Towards a Visual Notation, and Editor, for User Interface Design

Ian Rogers

School of Cognitive and Computing Sciences
Sussex University, Falmer, E. Sussex, UK.

with

Dr. Jonathan Cunningham

British Maritime Technology Ltd.
Orlando House, Teddington, Middlesex, UK.

Prof. Aaron Sloman

School of Computing Science
Birmingham University, Edgbaston, Birmingham, UK.

ABSTRACT

A visual programming language and editor is described that aims to support User Interface design through rapid, exploratory programming.

This paper describes work in progress in the User Interface Design Environment (UIDE) project (DTI/SERC: IED 4/1/1577), due for completion in August 1993.

CONTENTS

LIST OF FIGURES

Part 1 - Preface

1 INTRODUCTION

This paper describes work in progress in the UIDE[1] project, due for completion in August 1993.

1.1 Objective and Significance

The project is attempting to provide a toolset (UIDE-2) that aids the User Interface (UI) design process by supporting the opportunistic design patterns exhibited by expert designers[12].

A number of tools are available that allow a designer to specify the look of the UI by demonstration: X-Designer, DevGuide[18], FaceMaker etc. But this type of tool provides little or no help in designing the behaviour of the UI. Also, despite their claims for ease of use, they are extremely viscous [5] in that high level design decisions are not supported. They are very good at allowing the designer to make small, cosmetic changes, but it is not easy for the designer to make large changes while retaining the overall feel of the UI.

A number of other tools approach the problem of UI design from the "opposite" direction, that of the behaviour or "feel" of the UI: Garnet [14], Fabrik [9], Authorware etc. This type of system extracts the essence of UI behaviour programming into a visual language or a form-filling style of programming, but they provide little, or no, support for rapid exploration of different designs for the look of the interface.

There are systems that are having some success in trying to integrate these two aspects of UI design. E.g. the ACE system provides a set of pre-coded C++ "Selectors". The designer decides which type of data the user of the UI should provide, and the system chooses an appropriate selector. But this type of system still does not directly support high-level design decisions, or automate the UI layout based on machine readable style guidelines.

The UIDE-2 system attempts to support both these aspects of UI design through two integrated tools. One is a fluid, Object Oriented notation that supports rapid experimentation. The other is a constraint reasoner which automates the mundane aspects of laying out a UI.

1.2 Methods

In the UIDE-2 system the designs are as abstract, or as concrete, as the designer wishes. The Object Oriented aspects of the notation allow the designer to encapsulate designs

1. DTI/SERC Project: IED 4/1/1577

into a higher level class, to expand and re-implement part of a class, or to make a class more specific for a particular implementation.

A visual language is used to specify the behaviour of the interface. The language represents a stylised form of Object Oriented programming, appearing much like a dataflow language. Each node corresponds to an instance of a class, and the links denote messages that can be sent between them. Some nodes represent graphical UI components while others represent computational or control structures.

The UI nodes represent a UI component with as much abstraction as possible. An integrated constraint reasoner (based on an Assumption-based Truth Maintenance System) [13, 22] automatically decides which UI component, or components, to use and lays them out, as an interface, in accordance with style guidelines.

A librarian mechanism is supported which allows a designer to search for existing behaviour classes. Also, designers are encouraged to add their own classes to the library (which can be partitioned into public and private areas). In this way code re-use and collaborative work are supported by different team members working on different parts of the project, or by some team members providing low-level utilities for the high-level designers.

The cognitive load on the designer is greatly reduced by being able to rapidly scan a library and experiment with any suitable classes that are found. The automatic UI layout tool means that adding nodes to, or removing nodes from, the behaviour diagram is very easy.

1.3 Current Status

This particular graphical notation is still very new. Therefore the library is fairly empty of high-level, abstract behaviour classes.

1.4 Acknowledgments

The partners on the UIDE project are: Birmingham University, British Maritime Technology Ltd., Integral Solutions Ltd. and Sussex University.

Jonathan Cunningham of BMT was a major source of ideas on the Sibal implementation and comments on later revisions. All partners in the project produced comments on the working documents that produced this paper, but particular thanks are due to Aaron Sloman of Birmingham University and Alan Montgomery of ISL. Ben Rabau of ISL produced the GO (Graphics Objects) library which made the Behaviour Editor possible.

1.5 Intended Audience

This document discusses how OO behaviour notation has been implemented. The reader should have a knowledge of object-oriented programming, and Pop11 features such as: procedure composition (pdcomp), closures, processes, and the open stack.

The “objectclass” OOP system is used to provide classes, inheritance and methods.

1.6 Document Organisation

The next 2 parts of this document were originally internal design documents of the UIDE project. They reflect the working-design of UIDE-2 and as such may not correspond exactly with the final implementation of UIDE-2.

Part 2 concerns the semantics and implementation of the Sibal architecture.

Part 3 describes the visual notation for Sibal, called Behaviour Diagrams, and a design for an editor to support the notation.

It should be possible to read the two parts separately if desired.

Part 4 describes areas of the Behaviour editor that could provide the basis for future research.

Part 2 - Sibal Semantics and Implementation

2 INTRODUCTION

Sibal stands for "Specifying Interface Behaviour And Layout". It forms the underlying architecture to support a visual programming language, called Behaviour Diagrams. It is hoped that UI designers using UIDE-2 will use behaviour diagrams for most of the UI programming task.

This part of the paper describes the Sibal architecture, semantics and implementation.

The behaviour of a UI is defined by a network of Sibal objects. The behaviour of the network is defined by the **ports** on the objects, the **links** between them. The network acts much like a dataflow diagram in that behaviour is composed of **events** in which an item, or collection of items, is transmitted along a link. It is also possible to pass "control" along a link without passing any data items.

There are four types of ports:

- Ask
- Answer
- Receive
- Transmit

There are two types of links:

- Fetch
- Pass

Example network diagrams will be shown in order to illustrate implementation issues. In these cases the notation of Figure 1will be used. Part 3 of this paper describes the visual notation fully. The arrows indicate the direction of data-flow. If the arrow is outside the box then it is an "instigator" of control. If the arrow is inside the box then it passively responds to control.

As indicated in Figure 1, Answer and Transmit ports are "source" ports (relative to a link) and Ask and Receive ports are destination ports. For a given link between objects O1 and O2 one port must be a source port and the other a destination port. The objects containing those ports are source and destination objects, **for that link**. A link can use the same object as source and destination.

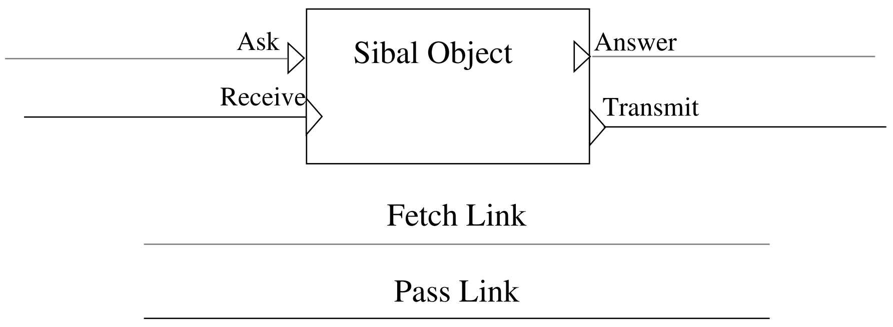


Figure 1 : Behaviour Diagram notation

Ask and **Answer** ports are linked by **Fetch** links. **Transmit** and **Receive** ports are linked by **Pass** links. Ports and links can have an "arity", an integer representing the number of items transmitted at each behaviour event (Pop11 also supports variadic ports and links). The number of values transmitted can be one or more for fetch links, and zero or more for pass links. Ask ports and Transmit ports are "active" and can initiate behaviour. Answer ports and Receive ports are passive, but they must respond to a request by an "active" port. Constraints on types of links and ports are mentioned below.

Buffered links are also supported, and can be used to support explicit concurrency in simulations, etc.

This paper lists the kinds of Sibal objects that already exist or are planned. The purpose of the paper is to define the functionality of the objects that will be provided, at a fairly abstract level, and to define the input and output ports for each class of object. Behaviour diagrams specifying these objects with pass and fetch links can determine the behaviour of a particular interface. For interfaces permitting dynamic creation and linking of objects additional notation is required (described in chapter 24).

2.1 Related work

[16] discusses optimising compiler techniques using a "dual graph" notation. The two types of links are **control** and **data** which correspond fairly closely to pass and fetch links respectively. The objects in these dual graphs are very primitive, sometimes representing a single machine code instruction. The links are also very simple. The data links correspond to a single data item; often a single word of store in memory. The control links represent pure "flow of control" information, and carry no data.

This makes dual graphs very amenable to code optimisation (the thrust of his thesis), but the graphs are very unwieldy.

The design of Sibal concentrates on providing very high-level control structures and primitives. This should enable the programmer to easily write powerful programs.

[8] gives a survey of a large number of visual languages, but does not discuss their implementation.

Statecharts [7] represent behaviour in terms of state transitions for a total system. Sibal Behaviour Diagrams correspond more closely to an architectural specification for a system. Side-effects of Sibal events can be thought of as state transitions. Thus Sibal Behaviour Diagrams and Statecharts complement each other by providing different views of a system.

3 FETCH LINKS

A fetch link can be made between an **answer** port on its source object, to an **ask** port on its destination object. The link corresponds to an "if-needed" call by the destination object to the passive source object. A fetch link always transfers at least one value (which may be a complicated structure, nested list etc.) or a tuple of objects.

3.1 Answer

An answer port corresponds to a *value* in an object. For example, a slider object has a value (set by moving the slider "thumb"). In some cases, the answer port corresponds to a tuple of values.

Implementation note: an answer port name is the name of a method, which, when applied to an instance of the appropriate class, returns the value or values

The simplest examples of answer port values are provided by instance variables (which behave like methods). In this case the name of the port is the name of the instance variable (because in objectclass, the name of an instance variable has as its value the accessor **method** for that slot). Note that the definition of an answer port value does not imply that there must be a corresponding instance variable. For example, the *value* of a slider may be computed, by a method, e.g. by interpolating between *low* and *high* instance variables using the position of the thumb.

3.2 Ask

Perhaps surprisingly, an ask port does **not** correspond to the updater of a value. Instead, it corresponds to the place where the information as to how to get the required value is stored. For example, suppose we have a fetch link from a slider to an object containing a gauge. We may wish the value of the gauge to be set from the value of the slider. Since fetch links are passive - they fetch values when needed - we need some method in the gauge that will access the slider value. This is achieved by storing in the gauge object a procedure of no arguments, which when called will return the answer value.

Implementation note: Storing the answer method may use an instance variable, and it is the name of **this** instance variable which is the name of the ask port. Note that whilst answer ports **may** use an instance variable this is not necessary, but ask ports are **always** associated with a corresponding instance variable (there are exceptions to this, described in section 4.5).

3.3 Fetch links

The construction of a fetch link can be summarised by the following implementation hint. This is intended to be suggestive rather than prescriptive. In the example, we assume that the *name* of a port is actually the method itself, rather than a word naming the method (this is to avoid problems with pop11 sections).

Implementation hint:

```
define:method make_fetch_link(
        ans_obj:sibal_object,  ans_slot:sibal_port,
        ask_obj:sibal_object,  ask_slot:sibal_port
    );
    ans_slot(% ans_obj %) -> ask_obj.ask_slot;
enddefine;
```

The answering (responding) method is closed on the answering instance to form a procedure. This procedure is then stored in the ask slot of the asking (initiating) instance.

Note: an Ask port can be connected to several answer ports. This poses no problem, the answer method is simply stored in many places (i.e. in **each** of the ask slots).

4 PASS LINKS

Pass links are used for forward chaining links, ie they correspond to data driven (or event driven!) computation. They have a more active kind of "feel" to them. Most existing visual programming languages are mainly based around this kind of dataflow. (Although there are sometimes differences that add a hint of "fetch" to the links.) Pass links pass flow of control, in that if A links to B links to C, then activating A will cause control to be passed to B. The receive method in B will then, typically, pass control to C.

A stack tuple of zero or more data items is also passed with the control event. The data items correspond to input arguments of the receive method.

4.1 Receive

A receive port is the complement of an answer port. Like an answer port it is the name of a method - the method that is to receive control as the control flow is passed forward. It is this method which receives the value passed forward (data flow) by transmitting objects.

Implementation note: A receive port is the name of a method, which should do whatever is appropriate when control is transferred to it. The data items passed along the link are taken as input arguments to the method (as well as the destination object itself). A zero arity input port takes no arguments (apart from the destination object itself). The method is also responsible for passing on control to the transmit ports of the object. There is an example in section 4.4.

4.2 Transmit

A transmit port is the complement of an ask port. It is the place where the information as to how to pass on control is stored. As such, like an ask port, it requires an instance variable.

<u>Implementation note:</u> The name of the receive port is the name of an **instance variable**. The value of this instance variable is a **procedure** that will receive control when a pass event occurs.

We now give an example of how an "incrementor object" might have its behaviour defined. Its semantics is to receive an integer, increment it, and then pass it along immediately. We assume it has two "pass/1" ports: an input port and an output port. (These are parts of two **different** pass/1 links - if they were the same link, then the incrementor would go into an infinite recursion.)

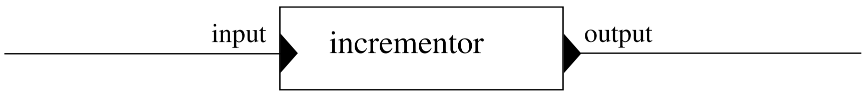


Figure 2 : A hard-coded incrementor

In Figure 2, the receive/1 port is called *input* and the transmit/1 port is called *output*. This is both arbitrary and unimaginative. Other names could be used. This is the Pop11 code that could implement the incrementor:

```
define:objectclass incrementor;
    output = erase; ;;; default for pass/0 output is identfn
enddefine;

define:method input(value, obj :incrementor );
    (obj.output)( value + 1 );
enddefine;
```

The default value of the output port is the procedure `erase`. This procedure consumes one item off the stack and then returns. It is useful in that it allows the incrementor to work before it has been connected to anything.

The incrementor object is an example of a Sibal behaviour object that has no representation in a User Interface (unlike the slider or gauge described below).

4.3 Pass links

We link objects with pass links in a very similar way to fetch links - but the other way around! As for fetch links, this is best explained by the implementation hint:

Implementation hint:

```
define:method make_pass_link(
        tran_obj:sibal_object, tran_slot:sibal_port,
        recv_obj:sibal_object, recv_slot:sibal_port
    );
    recv_slot(% recv_obj %) -> tran_obj.tran_slot;
enddefine;
```

The receive procedure, of the appropriate number of arguments, is constructed by forming a closure of the receive method on the receiving object. This procedure is then stored in the transmit slot of the transmitting object.

4.4 Example Pass Link

The following example, shown in Figure 3 shows two objects linked by a pass link.

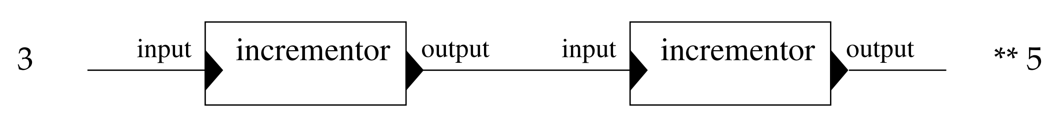


Figure 3 : Two linked incrementors

In raw Pop11 this is implemented as follows:

```
newincrementor() -> i1;
newincrementor() -> i2;
make_pass_link( i1, output, i2, input );
```

Now if we pass a number into **i1**, the code increments it twice and eventually calls the output method of **i2**. This can be seen replacing the `erase` in **i2**.

```
sysprarrow(% false %) -> i2.output;
```

Now we call the input method of **i1**.

```
input( 3, i1 ); ;;; send i1 an event with value 3.
```

Which prints the value.

```
** 5
```

Note that the same linking mechanism can be used for pass links of any arity. It is also possible to link a transmit port to a receive port of lower arity, but in this case it is necessary to use `pdcomp` with `erase`, or something similar, in order to erase the unneeded value from the stack. This is discussed further in section 6.4. It is not meaningful to connect a transmit port to a receive port of higher arity.

A transmit port can be connected to several receive ports. This requires the use of a procedure composition mechanism, as described in section 6.1.

4.5 Instance variables

Actually, neither receive ports nor ask ports need to correspond to instance variables. All that is **really** necessary is that they have methods and updaters for those methods. The updaters should expect to be used by the linking mechanism, described above, to store the pass/fetch procedures respectively.

5 GUI objects

Primitive GUI objects will be implemented (initially) as raw Sibal objects. Ie. either as objectclass objects, or objectclass wrappers around other types of code.

The following diagram shows a simple user interface. It contains a slider and a gauge. When the slider is moved, the pointer on the gauge changes to reflect the value of the slider.

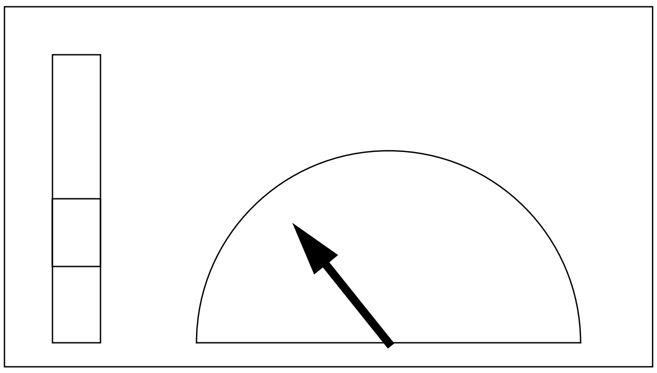


Figure 4 : Slider and Gauge user interface

This could be implemented by the following Sibal network, given the appropriate Sibal primitive objects.

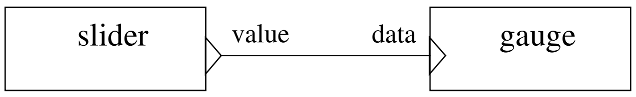


Figure 5 : Slider and Gauge network

A collection of GUI Sibal objects, and their associated links, can implement a whole user interface. Chapter 11 contains a description of how these user interface descriptions would be stored off-line.

Not every object in a Sibal behaviour diagram need correspond to a screen object. Some objects may represent applications that generate or request data for, or from, the interface. Some objects may represent transformation functions, buffers, integrators, logging mechanisms, error monitors etc.

Although a Sibal behaviour diagram specifies all the objects that compose the interface, it does not specify the appearance of the interface (ie. the layout on the screen).

One module of the UIDE2 architecture is a layout-constraint reasoner. It is this module that constructs a default layout of all the objects in the UI. It contains a set of heuristics, derived from style guides, which are used to generate a default layout of the interface. The designer can specify constraints that may override the defaults. In the example above, the designer may have specified that the slider should be "to the left of" the gauge.

6 MEANINGFUL LINKS

- It is not possible to connect two ports of the same behaviour type (e.g. ask to ask).
- It is not possible to connect a transmit port to a receive port of higher arity.
- It is not possible to connect an ask port to an answer port of lower arity.

The following four sections describe the labels shown in the table below.

Transmit							
Yes	Receive						
No	update-demon	Answer					
buffer	No	Yes	Ask				
No	Yes	No	buffer	Multiple Transmit			
pdcomp	No	update-demon	No	pdcomp	Multiple Receive		
No	update-demon	No	No	No	update-demon pdcomp	Multiple Answer	
pdcomp buffer	No	Yes	No	pdcomp buffer	No	No	Multiple Ask

Figure 6 : Meaningful links

6.1 pdcomp

As described in section 4.3, when a **transmit** port is connected to a **receive** port (a Pass link) the receiving object supplies a method that the transmitting object stores, and uses, to pass on the event. If the transmit port is connected to multiple receive ports, then many methods have to be stored in the transmitting object and applied in turn. A mechanism like `pdcomp` is needed concatenate the methods. For links of arity greater than 0, a mechanism is also needed to push copies of the arguments onto the stack before each receive method is called.

6.2 buffer

If a connection is attempted between a **transmit** port and an **ask** port, a buffer is automatically created. The transmit port places data into the buffer at will. The ask port fetches data from the buffer independently of the behaviour of the transmit port. When a new piece of data is transmitted to the buffer, it overwrites the item that is already there. If a fetch is attempted before any data has been placed in the buffer, the buffer is at liberty to return an undefined data item. Each data type has a notion of its "default value". The default value of the buffer will be the default value of the data type of the linked-to object.

6.3 update demon

It is conceivable that an answer port could be turned into a transmit port if the port refers to a simple value slot in the object. If an "update demon" was placed on the slot, an event could be generated, and transmitted, whenever the slot was updated. Initial versions of Sibal may not support promoting answer ports in this way.

6.4 Arity Differences

A source port must have a equal or greater arity than the destination port it is connected to. That is:

- It is not possible to connect a transmit port to a receive port of higher arity.
- It is not possible to connect an ask port to an answer port of lower arity.

If these restrictions were not followed, then Sibal would have to "invent" data items to satisfy the demands of the destination (receive or ask) port.

The question that remains is what to do with superfluous values. The convention in Pop11 is that the right-most arguments correspond with the right-most parameters (cf. closures binding the right-most formal parameters). The solution to superfluous values is to simply delete the values that correspond to the **left**-most arguments. This ensures that right-most common arguments always map to each other using the Pop11 stack mechanism for passing arguments.

<u>Implementation hint:</u>

The following procedure takes three argument: the arity of the source and destination ports respectively, and the "responding" method (this corresponds to a **receive** or **answer** method)

```
define make_handler(src, dst, meth);
    lvars src, dst, meth,
        ;
    if src == dst then
        meth
    else
        procedure(kill_num, save, numsave);
            lvars kill_num, save, numsave;
            fill(save) -> ;    ;;; save rightmost args
            erasenum(kill_num);;;; lose unneeded args
            explode(save) -> ; ;;; replace rightmost
            ;;; and so that args can be garbaged ...
            set_subvector(0, 1, save, numsave);
        endprocedure(% src - dst, initv(dst), dst %)
        <> meth
    endif;
enddefine;
```

7 EXECUTION SEMANTICS

A sibal graph has no execution semantics of its own. The behaviour of the network is dependent wholly on the methods that define each object. These methods can contain arbitrary code and may therefore produce arbitrary side-effects, including switching to a new context, or re-configuring the current interface.

7.1 Control of Control

At the implementation level, there is no way to prevent misuses of the fetch and pass linking mechanisms to implement other weird and wonderful control flows. Arbitrary procedures can be stored in a transmit port or an ask port. For example, a fetch "if-needed" method could treat the access of a value as an "event" and send activation down a pass link.

However, if only the linking mechanisms outlined are used, and if all objects with "ports" are written according to these conventions, (such as that "fetch" links should not cause side effects) then the control mechanisms will work cleanly as intended. Control will **forward** chain along **pass** links, and may **backward** chain along **fetch** links. By convention, control should not pass from fetch links to pass links.

See section 21.1 for further thoughts on extending control of control.

8 PORT NOTATION

Port titles consist of

- A name

and three pieces of type information

- Behaviour type (ask, answer, transmit, or receive)
- Data type (integer, list, pop11 etc.)
- Arity

The following is a grammar for the port titles.

Round brackets are literals and are part of the port notation. Square brackets denote optional segments. Bar means a choice. Curly brackets are a grouping notation for the BNF. All whitespace is removed from the final port title.

Port	:==	Name [Type]
Name	:==	<any Pop11 word>
Type	:==	([Behaviour-Type /] Arity [- Data-Type]) | ([Pass-Behaviour /] 0)
Behaviour-Type	:==	Pass-Behaviour | Fetch-Behaviour
Pass-Behaviour	:==	transmit | receive
Fetch-Behaviour	:==	ask | answer
Arity	:==	<positive, non zero integer>
Data-Type	:==	<Pop11 type> | <Pop11 type> * Data-Type

If a port doesn't have a Behaviour-Type, it is presumed to be both a **receive** and **answer** port. Ie. the receive port is implemented by the **updater** of the answer port

If a port doesn't have a Data-Type it is presumed to be the unifying type "pop11"

Some ports can have an arity higher than 1. This is represented by a sequence of data types separated by an asterisk (e.g. **int*string**[1]). These "compound" data items are implemented by **stack tuples**, effectively putting more than one item on the stack at a time.

E.g.

```
/***
    passive buffer - value(1-Pop11)
 ***/
define:objectclass passive_buffer;
    value = undef;
enddefine;
```

1. This part of the type notation is borrowed from the ML tuple notation [21]

```
/***
    passive queue - value(1-Pop11)
 ***/
define:objectclass passive_queue;
    queue = [];
enddefine

define:method value(q:passive_queue);
    if q.queue == [] then
        undef
    else
       dest(q.queue) -> q.queue ;;; the value is on the stack
    endif
enddefine;

define:method updaterof value(val, q:passive_queue);
    lvars Q = q.queue;
    ;;; add -val- to the end of the queue
    if Q == [] then
        [^val] -> q.queue;
    else
        ;;; q.queue already holds the list,
        ;;; so no need to put it back into queue.
        [^val] -> fast_back(lastpair((Q));
    endif;
enddefine;
```

8.1 Port Signatures

The port signature of an object consists of the class name and a list of port titles. For example:

multiplex	-	value(receive/1-Pop11), list(transmit/1-list), no_more(receive/0)

See chapter 9 for an explanation of this signature, and for further examples.

9 EXAMPLES

9.1 Control Objects

Here's an initial list of "control" objects that could be predefined primitives in Sibal. Experience will determine which subset is actually useful.

Each definition consists of the object's port signature, followed by a brief description of the semantics.

buffer	-	value(1-Pop11), got_one(transmit/0)

Receives a pass event on the **value**(receive/1-Pop11) port, and then splits the data from the control event (the value is buffered in the object, and a pass/0 event is sent out of the **got_one**(transmit/0) port). Values are not queued. Whenever a pass/1 is received the old value is over-written.

gather	-	value(ask/1-Pop11), propagate(receive/0), value(transmit/1-Pop11)

Gathers together a pass/0 control event and a data item (gained through the **value**(ask/ 1) port) and sends them off together from the **value**(transmit/1) port.

queue	-	value(1-Pop11)

Receives pass/1 events, and keeps the value in a queue that can be accessed by the **value**(answer/1) port. The values that are returned when the queue is empty, and the initial value of the answer port, are undefined.

listify	-	value(receive/1-Pop11), list(transmit/1-list), no_more(receive/0)

Repeatedly receives items on the **value**(receive/1) port and collects them into a list. When a pass/0 event is received on the **no_more**(receive/0) port, the list is sent out through the list(transmit/1) port. The internal list is then reset to nil.

delistify	-	list(receive/1-list), value(transmit/1-Pop11), no_more(transmit/0)

Receives a Pop11 list and transmits each item, in turn, through the **value**(transmit/1) port. When the end of the list is reached, a pass/0 event is sent from the **no_more**(transmit/0) port.

If a new list is presented on the receive port before a previous list is exhausted, the old one is replaced by the new. Items are then transmitted from this new list, starting at the beginning.

filter	-	value(receive/1-Pop11), function(receive/1-procedure), value(transmit/1-Pop11)

Receives an item through **value**(receive/1), applies a Pop11 function (gained through **function**(receive/1-procedure)), and passes it out through **value**(transmit/1). Compare `switch-on-eq`.

constant	-	value(answer/1-Pop11)

Holds a constant value, which can be attained through the **value**(answer/1-Pop11) port.

switch-on-type	-	data(receive/N-Pop11), data(transmit/N-T)*

This object has a arbitrary number of transmit ports. The types **T** on the transmit ports have to be equal, or sub, classes of the type on the receive port. The data types of the input data are analysed. The transmit ports are analysed in turn for a port whose data types are all equal, or super, classes of the actual data. The data is sent out of the first port that matches. As implied, transmit ports of the same type are redundant.

switch-on-bool	-	data(receive/N-Pop11), switch(ask/1-boolean) true(transmit/N-Pop11), false(transmit/N-Pop11)

When a pass event is received by the **data** port, the **switch** port asks for a boolean. If the boolean is false, then the data is sent out of the **false** port. If the boolean is non-false, the data is sent out of the **true** port.

switch-on-eq	-	data(receive/N-Pop11), eqpdr(ask/1-procedure) true(transmit/N-Pop11), false(transmit/N-Pop11)

This is a special case of `switch-on-bool` in that the **data** is first passed through the procedure in **eqpdr** to get the boolean to determine the switch.

timer	-	usecs(ask/1-int), tick(transmit/0) start(receive/0), stop(receive/0)

After the **start** event has been received, a system timer is started such that an event is generated on the **tick** port every **usecs** microseconds until a **stop** event is received. Note that, although the interval is specified in microseconds, the resolution of the timer is dependant on hardware. Ie. the interval will be rounded to the nearest appropriate interval according to the hardware (e.g. rounded **up** to 1/100th second on VAXen and Sun-4 systems, 1/50th sec on MC68000 systems, etc.)[1].

delay	-	data(receive/N-Pop11), data(transmit/N-Pop11), usecs(ask/1-int)

When a pass event is received by the **data** port, a delay of **usecs** microseconds occurs before the event is passed on via the transmit **data** port. The data is unchanged in any way. The resolution of the delay suffers the same restrictions as the `timer` object.

1. This information is taken from the Poplog reference entry: REF * SYS_TIMER

schedule	-	data(receive/N-Pop11), data(transmit/N-Pop11), scheduler(ask/1-scheduler)

When an event is received by the **data(receive)** port it is not sent out of the transmit **data(transmit)** port immediately. Instead, a closure is made of the transmit method and the data items. This closure is then placed, as a job, on the scheduler attached to the **scheduler** port. It is then up to the scheduler to determine the appropriate time to transmit the event (i.e. after other events on the scheduler have been dealt with, see section 9.2). Normal usage is to connect **many** schedule objects to a single, or few, schedulers.

scheduler	-	jobs_pending(answer/1-boolean), do_next_job(receive/0), scheduler(answer/1-scheduler)

See section 9.2 for a description.

9.2 Scheduling and Parallelism

Scheduler objects are objectclass classes with the following characteristics:

- They must be a subclass of `scheduler`
- They must respond to, at least, the methods: `jobs_pending`, `do_next_job`, `add_job`

`add_job(job:procedure, priority:number, s:scheduler)`

This is the only method actually required by the Sibal schedule object. It enables the Sibal object to place a job onto the scheduler `s`.

`jobs_pending(s:scheduler)`

`do_next_job(s:scheduler)`

These are used by the Poplog top-level control loop to discover any jobs that are pending, and to execute.

Possible schedulers include:

- First In First Out (**fifo**) - In this case the priority number is ignored. Each new job is placed in a queue behind any others that might be waiting.
- prioritised fifo (**p_fifo**) - A priority number decides where in the queue a job should go, but it still has to wait in line behind jobs of equal or higher priority
- Pre-emptive Round Robin (**perr**) - A Poplog process is wrapped around each job, which is then placed in a circular queue. Each job then gets a time

slice before being interrupted so that the next job gets a chance. The job is only removed from the queue when the process terminates naturally.

- Time Delayed fifo (**td_fifo**), Time Delayed perr (**td_perr**) - With these schedulers, the priority number is actually a time delay in microseconds. The job is put on hold for this length time, and then it is put into the queue. Unlike the `delay` Sibal object, this doesn't block, thus allowing other processes to execute.
- Multiple Agenda - several queues are maintained, one for each priority level. Each queue is treated as fifo, but a policy procedure determines the proportion of times or job slots to give to each queue. E.g. if there are 3 queues, then no. 1 might get 3 jobs in every six, no. 2 might get 2 jobs in every six, and no. 3 might get 1 job in every six. The policy could be a pattern-list e.g. [1 2 1 2 1 3] (this mechanism guarantees that nothing will be stuck in a queue forever).

9.3 Example: fifo scheduler

Implementation hint:

```
define:objectclass fifo;
    isa sibal_object;
    isa scheduler;
    job_queue == [];
enddefine;

define:method jobs_pending(f:fifo);
    lvars f;
    not(f.job_queue == []);
enddefine;

define:method do_next_job(f:fifo);
    lvars f;
    if jobs_pending(f) then
        fast_chain(fast_destpair(f.job_queue) -> f.job_queue);
    else
        mishap(f, 1, 'No jobs pending');
    endif;
enddefine;

define add_job(j, f:fifo);
    lvars q, b, f, procedure j;
    ;;; typing j to be -isprocedure- means that we
    ;;; can use -fast_chain- elsewhere

    f.job_queue -> q;
    if q == [] then
        [^j] -> f.job_queue;
    else
        ;;; find the end of the list and add the new job.
        ;;; f.job_queue already holds the list,
        ;;; so no need to put it back in.
        [^j] -> fast_back(lastpair((q));
    endif;
enddefine;
```

9.4 Example: Maplist

The following is a Behaviour Diagram for a "maplist" function. That is, a function that applies a procedure, in turn, to each item in a list, and makes a new list out of the results. The diagram has been defined using only the primitives given in section 9.1
.

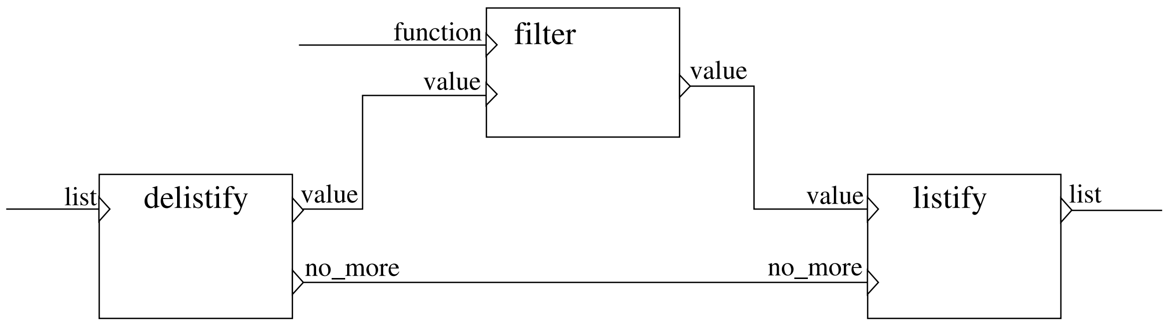


Figure 7 : -maplist- Behaviour Diagram

Encapsulating this network as a new class, called maplist, would produce the port signature:

maplist	-	list(receive/1-list), function(receive/1-procedure), list(transmit/1-list)

10 FOUR TYPES OF OBJECTS

There are four ways of implementing Sibal objects:

- Raw Sibal
- A Pop11 Function
- Encapsulated Sibal
- Constant object (a closure on `identfn`)

10.1 Raw Sibal

"Raw" Sibal objects are those which are constructed by Objectclass classes and methods (as has been described above).

10.2 A Pop11 Function

A sibal object that acts as a filter could easily be implemented by a pop11 function:

- Receive ports correspond to input parameters
- Transmit ports correspond to output parameters

E.g. the Pop11 procedure `rev`, which reverses a list, has the signature:

rev	-	input(receive/1-list), output(transmit/1-list)

Connections to the underlying application could be made via this kind of Sibal object.

Procedures with more than one input parameter (e.g. the arithmetic operator "plus") pose a problem. The arguments could all be passed together as a stack tuple (ie. a port with arity greater than one), but more interesting behaviour can be achieved if the input parameters are given a port each.

The definition of Sibal given above shows that receive ports, on the same object, are expected to operate independently. But the input arguments of a procedure (ie. the receive ports) must, effectively, accept pass events **simultaneously**.

This can be handled by nominating one of the input arguments as "special". A pass/1 event arriving at this port will cause the procedure to be executed. The other ports are buffered. A question remains as to whether the buffered values should be queued, or thrown away in the case where a new value arrives before the special port is fired. This could be an option set by the programmer at design time.

The nomination of the special port could be done at design time (ie. when the pass link is created).

Another possibility is that each of the receive ports would check to see if all the other ports have received their data item, ie. all the parameters are satisfied. When all items are received the procedure is fired and all the input ports are cleared ready for the next set of arguments. Alternatively, the values could be left in the ports - the procedure would be fired whenever a new piece of data arrived on any of the ports. This is similar to the normal data-flow execution metaphor [20].

Sibal "constant" objects (described in section 10.4) could be used to supply the values of some of the arguments. If all but one of the ports are bound to constant objects, then the remaining port is automatically promoted to be the "special" port.

So, another way of implementing the "incrementor" object (described above) could be as follows:

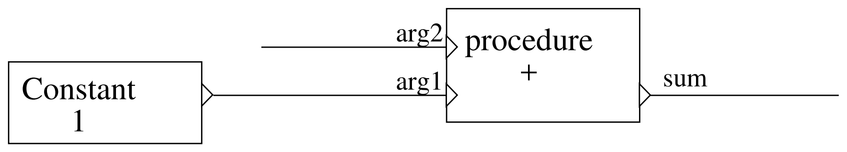


Figure 8 : "incrementor" Behaviour Diagram

If all the input ports are bound to constants, then the function is evaluated and the whole sub-network is promoted to be a constant.

10.3 Encapsulated Sibal

Once a large network has been constructed, the designer may wish to capture the network as a class (**encapsulate** it) for re-use at a later date. Encapsulated Sibal includes:

- A list of Sibal objects
- A table of port links
- Network layout information
- A table of graphical UI constraints
- A table of properties (e.g. does the network contain any GUI objects or application functions etc.)
- A port signature including port renames.

An encapsulated object may have been taken from the UIDE-2 library. It is not permissible for the designer to edit any of these class-tables as they stand (ie. in the libraries). If the designer does attempt to edit them, the network is re-classed automatically. Ie. the designer will then be editing a private copy of the library class.

See chapter 11 for a description of the off-line representation of sibal networks.

10.4 Constant Objects

Fetch links are normally implemented by placing an answer method into the ask port of the destination object. Constant objects can be implemented more efficiently by placing a closure of `identfn` into the ask port instead. When the Sibal object wants the value, the procedure in the ask port is run and the value is placed on the stack.

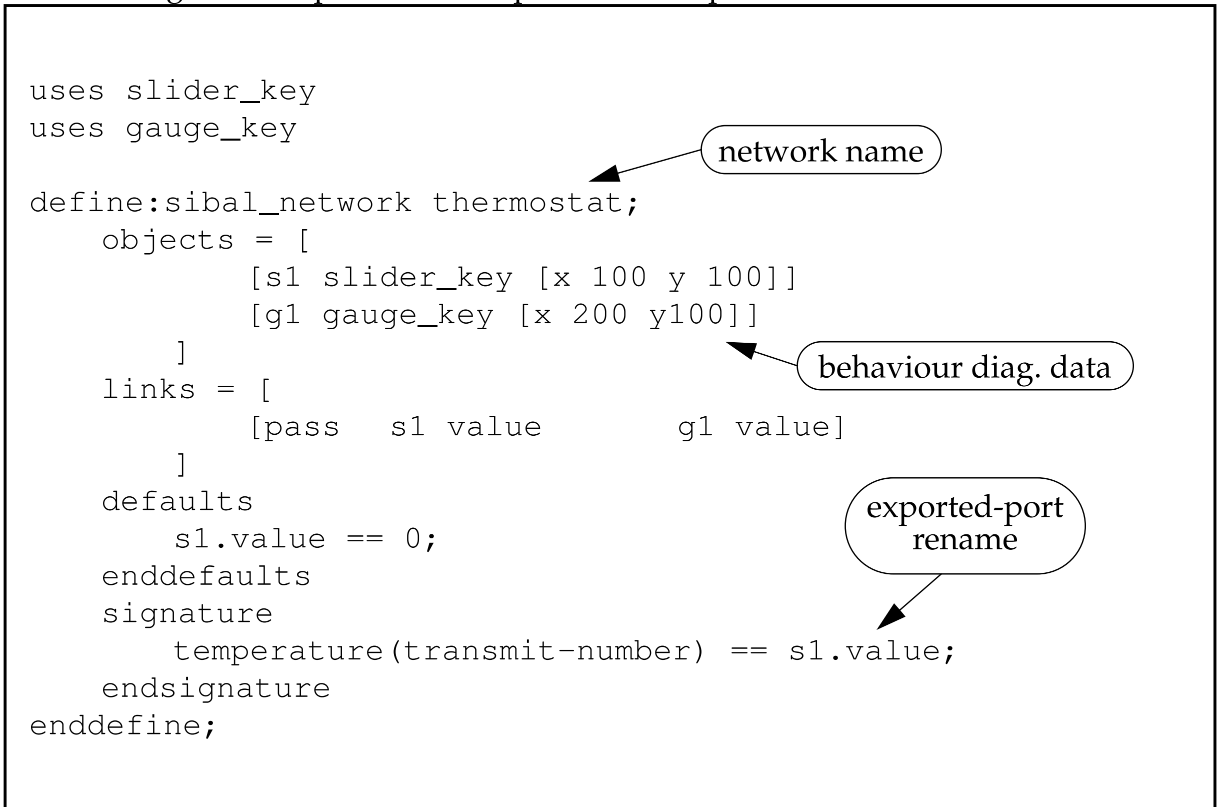
Implementation hint:

```
define:method make_fetch_link(
        sobj:sibal_constant, value:procedure,
        dobj:sibal_object, dprt:procedure
    );
    src_obj.closure -> dest_port(dest_obj);
enddefine;
```

11 ENCAPSULATED SIBAL NETWORKS

Implementation hint:

The following shows a possible example of an encapsulated Sibal network



```
uses slider_key
uses gauge_key

define:sibal_network thermostat;
    objects = [
            [s1 slider_key [x 100 y 100]]
            [g1 gauge_key [x 200 y100]]
        ]
    links = [
            [pass   s1 value       g1 value]
        ]
    defaults
        s1.value == 0;
    enddefaults
    signature
        temperature(transmit-number) == s1.value;
    endsignature
enddefine;
```

Figure 9 : Example Encapsulated Sibal

This network is a possible encapsulation of Figure 5. The UI in Figure 4 would have been generated automatically by the constraint reasoner.

12 NON-OBJECTCLASS OBJECTS

It is possible to treat non-objectclass objects as Sibal objects. The main consideration is to decide how the object can be described by a port signature, and to extend the linking software to reflect this.

For example, it would be possible to treat X toolkit based widgets as Sibal objects if Callback lists were treated as **transmit** ports. This would require an extension, to the software that creates pass links, to coerce the **receive** method, of the destination object, into something that could be used as a callback procedure.

Implementation hint:

```
define:method make_pass_link(
        widget:XptDescriptor, callback:string,
        dest_obj:sibal_object, dest_slot:procedure
    );
    define lconstant drop3
        = erasenum(% 3 %)
    enddefine;

    XtAddCallback(
        widget, callback, drop3 <> dest_slot(% dest_obj %), false
    );
enddefine;
```

Of course this loses all the useful information that X provides a callback procedure. This would be avoided if callback transmit ports were arity 2 (the third argument is the object holding the receive port, and is supplied through the "client data" argument).

Implementation hint:

```
define:method make_pass_link(
        widget:XptDescriptor, callback:string,
        dest_obj:sibal_object, dest_slot:procedure
    );
    XtAddCallback(widget, callback, dest_slot, dest_obj);
enddefine;
```

Part 3 - The Behaviour Editor

13 TERMS

This part of the paper will use some terms taken from the OpenLook document style guide [19][1]. The following terms describe actions performed with the mouse:

- **Press** a mouse button and hold it
- **Release** a mouse button to initiate the action
- **Click** a mouse button by pressing and releasing it before you move the pointer
- **Double-click** a mouse button by clicking twice quickly without moving the pointer
- **Move** the pointer by sliding the mouse with no buttons pressed
- **Drag** the pointer by sliding the mouse with one or more buttons pressed
- **Point** to a control or an object by moving the pointer to the appropriate place on the screen

14 INTRODUCTION

UIDE-2 contains three main tools (from the GUI designer's point of view) which are tightly coupled together:

- the Librarian
- the User View
- the Behaviour Editor

The Librarian is a suite of tools which store and maintain the various resources used by a user of UIDE-2, e.g. bitmaps, programs, partially, or fully, completed GUI designs etc.

The User View shows a sketched view of the look of the final UI. The graphical objects in the User View (e.g. buttons or sliders) may or may not look identical to their appearance in the final delivery system, they also may or may not be as "active" as expected in the final delivery system.

The Behaviour Editor provides a visual programming language to be used by the GUI designer to specify the behaviour of the User Interface under design. Programs written in this language are called Behaviour Diagrams.

This part of the paper concentrates on describing the Behaviour Editor.

1. page 343.

14.1 Intended Audience

Readers of this part of document are assumed to be familiar with some form of graphical, computational notation, e.g. flowcharts, state charts, or Petri nets etc.

A limited grasp of Pop11 [1, 2] may be useful, but is not essential.

14.2 Overview

Behaviour diagrams manipulated by the behaviour editor use a box-line notation [8]:

- objects are boxes
- links are lines connecting ports on the objects
- ports are arrowed line stubs

There are four types of port:

- Transmit
- Receive
- Ask
- Answer

They can be connected together to form a diagram very similar to a data-flow diagram:

- **Transmit** ports are connected to **receive** ports to form **pass** links
- **Ask** ports are connected to **answer** ports to form **fetch** links

The behaviour of a UI is defined by a network of Sibal objects. The behaviour of the network is defined by the ports on the objects and the links between them.

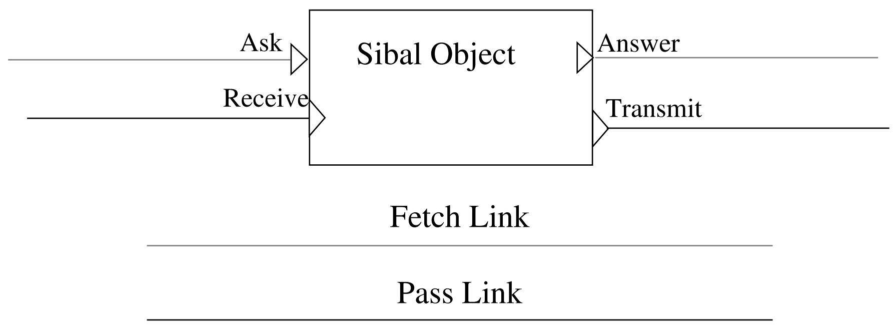


Figure 10 : Behaviour Diagram notation

As indicated in Figure 1, Answer and Transmit ports are “source” ports (relative to a link) and Ask and Receive ports are “destination” ports. For a given link between objects O1 and O2 one port must be a source port and the other a destination port. The objects containing those ports are source and destination objects, for that link. A link can use the same object as source and destination.

Zero or more items of data may flow along a pass link. One or more data items must flow along a fetch link. Chapter 6 gives a more detailed description of the semantics of the links.

The following table summarises the links that are possible. Links that are not possible are marked by a hyphen in the diagram. Some links require that a "buffer" object is automatically placed on the link. This is indicated by the word **buffer**.

Transmit							
Yes	Receive						
-	-	Answer					
buffer	-	Yes	Ask				
-	Yes	-	buffer	Multiple Transmit			
Yes	-	-	-	Yes	Multiple Receive		
-	-	-	-	-	-	Multiple Answer	
buffer	-	Yes	-	buffer	-	-	Multiple Ask

Figure 11 : Meaningful links

Even though the set of rules for making links is quite large, the programmer does not need to learn them. As will be seen later (in section 19.2), the behaviour editor only allows the programmer to make legal connections.

15 MAIN EDITOR

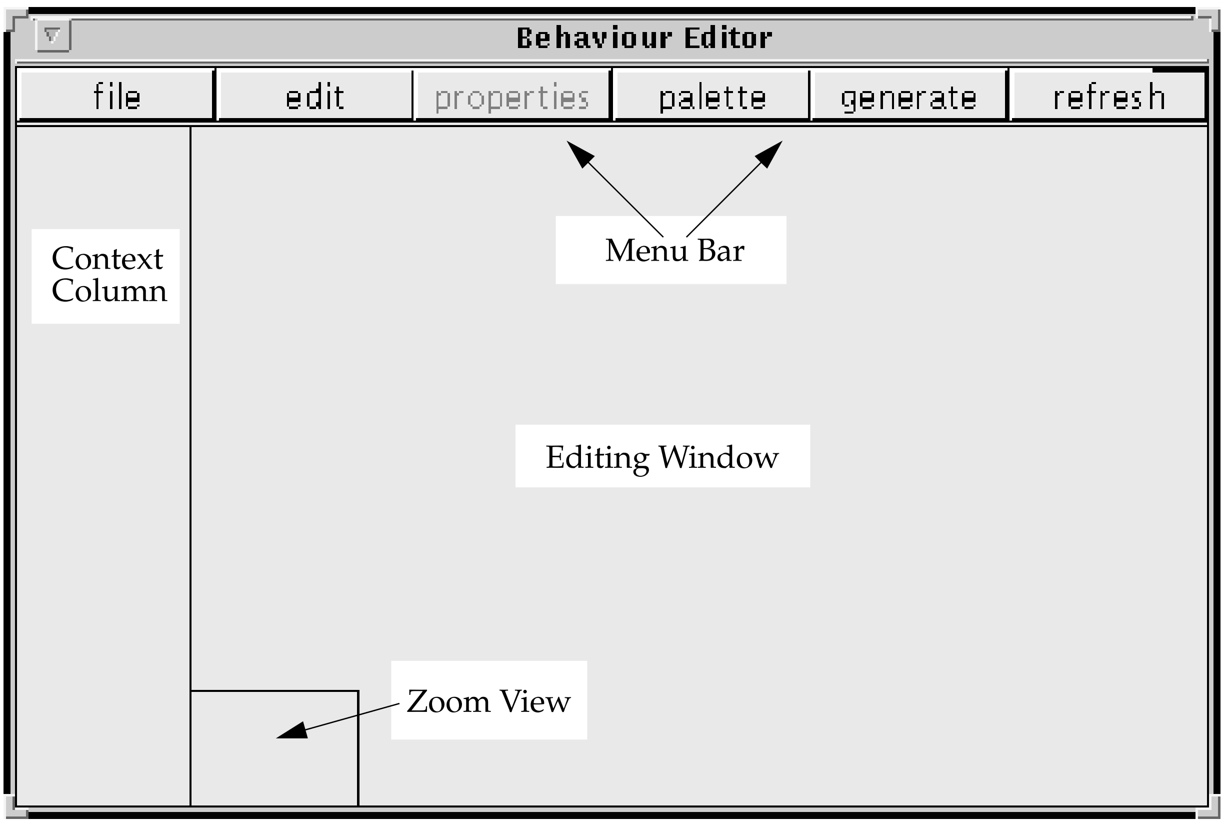


Figure 12 : The Behaviour Diagram Editor

The editor has three main areas:

- The main editing window (see section 15.1)
- A context column (see section 15.2)
- A menu bar (section 15.3)

The editing window is a window on to a (perhaps) much larger editing pane. The zoom view shows a reduced view of the entire pane and is used to scroll the editing window. It is this reduced view that is shown in the context column.

15.1 Editing Window (overview)

Figure 13 shows the editor being used to edit a network with no parents

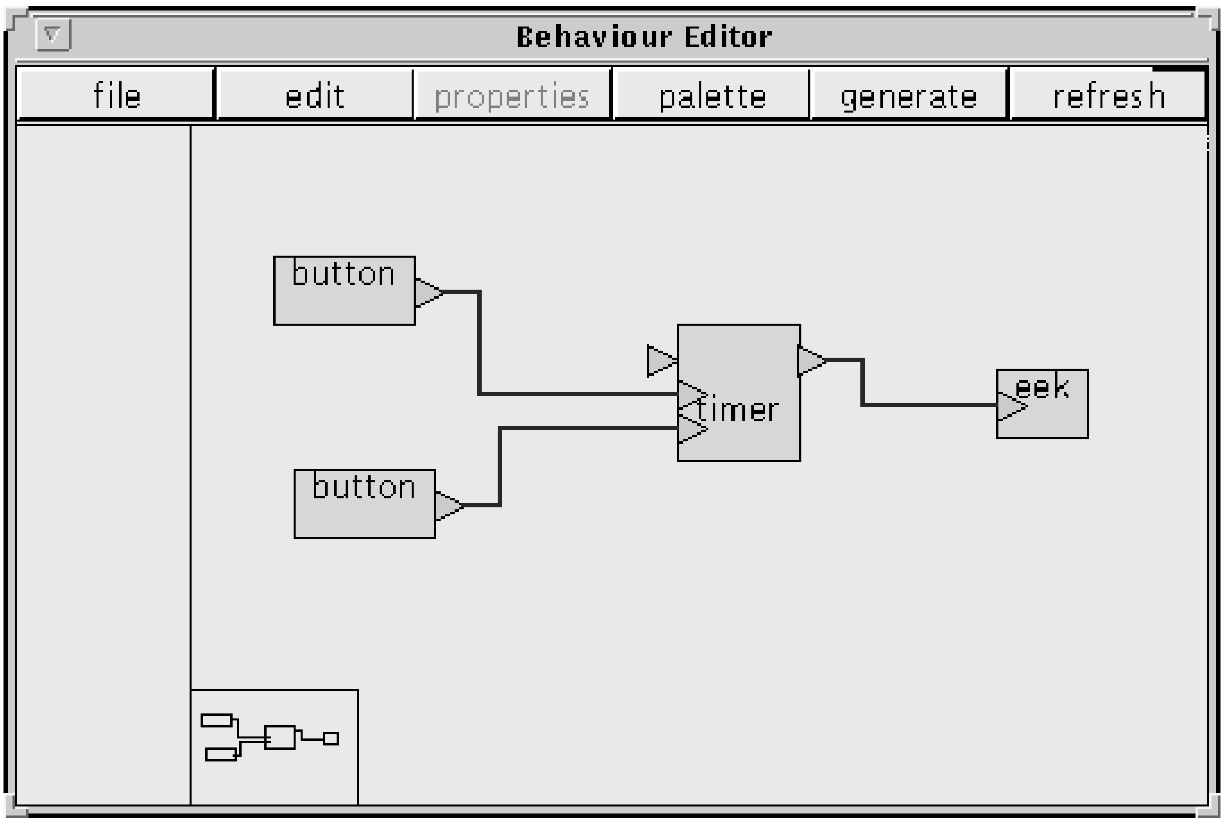


Figure 13 : Editing a network with no parents

See figure 16 (page 40) for the user-view of this network. Chapter 19 gives details of how a designer manipulates a network.

15.2 Context Column

Figure 14 shows the editor being used on a sub-network that is at the third layer in the hierarchy

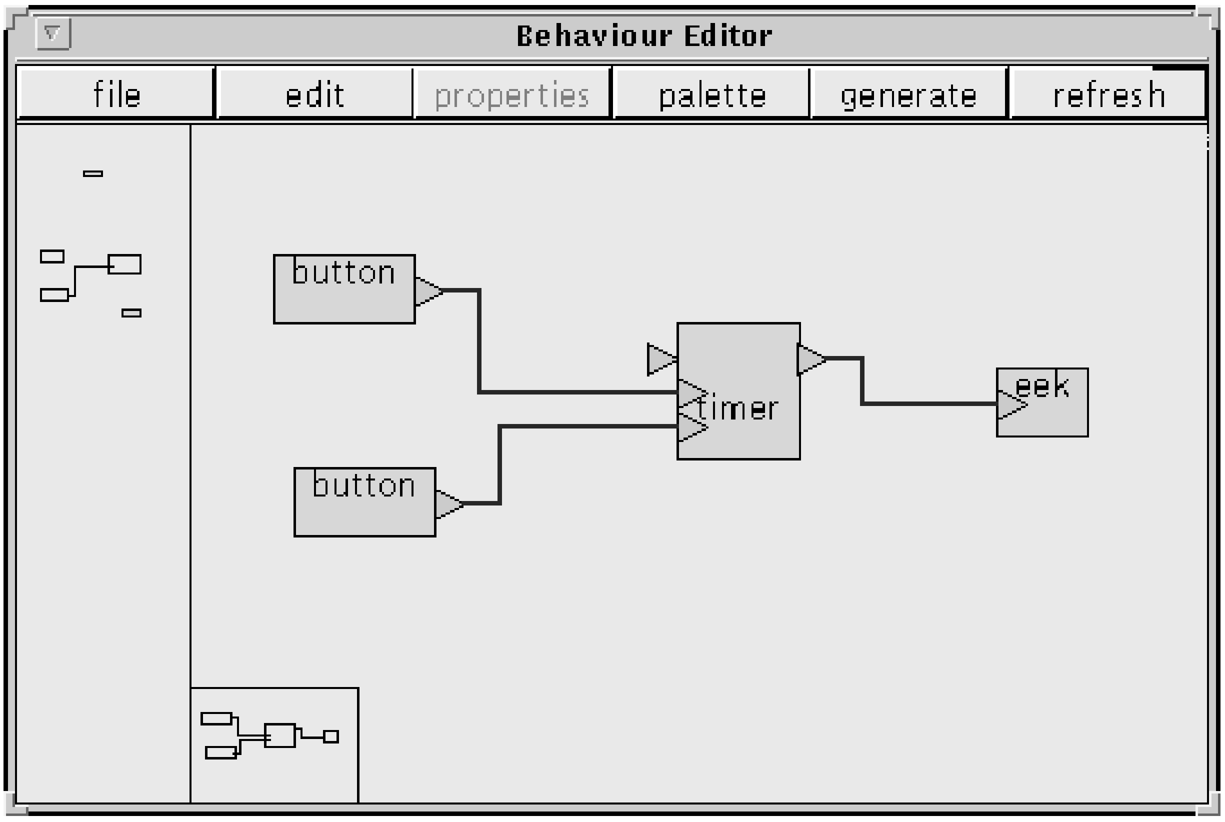


Figure 14 : Editing a third layer sub-network

The left hand side of the Behaviour Editor consists of a Context Column. This is for showing the context of network currently in the editing window. Each network in the context column contains a highlighted node. The highlighted node is shown in its expanded form either directly below in the context column or in the editor window (the highlight doesn't appear well in this screendump).

The implementation of the context column is not complete in this screendump.

15.3 Menu Bar

The following list indicates the structure of the menu bar (the items in italics have yet to be implemented):

- file
  - *new*
  - *open...*
  - *save*
  - *save as...*
  - *print...*
  - exit
- edit
  - expand
  - undo
  - encapsulate
  - *cut*
  - *copy*
  - *paste*
  - *properties*
  - delete
  - restart
- *view*
  - *link types*
  - *port titles*
  - *data types*
  - *arity*
- palette
  - glasi
  - GUI
- generate
  - User View
  - *UI...*
- refresh

The menu items are **always** present in the menu bar, or sub-menus, but there are some occasions when a particular option is not appropriate. Under these circumstances the menu option will be greyed out.

Some of the menu options are explained below.

15.3.1 properties:diagram properties

These are properties used by the hint manager, and by the generate:UI option.

E.g one property could be "predominant target" which would be a choice of: openlook, motif, mac, windows, sketch etc.

Hints are probably only useful for nodes with more than one implementation.

15.3.2 edit:expand

This option is only available if a node has been selected with the mouse. Puts the current network on the history column. Opens, for editing, the network that implements the currently selected node (see figure 14). See also section 19.1.2.

16 LIBRARIAN PALETTE

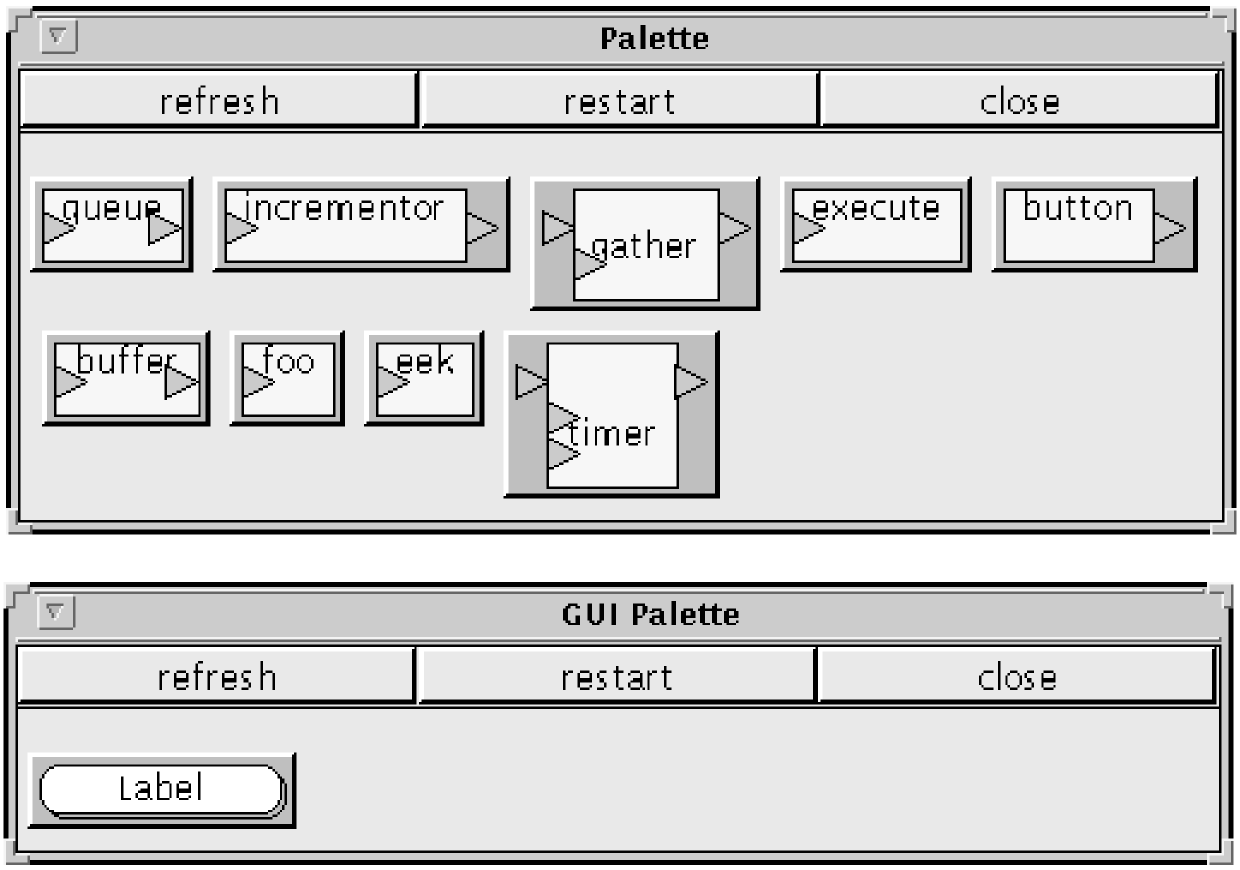


Figure 15 : The GUI and standard palettes

The Palette is the programmer's interface to the library. Behaviour classes are collected onto separate pages depending on type. Each class is represented by an icon similar to the icon used in the behaviour diagram. There is an integrated browser to explore the properties of each class.

When a class is selected in the palette, the mouse pointer takes on a different shape in the behaviour editor window. The shape of the pointer is dependent upon the icon used for the class. When the mouse is clicked in the editor window an instance of the selected class is placed at that position and the pointer returns to normal (and the item in the palette is deselected).

Figure 15 also shows the GUI palette. This palette contains a visualisation of the behaviour classes that represent GUI objects. If a GUI object is selected from either the GUI palette or the normal palette, then the instance can be placed in either the userview or the behaviour diagram. Whichever the designer chooses to do, the system will complete the other option. Ie. if the designer selects the button class (from either of the palettes), the cursor in the userview will change to a button and the cursor in the behaviour editor will change to the appropriate behaviour node. Placing the button in the userview will also place a "button" behaviour node in the behaviour diagram.

16.1 On-the-fly palettes

As the number of classes in the library is expected to be large, the palette should be split into groups on separate palettes. Programmers could do this when they add a new class to the library. Another possibility is to generate restricted palettes "on the fly" by using a searching mechanism.

The designer constructs a new palette by selecting a node that is currently in the behaviour diagram. A search tool is then selected from the "edit" menu. The search window contains a copy of the selected node. The designer then selects the port of current interest. In response to the "ok" button, the search tool then scans the library for behaviour classes that contain a port that could make a legal connection. A new palette is made of all these classes and presented to the designer.

The UIDE project is also investigating allowing designers to add stylised and machine readable comments to new behaviour classes. This would make it easier for later designers to search for an appropriate class.

Obviously, the searching mechanisms provided by the librarian play a key role in the designer's ability to re-use classes. No claims are made by this paper for the effectiveness of the librarian used in UIDE-2. Code re-use is a difficult, and ongoing, area of research [6].

17 DRAG AND DROP

The Behaviour Editor should support having objects dropped onto it. Ie a Sibal file could be located by the OL file manager and dropped onto a Behaviour Editor, or an XVed window etc. This would result in a node being added to the current diagram (in the former case) or a new Behaviour Editor being launched for the network file (in the latter case).

This implies that Sibal files must have a file extension, or magic number, that can be recognised by Poplog.

18 USER VIEW

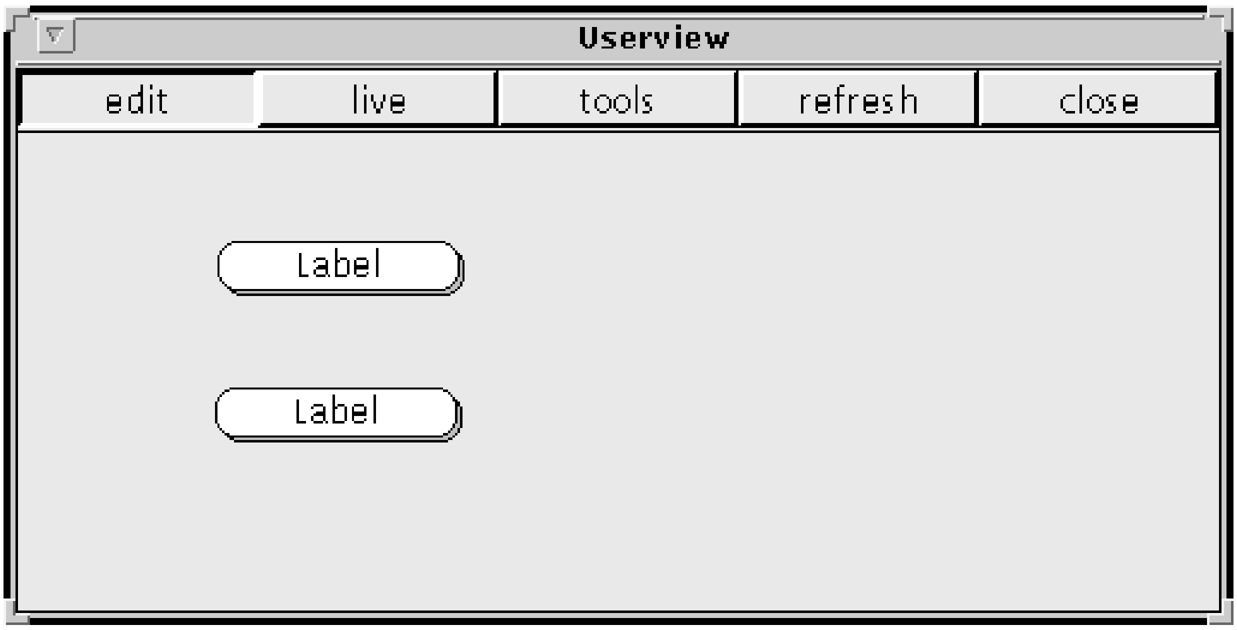


Figure 16 : The Userview. A UI generated by figure 13.

If the User View window is open, then any changes made to the GUI aspects of the behaviour diagram are immediately reflected in the User View.

The designer may directly edit the User View. If this is done, the behaviour diagram is updated in the appropriate way. E.g.:

- Graphical constraints will be added, updated, or deleted
- Sibal objects will be added to, or deleted from, the diagram.

19 EDITING A NETWORK (DETAIL)

Many functions are available from the *Edit* menu (section 15.3) in the menu bar, but the most frequently needed facilities are available by manipulating the diagram directly.

19.1 Nodes

New nodes can be selected from the Librarian palette and added to the editor window. If they are placed on another part of the behaviour editor they will be ignored.

19.1.1 Edit Menu

Clicking on a node selects it; options from the Edit menu are then available (deleting, copying to the clipboard etc.). See section 15.3 above.

19.1.2 Expanding a node

Click middle button on node: The current behaviour diagram is moved to the context column (see Figure 14). The sub-network that the clicked on node represents is then "opened" in the editing window.

This is also available from the *Edit:expand* menu option (section 15.3.2) if a node has been selected.

19.1.3 Editing node details

Click right button on node: a "properties sheet" or other appropriate dialogue box will be opened. The details of the dialogue box are dependant on the type of the node. Currently only nodes representing UI objects are supported in this way. The dialogue box gives a menu for setting various properties of the UI object; e.g. colour, label text etc.

19.1.4 Moving a node

Pressing on, and dragging, a node allows the designer to drag the node around the editor window. The links will automatically follow the node.

19.1.5 Making a constant node

Double click on ask arrow: a "constant" box, of the appropriate type, will be automatically created. The type of constant will depend on the data type of the ask port. E.g. if the constant box is being connected to a port that requires an integer, then the box will contain a space for typing the number plus increment/decrement buttons.

Constant boxes will be available from the palette, but the designer should usually find it easier to "open up" the ask port by double-clicking on it.

19.2 Links

19.2.1 Making a link

Click left button on port arrow: all legal links from the selected port are shown with a "tentative" thin black line. The programmer then clicks on the required destination port to specify a connection. The link then turns to a thick blue line. This is similar to the behaviour exhibited by AVS [3 pp. 67 - 71]. The programmer can actually click anywhere in the diagram to complete the link; the system will select the legal port which is physically closest to the mouse click. If none of the tentative links are required, the programmer clicks anywhere in the diagram and then selects the *Edit:Undo* menu option Some links will require buffers. In this case the buffers will be created automatically.

This "tentative link" behaviour has two advantages:

- Only legal connections can be made. This means that a **syntactically** incorrect network can **never** be constructed.
- The programmer does not need to know whether the link they are constructing is a pass or a fetch link.

The current implementation of the behaviour editor uses "Manhattan" links[1]. Possibilities that may appear in later versions include: allowing the designer to draw curved, "free-hand" links.

19.3 Diagram Properties

Double clicking on the edit window background accesses the Diagram Properties dialogue box. This option is also available on the *Properties:diagram properties* menu option.

19.4 Grouping a Subnet

Press and drag on the background: will create a "rubber-band" bounding box that is used to select a set of nodes. A number of options are then available from the *Edit* menu:

- Delete the group
- "Fold" the group into a single node.
- Create a new class (ie. available from the palette) out of the group. This will also fold the group.

Chapter 20 describes how the last two options support bottom-up design.

20 NORMAL USAGE OF THE BEHAVIOUR EDITOR

Two typical design styles are supported by UIDE-2:

- Bottom-up
- Top-down

The bottom up approach corresponds closely to the GUI **construction** approach supported by many available UIDEs (e.g. Fabrik [9], Garnet [14], X-Designer, TeleUse, SUIT[15] Etc.)

Support for the top down approach is fairly rare in UIDEs. It aids the designer by enabling step-wise refinement of a design.

20.1 Bottom Up

The bottom-up approach is likely to be the most familiar to a designer. It uses the User View and the "GUI primitives" part of the Sibal Objects palette.

Within this style of GUI construction, the designer will typically drag low level GUI objects from the palette directly onto the User View. It will then be possible to add graphical constraints to those objects (e.g. alignment, nearness etc.). Behaviour (ie. callbacks) can be added to the objects using a properties sheet.

1. Also known as "orthogonal" links, [11] page 173.

Any changes, or additions, to the User View are automatically reflected in the Behaviour Diagram. This means that object behaviour, and graphical constraints, can be edited by direct manipulation within the behaviour editor.

If the designer decides that a portion of the design would be useful as a new primitive object, the behaviour editor can be used to group that section of the diagram. This group can then be encapsulated into a new class (by selecting a menu option).

20.2 Top Down

The top-down approach is useful for exploratory design. With this method, the designer creates a first draft of the GUI, that captures the overall behaviour of the whole interface, using very abstract components. This design is then subsequently refined using more concrete GUI primitives. It uses the Behaviour Editor and the "Functional Primitives" part of the Palette.

The designer will typically drag some very abstract, functional components onto the behaviour editor. Examples of these components are: "dialogue box", UI screen etc.

Even the most abstract, overview design of a UI can be tested in the user view as even the most abstract component has some form of default, sketch representation (e.g. a box with a text label, an "Accept" button, or both)

Once the overall behaviour of the UI is satisfactory, the designer can construct more detailed implementations of the abstract nodes in the behaviour diagram. E.g. a very abstract specification of an "obtain numeric value" dialogue (a text field) could be replaced later by a slider. Further refinements or extensions could replace this with a more complicated behaviour diagram containing many buttons and menus (e.g. for specifying the units of the numeric value).

Part 4 - Future Research and Development

21 TOWARDS A STRONGER EXECUTION MODEL

This design document, so far, has been quite lax in addressing issues like execution semantics and type coherence etc. This has been intentional. Addressing these issues in a rigorous manner is beyond the scope of the UIDE project. It could form a major part of another project.

The next few sections give a rough discussion of the issues involved.

21.1 Consequences and Dependencies

The version of Sibal that has been described by this document, only allows for very simple forms of consequence and dependency information. This information is implied by pass and fetch links.

Pass links denote consequence information. The consequence of an object transmitting an event along a pass link, is that the receive port at the other end of the link is obliged to do something (anything) with the event.

Fetch links imply a very weak dependency relationship. An ask port is dependant on the answer port, that it is connected to, having a viable piece of data. But there is no way for the answer port to communicate its readiness to the ask port. This version of Sibal has avoided problems by stating that an answer port is allowed to return **any** default data item, if it is not ready, as long as it is of the correct type.

As can be seen, the information currently supplied is very weak. Consequence information is useful in detecting areas of “dead” code, and infinite recursions etc. Dependency information is useful in proving program correctness etc.

What is needed is an extension to the port signature. The extra information would describe how the ports within an object relate to each other. For example, the answer port of a buffer is not valid until at least one event has been received by the receive port. This information should be supplied by a dependency relationship.

21.2 Type propagation

Another form of dependency information is to do with type propagation.

Consider the buffer shown below in Figure 17

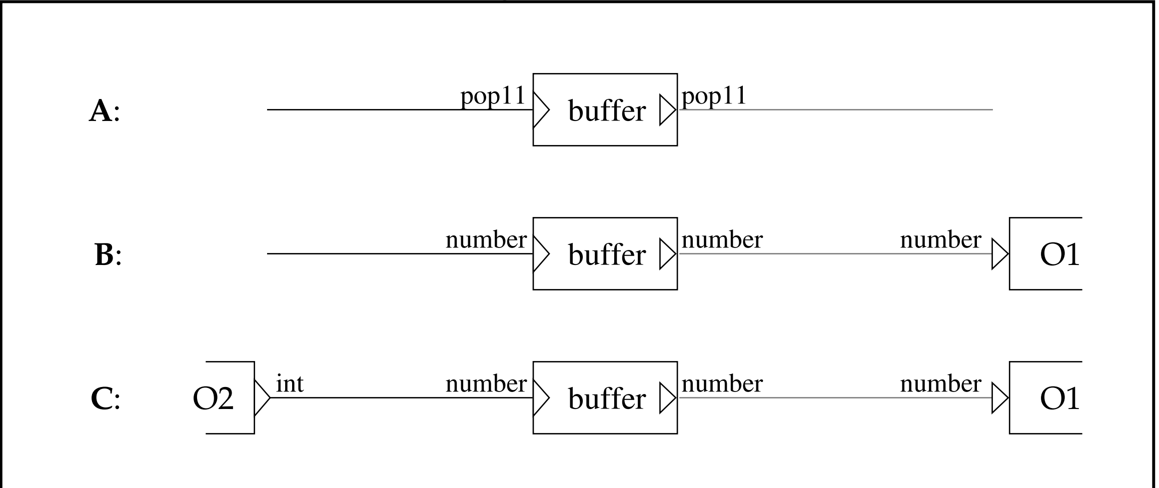


Figure 17 : Type propagation

Only the data-type part of the port labels is shown.

Part A shows the buffer on its own. The two ports have the data type **pop11** which corresponds to the unifying Pop11 type (ie. all Pop11 types are descendants of the type **pop11**).

Part B shows the output, answer, port connected to another object O1. The data type of the ask port on O1 is **number**. This means that the type of the answer port of the buffer should also be **number**. If the correct "type propagation" information was associated with the buffer, the data type could be propagated to the receive port (causing it also to be of type **number**).

This would then, correctly, restrict the type of objects that could be connected to the input side of the buffer. Ie. the transmit port would have to be of type **number** or one of its descendants.

The current definition of Sibal doesn't encompass this at all.

22 DISTRIBUTED APPLICATIONS

22.1 Distributed Interfaces

Some windowing technologies (e.g. the X Window System [17]) allow applications to utilise multiple displays in one application. This is done through a technique known as "network transparency". All drawing requests, made by UI widgets, are done through a simple, well defined protocol. This protocol is amenable to being transmitted over an Inter Process Communication (IPC) socket. Each widget knows which channel it should use to send requests and receive events. This is all transparent to the application, which maintains a single set of widgets in its process memory.

Sibal will support this type of multiple display interface. Some Sibal objects “know” that they are GUI objects. When constructing a behaviour network using these objects, the designer could “annotate” them such that they will appear on separate screens when the interface is built. There will only be a single process that controls them though.

22.2 Distributed Processes

Poplog supports the IPC and RPC (Remote Procedure Call) protocols [4]. This allows a process to start a number of other processes on other machines (or on the same machine) and to exchange data with them.

Special Sibal objects could be constructed to directly support this kind of spawning and communication behaviour.

23 BEHAVIOUR EDITOR EXTENSIONS

23.1 Editing Graphical Constraints

Certain user interfaces require that graphical constraints are placed on the UI objects (e.g. “align bottom edge”). Normally, the built in style guides should be sufficient to correctly lay out the UI. But occasionally the designer may wish to override the defaults. Figure 18 shows how a graphical constraint may be specified in a behaviour diagram.

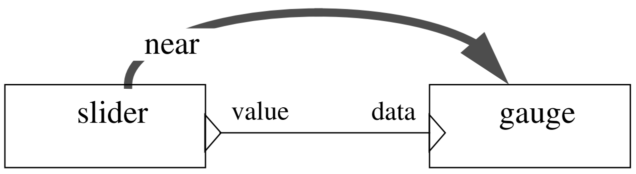


Figure 18 : Adding a “near” constraint

The Rockit system [10] uses an interesting technique of inferring graphical constraints from the way the designer manipulates UI objects in the “User View” (UIDE terminology). The Rockit system only uses six constraints: Connector, Spacer, Attractor, Repulser, Container, Aligner. Experience will show if any others are needed.

24 NODES AS 1st CLASS DATA

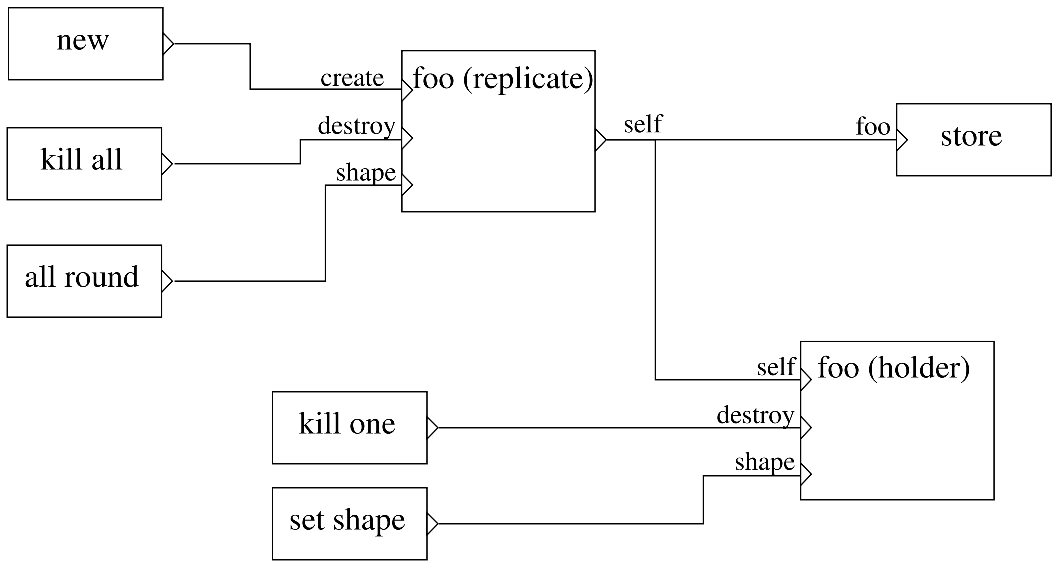


Figure 19 : A network with a Replicator and a Placeholder

Behaviour Diagrams are unusual in that it is possible to promote a node, normally assumed to be a function, to be a first class data item. The node can be either a primitive or an arbitrarily complex network.

To facilitate this there are two, complimentary, possible promotions:

- promotion to a “Self Replicating” node
- promotion to a “Place Holder” node

24.1 Self Replicating Nodes

Any node, or network, can be converted into a self replicating node. This enables the extent, and behaviour, of the network to change dynamically at runtime.

The effect of promoting a node like this is to add three ports to its signature:

create(receive/0), self(transmit/1-sibal_net), destroy(receive/0)

When a trigger is received on the create port three things happen:

- a new instance of the net is created.
- Any other connections to the node which were specified in the behaviour diagram are duplicated (apart from the create and self links).
- a **sibal_net** object is generated on the **self** port

If the network which implemented the node contained any GUI objects then these are, naturally, duplicated as well. In this way the number of GUI objects appearing on the screen can be altered at run time. The behaviour networks, associated with these new screen objects, are also duplicated. This means that the new objects can behave as autonomous agents, responding to the mouse or the application as required independently of their "siblings".

Because all the links to the node are duplicated as well, any answer ports on the node must be unbound. Figure 11 (page 33) shows that it's not possible to connect more than one answer port to an ask port, which would be the effect of duplicating a node containing a bound answer port.

Links to receive ports effectively become "broadcast" links. Because the link is duplicated, all events transmitted along the link are sent to all instances of the node.

24.2 Placeholder nodes

These are the natural compliment of self replicating nodes.

The effect of promoting a node likes this is to add two ports to its signature:

self(receive/1-sibal_net), destroy(receive/0)

The node then becomes a place holder for a net-instance (as created by a self replicating node). The links into the node only become valid when a net-instance of the appropriate class has been received by the **self** port. An error is generated if any activity is attempted on the links before a net-instance is received by the node.

Only one instance of a node can be resident in a placeholder at any one time. Therefore:

- pass-in links (receive ports) are not "broadcast" links
- answer ports may be bound

Nodes can also be stored in lists, and other such data structures, while they are not "in use".

24.3 Example Replicating Node

In figure 19, the application procedures `new`, `kill all`, and `all round` are connected to a self replicating node (of the class `foo`). The class `foo` normally has only one port:

foo - shape(answer/1-shape)

The port is an answer port of arity 1, of type **shape**. The value expected by the shape port is one of "circle", "triangle", "square" etc. This class is, of course, entirely imaginary, but plausible.

The application procedures `kill all` and `all round` are connected to the instance of `foo` by **broadcast** links (by virtue of being connected to receive ports on a replicating node).

Thus `kill all` will destroy **all** instances of `foo` that were created by this node, and `all round` will, similarly, set the shape of all instance of `foo` that were created by this node.

The `self` port is connected to two objects: a node called `store` (which may well simply keep the instances safe somewhere) and a **placeholder** version of the class `foo`.

The application functions `kill one` and `set shape` only ever affect the instance of `foo` that is currently residing in the placeholder.

25 REFERENCES

1. Anderson, James, **"Pop-11 Comes of Age"**, 1989, Ellis Horwood
2. Barrett, Ramsay and Sloman **"POP-11: a Practical Language for Artificial Intelligence"**, 1985, Ellis Horwood
3. Earnshaw, R. A. and Wiseman, N. **"An Introductory Guide to Scientific Visualization"**, 1992, Springer-Verlag
4. Gaizauskas, R. J. **"Building a Distributed Poplog Application Using RPC"**, 1992, in Rogers I. and Goodlet J. Proceedings of Plug92, CSRP 271, School of Cognitive Sciences, Sussex University, UK
5. Green, T. R. G. **"The cognitive dimension of viscosity: a sticky problem for HCI"** In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.) Human-Computer Interaction - INTERACT'90. Elsevier
6. Green, T. R. G., Gilmore, D. J., Blumenthal, B. B., Davies, S. and Winder, R. **"Towards a cognitive browser for OOPS."**, 1992, Int. J. Human-Computer Interaction, 4(1), 1-34
7. Harel, David et al **"STATEMATE: A Working Environment for the Development of Complex Reactive Systems"**, 1990, IEEE Trans. Soft. Eng., Vol. 16, pp. 403-414
8. Hils, Daniel D. **"Visual Languages and Computing Survey: Data Flow Visual Programming"**, 1992. Journal of Visual Languages and Computing - vol. 3 no. 1 pp. 69-101
9. Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, Ken Doyle **"Fabrik: A Visual Programming Environment"**, 1988, OOPSLA'88 Proceedings
10. Solange Karsenty, James A. Landay, Chris Weikart **"Inferring Graphical Constraints with Rockit"**, in A. Monk, D. Diaper, M. D. Harrison (Eds.) People and Computers VII, Proceedings of HCI'92, York, September 1992
11. David Kirk (Ed.) **"Graphics Gems III"**, 1992, Academic Press
12. Koenemann, J. and Robertson, S. P. **"Expert Problem Solving Strategies for Program Comprehension"**, in Proc. CHI'91, ACM Press, pp. 125-130
13. Mott, D.H., Cunningham, J., Kelleher, G. and Gadsden, J.A. **"Constraint-based reasoning for generating naval flying programmes"**, in Expert System August 1988, Vol. 5, No. 3, pp. 226 - 246
14. Brad Myers et al, **"GARNET: Comprehensive support for graphical, highly interactive user interfaces"**, November 1990, IEEE Computer, pp. 71 - 85
15. Randy Pausch, Nathaniel R. Young II, and Robert DeLine **"SUIT: The Pascal of User Interface Toolkits"** November 1991, Proc. User Interface Software and Technology, ACM Press
16. Klaus Erik Schauser **"Compiling dataflow into threads: efficient compiler-controlled multithreading for lenient parallel languages"**, July 1991. Report no.

UCB/CSD/91/644, Computer Science Division, University of California, Berkeley, CA 94720

17. R.W. Scheifler, J. Gettys **“The X Window System”**, October 1986. Report no. MIT/LCS/TR-368, AI Lab., MIT, Cambridge, Mass.
18. Sun Microsystems Inc. **“DevGUIDE, OpenWindows Developer’s Guide”** 2550 Garcia Ave., Mtn. View, CA 94043
19. Sun Microsystems Inc. **“OPEN LOOK Graphical User Interface Application Style Guidelines”**, Addison-Wesley
20. Craig Upson **“Visual programming in data flow environments”**, 1992, The distinguished lecture series IV (Video), University Video Communications, Stanford CA 94309
21. Ake Wikstrom **“Functional Programming Using Standard ML”**, 1987, Prentice Hall
22. Wilson, S., Markopoulos, P., Pycock, J., and Johnson, P. **“Modelling Perspectives in User Interface Design”**, 1992, EWHCI’92 International Conference on Human-Computer Interaction, pp. 210 - 217

26 SELECTED BIBLIOGRAPHY

The references in this bibliography, amongst many others, have a bearing on this project, but have not been explicitly mentioned in this paper.

i. P. T. Cox, F. R. Giles and T. Pietrzykowski, **"Prograph: a step towards liberating programming from textual conditioning"** 1989, IEEE Workshop on Visual Languages, pp. 150 - 156

ii. Mark Green, **"A Survey of Three Dialogue Models"**, July 1986, ACM Transactions on Graphics, Vol. 5, No. 3, pp. 244 - 275

iii. T. R. G. Green, M. Petre and R. K. E. Bellamy, **"Comprehensibility of visual and textual programs: a test of superlativism against the match-mismatch conjecture"**, Aug 1991, JCI summer school

iv. M. Helander (ed.), **"Handbook of Human-Computer Interaction"**, 1988, Elsevier Science

v. Mainstay, **"The V.I.P. Treatment"**, June 1987, MacUser, pp. 132, 134, 136, 138 and 184

vi. Dan R. Olsen, Jr. **"User Interface Management Systems: Models and Algorithms"**, 1992, Morgan Kaufmann

vii. M. Petre and T. R. G. Green, **"Is graphical notation really superior to text, or just different? Some claims by logic designers about graphics in notation"**, July 1990, CITE Report 113, Open University

viii. Ben Shneiderman, **"Direct Manipulation: A Step Beyond Programming Languages"**, 1983, IEEE Computer, Vol. 16, No. 8, pp. 57 - 69

ix. Bruce Tognazzini, **"TOG on Interface"**, 1992, Addison-Wesley