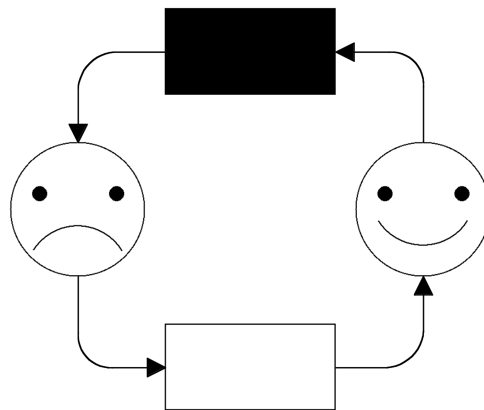
Computer Science Department
Aarhus University
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark

December 1995

Mimic/CPN

A Graphic Animation Utility for Design/CPN



User's Manual

Version 1.5

by

Jens Linneberg Rasmussen & Mejar Singh

Contents

1 Introduction **2**

2 Examples **2**
2.1 Security System Examples 2
2.2 Radar Example 5
2.3 Telephone Example 5

3 Graphic Objects of Mimic/CPN **6**
3.1 Object Structure 6
3.2 Constraints 7
3.3 Constructing Graphics 8

4 How to Use Mimic/CPN **10**
4.1 Types 10
4.2 Initialisation 11

5 Functions on Mimic-Objects **12**
5.1 Functions for Hiding and Showing 12
5.2 Functions for Moving and Aligning 12
5.3 Functions for Selecting 14
5.4 Text Functions 15
5.5 Information Functions 16
5.6 Snapshot Functions 16

6 General Functions **18**
6.1 Dialogue Boxes 18
6.2 Miscellaneous 19
6.3 Operation Modes 20

7 Mimic Exceptions **20**
7.1 List of Exceptions 21
7.2 Handling Exceptions 21

8 Limitations **22**

Function index **24**

1 Introduction

Mimic/CPN is a graphic animation utility which allows manipulation of existing graphic objects in Design/CPN simulations. By using the animation utility it is easy to make tailor-made graphic interfaces to simulations. The utility is a stand-alone code-library whose main functions are to:

- Hide or show regions or connectors.
- Select regions or connectors by using the mouse.
- Move or align regions.
- Save and restore positions and appearances of graphic objects.

The animation utility can be used for making a graphic display of the *state* of the system that a CP-net describes. Furthermore, this utility enables the user to *control* simulations. The feedback from simulations becomes more comprehensible, and it becomes easier to make simulations which resemble real-world use of the system which is being simulated.

This document serves as a user's manual for Mimic/CPN. Readers are expected to be familiar with Design/CPN, and a certain knowledge of functional languages (in this case Standard ML) can be useful. Mimic/CPN operates as an interface to the OA-functions[1] that are available in Design/CPN. The OA-functions are described in [OAF93]. Some of the functions of the animation utility resemble certain OA-functions, and others contain considerable refinements. As they are all included, it should be possible to avoid use of OA-functions directly — thereby avoiding looking up the right functions in a larger manual. While OA-functions refer to objects by means of ID numbers, the animation utility uses mnemonic names defined by the user. In this way, there are two different levels of accessing the objects. One level (used by OA-functions) contains ID numbers which depend on whether the diagram is reloaded and on which machine the diagram is edited. The other level contains names which remain the same on different machines. The animation utility makes use of names but also allows retrieval of object IDs. This makes it possible for OA-functions to be used as a supplement to Mimic/CPN functions.

The functions of the animation utility are intended for use in code-regions of transitions. In the following section, we give some examples of applications of the animation utility in connection with various CP-nets.

2 Examples

Mimic/CPN was developed in connection with the designing of a security system by means of CP-nets. By way of introduction, we give some examples from this project.

2.1 Security System Examples

For an illustration of the use of the animation utility, consider Figure 1. It shows a code unit applied for access control in the security system (described in [Dal94]). In the process of constructing a new version of the security system, Design/CPN is used as a modelling/designing

[1] OA is an acronym of Meta Software's Open Architecture

tool, and Mimic/CPN is used for controlling and monitoring the simulations. For example, user-codes can be entered (when simulating the CP-net) by clicking the buttons of the scanned picture[2] of the code unit in Figure 1. Moreover, a green and a red light-indicator above the '0' and '1' keys can be shown/hidden during the simulation. We will use the code unit as our main example throughout this manual.

Figure 1: Code unit

The control panels used for controlling the security system are handled more or less like the code units. In Figure 2, the control panel used during simulations is shown. Compared

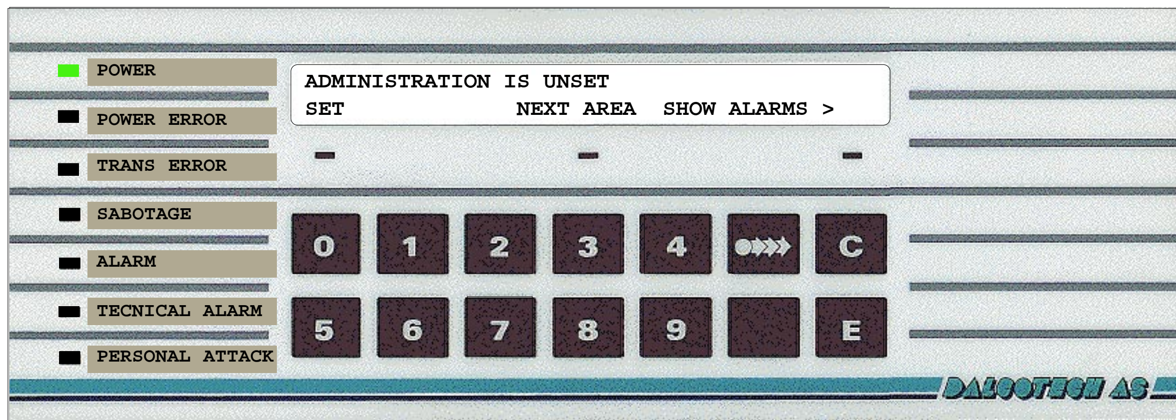


Figure 2: Control panel

to the code unit, the control panel has seven light-indicators (in the left column), a display,

[2]Bitmap graphics in Design/CPN is currently only available for Macintosh

and some additional buttons. From the menu displayed in Figure 2 it is possible to, e.g. set the administration area (which means that alarm surveillance in the area is activated).

The 'mimic' name is actually inherited from the old security system, in which a *mimic-board* was an optional extra. A mimic-board shows a drawing of the supervised building with light-indicators mounted underneath, indicating different system-states. In order to model the mimic-board, we have made a page with a scanned picture of a building. Detectors are represented as circles, output (lamps or indicators) as squares, and icons stand for several devices: clock, horn, code entry units, control panels, and printers (see Figure 3).

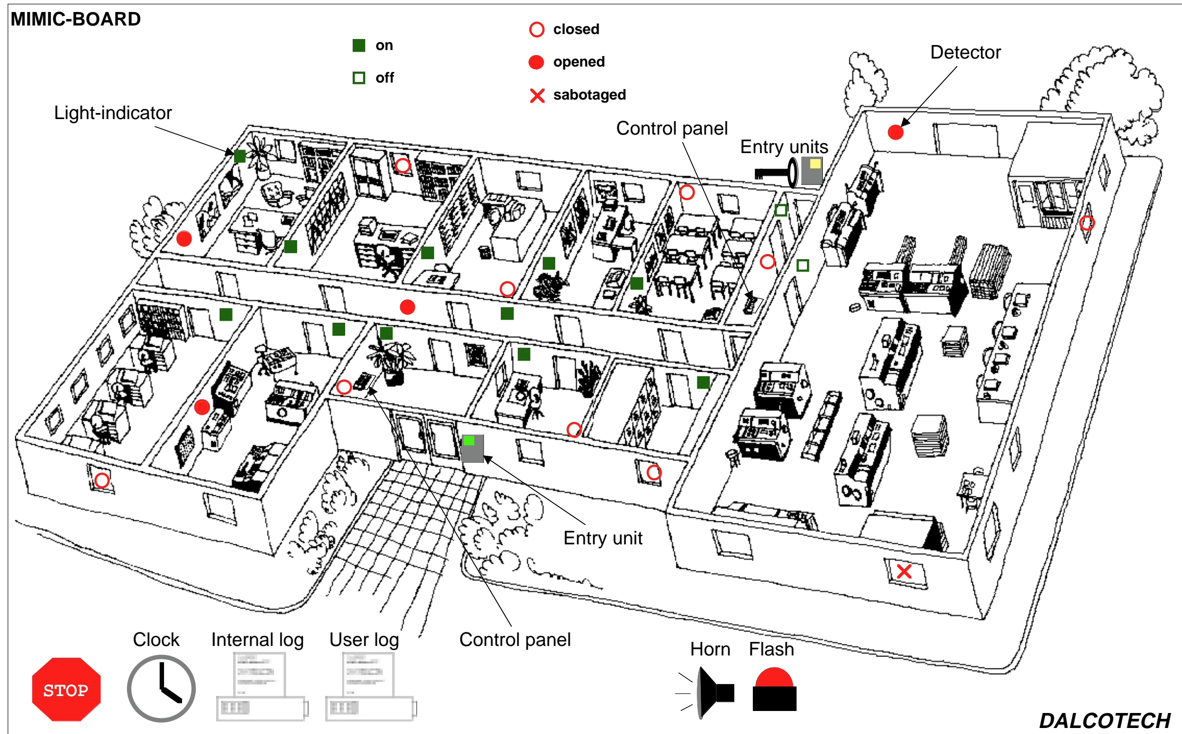


Figure 3: Mimic-board

The security system CP-net has a subnet which models the physical environment. Whenever an output change is issued by the security system, it will report this change to the environment subnet, which then will apply Mimic/CPN functions in code-segments of its transitions to update the graphic representations of the output on the mimic-board page. When, for example, an area of the building is set in surveillance-mode, it will cause the outputs showing the area-states in the building (the squares) to change their appearance accordingly.

The mimic-board is also used for controlling the simulations of the security system CP-net. During simulations, the user is requested to select from the mimic-board using the mouse — thereby creating environmental changes for the security system to react upon. For instance, if the user clicks on a detector (a circle), he/she is asked to choose a new state of the detector by selecting one of the closed/opened/sabotaged symbols. If opened is selected, the user has simulated a situation where, e.g. a glass-breakage detector is activated due to a burglar trying

to get unauthorised access to the building. In the underlying net, the detector-change will be reacted upon by the CP-net. The horn is turned on and a message is sent to an alarm-receiving-central. On the mimic-board page, the activation of the horn is shown by a change in the appearance of the horn-icon.

Another example of selecting from the mimic-board: a user clicks on the code entry unit icon, situated next to the front door. This brings forward a page containing the code unit (see Figure 1) on which the user can enter his user-code in order to get access to the building.

Thus, the mimic-board page is used as a control-menu for the underlying CP-net as well as for displaying the state of the system.

2.2 Radar Example

The simple radar screen displayed in Figure 4 could be used in a model of a radar system of an air-port. The radar is an example of a system in which it is useful to show *movement* of objects. The animation utility provides functions that makes it possible for a CP-net to move objects (e.g. on a radar screen) during simulations. In addition to controlling object positions,

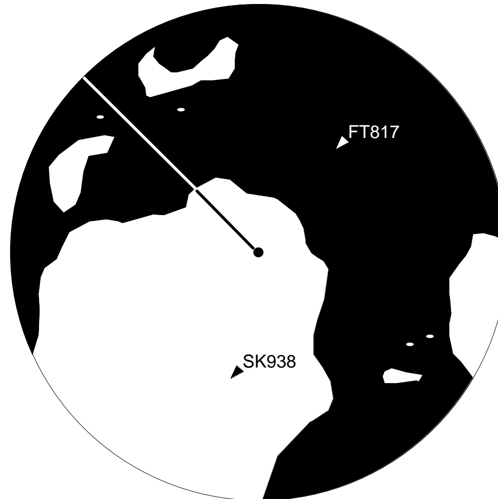


Figure 4: Radar screen

Mimic/CPN makes it possible to display textual information (such as velocity/altitude) on the radar screen during simulations.

2.3 Telephone Example

Figure 5 shows the state of a CP-net which models a telephone system. Each phone is represented by a rounded box. The icon of a telephone depends on whether the telephone receiver is on or off the hook. The bottom text in the rounded box tells whether the phone is ringing, inactive, connected, etc. Various types of connectors indicate which phones are connected and which phones are trying to establish connection. When simulating the underlying CP-net, the icons, connectors, and text change appearance in accordance with the state

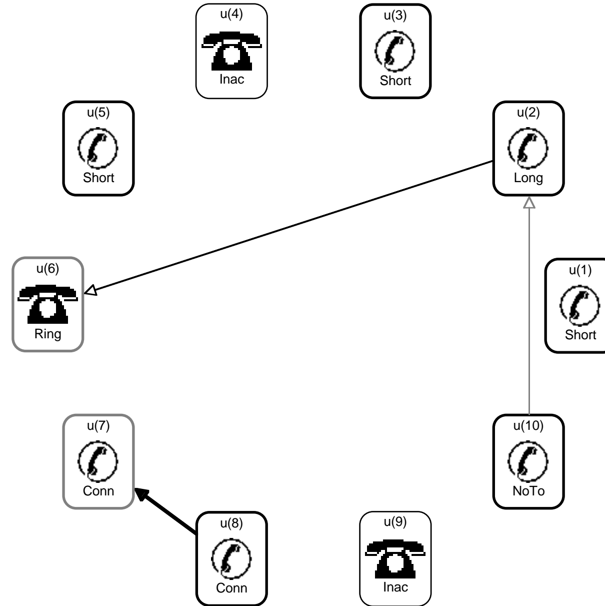


Figure 5: Telephone example

of the simulation. In this way, Figure 5 reflects a certain state of a simulation where, e.g. phone number eight and seven are connected. Furthermore, when a "dial" transition occurs, the user is asked to select (using the mouse) which telephone should be the recipient. The telephone example is from [Jen92] in which pages 184-186 describe a graphic representation using OA-functions directly.

The telephone example demonstrates how the animation utility can give an *abstract overview* of the state of a system. In the security system model, the utility was used for constructing a simulation *interface* which resembled the interface which was used in the actual security system.

3 Graphic Objects of Mimic/CPN

In this section, we describe the way in which graphic objects relate to each other in Design/CPN. Furthermore, we present some constraints that graphic objects must comply with in order to be manipulated via Mimic/CPN. Finally, we show *how* the graphics are constructed using the Design/CPN editor.

3.1 Object Structure

All graphic objects in Design/CPN are part of a tree-structure, which relates the objects vertically in a father/son relationship and horisontally as siblings. The *node* is at the root of the tree, and its sons on the next level are called *regions*. The objects on the third level (regions of the regions) are *subregions*. The tree-structure can be deeper, but only the first three levels of objects are important when using Mimic/CPN.

The tree-structure is reflected when moving from one object to another by using the arrow keys[3] in non-text mode:

- $\leftarrow$ moves left in the tree, so that you choose the sibling to the immediate left.
- $\rightarrow$ moves right in the tree, so that you choose the sibling to the immediate right.
- $\downarrow$ moves down in the tree, so that you choose the leftmost son.
- $\uparrow$ moves up in the tree, so that you choose the father.

The regions inherit some properties of their ancestors. For example, moving an ancestor makes its descendants move, so that they all retain their relative positions, and when hiding the ancestor, all the descendants are automatically hidden.

Along with the nodes and regions, it is of course possible to construct connectors. They can have any pair of nodes or regions as endpoints, and they can also have regions of their own. Connectors are *not* part of the object structure which encompasses the node and the regions. Instead, connectors are individual objects that can have a structure of regions — as nodes can.

3.2 Constraints

We have chosen to identify a node by the name of the page on which it is placed and by the text in its leftmost region. The regions of the node are also referred to by the text inside their leftmost region, i.e. the leftmost subregion. It is possible to control the order of regions in Design/CPN, and in this way, the names can be placed correctly (see Section 3.3).

In Figure 6, the object structure of the graphic objects representing the code unit is displayed. The thin circles contain the names of the node/regions that are the actual graphic objects to be manipulated. All the objects are auxiliary as they will normally be, simply because they are made for displaying graphics. However, places and transitions can be nodes of mimic-objects[4] if that is desired.

In general, there can be more sub-regions than the regions which contain names, and the sub-regions can have regions of their own. However, these are not important for the behaviour of simulations with Mimic/CPN. Moreover, Mimic/CPN only recognises connectors which have a name (the text in its leftmost region) and which interconnect two regions which belong to the same mimic node. The following rules define the components of mimic-objects.

Mimic names A mimic name of an object is the text in its *leftmost* region. If the object has no regions or if the leftmost region contains no text, it does not have a mimic name.

Mimic nodes A node is a mimic node (can be initialised as a mimic-object) if it has a mimic name.

Mimic regions A region is a mimic region (can be manipulated by the Mimic/CPN) if it has a mimic name and if its father is a mimic node. Furthermore, the region cannot be the leftmost region of the node.

[3] On UNIX versions you have to press the META key simultaneously

[4] We use the term *mimic-object* to denote a node (and its associated objects) that has been initialised by Mimic/CPN

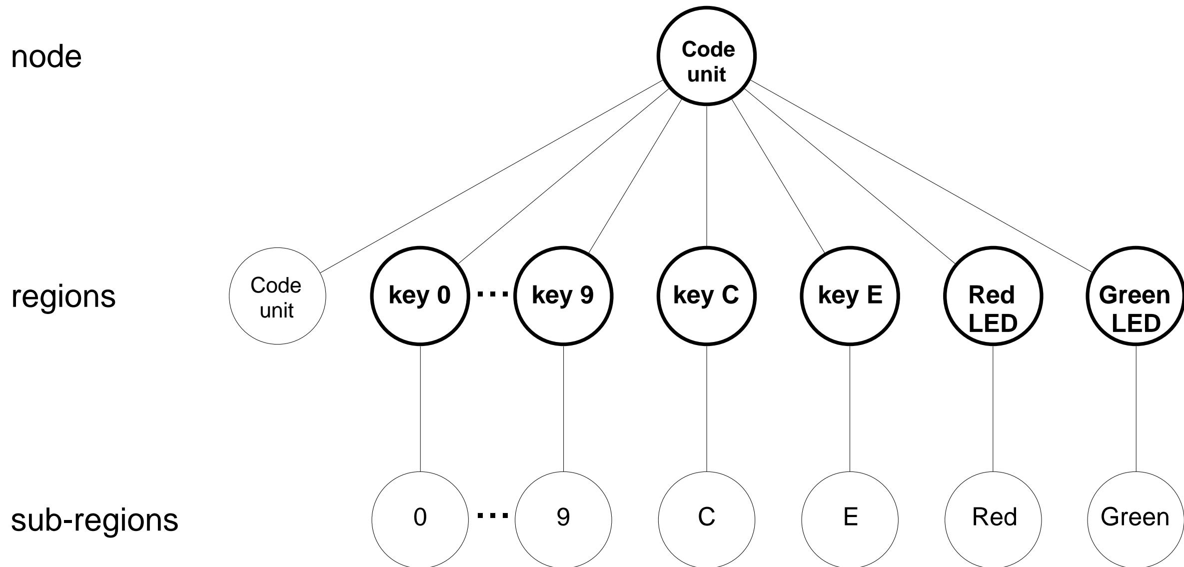


Figure 6: Object structure of code unit

Mimic connectors A connector is a mimic connector if it has a mimic name and if it interconnects two mimic regions of the same node.

When using Mimic/CPN, only mimic nodes, mimic regions, and mimic connectors can be manipulated. However, this manipulation may also influence some non-mimic objects. As an example, connectors will always follow their endpoint-objects — no matter whether they are mimic connectors or not.

In this version of Mimic/CPN, a mimic name can be any SML string except for strings including double quotes("'s). This means that, e.g. `"Blue lamp"` and `"Blue  lamp"` can be mimic names, which are different from each other because of the extra space.

As we use names to identify the objects, we demand that

- Mimic nodes on the same page must have unique mimic names.
- Mimic regions of the same mimic node must have unique mimic names.
- Mimic connectors of the same mimic node must have unique mimic names.

3.3 Constructing Graphics

A mimic-object consists of a structure of objects constructed as usual in the Design/CPN editor. In the **Aux** menu, a series of different objects (boxes, ellipses, etc.) can be constructed. When constructed they will be auxiliary nodes, except for the connectors, which are auxiliary connectors. The shape and appearance of these objects can be adjusted by changing shape and graphic attributes in the **Set** menu. On the Macintosh, you can also use the cut and paste facility to include (any) bitmaps from other programmes. When a bitmap is pasted into Design/CPN, it will become a node which can be treated like the other graphic objects. Typical ways of making the components of a mimic-object are described below.

Making the node In a mimic-object there has to be exactly *one* node, which will be parent of the regions. The node can have any form, but it is advisable to let it surround all its regions. If no graphic background is needed for the regions, one can for example make a big box and then change its graphic attributes in order to make it invisible.

Giving mimic names The **Label** entry in the **Aux** menu is convenient for giving objects mimic names (as discussed in Section 3.2). Just make a label with the desired text and turn it into a region of the object by means of the **Make Region** entry of the **Aux** menu. If other regions of the object already exist, it is necessary to change the succession of the regions in order to comply with the constraint concerning names in leftmost regions. This can be done by selecting the other regions one by one and using the **Bring Forward** entry in the **Makeup** menu. The **Bring Forward** command turns the previously selected region into the sibling immediately to the right of the region which is selected after issuing the command. When a new region is made, it becomes the rightmost region and will be displayed on top of all the other regions. By making the mimic name region first, it will automatically become the leftmost region. If many regions exist before making the mimic name region, it can be easier to turn all the preexisting regions into nodes before making the mimic name region and then turn them into regions again afterwards.

Making a region By using the **Make Region** entry of the **Aux** menu, it is easy to change a node into a region of another node or region. This way you can turn all other nodes into regions of the background node. If you are not working on a Macintosh (where you can use, e.g. scanned pictures) you will probably need to join some objects in order to get the desired appearance. For example, a car-object can consist of several polygons, circles, lines, etc. Graphic objects can be combined by selecting a parent and turning the other objects into its regions. This parent must then be changed into a region of the background node. The other objects will now be subregions of the parent, and they cannot be manipulated individually by Mimic/CPN. As we recommend having the node surrounding its regions, we advise that a parent region surrounds its subregions.

Making a connector A connector is made by using the **Connector** entry of the **Aux** menu. If the endpoints of the connector are not regions of the same mimic node, it will not be possible to use the connector as a mimic connector.

The subregions (mimic names, etc.) are not treated individually by Mimic/CPN. It is advisable to hide them or to make them unselectable, so that they will not interfere with, e.g. select-operations performed on their parents. The **Graphic Attributes** entry in the **Set** menu enables you to do this. The **Hide Regions** and **Show Regions** entries of the **Makeup** menu enables you to hide or show region structures. In this way, the name-regions can be concealed. If you have unselectable or hidden regions, it is a good thing to remember how to navigate the object structure (see Section 3.1). When navigating the object structure, all objects can be visited — also the unselectable and hidden ones.

As regards the code unit example, the following objects constitute the mimic-object displayed in Figure 1:

- The node is the scanned picture.
- The mimic name of the node is in this case situated in an invisible label (which is the leftmost region of the node).

- The numeric keys '0'–'9', the enter key 'E', and the cancel key 'C' each has an invisible box on top of it. These boxes are mimic regions of the node, and again their mimic names are situated in invisible labels.
- The green and red light-indicators are represented by a green box and a red box (hidden in Figure 1). These are mimic regions of the node in a similar way as the keys.

Figure 6 shows the object structure — the mimic names (labels) are represented by circles with thin borders.

When the mimic node, the desired mimic regions, and the desired mimic connectors have been constructed, it will be possible to use them in a simulation applying Mimic/CPN as described in the next section.

4 How to Use Mimic/CPN

In order to include the Mimic/CPN code-library in a CP-net we suggest using the `use`-command. Put the following code into a declaration node:

```
use "<path>mimics.<extension>";
```

There are two versions of Mimic/CPN — one for ESML (on Macintosh) and one for SML/NJ (on UNIX and Macintosh). They can be distinguished by their file-extensions: `esml` and `njsml`, respectively. It is essential that the correct one is used. In this manual, we use the SML/NJ syntax[5]. The functions are encapsulated in a structure called `mimic` (a structure in SML compares to a module in other languages). This means that you must write, e.g. `mimic.hide_regions` to use the hide_regions function. You can also write `open mimic`, which makes all functions in the signature visible, and then you can skip the `mimic.` prefix. More information on the SML language can be found in, e.g. [MTH90], [Ull94], and [Pau91].

A global reference to a mimic-object is made in the declaration node (after the `use`-command), as follows:

```
globref mimref = mimic.dummy;
```

For a description of `globref` see [CPN93] Chapter 35. The `dummy` value contains no information and is only used to construct references to values of the proper type. With global references declared as above (one for each mimic-object), no further changes are needed in the declaration nodes. Later on, we describe how the global references are applied to the initialisation of mimic-objects.

4.1 Types

In order to enhance the readability of the function specifications, we use the following type-aliases in Mimic/CPN:

```
type PageName, NodeName, RgnName, ConnName, ObjName, SnapshotName = string
type ALN_Type, x, y  = int
```

[5] The ESML (Edinburgh) version differs slightly from the SML/NJ (New Jersey) code displayed. However, the same functionality is implemented by the two versions

4.2 Initialisation

The `init` function described below is essential for using Mimic/CPN.

```
fun init:  PageName -> NodeName -> Mim
```

The above specification shows that the `init`-function takes the name of a page and the name of a node (its mimic name) and returns a value of type `Mim`. The type `Mim` is used for storing all information about mimic-objects. Objects of this type have to be declared in order to use the mimic functions. However, it is not necessary for the user of Mimic/CPN to know anything about the information stored in the objects. The data is solely used by the mimic functions, which need a reference to a `Mim`-object when called upon.

`init` initialises the data-structures used and returns a mimic-object (of the `Mim` type). The initialisation is done automatically by looking at the graphic structure of the denoted node and its regions. By identifying objects by means of mimic names we avoid directly referring to object IDs in, e.g. code-segments which use Mimic/CPN functions. Since object IDs change (in the UNIX version of Design/CPN) when reloading a diagram, we need to retrieve the IDs of the objects by initialising the mimic-objects. In this way, Mimic/CPN can translate the names given by the user into object IDs and then manipulate the intended objects.

You can choose a transition or construct a new one, which will be responsible for initialisation of the mimic-object. In our security system example, we use a separate transition for initialising the mimic-objects. The code unit is initialised in the following statement:

```
code_unit := mimic.init "MIMcu" "Code unit";
```

because it is placed on the `MIMcu` page and because the mimic name of the mimic node is `Code unit`. The mimic-object (which `code_unit` refers to) is initialised each time the transition occurs.

Initialisation *must* be performed whenever any manual changes of the mimic-object are made — by editing in Design/CPN. The following actions, done via the menus, the mouse/keyboard, or OA-functions, can result in inconsistencies between the graphic structure and the internal mimic-object data-structures.

- Hiding/showing of regions, connectors, or nodes.
- Moving of regions or nodes.
- Inserting/deleting text in regions or nodes.
- Changing the object structure (this includes deleting objects).

For example, when the **Hide Regions** command from the **Makeup** menu is invoked on the node, all regions will be hidden. But this action will not be detected by Mimic/CPN until `init` is called. It is possible to initialise mimic-objects by evaluating an `init` function call in an auxiliary box. However, we recommend using the `init`-function each time a new simulation starts, in order to ensure that the internal data-structures are consistent with the actual graphic structure. For this reason, we advise putting the `init` statement(s) into code-segments of transitions which occur in the beginning of new simulations.

It should be noted that `init` only reads the structure — it does not change the appearance of mimic-objects. Normally, it is desirable to start each simulation with the same appearance

of the mimic-object. To do this, you must succeed the initialisation with function calls that ensure the right state of the mimic-object. See Section 5.6 for more information on this.

Warning: If Design/CPN crashes during execution of `init`, it is probably because of too much text in a region on the mimic page. We use an OA-function to read the text information on the mimic pages, and this function causes a program crash if the length of a text string exceeds 4084 characters.

5 Functions on Mimic-Objects

The basic rule of the functions in this section is that they need a reference to a (previously initialised) mimic-object as the first parameter. The types of the parameters are displayed and explained in the function-descriptions. If you are unsure about the effect of a function, just evaluate it in an auxiliary box and look at the outcome. This, of course, provided that the Mimic/CPN code has been evaluated and a mimic-object has been initialised.

5.1 Functions for Hiding and Showing

```
fun hide_regions:  Mim ref -> RgnName list -> unit
fun show_regions:  Mim ref -> RgnName list -> unit
fun hide_conns:    Mim ref -> ConnName list -> unit
fun show_conns:    Mim ref -> ConnName list -> unit
fun hide_node:     Mim ref -> unit
fun show_node:     Mim ref -> unit
```

`hide_regions` hides the regions specified. Subregions of the regions are also hidden. The `show_regions` functions redisplays hidden regions. The remaining four functions have equivalent effects on connectors/nodes. It should be noted that `hide_node` hides the node and all its regions, but not the connectors. In the code unit example, the following two function calls

```
mimic.hide_regions code_unit ["Red"];
mimic.show_regions code_unit ["Green"];
```

turn off the red light-indicator and turn on the green light-indicator. In the telephone example, we use the following function call to display the `Request` connector between sender `x` and recipient `y` of colour `U`[6], in the code-segment of the "dial" transition:

```
mimic.show_conns phones [(mkst_col'U x)^"->"^(mkst_col'U y)^".Request"]
```

Here we assume that `phones` refers to the initialised mimic-object, and that is has a connector with mimic name `"u(x)->u(y).Request"`.

5.2 Functions for Moving and Aligning

```
fun move_regions_abs:  Mim ref -> RgnName list -> x*y -> unit
fun move_regions:      Mim ref -> RgnName list -> x*y*int -> unit
fun multi_move:        Mim ref -> (RgnName*x*y) list -> int -> unit
fun move_node_abs:     Mim ref -> x*y -> unit
fun move_node:         Mim ref -> x*y*int -> unit
```

[6] `U` is declared as follows: `color U = index u with 1..10;`

In these specifications, the `x` and `y` types represents integer coordinates. `move_regions_abs` moves the regions to the *absolute* position determined by the values of `x` and `y`. `move_regions` moves the regions *relatively* along the (x,y) vector given. The `int` parameter gives the option of dividing the move into small steps, making the move look more smoothly animated. The number tells how many sub-steps the move is supposed to consist of. The `multi_move` function allows movement of regions in multiple directions. For each region a vector (x,y) is given. Again, the `int` parameter tells the number of steps the move is to be divided into. With more steps it looks like the regions move simultaneously in the (possibly different) directions specified for them. `move_node_abs` and `move_node` are similar to `move_regions_abs` and `move_regions` except that they move the entire node.

Consider the radar screen in Figure 4. We assume that `radar` is a reference to a mimic-object (of the radar) and that the two aeroplanes are mimic regions with the names `FT817` and `SK938`. Then the statement:

```
mimic.multi_move radar [("SK938",100,~50),("FT817",~40,170)] 10;
```

would imitate the aeroplanes moving (simultaneously, in 10 steps) in two different directions.

```
fun align_regions:  Mim ref -> RgnName list -> RgnName -> ALN_Type -> unit
fun align_BETW:     Mim ref -> RgnName list -> RgnName*RgnName -> unit
fun align_PROJ:     Mim ref -> RgnName list -> RgnName*RgnName -> unit
```

The `align_regions` function aligns the regions in the list with respect to the region denoted in the third argument. The type of alignment is determined by the `ALN_Type` argument, which should be one of the following constants:

ALN_H	aligns Horisontal.
ALN_V	aligns Vertical.
ALN_LL	aligns Left to Left.
ALN_LR	aligns Left to Right.
ALN_RL	aligns Right to Left.
ALN_RR	aligns Right to Right.
ALN_TT	aligns Top to Top.
ALN_TB	aligns Top to Bottom.
ALN_BT	aligns Bottom to Top.
ALN_BB	aligns Bottom to Bottom.
ALN_CENT	aligns Center.

The other two alignment functions align the list of regions with respect to *two* other regions. `align_BETW` aligns between the two regions while `align_PROJ` projects the list of regions equidistantly in the direction determined by the two regions. In the radar example we could imitate the emergence of a new aeroplane `NA027` between `FT817` and `SK938`. Using (previously hidden) mimic region `NA027`, this is done by the following statements:

```
mimic.align_BETW radar ["NA027"] ("FT817","SK938");
mimic.show_regions radar ["NA027"];
```

5.3 Functions for Selecting

```
fun select:        Mim ref -> RgnName list -> ConnName list -> ObjName
fun multi_select:  Mim ref -> {regions:  RgnName list,
                               conns:  ConnName list,
                               startvalue:  'a,
                               term:  ('a*ObjName) -> bool,
                               comb:  ('a*ObjName) -> 'a,
                               escape:  bool} -> 'a
```

These selecting operations work on regions and connectors. The ObjName-type denotes the mimic name of either a region or a connector.

The select function allows the user to select (using the mouse) *one* region or connector from the ones listed. The specified regions and connectors will flash when the mouse is moved on top of them and the selection can only be made from these regions. A selection cannot be aborted without selecting one of the regions or connectors listed. Pressing the ESC key or clicking at the wrong places will cause the window to scroll to the mimic-object (a good thing if the window has scrolled away from the object in question).

The multi_select function performs multiple selections (select's) in a sequence, combining the selected object names (using the comb function) and returning the result when the termination function (term) is fulfilled. The regions and conns lists contain the objects that the selection is to be performed from. The startvalue is used in the first call of the comb function. When a region is selected during a multi_select function-call, term is called *before* comb. If term returns false, multi_select returns the outcome of the subsequent comb function-call. If term returns true, the outcome of the comb function-call will be the first parameter in the next call of term and comb.

If escape is false, the selection cannot be aborted (like in select). If it is true, the selection can be aborted by pressing ESC or clicking outside the selectable regions and connectors. If you are unsure about whether the term function works, it can be wise to set escape to true — otherwise, you cannot abort an "infinite" selection without aborting Design/CPN. When the selection has been tested sufficiently, escape can be set to false, which will inhibit annoying abortions caused by, e.g. clicking at the wrong regions. In Section 6.3, the behaviour of the select functions in disabled-interaction-mode is described.

In the code unit example, we use the function multi_select in order to get a user-code:

```
val usercode =
  mimic.multi_select code_unit
                    {regions = ["1","2","3","4","5","6","7","8","9","0","E","C"],
                     conns = [],
                     startvalue = "",
                     term = fn (s1, s2) => s2="E",
                     comb = fn (s1,"E") => s1
                             | (_ ,"C") => ""
                             | (s1, s2) => s1^s2,
                     escape = false};
```

The term-function implies that the selection will end when the "E"-key is pressed. The comb-function combines the region names (digits) selected to produce a string representation of the user-code. The "C"-key discards previously typed digits. Furthermore, the comb-function prevents the "E" from being included in the resulting string. Having escape set

to false means that the selection cannot be terminated unless the `term`-function evaluates to true ("E" is pressed). The resulting code is now contained in a string. However, the `select` function is *polymorphic* and can be set to return an integer if desired. Such a selection could look like:

```
val usercode =
  mimic.multi_select code_unit
                    {regions = ["1","2","3","4","5","6","7","8","9","0","E","C"],
                     conns = [],
                     startvalue = 0,
                     term = fn (i1, s2) => s2="E",
                     comb = fn (i1,"E") => i1
                             | (_ ,"C") => 0
                             | (i1, s2) => (10*i1)+(ord(s2)-ord("0")),
                     escape = false};
```

which makes the `comb`-function slightly more complicated but may save you the trouble of converting the string to an integer afterwards. Having a polymorphic function also makes it possible to make a fixed number of selections before terminating. Such a selection can look like (in pseudo-code):

```
val result =
 (mimic.multi_select obj {regions = <region-names>,
                          conns = <connector-names>,
                          startvalue = (v,N),  (* N > 0, the number of selections *)
                          term = fn ((v1,n),s2) => n=1,
                          comb = fn ((v1,n),s2) => (<combining v1 and s2>,n-1),
                          escape = false})
```

`result` is now a pair with the first element containing the "real" result, while the second element is `0`

5.4 Text Functions

```
fun append_in_region:  Mim ref -> RgnName -> string -> unit
fun put_in_region:     Mim ref -> RgnName -> string -> unit
fun read_from_region:  Mim ref -> RgnName -> string
fun append_in_conn:    Mim ref -> ConnName -> string -> unit
fun put_in_conn:       Mim ref -> ConnName -> string -> unit
fun read_from_conn:    Mim ref -> ConnName -> string
fun append_in_node:    Mim ref -> string -> unit
fun put_in_node:       Mim ref -> string -> unit
fun read_from_node:    Mim ref -> string
```

The first three functions are used for manipulating the text of a mimic region. The `append_in_region` function adds the specified text to the existing text, while `put_in_region` replaces the existing text with the specified text. `read_from_region` returns the existing text. If no text is found, the `""` string is returned. The remaining six functions work in a similar way. They are used for manipulating the text of mimic connectors and mimic nodes.

As regards the control panel of Figure 2, we apply some of these text functions to control the text in the display. The text shown can be put in the display by the following function call:

```
mimic.put_in_region control_panel "display"
    " ADMINISTRATION IS UNSET                 "^
    " SET             NEXT AREA  SHOW ALARMS > ";
```

where `control_panel` is a reference to an initialised mimic-object and `display` is the mimic name of the mimic region containing the display.

5.5 Information Functions

```
fun all_regions:          Mim ref -> RgnName list
fun shown_regions:        Mim ref -> RgnName list
fun hidden_regions:       Mim ref -> RgnName list
fun conns_of_regions:     Mim ref -> RgnName list -> ConnName list
fun get_regions_coors:    Mim ref -> RgnName list -> (x*y) list
fun get_regions_ids:      Mim ref -> RgnName list -> int list
fun are_regions_hidden:   Mim ref -> RgnName list -> bool list
---------------------------------------------------------------------
fun all_conns:            Mim ref -> ConnName list
fun shown_conns:          Mim ref -> ConnName list
fun hidden_conns:         Mim ref -> ConnName list
fun regions_of_conns:     Mim ref -> ConnName list -> RgnName list
fun get_conns_ids:        Mim ref -> ConnName list -> int list
fun are_conns_hidden:     Mim ref -> ConnName list -> bool list
---------------------------------------------------------------------
fun get_node_coor:        Mim ref -> x*y
fun get_node_id:          Mim ref -> int
fun is_node_hidden:       Mim ref -> bool
```

The functions above are used for obtaining information about the mimic-object. `all_regions` returns a list of all mimic regions of the mimic-object. `shown_regions` and `hidden_regions` return lists of the shown/hidden regions. The function `conns_of_regions` returns the list of the mimic connectors which starts or ends in one of the mimic regions listed. The functions `get_regions_coors` and `get_regions_ids` return the coordinates/IDs of a list of mimic regions. Finally, `are_regions_hidden` returns true/false for each hidden/shown region in the given list.

The remaining nine functions have similar effects. They are applied to connectors and nodes instead of regions. The IDs returned by, e.g. `get_regions_ids` are the Design/CPN editor's IDs of the objects. These are used in calls of OA functions. Normally, it should not be necessary to use functions other than those provided by Mimic/CPN, but if it is desired to, e.g. delete an object through simulation, an OA function has to be used.

An example of using an information function:

```
mimic.hide_regions radar (mimic.all_regions radar);
```

which hides all regions on the radar screen — thereby imitating that there is no contact with any aeroplanes.

5.6 Snapshot Functions

The snapshot functions are intended for storing and retrieving “interesting” mimic states. The mimic state consists of the following:

- Node state
  - Coordinates of the node.
  - Text inside the node.
  - Is node hidden/shown?
- Region state
  - Coordinates of the region.
  - Text inside the region.
  - Is region hidden/shown?
- Connector state
  - Text “inside” the connector.
  - Is connector hidden/shown?

The state is saved by creating Mimic/CPN function calls and putting them in special regions of the mimic-object. This code can then be executed whenever the user wants to restore a mimic state. It should be noted that snapshots cannot be restored if, e.g. a region has been deleted manually after the snapshot was made.

If any manual changes are made after a snapshot, you have to report these to the mimic-object by executing the `init` function (from Section 4.2) in order to restore the snapshot correctly. The reason for this is that an internal state of the mimic-object is maintained in order to make the graphic operations more efficient. For instance, calling the `hide` function will not result in the use of the corresponding OA-function (which hides the objects) if the internal state says that the objects in question are already hidden. When restoring a snapshot, the internal state is compared with the state to be restored, and the necessary operations are made. If the internal state did not exist, many time-consuming OA-function calls would be necessary.

```
fun snapshot:          Mim ref -> SnapshotName -> unit
fun restore:           Mim ref -> SnapshotName -> unit
fun remove_snapshot:   Mim ref -> SnapshotName -> unit
fun all_snapshots:     Mim ref -> SnapshotName list
```

By issuing the `snapshot` command, the mimic-object state is saved in a special region structure named by the `SnapshotName` argument. If a snapshot of the given name pre-exists, the user is asked whether or not the old one should be overwritten. The `snapshot` function can, for example, be used to save the mimic state that is meant to represent the initial state of the model. In the security system project, a snapshot of the mimic board from Figure 3 was made by the commands:

```
mimic_board:= mimic.init "MIM" "MIMICS";
mimic.snapshot mimic_board "START";
```

This was done *while editing* the diagram, and not during simulation. The mimic-object (in mimic node ”MIMICS” on page ”MIM”) had previously been put in the graphic state by which we wished to start each simulation. `init` has to be executed before taking the snapshot, and

we recommend using it immediately before the `snapshot` statement, so as to ensure that the mimic-object is up to date.

The `restore` command restores a mimic state of the given mimic-object. If a snapshot of a state has been made, it can be restored by this `restore` function in the beginning of each simulation (after the initialisation of the mimic-object). In the security system example, we succeed the initialisation of the mimic board (in the initialisation transition mentioned in Section 4.2) by a `restore` command:

```
mimic_board:= mimic.init "MIM" "MIMICS";
mimic.restore mimic_board "START";
```

By firing the initialisation transition after each **Initial State** of the simulation it is ensured that the mimic-object has the same appearance in the beginning of every simulation.

The `remove_snapshot` command deletes a snapshot of a mimic-object, while the function `all_snapshots` lists the names of the existing snapshots of the mimic-object.

6 General Functions

The functions described in the previous section all have one thing in common: they need a reference to a mimic-object as argument. The functions in this section are independent of mimic-objects and can be applied at any time.

6.1 Dialogue Boxes

Among the OA-functions in Design/CPN there are four functions which put a dialogue box on the screen and wait for the user's action. The following functions extend the OA-functions with, e.g. error-checking and disabled-interaction-mode behaviour.

```
fun get_integer:  {prompt:string, min:int, max:int, def:int} -> int
fun get_string:   {prompt:string, def:string} -> string
fun get_boolean:  string -> bool
fun get_ok:       string -> unit
```

The `get_integer` function writes the `prompt` text in a dialogue box and waits for the user to input an integer in the range from `min` to `max`, having the default value set to `def`. Clicking the cancel-button or writing an integer out of range has no effect other than inducing a new dialogue box asking the same question. Due to the behaviour of the OA-function applied, `get_integer` returns immediately if a non-integer is entered[7].

`get_string` writes the `prompt` text in a dialogue box and waits for the user to input a string (with `def` as default string) and press the ok-button. Again, the cancel-button has no effect. The `get_boolean` function displays a string in a dialogue box and waits for the user to click the yes-button or the no-button. `get_ok` also displays a string but only has an ok-button to click. See also Section 6.3 for information on how the dialogue box functions behave in disabled-interaction-mode.

In the security system example, the user can select the clock on the mimic board (Figure 3) and then be prompted for a number which will serve as a number of clock-ticks (seconds), simulating that time is passing. We used the following statement to enter this number:

[7] In the ESML version of Mimic/CPN zero is returned while the NJSML version returns the default value

```
mimic.get_integer {prompt="Input number of seconds",
                   min=0,
                   max=1000,
                   def=1};
```

The statement results in the appearance of the dialogue box shown in Figure 7 when the code-segment containing it is executed.

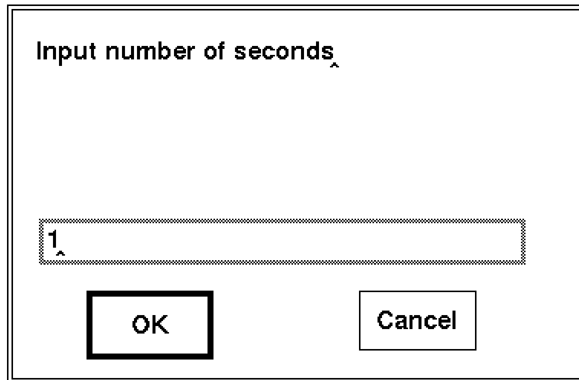


Figure 7: `get_integer` dialogue box (on UNIX version of Design/CPN)

6.2 Miscellaneous

These functions are more or less trivial. Some of them are actually renamings of existing OA-functions in Design/CPN. However, they can be quite useful and are therefore included in Mimic/CPN.

```
fun close_page:          PageName -> unit
fun goto_page:           PageName -> unit
fun write_in_statusbar:  string -> unit
fun beep:                unit -> unit
fun pause:               int -> unit
```

`close_page` closes the page of the given name while the `goto_page` function opens the page, making it the current page. `write_in_status_bar` writes the string in the Design/CPN statusbar, overwriting previous text. `beep` makes the computer beep and `pause` pauses the simulation for the given number of seconds. See Section 6.3 for the behaviour of these functions in disabled-interaction-mode.

In the security system example, we used the statusbar to inform the user about what to do during the automatic simulations. For example, our select-statements looked like:

```
let
  val usercode =
   (mimic.write_in_statusbar "Enter User-code";
    mimic.multi_select code_unit <selection-parameters>)
in
  mimic.write_in_statusbar ("User-code "^usercode^" entered");
end;
```

6.3 Operation Modes

```
fun disable_interaction:  unit -> unit
fun enable_interaction:   unit -> unit
```

By default, Mimic/CPN allows the user to interact with the simulation by for example selecting objects with the mouse. The `disable_interaction` function enters *disabled-interaction-mode* in which some of the mimic facilities are disregarded in order to make simulation faster. The user will not be involved in selections or dialogue boxes anymore. Hence, simulations can be performed without user interaction. The `get_ok`, `goto_page`, `close_page`, `beep` and `pause` functions will have no effect, while `select` and `multi_select` will make selections at random instead of by using the mouse. `get_integer` will choose an integer at random in the given range, and executing `get_boolean` will work like tossing a coin. The `get_string` function will return the default value.

The disabling of interaction can be useful when extensive test simulations are to be made on a model, which has already been tested by hand using Mimic/CPN. Instead of removing all Mimic/CPN function calls, simply evaluate the `disable_interaction` command in an auxiliary box and initiate a simulation. The `enable_interaction` command returns simulations to normal, making all Mimic/CPN functions perform as described in the previous sections.

```
fun disable_error_reporting:  unit -> unit
fun enable_error_reporting:   unit -> unit
```

If the graphic structure of a mimic-object does not fulfil the constraints from Section 3.2 or if Mimic/CPN functions are given wrong arguments, exceptions are raised. In Section 7, we describe these exceptions and discuss how they can be handled. These two functions above influence the way a raised exception will be reported to the user.

By default, Mimic/CPN reports raised exceptions in a dialogue box (the `get_ok` type). A typical report will look like

```
MIMIC EXCEPTION
Unknown_region: (hide_regions) "some-name"
```

which tells the user that the `Unknown_region` exception has been raised with `"some-name"` during a `hide_regions`-operation. In this way, you get to see in which function-call something went wrong.

In Section 7, we discuss the possibility of handling exceptions raised by Mimic/CPN. If you intend to use exceptions in this way, it would probably be annoying to see the abovementioned dialogue box in every simulation even though nothing has gone wrong. By applying `disable_error_reporting`, this dialogue box will be suppressed for all exceptions raised later on (until `enable_error_reporting` is invoked). If an exception is un-handled when error reporting is disabled, the simulation will halt without telling the user which exception was raised.

7 Mimic Exceptions

In this section, we list the exceptions which can be raised by Mimic/CPN and discuss how they can be handled.

7.1 List of Exceptions

The following table lists the exceptions which can be raised in Mimic/CPN and handled in the code-segments using the mimics. For each exception we list the functions that can raise it. Some of the exceptions return an argument which contains the name of the page, node, etc. that caused the problem.

Exception name	Type	Raised by
Unknown_page	PageName	init, close_page, goto_page
Unknown_node	NodeName	init
Unknown_region	RgnName	hide_regions, show_regions, move_regions_abs, move_regions, multi_move, align_regions, align_BETW, align_PROJ, select, multi_select, append_in_region, put_in_region, read_from_region, conns_of_regions, get_regions_coors, get_regions_ids, are_regions_hidden
Unknown_connector	ConnName	hide_conns, show_conns, select, multi_select, append_in_conn, put_in_conn, read_from_conn, regions_of_conns, get_conns_ids, are_conns_hidden
Unknown_snapshot	SnapshotName	restore, remove_snapshot
Identical_node_names	NodeName	init
Identical_region_names	RgnName	init
Identical_connector_names	ConnName	init
Select_error	unit	multi_select
Empty_namelists	unit	select, multi_select
Invalid_alignment_type	unit	align_regions
Range_error	unit	get_integer
Text_overflow	string	append_in_region, put_in_region append_in_conn, put_in_conn, append_in_node, put_in_node

The Unknown_... exceptions are raised when a name (given as argument to a function) cannot be resolved by the function in question. The Identical_... exceptions are raised by init if the mimic-object cannot be initialised because two nodes on the same page have the same mimic name or because two regions/connectors of the same mimic node have the same mimic name.

Select_error is raised in calls to multi_select if escape is true and the selection is aborted or if something else than the specified regions is clicked upon. Empty_namelists is raised by the two select functions if the list of regions names and the list of connector names are empty. Invalid_alignment_type is raised when the align_regions function is called with an illegal alignment type. Range_error is raised by get_integer if min is greater than max or if def is out of range. The Text_overflow exception is raised when the text of the node/region/connector in question exceeds 4084 characters. The string parameter names the node, region, or connector that caused the problem.

7.2 Handling Exceptions

The exceptions which can be raised in Mimic/CPN all somehow indicate that something has gone wrong. However, they can be handled in the code-regions calling the mimic-functions, thereby inhibiting a termination of the simulation. An example of this is:

```
mimic.multi_select obj {regions = [],
                        conns = ["A","B"],
                        startvalue = "",
                        term = fn (s1,s2) => s2="B",
                        comb = fn (s1,s2) => s1^s2,
                        escape = true}
  handle mimic.Select_error => "B";
```

Here, you can interrupt a selection because `escape` is set to `true`, but then the "B" string will be the result (as a default value). Thereby, exception handling will operate as a default-mechanism in this case. Another example could be:

```
mimic.hide_regions obj ["some_unknown_name"]
  handle mimic.Unknown_region rgn_name => ();
```

which implies that the simulation can continue without terminating even if the `hide_regions`-operation does not succeed. The `rgn_name` reported by `Unknown_region` will in this case be `"some_unknown_name"`. Note that you have to write the `mimic.`-prefix in order to capture the exceptions (unless the mimic structure has been opened).

If you handle the Mimic/CPN exceptions in your model, you probably do not want to see the error dialogue box in every simulation. `disable_error_reporting` will stop the error reporting as described in Section 6.3.

8 Limitations

Initially, Mimic/CPN only allowed hiding, showing, and selecting of regions. In this way, it was easy to restore old states of mimic-objects — only few hide/show commands were needed. Later on, we introduced functions for moving and aligning, functions for text, and functions which operate on connectors. In order to be able to restore old states, we made the snapshot-mechanism, which can restore object positions, text in objects, and visibility of objects. Having constructed this snapshot-mechanism it will also be possible to make mimic-functions which, e.g. change the size of regions.

We have already implemented functions for changing some graphic attributes, but we have not included them in this version, because they need to be properly checked and because we would like to have all the relevant functions available. Until then, OA functions (and `get_..._id` functions) must be applied if object appearances (e.g. sizes and colours) are to change during simulation.

At a later stage, functions that allow dynamic creation and deletion of objects might be considered. This would for instance be useful in the telephone example, where the connectors could be constructed when needed.

Acknowledgments

Mimic/CPN has been developed as part of our master's thesis project. Many valuable ideas and comments have been given by the other parties of this project: Kurt Jensen, Søren Christensen, John Mølgaard, and the Dalcotech employees who worked on the project.

References

[CPN93] Meta Software Corporation, Cambridge, MA, USA. *Design/CPN Reference Manual for the Macintosh (vers. 2.0)*, 1993.

[Dal94] Dalcotech A/S. *PRISMA C-91 System Manual*, 1994.

[Jen92] Kurt Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use, Volume 1.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Defintion of Standard ML.* MIT press, 1990.

[OAF93] Meta Software Corporation, Cambridge, MA, USA. *Design/CPN Internal Functions Programmer's Reference (vers. 2.0)*, 1993.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Ull94] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice-Hall International Editions, 1994.

Function index

align_BETW, 13
align_PROJ, 13
align_regions, 13
all_conns, 16
all_regions, 16
all_snapshots, 17
append_in_conn, 15
append_in_node, 15
append_in_region, 15
are_conns_hidden, 16
are_regions_hidden, 16

beep, 19, 20

close_page, 19, 20
conns_of_regions, 16

disable_error_reporting, 20
disable_interaction, 20

enable_error_reporting, 20
enable_interaction, 20

get_boolean, 18, 20
get_conns_ids, 16
get_integer, 18–20
get_node_coor, 16
get_node_id, 16
get_ok, 18, 20
get_regions_coors, 16
get_regions_ids, 16
get_string, 18, 20
goto_page, 19, 20

hidden_regions, 16
hidden_conns, 16
hide_conns, 12
hide_node, 12
hide_regions, 12, 16, 22

init, 11, 17, 18
is_node_hidden, 16

move_node, 12
move_node_abs, 12
move_regions, 12
move_regions_abs, 12
multi_move, 12, 13
multi_select, 14, 15, 20, 22

pause, 19, 20
put_in_conn, 15
put_in_node, 15
put_in_region, 15, 16

read_from_conn, 15
read_from_node, 15
read_from_region, 15
regions_of_conns, 16
remove_snapshot, 17
restore, 17, 18

select, 14, 20
show_conns, 12
show_node, 12
show_regions, 12, 13
shown_conns, 16
shown_regions, 16
snapshot, 17

write_in_statusbar, 19