ECL^{i}PS^e

Visualisation Manual

Release 7.1

Kish Shen (IC-Parc)
Josh Singer (Parc Technologies Ltd.)
Andrew Sadler (IC-Parc)

September 3, 2022

Trademarks

Java is a trademarks of Sun Microsystems, Inc.

Contents

Chapter 1

Introduction

This manual contains information on the modelling level search and propagation visualisation tools available in ECL^{i}PS^e.
In order to investigate the behaviour of your constraint logic programs at a level of abstraction above that provided by the source level debugger, ECL^{i}PS^e provides the following visualisation tools.

Chapter 2

Program annotation

When visualising CLP program behaviour, not all the variables of the program are of interest. ECL^{i}PS^e supports the concept of a set of **viewable** variables whose state over the course of a program run are of interest to the user. The library **lib(viewable)** contains the annotation predicates that allow a programmer to define (and expand) these **viewable** sets.

2.1 Viewables

By collecting together related program variables into a logical, multidimensional array-like structure called a **viewable**, the user can view the changing state of these variables in a number of ways using the provided visualisation clients (these will be covered in depth later (section 3)). As an example of how to annotate an ECL^{i}PS^e program, consider the following classic cryptographic example, `SEND+MORE=MONEY`

```
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    Carries = [C1,C2,C3,C4],
    Carries :: [0..1],
    alldifferent(Digits),
    S #\= 0,
    M #\= 0,
    C1         #= M,
    C2 + S + M #= O + 10*C1,
    C3 + E + O #= N + 10*C2,
    C4 + N + R #= E + 10*C3,
         D + E #= Y + 10*C4,
    labeling(Carries),
    labeling(Digits).
```

It is hopefully clear from the code that this formulation of the classic puzzle uses four variables `[C1,C2,C3,C4]` to indicate the *carry* digits. If we suppose that the user is only interested in the behaviour of the program with respect to the primary problem variables, which in this case corresponds to the variables [S,E,N,D,M,O,R,Y], then we can annotate the program code by declaring a **viewable** which contains these variables.

```
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    viewable_create(digits, Digits),
    ...
    ...
    labeling(Carries),
    labeling(Digits).
```

As can be seen, **viewable**s are declared using the **viewable_create/2** predicate, the first parameter of which is an atom which will be used to uniquely identify the **viewable** later, and the second argument is a (possibly nested) list of variables.
Declaring **viewable**s has little performance overhead when running code normally (that is to say, without any visualisation clients), and so it is safe to leave the visualisation annotations in the code even when not visualising.

2.1.1 2D and beyond

In the previous example, the created **viewable** was a simple one dimensional structure, it is possible however to create multi-dimensional structures if the problem variables are so related. For example one could choose to group the variables in a way that mirrors the problem structure, for example a 2D array representing the equation

$$\begin{array}{ccccc} & \mathrm{S} & \mathrm{E} & \mathrm{N} & \mathrm{D} \\ + & \mathrm{M} & \mathrm{O} & \mathrm{R} & \mathrm{E} \\ \hline \mathrm{M} & \mathrm{O} & \mathrm{N} & \mathrm{E} & \mathrm{Y} \end{array}$$

would be the array

$$\begin{pmatrix} 0 & S & E & N & D \\ 0 & M & O & R & E \\ M & O & N & E & Y \end{pmatrix}$$

and would be declared in the program as nested lists

```
viewable_create(equation,[[0, S, E, N, D],[0, M, O, R, E],[M, O, N, E, Y]]
```

or it could be declared in the program using ECL^{i}PS^e array syntax

```
viewable_create(equation,[]([](0, S, E, N, D),
                            [](0, M, O, R, E),
                            [](M, O, N, E, Y)))
```

Three points should be noted here,

1. **viewable_create/2** accepts both nested lists and arrays.
2. Variables may occur more than once in **viewable**.
3. Constants may occur in **viewable**s.

2.1.2 Growth

So far we have introduced only static (or *fixed* dimension) **viewable**s, but it is conceivable that during the course of program runs new variables may be introduced which the user has an interest in looking at. In order to accommodate this, **viewable**s may be declared as having *flexible* dimensions.

To declare a **viewable** with flexible dimensions, the three argument form of the **viewable_create/3** predicate is used. The third argument specifies the type of the **viewable** and at present the type must be of the form `array(FixityList, ElementType)` where

`FixityList` is a list with an atom `fixed` or `flexible` specifying the fixity for each dimension. The fixity denotes whether the dimension's size is fixed or may vary during the time when the viewable is existent.

`ElementType` is a term which specifies the type of the constituent viewable elements. Currently there are two supported element types:

- `any` which includes any ECLiPSe term.
- `numeric_bounds` which includes any ground number, integer **fd** variables, **ic** variables and **range** variables (including **eplex** and **ria** variables).

Let us expand our example by assuming that, during the program run our user is only interested in the *digit* variables but once labelling has finished they wish to also see the *carry* variables. Clearly the user is free to simply print out the *carry* variables after completing the labelling, but within the visualisation framework they may also expand the viewable by adding the *carry* digits to it. The code to do this is

```
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    viewable_create(equation,
                    []([](0, S, E, N, D),
                       [](0, M, O, R, E),
                       [](M, O, N, E, Y)),
                    array([flexible,fixed], any)),
    ...
    ...
    labeling(Carries),
    labeling(Digits),
    viewable_expand(equation, 1, [C1, C2, C3, C4, 0]).
```

Once declared as flexible, dimensions may be expanded by the **viewable_expand/3** predicate. The predicate specifies which dimension to expand and which values should be added. Had the **viewable** been 3 dimensional, then the values to be added would need to be 2 dimensional. In general for an N dimensional **viewable**, when expanding a flexible dimension, the values to be added must be N-1 dimensional arrays or nested lists.

As with **viewable_create/2** and **viewable_create/3**, **viewable_expand/3** silently succeeds with little overhead at runtime, so it too may be left in code even when not visualising.

2.1.3 Types

As mentioned briefly in the previous section, **viewable**s have a type definition which determines what sort of values may be stored in them. This type information allows visualisation clients to render the values in a fitting manner.
Explicitly stating that the variables in a viewable are `numeric_bounds` where known will increase the number of ways in which the **viewable** elements may be viewed.

2.1.4 Named dimensions

Each position in a **viewable**'s dimension has an associated name. By default, these names are simply the increasing natural numbers starting from "1". So, for example, in the previous code samples the variable `R` would be at location `["2","4"]`.
By using the most expressive form, the **viewable_create/4** predicate allows the user to assign their own symbolic names to dimension locations.
In our ongoing example, we could name the first dimension positions `["send", "more", "money"]` and the second dimension positions `["ten thousands", "thousands", "hundreds", "tens", "units"]`.
A version of **viewable_expand/4** exists also which allows the user to assign a name to the new position of an expanded dimension.
Our completed example now looks like this

```
:-lib(viewable).

sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    viewable_create(equation,
                    []([](0, S, E, N, D),
                       [](0, M, O, R, E),
                       [](M, O, N, E, Y)),
                    array([flexible,fixed], numeric_bounds),
                    [["send", "more", "money"],
                     ["ten thousands", "thousands",
                      "hundreds", "tens", "units"]]),
    Carries = [C1,C2,C3,C4],
    Carries :: [0..1],
    alldifferent(Digits),
    S #\= 0,
    M #\= 0,
    C1         #= M,
    C2 + S + M #= O + 10*C1,
    C3 + E + O #= N + 10*C2,
    C4 + N + R #= E + 10*C3,
         D + E #= Y + 10*C4,
    labeling(Carries),
    labeling(Digits),
```

```
    viewable_expand(equation, 1, [C1, C2, C3, C4, 0], "carries").
```

2.1.5 Structured data

In an effort to increase the ease with which program behaviour can be viewed and to provide tighter integration between ECL^{i}PS^e modules, data held in graph structures can also be annotated.
The following code demonstrates how a simple graph structure from the **lib(graph_algorithms)** library can be used to define a **viewable**.

```
:-lib(graph_algorithms).
:-lib(viewable).
:-lib(ic).

test:-
    make_graph(7,
               [e(1,2,F12), e(2,3,F23), e(2,4,F24), e(3,5,F35),
                e(4,5,F45), e(4,6,F46), e(5,6,F56), e(6,3,F63),
                e(6,7,F67)],
               Graph),
    Flows = [F23,F24,F35,F45,F46,F56,F63],
    Flows :: 0..5,
    (for(Node, 2, 6), param(Graph) do
        graph_get_incoming_edges(Graph, Node, InEdges),
        graph_get_adjacent_edges(Graph, Node, OutEdges),
        (foreach(e(_From, _To, Flow), InEdges),
         foreach(Flow, InFlow) do true),
        (foreach(e(_From, _To, Flow), OutEdges),
         foreach(Flow, OutFlow) do true),
        sum(InFlow) #= sum(OutFlow)
    ),
    F12 #= 9,
    viewable_create(flow_viewable, Graph, graph(fixed),
                    [node_property([0->[name(nodes), label]]),
                     edge_property([0->[name(edges), label]])
                    ]),
    labeling(Flows).
```

This simple network flow problem uses the graph structure to hold the problem variables and also to define the network topology. Note the single **viewable_create/4** statement immediately before the labeling step.
As with the regular list/array based viewable create calls, the first argument specifies the viewable name and the structure containing the variables of interest (in this case the graph) comes second. The third argument defines the type as being a graph whose structure is fixed (as all graph_algorithms graphs are). Currently only fixed graphs are supported. The final (optional)

argument defines a mapping between the node/edge structures within the graph and properties useful for visualisation. The table below outlines the currently supported properties.

markup	meaning	applicability	required
name(String)	A unique name to refer to this property	both	yes
label	This property should be used as the node/edge text label	both	yes

For more control over the display of graphs structures, consider using the **lib(graphviz)** library.

2.1.6 Solver variables

The program annotations shown so far will work with most solvers in ECL^{i}PS^e but not all. Generally speaking if the solver operates by monotonically reducing the domain of its variables then no further annotations are required. There are solvers however which do not manipulate variables in this way. For instance the **lib(eplex)** library uses ECL^{i}PS^e program variables as handles to the values calculated by an external solver. When solutions are found by the external solver, the ECL^{i}PS^e variables are not (always) instantiated but rather must be queried to obtain their values.

In order to facilitate the visualisation of such variables, the same **viewable**creation annotations can be used, but the name of the solver must be given explicitly. As an example consider the following **lib(eplex)** model of a simple transportation problem involving 3 factories `1,2,3` and 4 clients `A,B,C,D` taken from the ECL^{i}PS^e examples web page.

```
%----------------------------------------------------------------------
% Example for basic use of ECLiPSe/CPLEX interface
%
% Distribution problem taken from EuroDecision chapter in D4.1
%----------------------------------------------------------------------

:- lib(eplex_xpress).
:- eplex_instance(foo).

%----------------------------------------------------------------------
% Explicit version (clients A-D, plants 1-3)
%----------------------------------------------------------------------

main(Cost, Vars) :-
        Vars = [A1, B1, C1, D1, A2, B2, C2, D2, A3, B3, C3, D3],
        foo:(Vars :: 0.0..10000.0),             % variables

        foo:(A1 + A2 + A3 $= 200),              % demand constraints
        foo:(B1 + B2 + B3 $= 400),
        foo:(C1 + C2 + C3 $= 300),
        foo:(D1 + D2 + D3 $= 100),

        foo:(A1 + B1 + C1 + D1 $=< 500),        % capacity constraints
        foo:(A2 + B2 + C2 + D2 $=< 300),
```

```
        foo:(A3 + B3 + C3 + D3 $=< 400),

        foo:eplex_solver_setup(
                       min(                       % solve
                           10*A1 + 7*A2 + 11*A3 +
                           8*B1 + 5*B2 + 10*B3 +
                           5*C1 + 5*C2 +  8*C3 +
                           9*D1 + 3*D2 +  7*D3)),

        foo:eplex_solve(Cost).
```

Adding the following line immediately before the call to `eplex_solve/1` indicates that the solution values computed by the eplex instance `foo` are of interest. Note the *element type* field of the third argument says that the values of interest may be changed by the solver `foo`. Further note that you will need to load the **viewable**library inorder to access these annotations.

```
viewable_create(vars, Vars
                array([fixed], changeable(foo, any))),
```

This *changeable* element type can appear in any form of the annotations, so as another example, the following annotation gives more structure to the variables.

```
viewable_create(vars, []([](A1, A2, A3),
                         [](B1, B2, B3),
                         [](C1, C2, C3),
                         [](D1, D2, D3)),
                array([fixed,fixed], changeable(foo, any))),
```

As a final example, adding these two lines will make the structure of the problem even more explicit.

```
make_graph_symbolic([]('A','B','C','D',1,2,3),
                    [edge(1,'A',A1),edge(2,'A',A2),edge(3,'A',A3),
                     edge(1,'B',B1),edge(2,'B',B2),edge(3,'B',B3),
                     edge(1,'C',C1),edge(2,'C',C2),edge(3,'C',C3),
                     edge(1,'D',D1),edge(2,'D',D2),edge(3,'D',D3)],G),
viewable_create(network, G, graph(fixed,changeable(foo,graph_data))),
```

viewable_create/2/3/4 used to group problem variables for visualisation purposes. Groupings referred to as **viewable**s.

viewable_expand/3/4 **viewable**s can be of a fixed size, or can expand and shrink.

types elements of a **viewable** may be defined as being numeric values or may be any ECL^{i}PS^{e}term. The type of a **viewable** will determine how it can be visualised.

structure interesting variables contained within graph structures can be directly annotated using the `graph(static)` viewable type.

Figure 2.1: Overview of program annotation

Chapter 3

Visualisation clients

To visualise the **viewable**s of an annotated program, the library **lib(java_vc)** provides a Java based graphical visualisation client.
A new Java visualisation client (Java VC) can be started from the tools menu of tkECLiPSe or using the predicate **start_vc/1** in **lib(java_vc)**. The single argument will return a unique name for the created client which can be used to close the client if required. While the Java VC is running, it will react automatically to the creation of **viewable**s during ECLiPSe execution, but it cannot visualise **viewable**s which were created before the Java VC was running.

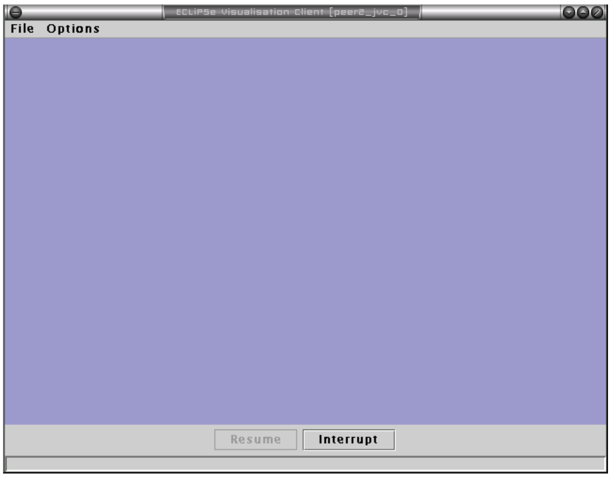

Figure 3.1: The initial Java VC screen before any viewables have been created.

3.1 Control

When running a visualisation-annotated ECL^{i}PS^e program with a Java VC attached, control of the ECL^{i}PS^e process may pass between ECL^{i}PS^e and the VC throughout the program run. That is to say at certain key events in the program, ECL^{i}PS^e will pause in its running of the program and wait for user interaction with the VC before continuing. In such circumstances, the VC is said to *hold* the control.
Table 3.1 details the default behaviour for each of the visualisation events which may occur, and indicates whether or not this default behaviour can be altered.

Event	Triggered by	Default hold	Alterable
viewable creation	**viewable_create/2** **viewable_create/3** **viewable_create/4**	yes	no
viewable expansion	**viewable_expand/3** **viewable_expand/4**	no	yes
viewable contraction	Backtracked over a viewable expansion	no	yes
viewable destruction	Backtracked over a viewable creation	yes	yes
forward update	One or more elements in a viewable have been updated, ie. had their domain reduced or have been instantiated	no	yes
backward update	A forward update has been backtracked over	no	yes

Table 3.1: VC default behaviour for visualisation event.

Should the VC hold, control can be passed back to ECL^{i}PS^e by pressing the **Resume** button at the bottom of the VC window, or by setting the **auto resume** timer. The **Resume** button and the **auto resume** timer are disabled when ECL^{i}PS^e has control, see Figure 3.2.

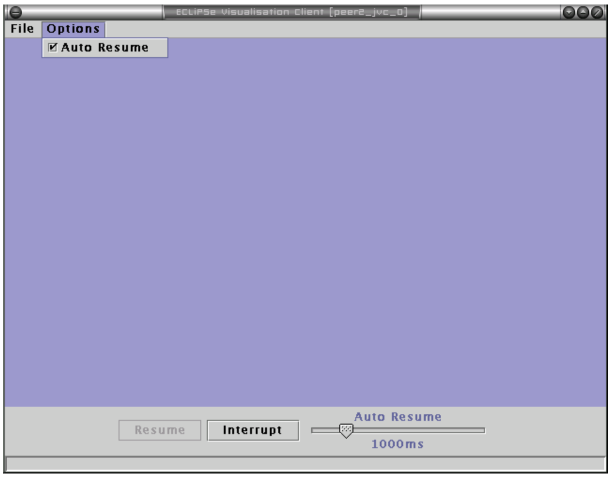

Figure 3.2: The VC showing the auto resume menu option and timer slider.

3.2 Viewlets

The Java VC provides many ways of visualising any single element of a **viewable**.

1. Textually, as though the element had been printed with **write/1**. This is suitable for all **viewable** types.

2. As a rectangular bar on a scale representing the current bounds of a *numeric_bounds* type **viewable** element. Bounds **viewlet**s can be aligned either vertically or horizontally.

3. As a node in a graph, similar to the simple textual representation but enclosed in a geometric shaped node.

4. As an edge in a graph, with the textual representation attached as a label to the edge.

5. With a colour which varies in shade and hue in response to events occurring on the variable.

When rendered on the screen these representations are referred to as **viewlet**s. Figure 3.3 shows the same variable rendered using a number of **viewlet** types.

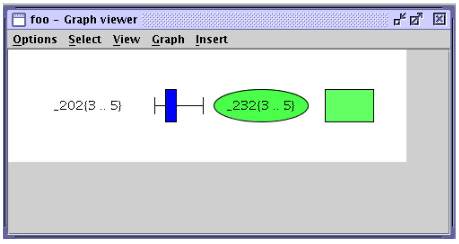

Figure 3.3: The FD variable with initial domain 0..10, reduced to 3..5 as rendered by text, bound, node and fade **viewlet**s.

3.3 Viewers

The Java VC currently contains five different methods for rendering an entire **viewable**. Each of these methods can be thought of as a window looking onto the **viewable** and is referred to as a **viewer**.
Upon a **viewable** being created, the user is presented with a dialog box asking which of the available **viewer**s they wish to view the **viewable** with.
The currently available viewers are

TextTable Renders any type of 1D and 2D **viewable**s as a grid of textual descriptions of the elements.

BoundsTable Renders numeric_bounds 1D and 2D **viewable**s as a grid of rectangles representing the size of the numeric domains.

FadeTable Renders 1D and 2D **viewable**s as a grid of coloured rectangles whose colour changes represent domain changes in the **viewable** elements.

Desktop Allows the user to place all available representations of the **viewable** elements anywhere on a desktop window. Also enables the loading of an arbitrary background image from file, and for placing images alongside **viewlet**s.

Network Renders `graph(fixed)` **viewable**s graphically as connected nodes, where the textual representation of the **viewable** elements is displayed at nodes and along edges.

Network (0/1) Similar to the Network viewer except that if the edge annotation can be interpreted as the number 0, then the edge is not drawn. If it can be interpreted as the number 1, it is drawn in black. Any other value has the edge draw in gray.

Network (Capacity) Similar to the Network viewer except that the edge labels are interpreted as fractions indicating the capacity of a link in a flow network. 0.0 indicating unused (thin black line) up to 1.0 indicating full usage (thick black line) and any number greater than one indicating over utilisation (very think red line). If the edge data cannot be interpreted as a number (eg. it is a variable) it is assumed to be 0.

Gantt Interprets the first three rows of any 2D viewable with `numeric_bounds` elements (and at least 3 rows) as being the start times, durations and resource requirements of a scheduling problem. The resulting schedule/partial schedule is rendered as a gantt chart.

Bar chart Renders any n-dimensional `numeric_bounds` viewable as a bar chart. Extra dimensions will be separated by gaps in the chart.

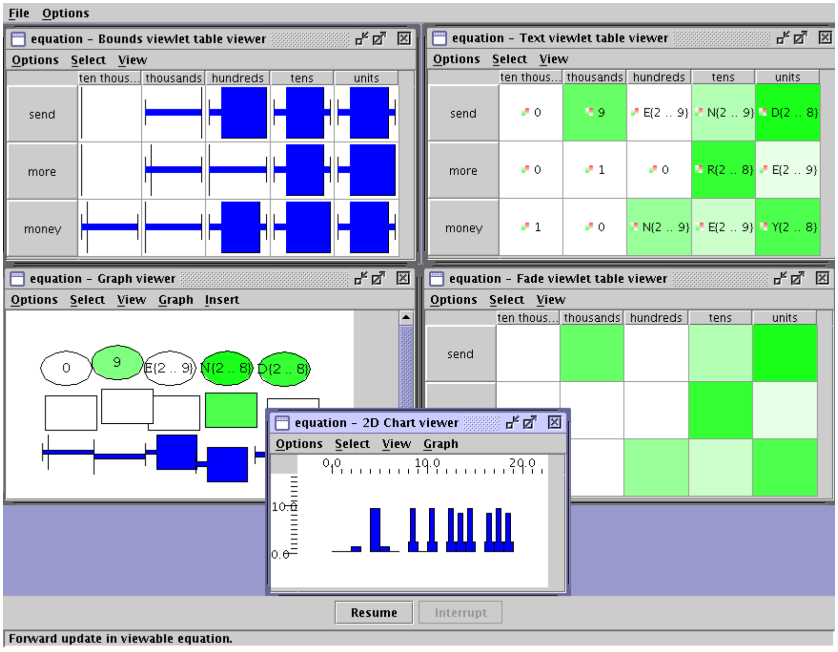

Figure 3.4: The VC showing some of the applicable viewers for the `SEND+MORE=MONEY` example.

Common to all viewers are the three menus **Options**, **Select** and **View**, the latter two also being accessible by pressing the right mouse button.

3.3.1 Options menu

The options menu contains controls for **viewer**-wide properties.

Hold at expansions Determines whether this **viewer** will hold control when the **viewable** is expanded.

Hold at contractions Determines whether this **viewer** will hold control when the **viewable** is contracted.

Hold at destruction Determines whether this **viewer** will hold control when the **viewable** is destroyed. This option is useful for examining the state of the **viewable** immediately before the creation is backtracked over.

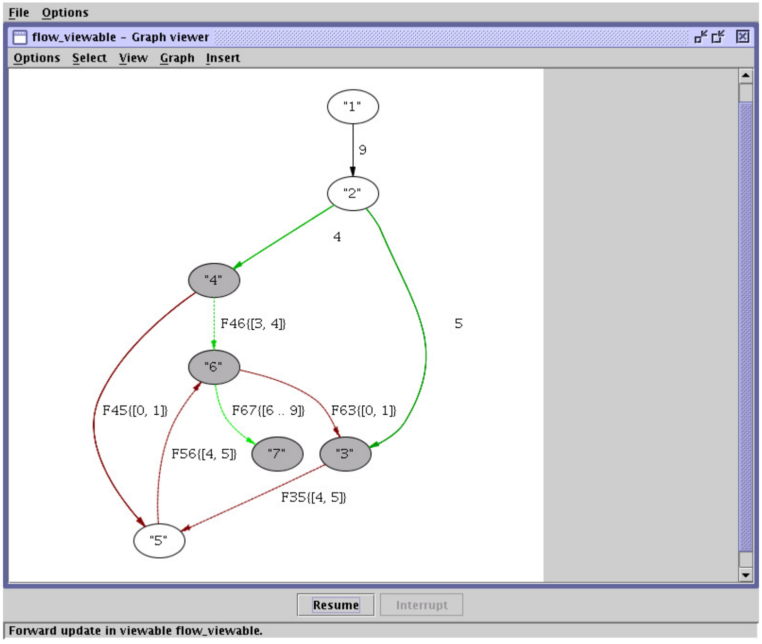

Figure 3.5: The VC showing the network viewer displaying the graph example.

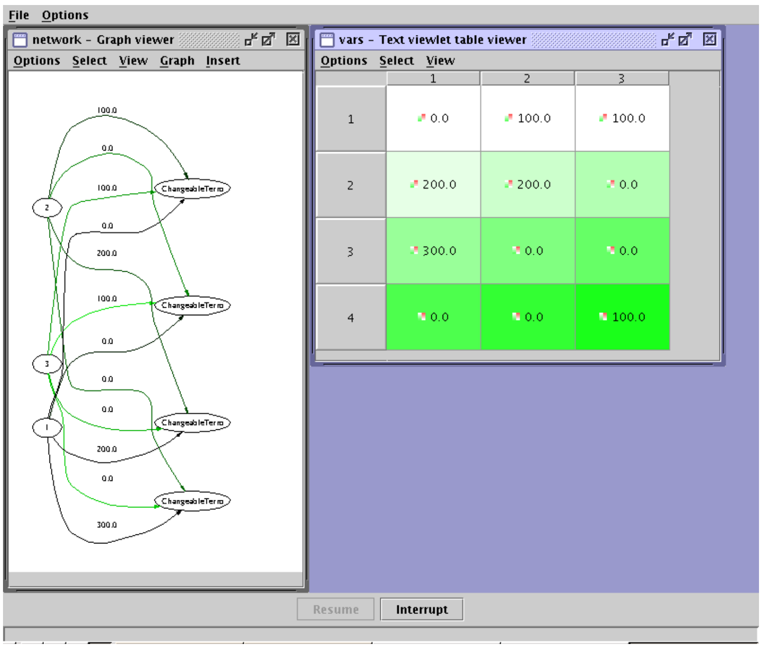

Figure 3.6: The VC showing various viewers for the changeable solver example.

View propagation steps Controls how frequently the visualisation client is informed of *forward update* events.

fine Events are sent as soon as they occur.

coarse Events are sent at priority 8 in the ECL^{i}PS^e program. Typically this means that all the propagation that occurs as a result of a single user level search step are sent together.

timed Events are collected and sent at regular timed intervals.

Track updates When set, the **viewer** will attempt to ensure that all updates are visible within the window. This can be important when visualising large **viewable**s which may not easily fit the window.

Figure 3.7 shows the default settings for the **Options** menu. Note that the **View propagation steps** options are disabled because ECL^{i}PS^e has control and the update granularity can only be changed when the Java VC is holding control.

Figure 3.7: The options menu, common to all viewers.

3.3.2 Select menu

Contains convenience commands for dealing with the currently selected set of **viewlet**s.
Selecting individual **viewlet**s can be done clicking on them with the left mouse button, whilst selecting ranges can be done by dragging the mouse across a range of **viewlet**s.

Select all viewlets Sets the selection to the entire **viewable**.

Select updating viewlets(s) Sets the selection to only those **viewlet**s which have been marked as updating (either *forward* or *backward*). This option is only enabled when the Java VC has control, since it requires the state of the viewables to remain constant during the selection process.

Clear selection Clears the selection.

Figure 3.8: The select menu, common to all viewers.

3.3.3 View menu

So as to facilitate visualisation of large **viewable**s, all **viewer**s have the ability to zoom in and out. All the options are self explanatory and will not be expanded further upon except to mention that the **Zoom to fit width** and **Zoom to fit height** options operate on the whole **viewer** and not just the selected **viewlet**s.

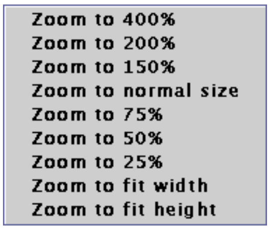

Figure 3.9: The view menu, common to all viewers.

Both the network and desktop viewers have an extra item on the view menu, *Toggle high quality*. This toggles between quick rendering and high quality views, and may help to make the VC more reactive under high load.

3.3.4 Viewlet actions

Within a **viewer**, as previously mentioned, any number of **viewlet**s may be selected. These **viewlet**s, once selected can have actions performed on them. The actions are selected by pressing the right mouse button in order to bring up the context sensitive actions menu. If the **viewlet**s in the selection are of different types then all the available actions are displayed and once one has been selected, it will be applied to all applicable **viewlet**s in the selection. This is a change from previous versions of the visualisation client, which would display only those actions common to all **viewlet**s.

Hold on update

The most common action, which can be performed on any type of **viewlet** is the *Hold on updates* action, which, when set, indicates that the Java VC should hold control whenever any sort of update event is issued for the corresponding **viewable** element. The *Hold on updates* property of a **viewlet** is indicated by a slight "greying" out of the **viewlet**, or in the case of **viewlet**s attached to edges in the network viewer, the edge is drawn "dotted" instead of solid.
Figure 3.10 shows the graphical effect of setting the *Hold on update* property of a text **viewlet**. Table 3.2 lists the available **viewlet** actions and indicates for which type the actions are valid.

3.3.5 Desktop/Network viewers

All the **table** viewers have essentially the same functionality – they do not allow flexible placement of viewables and both deal only with 1 or 2 dimensional **viewable**s. A more flexible **viewer** is provided in the **Desktop viewer**.
This **viewer** aims to implement the common *desktop* metaphor by providing the user with a rectangular region of the screen upon which **viewlet**s can be dropped, stacked and moved around as though they were pieces of paper on a desk.

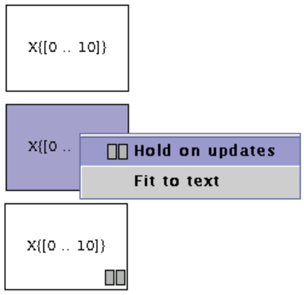

Figure 3.10: The sequence of actions required to select **Hold on update** for a **viewlet**

Name	Description	Applicable
Hold on updates	Causes the VC to hold control on forward or backward update events for the selected **viewlet**s.	all
Fade update history	Toggles using the background color of the viewlet to indicate recent update history. This has the effect of fading from green to white in the event of a forward update and from red to white for backward updates.	text, node, fade, edge
View bounds in detail	Pops up a window detailing the original bounds and the current bounds for the single selected **viewlet**.	bound
Align bounds	Causes the selected **viewlet**s to use the same underlying scale when displaying the bounds. This allows variables whose initial bounds were different to be visually compared.	bound
Toggle horizontal/vertical range bar	Toggles the rotation of the bar for all bounds **viewlet**s	bound

Table 3.2: The available **viewlet** actions and associated types.

Adding viewlets

Typically, **viewlet**s will be added to a desktop immediately after the **viewer** has been created. To minimise the overhead of having to layout the **viewlet**s each time the user's program is run (a potentially time consuming task), the Java VC provides an automatic *recording and repeat* mechanism which is triggered every time a **viewer** is created. Section 3.4 explains this feature in more detail.

Adding **viewlet**s to a Desktop **viewer** is done by selecting the required **viewlet** type from the **Insert** menu. This menu will contain only those **viewlet** types which are appropriate for the type of the **viewable**.

Once an appropriate **viewlet** type has been selected, the range selection dialog will pop up, from which any combination of dimension ranges may be selected.

Figure 3.11 shows the range select dialog for the on going `SEND+MORE=MONEY` example.

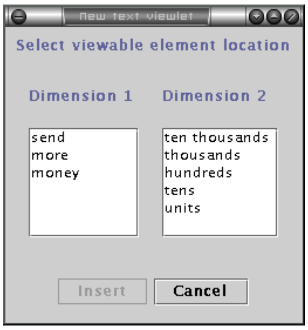

Figure 3.11: The range selection dialog for the `SEND+MORE=MONEY` example

At least one selection must be made from each of the dimensions, though it is possible to select multiple values in each dimension.

Figures 3.12 and 3.13 illustrate the default layout of **viewlet**s when 1 and 2 dimensional ranges are selected. The desktop will automatically resize to ensure that all **viewlet**s fit. Attempts to move a viewlet *off* the desktop will cause it to grow.

Higher dimension range selections result in a stacked 2D grid, with progressive dimensions appearing underneath the initially visible grid.

3.3.6 Adding images

As well as **viewlet**s, the **Desktop viewer** can show icons loaded from disk by selecting the **Image** option from the **Insert** menu. This brings up a file selection dialog from which the user may select an image file to load. The loaded image will be added to the **viewer** as a small icon which is selectable and movable like other items on the desktop. Currently there is no way to increase the size of the loaded image.

Background images

In keeping with the computer GUI *desktop* metaphor, the user may set the *background* image for the desktop **viewer**. Aside from making the **viewer** look pretty this feature is intended to allow graphical context to be associated with the visualisation of a program. For example the background image could be a diagram representing the network topology and the values being visualised could be the flows through various parts of the network. By placing the **viewlet**s near

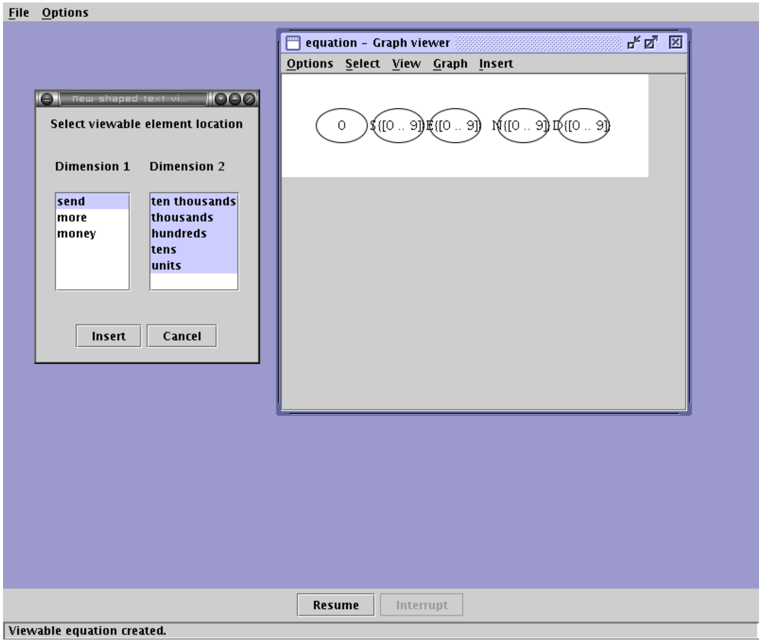

Figure 3.12: The result of selecting a 1D range

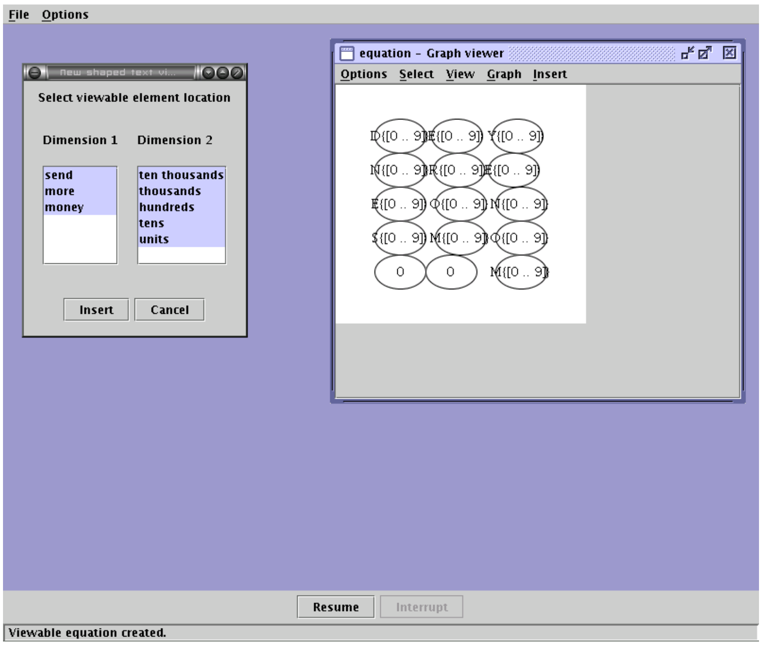

Figure 3.13: The result of selecting a 2D range

the appropriate nodes on the background image the user could more easily visualise the network flow problem.

Background images are loaded by selecting the **Import background image** option from the **Background** menu and are removed by selecting the **Clear background** option. Currently only **GIF**, **PNG** and **JPEG** format images can be loaded.

In keeping with our `SEND+MORE=MONEY` example, figure 3.14 shows the problem visualised on a desktop viewer, placed over a background image[1].

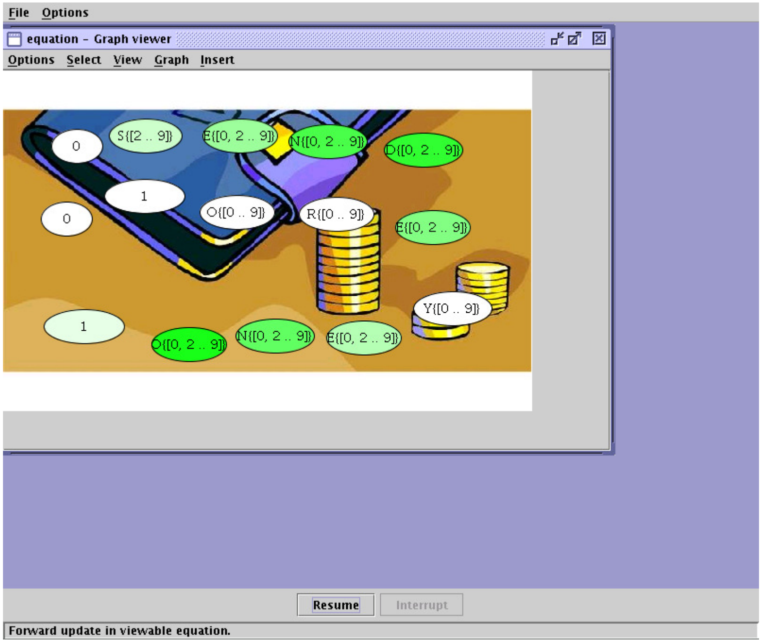

Figure 3.14: The `SEND+MORE=MONEY` example displayed on a Desktop **viewer** with a background image

3.3.7 Layout

Items on the desktop may be manually positioned by selecting (single click) and dragging (click-and-move) them. New items may be added to the current selection by holding down the **Ctrl** key whilst clicking with the left mouse button. Ranges of items are selected by clicking on the background of the desktop and dragging a selection rectangle around the desired items. When dragging a selection all items move, except lines on the **Network viewer**.

It is also possible to use one of the automatic layout options available from the **Graph** menu. These options make use of the external graph layout tools `dot`, `neato` and `twopi` from the AT&T Labs Research project *Graphviz*[2]. These tools should be automatically installed as part of the ECL^{i}PS^e installation procedure.

[1]Background image ©1999-2003 `www.barrysclipart.com`

[2]`http://www.research.att.com/sw/tools/graphviz/`

3.3.8 Gantt charts

The Gantt chart viewer has many of the same options as the Network viewer previously mentioned but in addition, the **Gantt** menu provides access to options that control how transparent the individual gantt task bars are drawn. By selecting the **transparent** option, regions where tasks overlap will be rendered in a darker colour. The darker the colour, the more tasks overlap there.
When either the start time or the duration of a task is a variable, then the task will be draw as two connected bars indicating the earliest & shortest possible occurrence of the task and the latest & longest possible occurrence.
Above the gantt chart is a numeric scale indicating time. By clicking and dragging this scale can be expanded or shrunk so as to fit the gantt chart into the window. This feature works independently of the zoom.

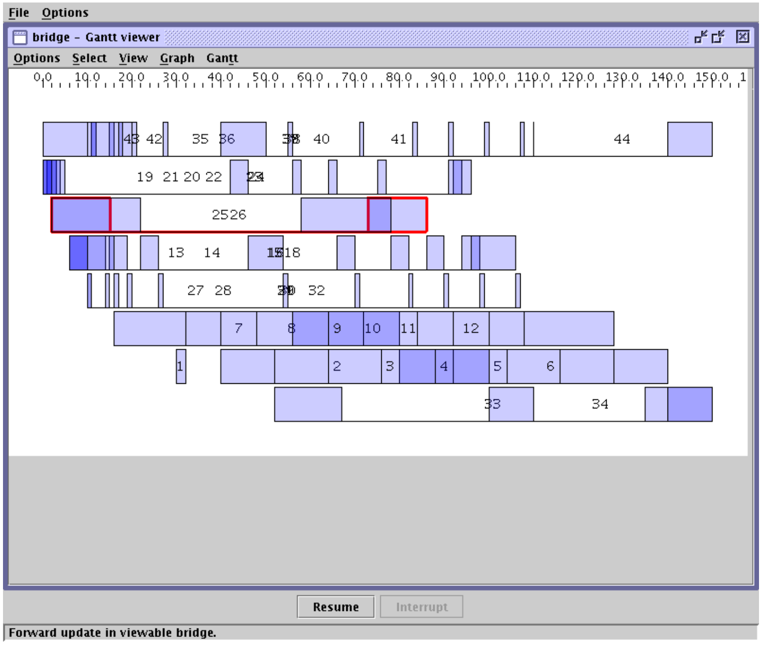

Figure 3.15: The VC showing the Gantt viewer for a scheduling example. Note the highlighted task showing the earliest start/shortest and latest/longest times of the task.

3.3.9 Printing

To print the current state of almost all viewers, right-click on the background and select the **Print** option from the popup menu. This will bring up the print dialog as shown in figure 3.16.

3.4 Scenarios

To make the process of setting up the visualisation environment and the laying out of **viewers** and **viewlets** quicker, the Java VC provides a *record and playback* feature where all user visible changes and actions that are performed following the creation of a **viewable** are recorded in a

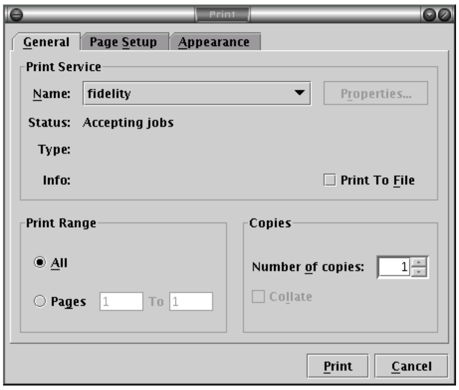

Figure 3.16: The print options dialog box.

visualisation **scenario**. This action sequence can then be optionally re-played the next time a **viewable** of the same name is created.
The common use case is as follows.

1. Start Java VC.
2. Run program which creates **viewable** "foo" for the first time.
3. Select **viewer**s for "foo".
4. Arrange **viewer** windows on screen, resize and scale to taste. Optionally insert and layout **viewlet**s on a Desktop viewer.
5. Press **Resume** button to continue running program.
6. Watch visualisation of program run until **viewable** is destroyed (ie. is backtracked over).
7. Re-run program, after having made some changes.
8. Answer **yes** to the prompt to *reinstate visualisation preferences for viewable "foo"*.
9. Watch as things magically re-arrange themselves into the configuration you previously had.
10. (optional) Make some more layout changes.
11. Press **Resume** again.
12. Repeat.

To make long running visualisation projects easier and also to assist in running demonstrations, these visualisation preferences can be saved to disk and loaded back into memory at any time. The loading and saving of scenarios is achieved by using the **Load** and **Save** options of the **File** menu. The most common point at which a **scenario** is saved is just after laying out all the **viewer**s and just before passing control back to ECL^{i}PS^e. It should be noted that the scenarios (settings) for many different viewables can be saved into/loaded from a single file, this is to aid visualisation of large programs.

Index