided for the embedded subunits of the

[20]Later, when a circuit is saved back to file, all its used folders are written in a similar entry and path prefixes are given only for those folders that were also specified with a path prefix during loading *and* that were found via that path prefix (and did thus not need the use of a search path).

container; in this way, the unit hierarchy can be easily traversed with a HTML browser). After the `BEGIN` line, there follow a number of lines with *keyword: value* pairs. The specification `INTERFACE: {..}{..}` must be present (and must be given on a single line!), all other specificications (such as `LABEL:` and `NAME:`) are optional or only required for some particular types of units (e.g., `OPERANDS:` for some types of operator units). This part of the specification *MUST* be delimited by a line of at least 5 minuses (`-----`), after which definitions of dialog parameters follow (cf. below). Lines starting with `'#'` indicate comments.

16.3 By-reference entries:

A star `'*'` as the first letter of the class name indicates that the entry is *by reference*. In this case the `INTERFACE:` line will be preceded by an additional line

```
ORIGIN:    foldername folderversion circuitversion
```

specifying the folder, the folder version and the version of the circuit which is referenced by the name `theClass` (after removal of the `'*'`.

16.4 Dialog Parameters of a unit:

Symbols `@0..@k` refer to the corresponding parameter positions in the creation dialog window of the unit, and `value0..k` specify the values for the corresponding creation parameters (the values for the settings of the `Cancel` and `Ok` buttons are also always included in the list). String values are given in apostrophes and are broken into lines, e.g,

```
...
@42  = "this is a long string"
       "extending over two lines"
@43  = ...
```

If the unit also has a configuration window, its values follow after the `------` - delimiter in a similar entry. If the unit also has a user defined window, it also has a similar entry that precedes the shown entry for the creation window; in this case, there will be a further entry (preceding the parameter entry of the user defined window) that shows the NST window format string of the user defined window (create an example file to see the details).

Variables may be given in any order, but ascending order is processed more efficiently than any other order.

If the window format of any of the dialog windows contains after a widget the directive `%_name`, the variable specifier `@i` for that widget's value becomes replaced by `name` in the output file (before making extensive use of this feature we should first establish a good general naming policy for parameters).

16.5 Patch Level:

Some dialog window sections may show as their first entry a line

```
@PLEV = n
```

Entries of this kind specify an integer *patch level n¿0* for the dialog window. The presence of a patch level specification indicates that the definition of the dialog window has later been extended by additional numeric or textual widgets. Whenever this is done, the patch level of the window (which starts with 0 for the initial definition of the window) must be raised by 1, and the new patch level must be assigned to the newly added widgets (this is achieved by placing before each newly added widget the format directive `%n'` where `n` is the new patch level value; the patch level `n` of the window is then the highest patch level number that occurs in any of its %n' directives and if $n > 0$ it is saved with the above line when the dialog window parameters are written to a file).

The patch level specification ensures that older versions of circuit files (written before the introduction of the new widgets) remain readable also with the extended dialog window. This is achieved by ignoring when reading back the parameters the presence of all those window widgets whose patch level exceeds the patch level specified by the @PLEV-line in the file. By this, the read parameters become automatically matched against the same subset of widgets that was in place when the file was written. Of course, this is only necessary for variables @i that are identified solely by their serial position i; variables with non-positional names can always (provided their names are unique) be matched correctly, thus, the patch level specification can be omitted when all widgets carry names (which can be assigned by means of the %_ directive, as explained in Sec. 16.4).

16.6 Wires:

Each wire has an entry of the form

```
WIRE           0 0 -> 1 0 %227.5,340+0
CONNECTIONS:   3 0->0 1->1 2->2
```

The first line WIRE n f -> n' f' %x,y+z specifies the position n of the source icon in its container (0=leftmost icon) and the position f of the source field. Likewise, n' and f' specify the analogous data for the destination icon. Position values n=-1 or n'=-1 indicate the enclosing container unit. The %227.5,340+0 specify the geometry of the wire (its center coordinates and the length of its middle piece).

The second line specifies the nr (here: 3) of actual pin-wise connections followed by a listing of these (may extend over several lines, with CONNECTIONS: not repeated).

If wires belong into a container, their entries are given *before* the closing ENDOF container } entry.

16.7 Annotations:

The annotations of a container are given as an entry

```
ANNOTATIONS: k
x1 y1 statusflags1 :
"title text of annotation1"

"body text of annotation1"
...

xk yk statusflagsk :
"title text of annotationk"

"body text of annotationk"
```

I.e., each annotation starts with a position specification (which MUST end with a colon), followed by the text lines (enclosed in apostrophes) of title and body texts. Title and body parts are separated by a blank line.

16.8 File end:

The circuit file ends when EOF or the special comment ##___END_OF_CIRCUIT___ is encountered. It will be followed by a further comment line #</PRE></BODY></HTML> that is present for HTML purposes.

NEW

When no data file or data section has been saved, theHTML comment line will be followed by a further section holding window position info. This section is enclosed in pair of lines

```
##_____BGN_OF_POSINFO_____
... position info for each window ...
##_____END_OF_POSINFO_____
```