

Steven P. Reiss
Brown University

FIELD

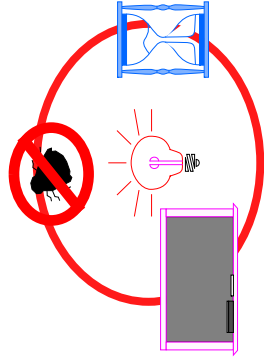
❖ A ❖

- ✓ **F**riendly,
- ✓ **I**ntegrated
- ✓ **E**nvironment for
- ✓ **L**earning and
- ✓ **D**evelopment

Overview of Talk

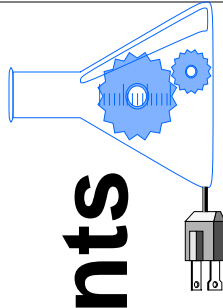
- ❖ **Previous work**
- ❖ **Motivation of FIELD**
- ❖ **The integration mechanism**
- ❖ **Data structure display**
- ❖ **Current research directions**

Programming Environments

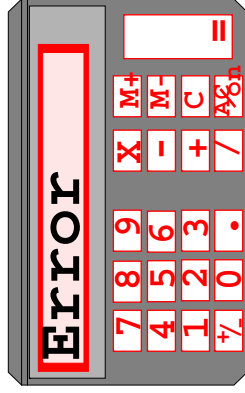


1. Loose collections of tools

2. Closed, powerful environments

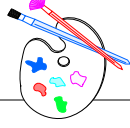


3. Student environments



*We want the best of all these worlds—
Without the drawbacks.*

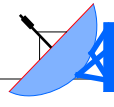
Motivations for FIELD



Graphical UNIX programming



Non-toy Environment for teaching



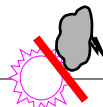
Showcase for research at Brown



Basis for future research projects



Front-end for algorithm animations



Workstation-independent interface








Distributed and threaded programs

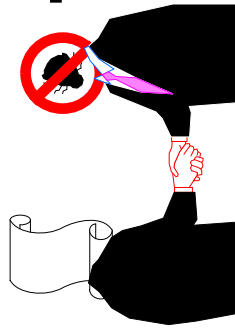


To do as little work as possible

Overview of FIELD

-  **Open collection of tools**
-  **Selective Broadcasting**
-  **Source annotations**
-  **Emphasis on visualization tools**
-  **Standard workstation interface**

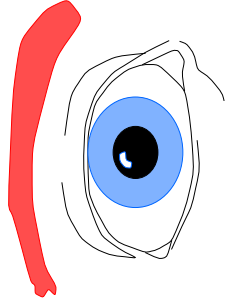
Integration Requirements



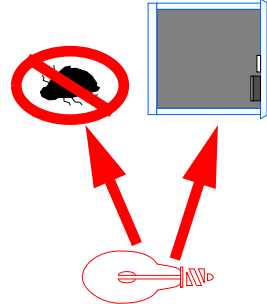
Tools must be able to interact with each other directly.



Dynamic information must be shared.



All source action should be through a single view.




Specialized information must be available to all tools.

Our Requirements

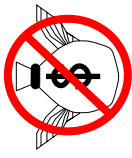
 **Extensible**

 **Simple enough for novices**


 **Rich enough for research**

 **Multiple Processes & machines**


 **Excellent student environment**

 **Inexpensive to build and maintain**

Integration Alternatives

 **Graft Tool Functionality**

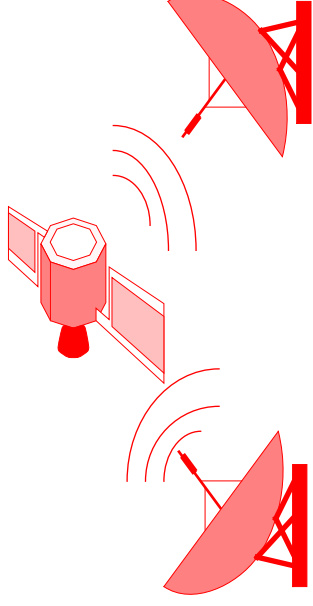
 **Program Database**

 **Single System**

 **Selective Broadcasting**

Selective Broadcasting

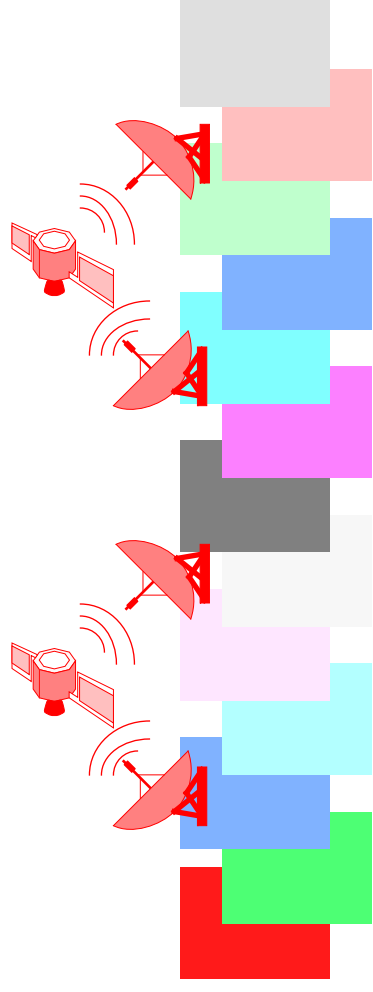
☞ Central Message server (MSG)



☞ Clients

- Send and receive messages
- ✗ Command requests
- ✗ Information
- Can be added or removed

Message Patterns

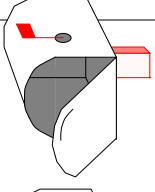
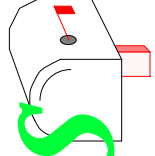
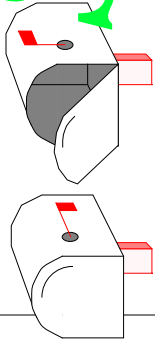


☞ Messages are strings

☞ Conventions for encoding

- ✗ WHO CMD system args...
- ✗ ID WHAT system info...

Sample Messages



DDT VIEW *sys file fct line cnt stkdelta*

DDTR RUN *sys args in out new*

DEBUG AT *sys file fct line*

DEBUG VALUE *sys file line var value*

XREF QUERY *sys args conditions*

XREF FLOW SET *file line button*

BUILD MAKE *system*

FORM CONFIG *dir file CHECKIN mode*

MSG Operation

- 👉 **Clients register patterns**
 - **Scanf- based pattern matching**
 - **Normal text matches exactly**
 - **%s, %d, %f — argument match**
- 👉 **Tools send messages to server**
 - **Rebroadcast selectively**
 - **Printf formatting for send calls**

MSG: Synchronization

 Messages can be asynchronous

- **Sender continues immediately**

 Messages can be synchronous

- **Wait for 1st non-null reply**
- **Wait for all receivers**

 Messages are decoded by MSG

- **Receiver gets args & reply id**

MSG Utility Services

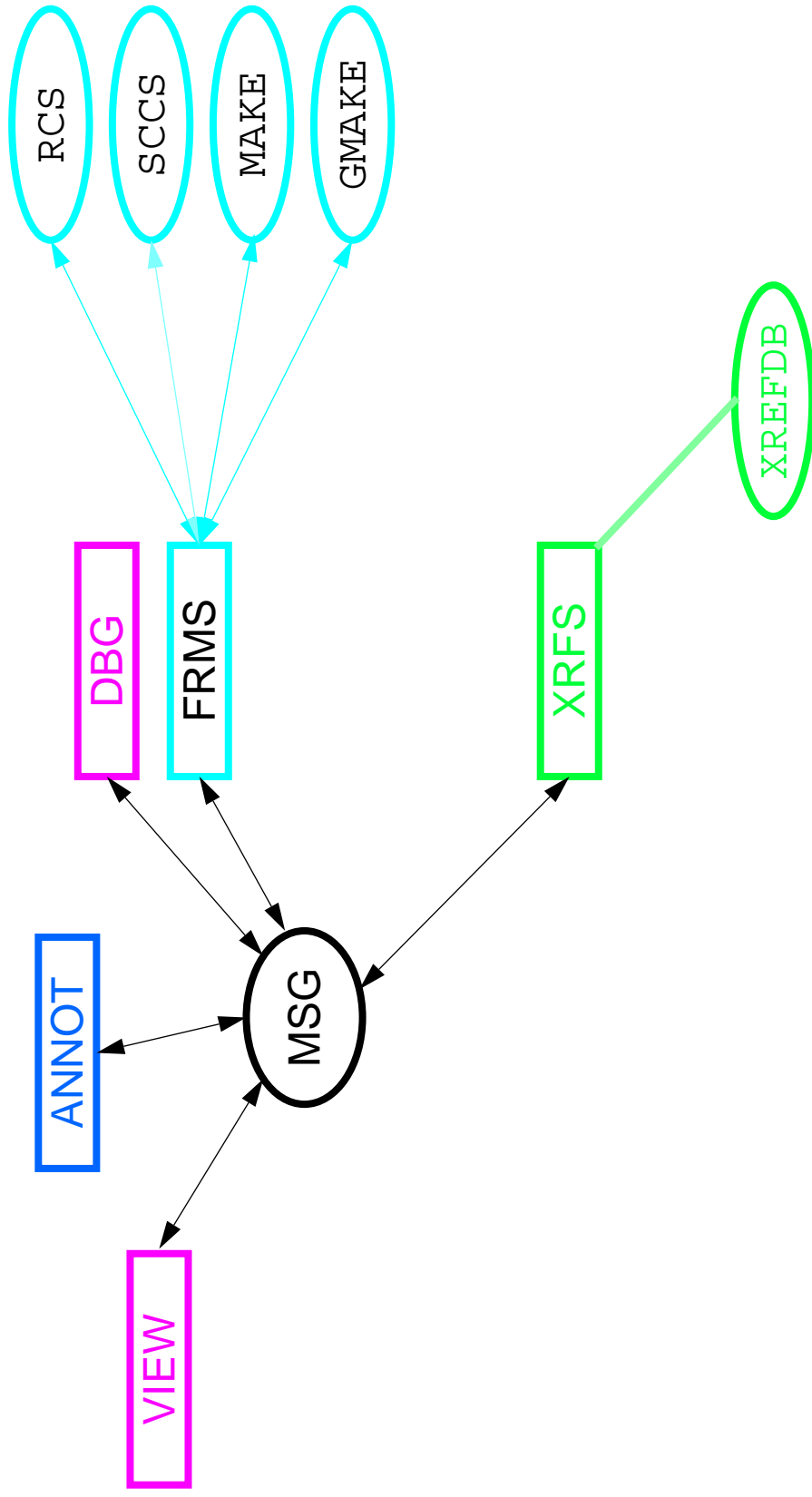
- ☞ **MSG is responsible for services**
 - **Central to the environment**
 - **Every tool talks to it**

☞ **Working Directory**

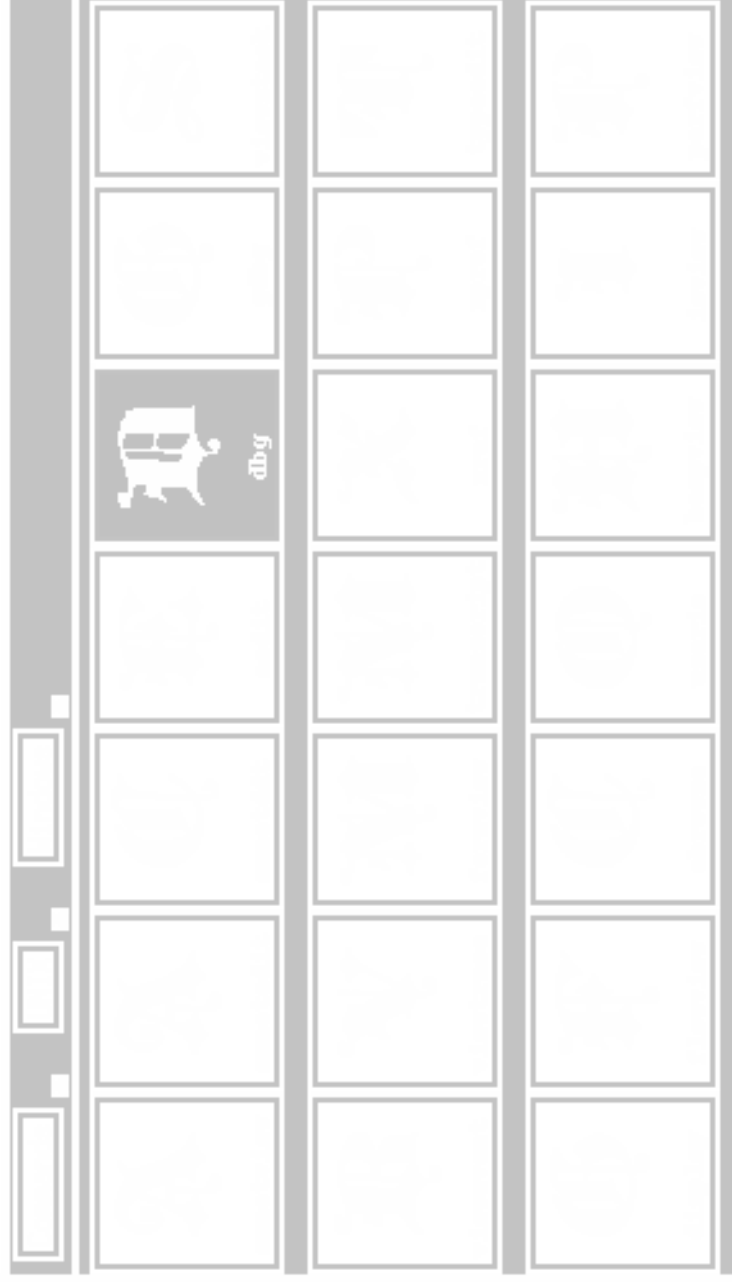
☞ **Check and start services**

☞ **Environment**

Basic Organization



Control Panel



Overview Screen

The screenshot displays a debugger's Overview screen. At the top, a grid of 12 small code windows is visible. Below this, a larger window shows the source code for the function `tree_walk`. The code includes a `main` function that initializes a tree, a `tree_walk` function, and a `static TREE` structure. The current execution point is at line 31 of `tree.c`, where `root = insert_tree(root, j);` is being executed. The right side of the screen shows the debugger's control panel with buttons for `Step`, `Next`, `Continue`, `Run`, `Stop`, `Print It`, `View`, `File`, `Edit`, and `Move`. The status bar at the bottom indicates the current file and line number.

```
static void tree_walk()
{
    main()
    {
        Integer i, j, ct;
        ct = 30;
        root = NULL;
        For (i = 0; i < ct; ++i) {
            j = random() % 1024;
            root = insert_tree(root, j);
        }
        tree_walk(root);
    }
}

static TREE
insert_tree(t, w)
TREE t;
Integer w;
{
    if (t == NULL) {
        t = (TREE) malloc(sizeof(TREE));
        t->lson = NULL;
        t->rson = NULL;
        t->v = w;
    }
    else if (w < t->v) {
        t->lson = insert_tree(t->lson, w);
    }
    else {
        t->rson = insert_tree(t->rson, w);
    }
}






ReadOnly Insert_Indent
```

Buttons: File Edit Move
Reading symbolic information...
Read 584 symbols
Signal 10 will be ignored
Signal VTRAP will be ignored
<ctrl> run
Running: tree
Stoppoint [1]: Stopped at line 31 in main of file tree.c
31 root = insert_tree(root, j);
<ctrl> 2)

Step Next Continue Run Stop Print It
View File Edit Move
j: main(), at line 31 of tree.c
ct = 30
i = 0
j = 445
GLOBALS:
root = (nil)

Annotations Search Comments Edit Move
Breakpoint
BREAK
STEP
FORCE
DEBUG
EVENT
tree.c 31/86

DDT Program Control

-  **dbx or gdb as low-level interface**
-  **MSG-based debugger back end**
-  **Textual debugger front end**
-  **Window interface to debugger**
-  **Separate interface for I/O, Views**

Debugger Extensions

 **dbx + gdb command languages**

 **Programmability**

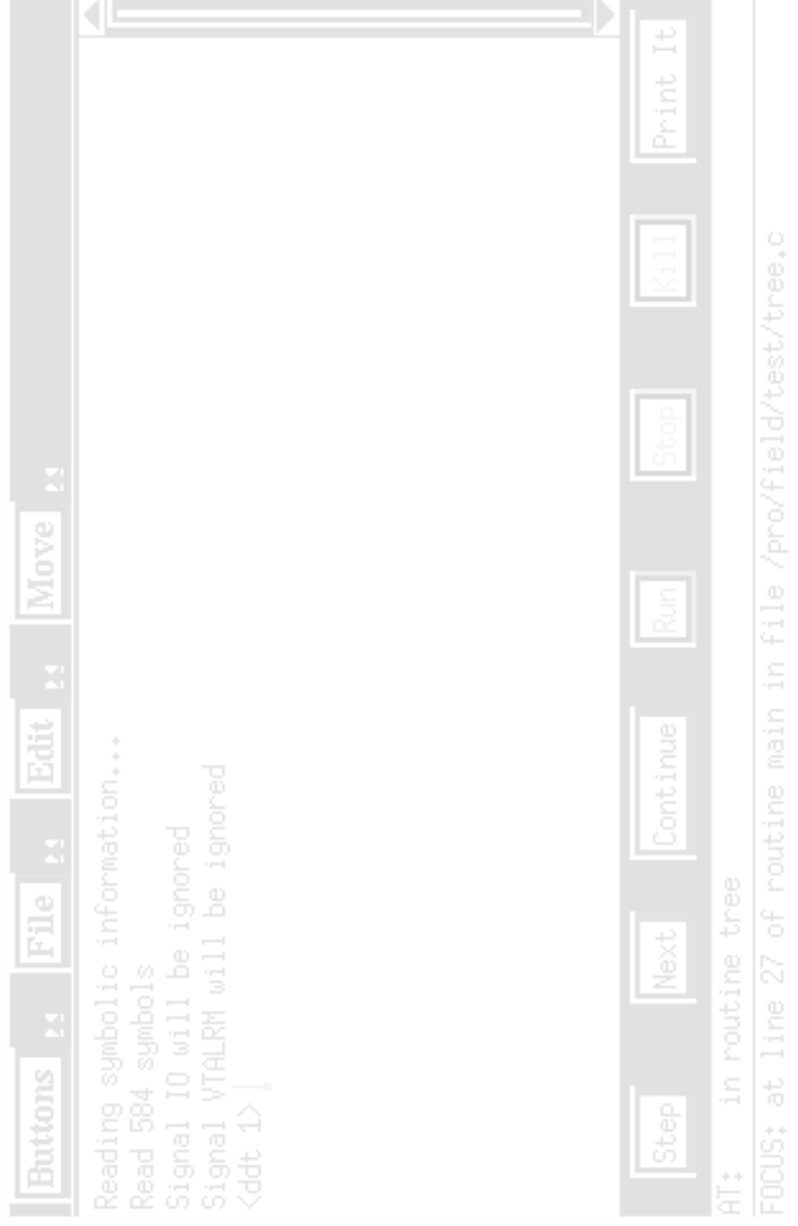
 **C++, Pascal, Modula 3 Support**

- **Full name mapping**
- **Dynamic type information**

 **Extended commands**

- **Stop in all <pattern>**

Debugger Interface



Annotation Editor

The screenshot displays the Annotation Editor interface. The main window contains a code editor with the following C++ code:

```
static void tree_walk();  
  
main()  
{  
    Integer i,k,j,ct;  
    ct = 30;  
    root = NULL;  
    for (i = 0; i < ct; ++i) {  
        j = random() % 1024;  
        root = insert_tree(root,j);  
    }  
    tree_walk(root);  
};  
  
static TREE  
insert_tree(t,v)  
TREE t;  
Integer v;
```

Annotations are visible in the left margin of the code editor:

- A mouse cursor icon is positioned over the line containing `Integer i,k,j,ct;`.
- A mouse cursor icon is positioned over the line containing `root = insert_tree(root,j);`.
- A mouse cursor icon is positioned over the line containing `tree_walk(root);`.

On the right side of the interface, there is an "Annotations" panel. It features a vertical scrollbar and a list of annotation actions: BREAK, UPDATE, TRACE, FOCUS, and EVENT. Below the code editor, there is a status bar with the text "Insert Indent" and "tr 25/80".

Annotations

- 👉 **Annotations relate source to tools**
- 👉 **Integrated through messages**
- 👉 **Defined in resource files**
- 👉 **Can be permanent**
 - **Original file, annotations saved**
 - ***Fixannot* uses *diff* & heuristics**

Configuration Management

 **Formview offers interactive MAKE**

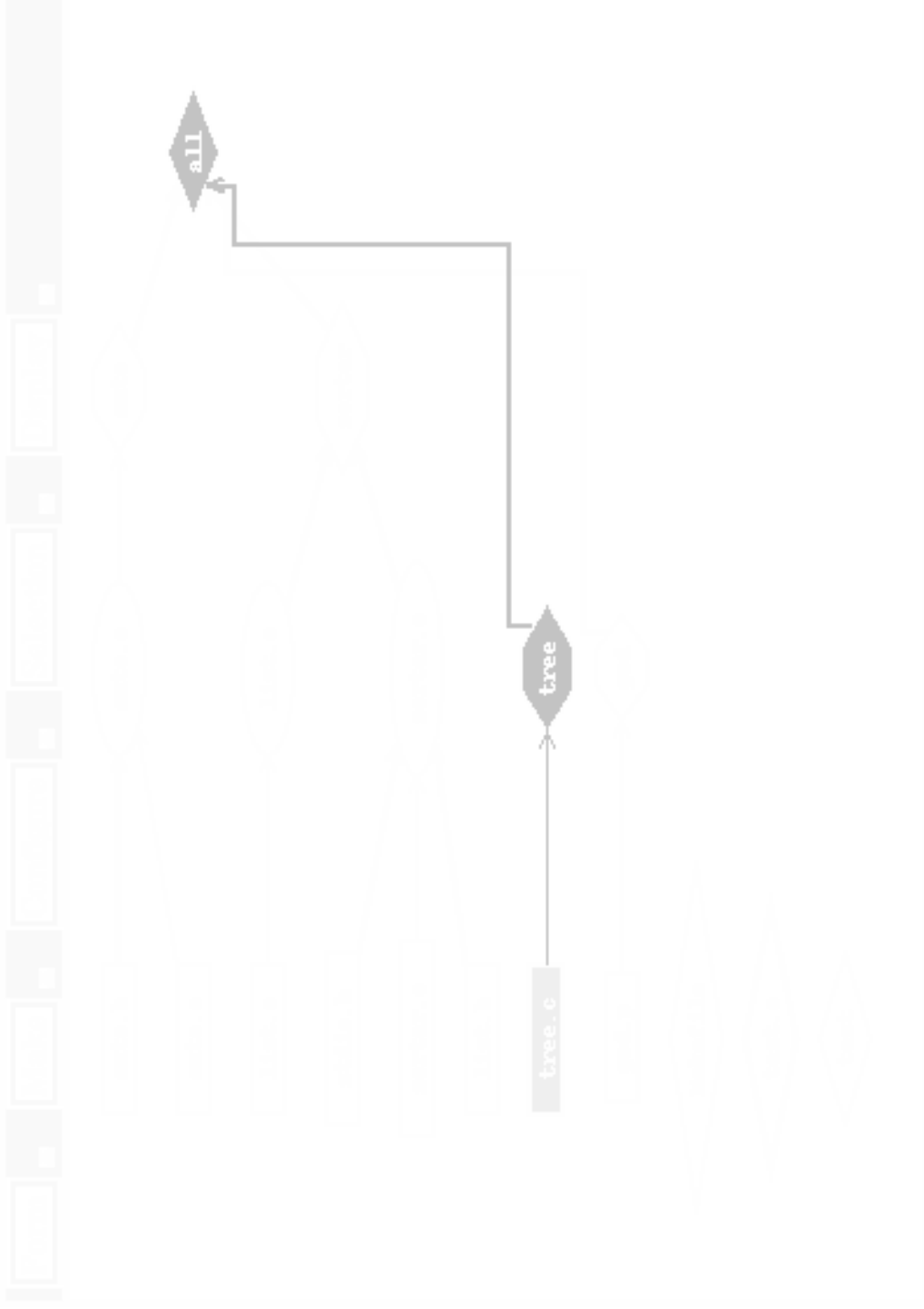
 **MSG interface**

- **Handle: BUILD, COMPILE, CONFIG**
- **Generate: ERROR, WARNING**

 **Graphical dependency Display**

 **RCS support**

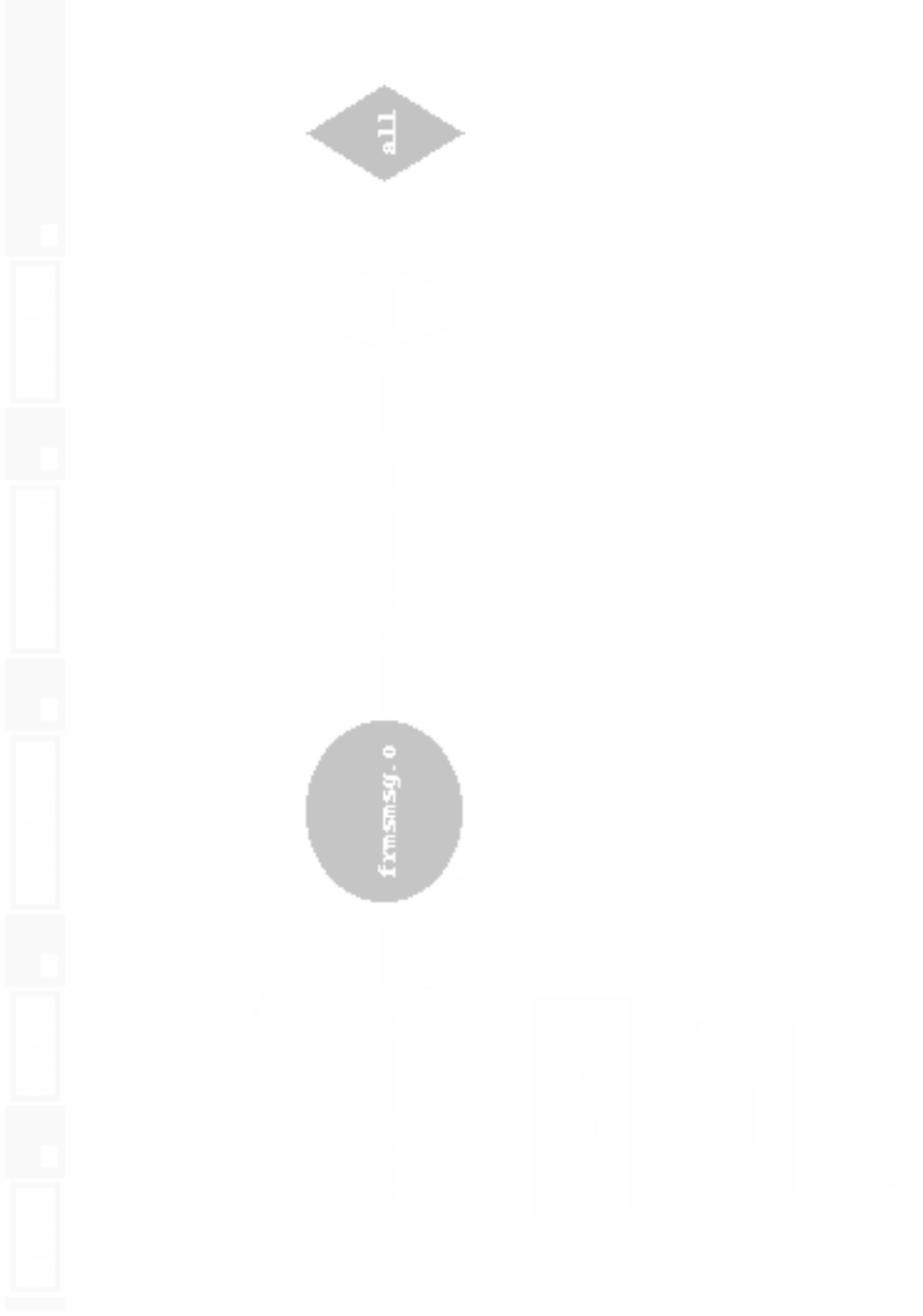
FORMVIEW Example



FORMVIEW Example



FORMVIEW Example



Cross Referencing

 **XREF offers interactive database**

- **Definable QBE-like queries**
- **Full relational database**

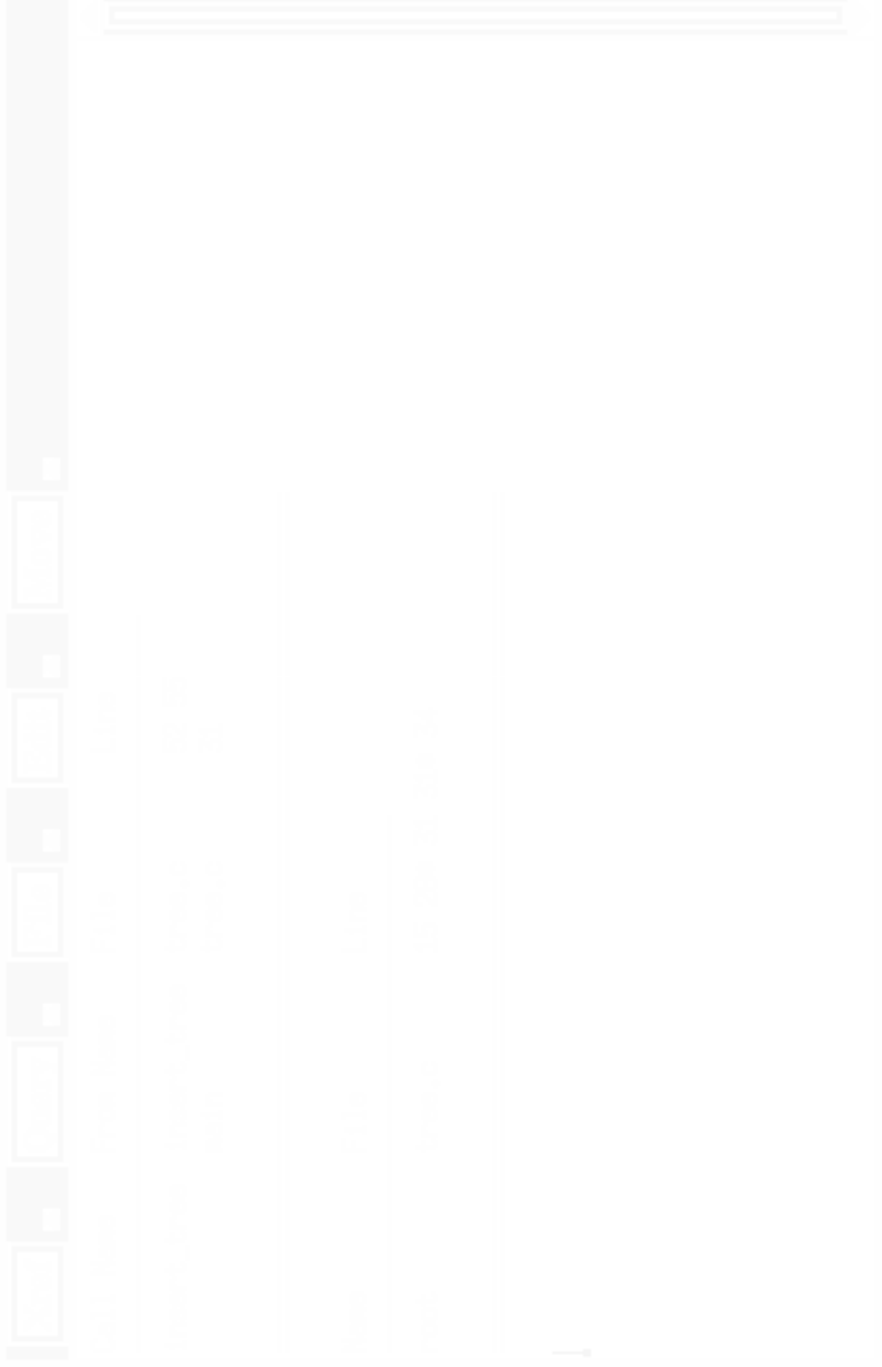
 **XREFDB provides services**

- **Relational calculus queries**

 **MSG interface**

- **Handles: XREF QUERY**
- **Generates: XREF SET**

XREF Example

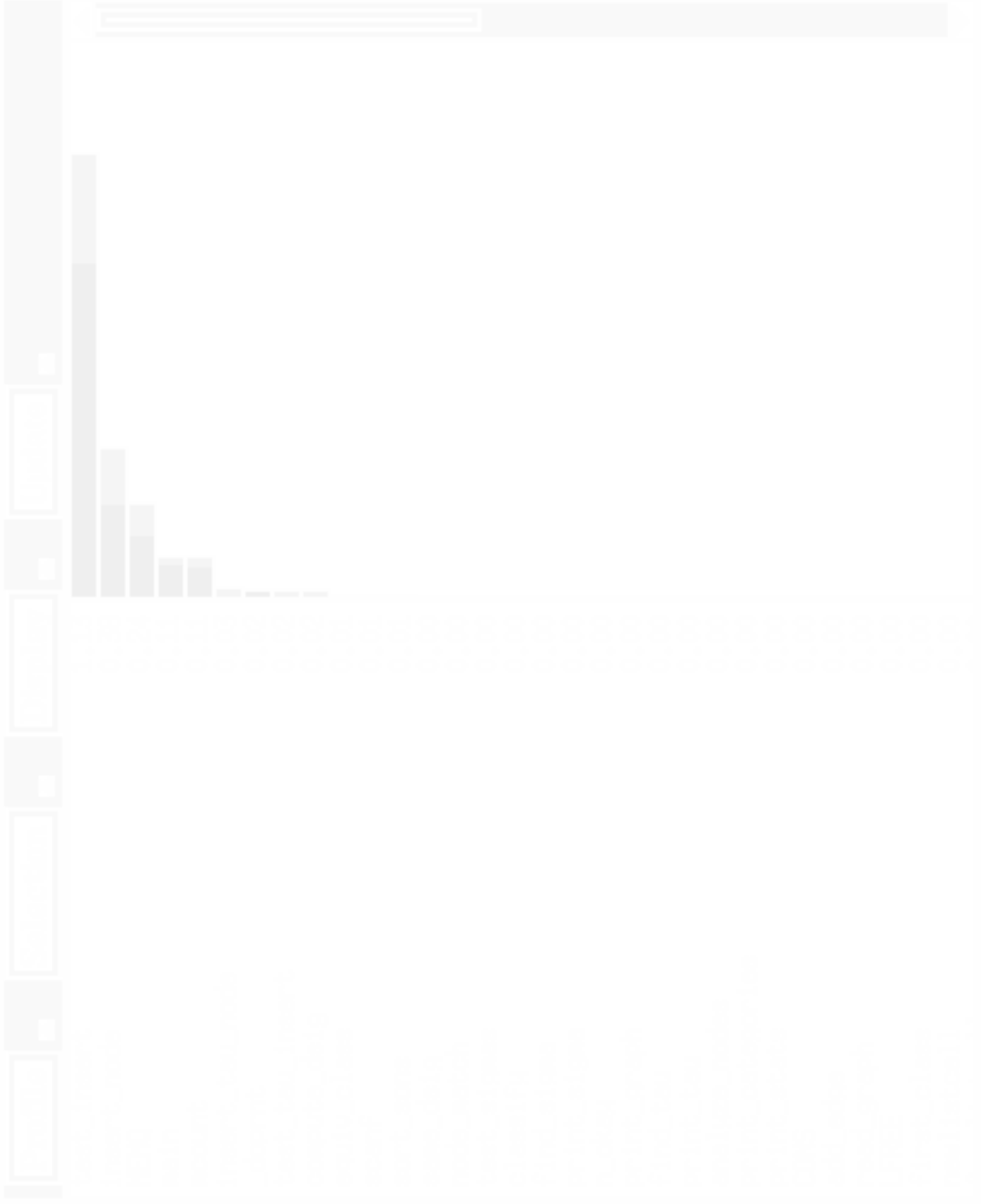


Profiling Utility

- 👉 Visual front end to UNIX profiling
 - *prof, gprof, pixie, iprof, fgprof*
 - Selective viewing -- file, fct, line
 - Incremental profiling capability

- 👉 MSG interface
 - Generates: XREF PROF SET

XPROF Example



Flowgraph Visualization

Call Graph Display

- Use XREFDB to get information
- Hierarchical display; Grouping

Browsing Capabilities

- Using hierarchy
- Selective include and exclude
- Local graph for a node

Tracing Program Execution

FLOWVIEW Example



FLOWVIEW Example



Class Visualization

 Graphical view of class hierarchy

- Full browsing capabilities

 Shows wealth of information

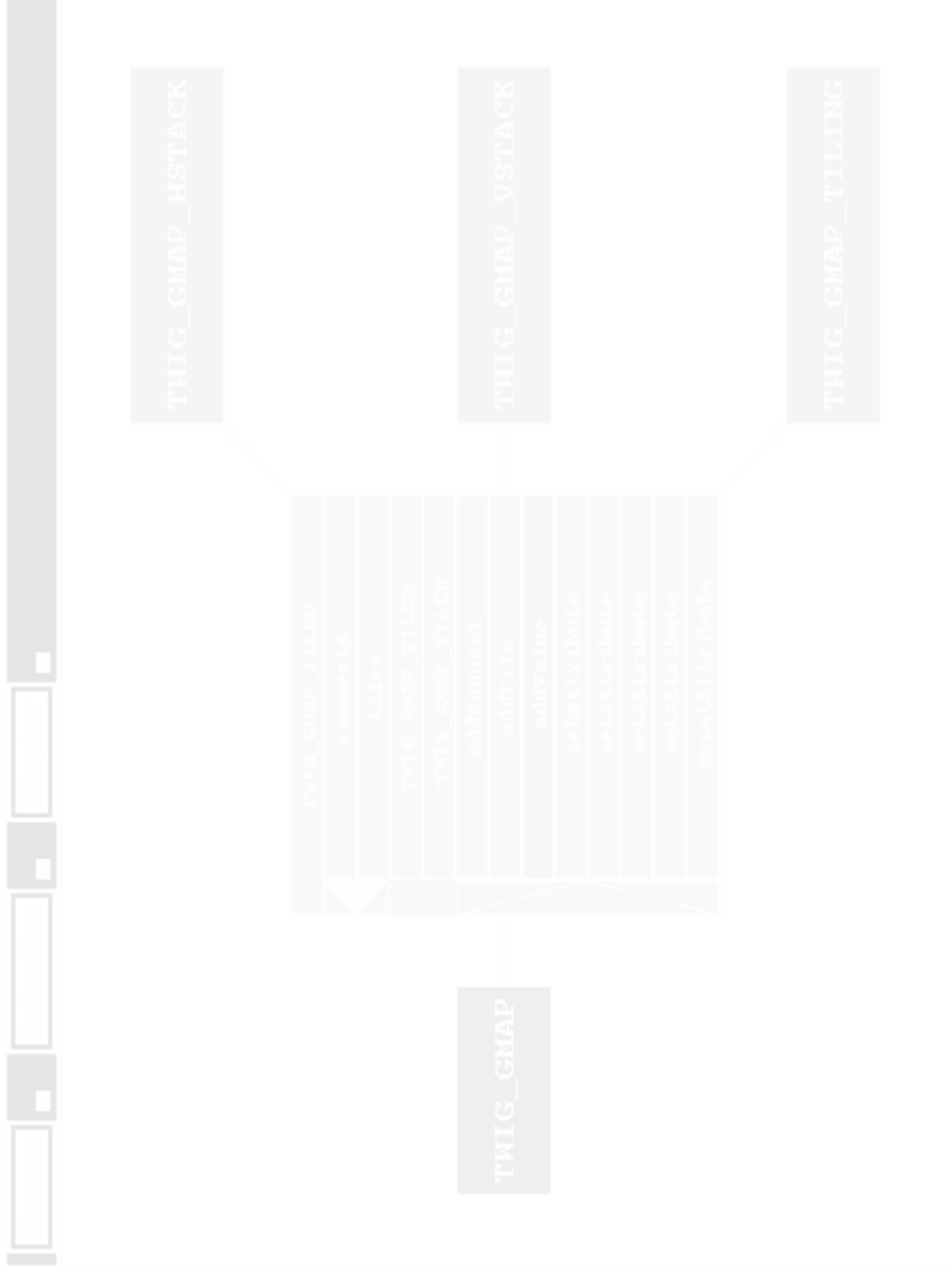
- Class relationships
- Member information
- Member inheritance & props
- Information window & dialogs

 Tied to rest of system thru MSG

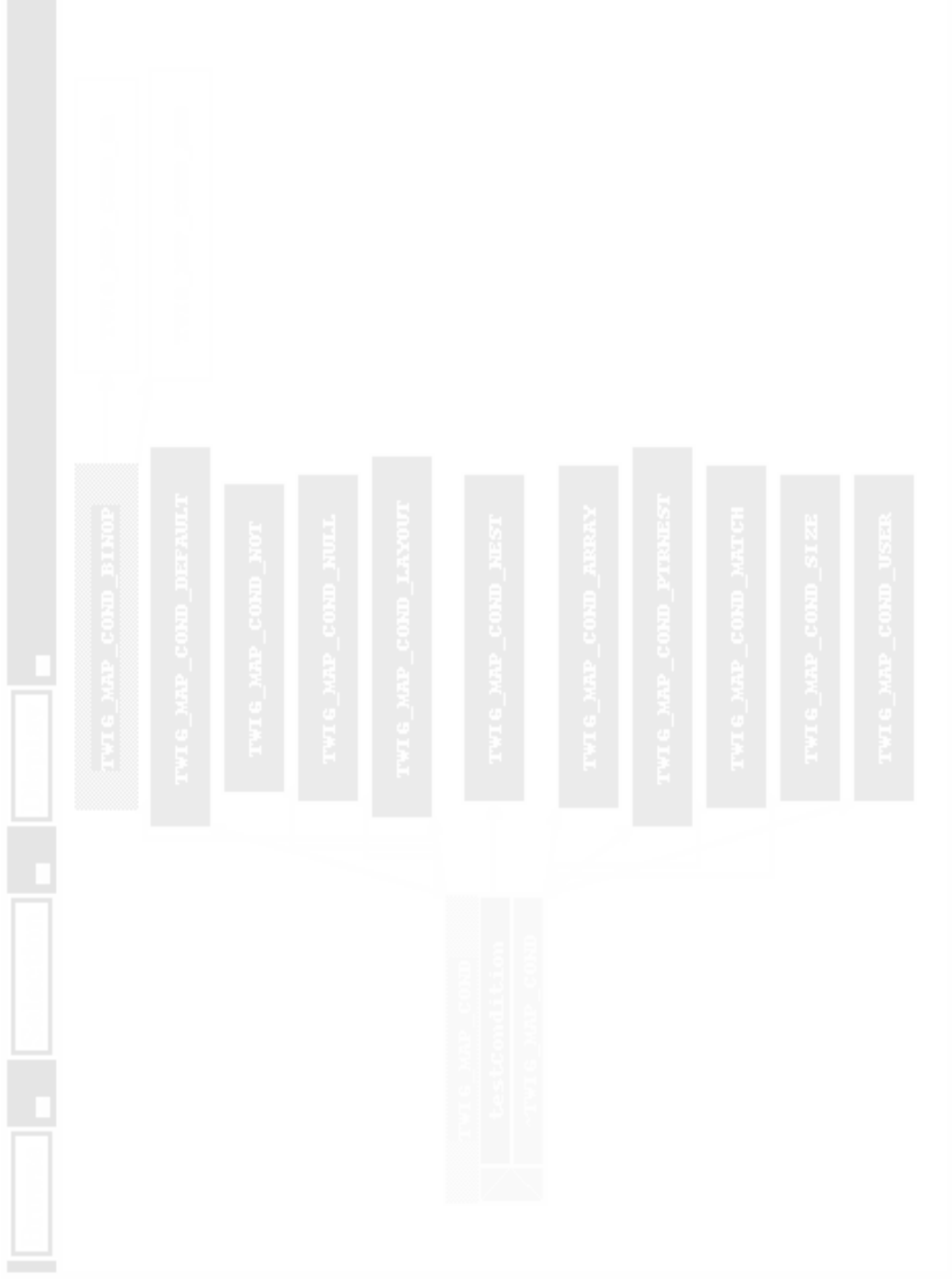
CBROWSE Example



CBROWSE Example



CBROWSE Example



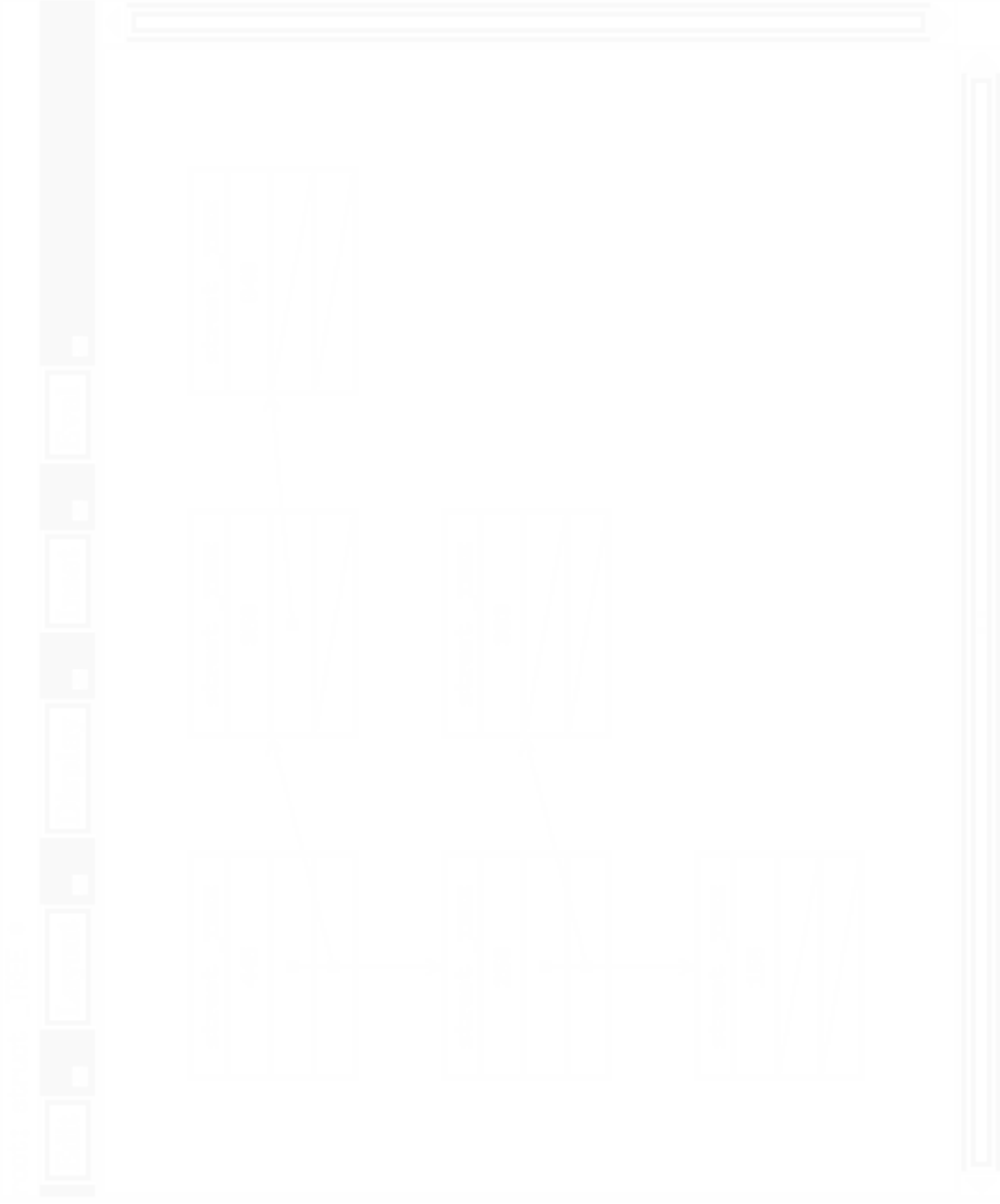
Data Structure Display

-  Graphical Display of arbitrary data
 - **Dynamic updating**
 - **User definable displays**

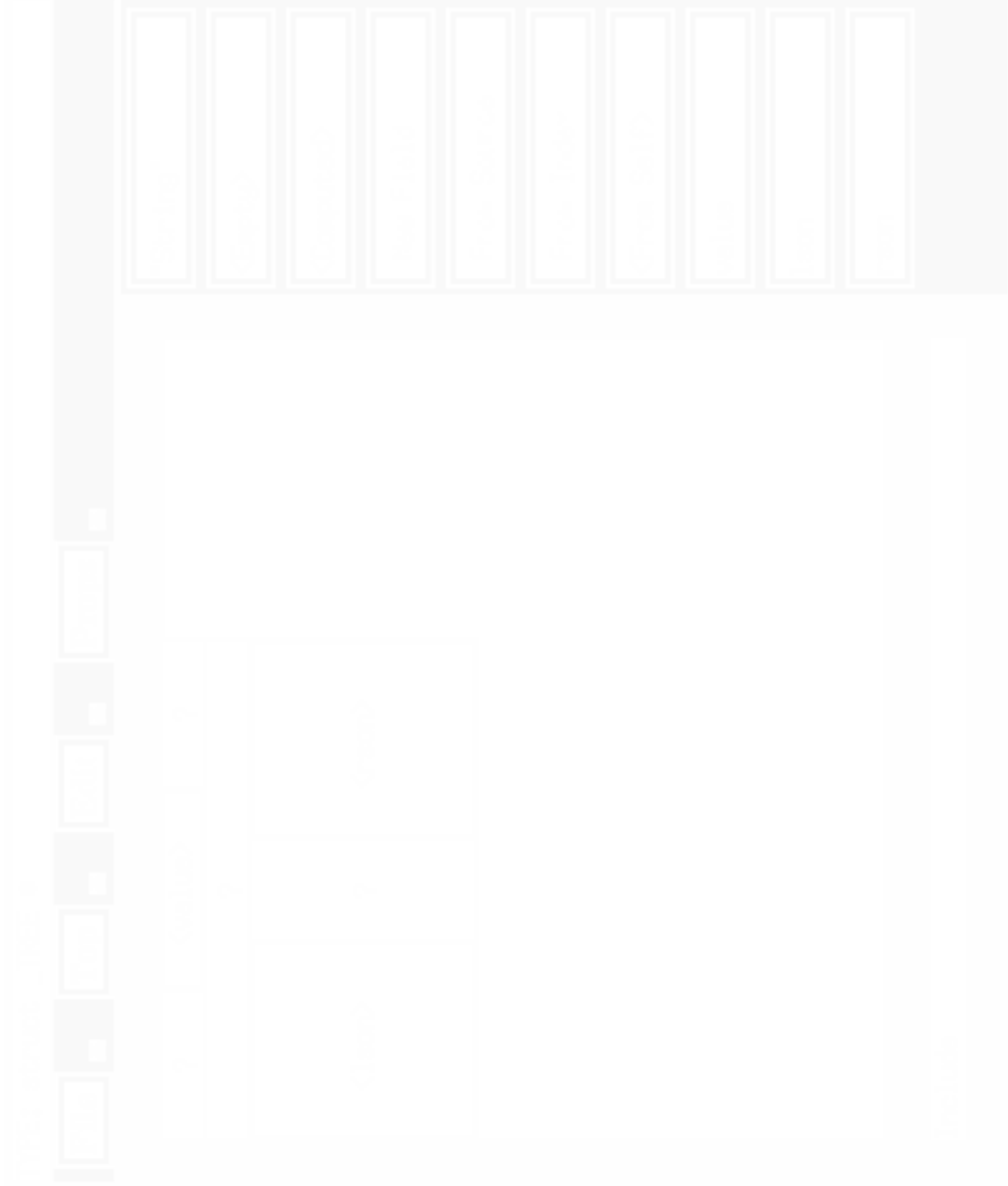
-  **DISP -- interface to GEL**
 - **Boxes, Tilings, Layouts**
 - **MSG interface to DDT**

-  **TYPEEDIT -- interface to APPLE**

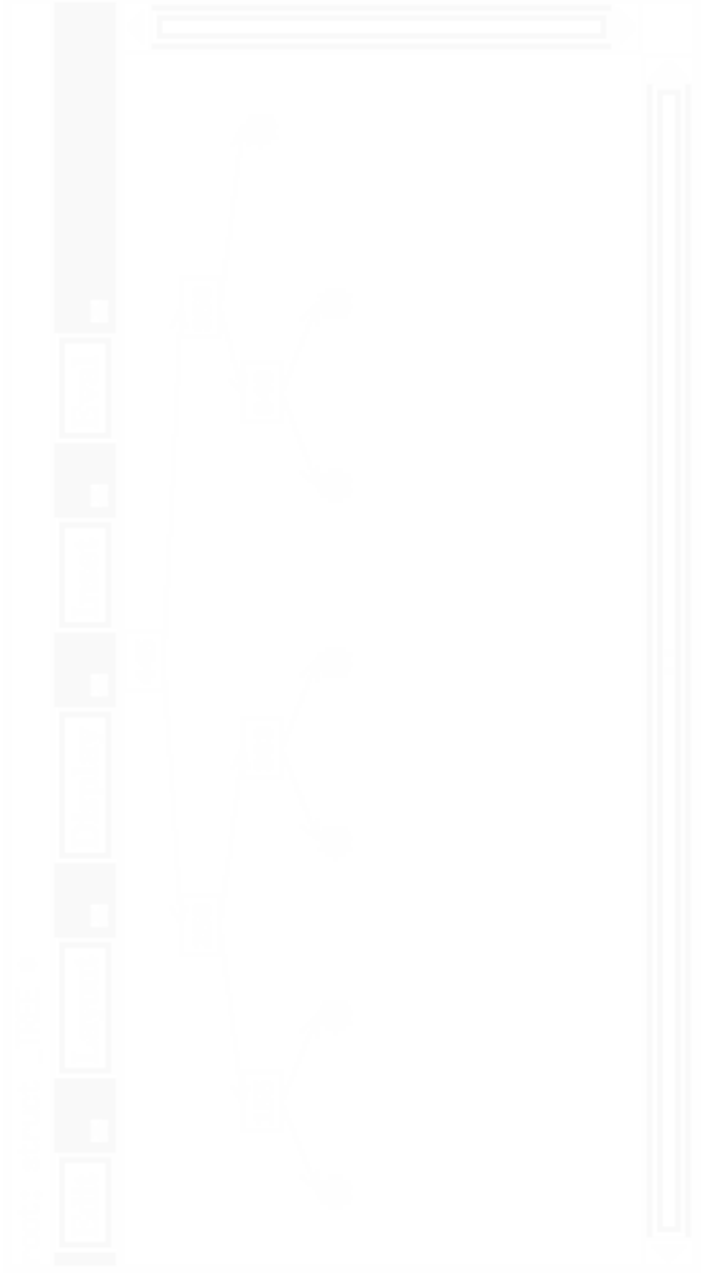
DISP Example




TYPEEDIT Example



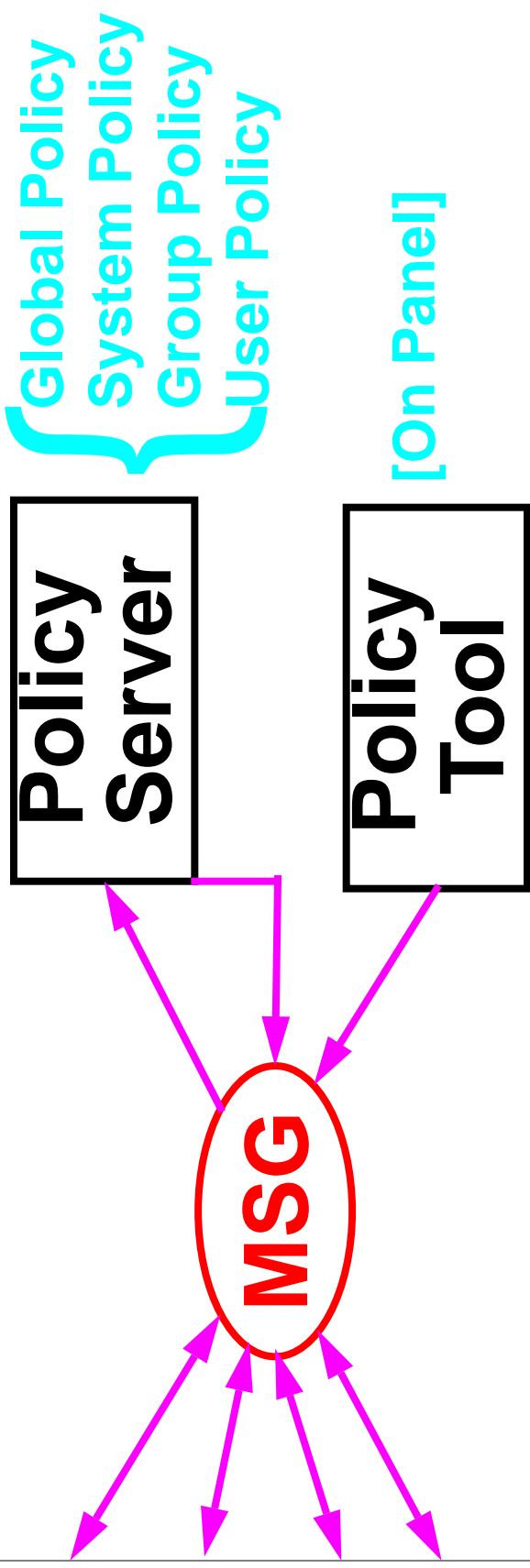
DISP Example



DISP Example



FIELD Policy Tool



 **Policy Programs are used**

- **FOREST: Garlan, Ilias**
- **Mediators: Sullivan, Notkin**

Policy Programming

Declarations

- Levels, users, tools, patterns, variables, load requests

Simple actions

- **WHEN** pattern **IF** cond **DO** action
- Done on a tool Basis

Actions -- conditional; required

- Send, Setenv, Set, Call, Null

Policy Programming

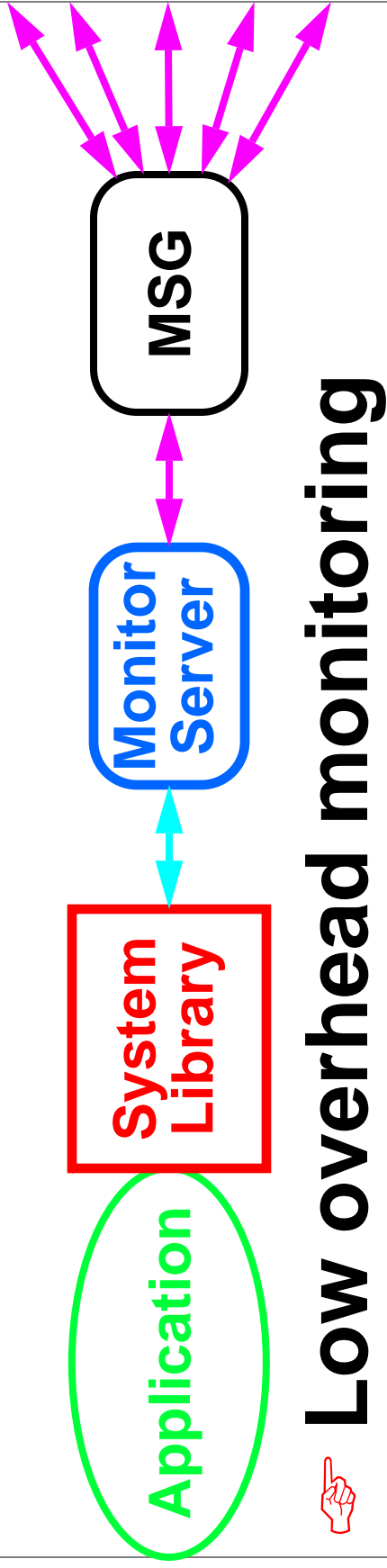
-  **Complex Actions**
- **Sophisticated policy programs**
- **Based on path expressions**

 **FOR patterns [WITH locals] DO**
WHEN path_expr : actions
...
END

 **Multiple actions can occur at once**

Program Monitoring

➡ **Modified System (Shared) Library**

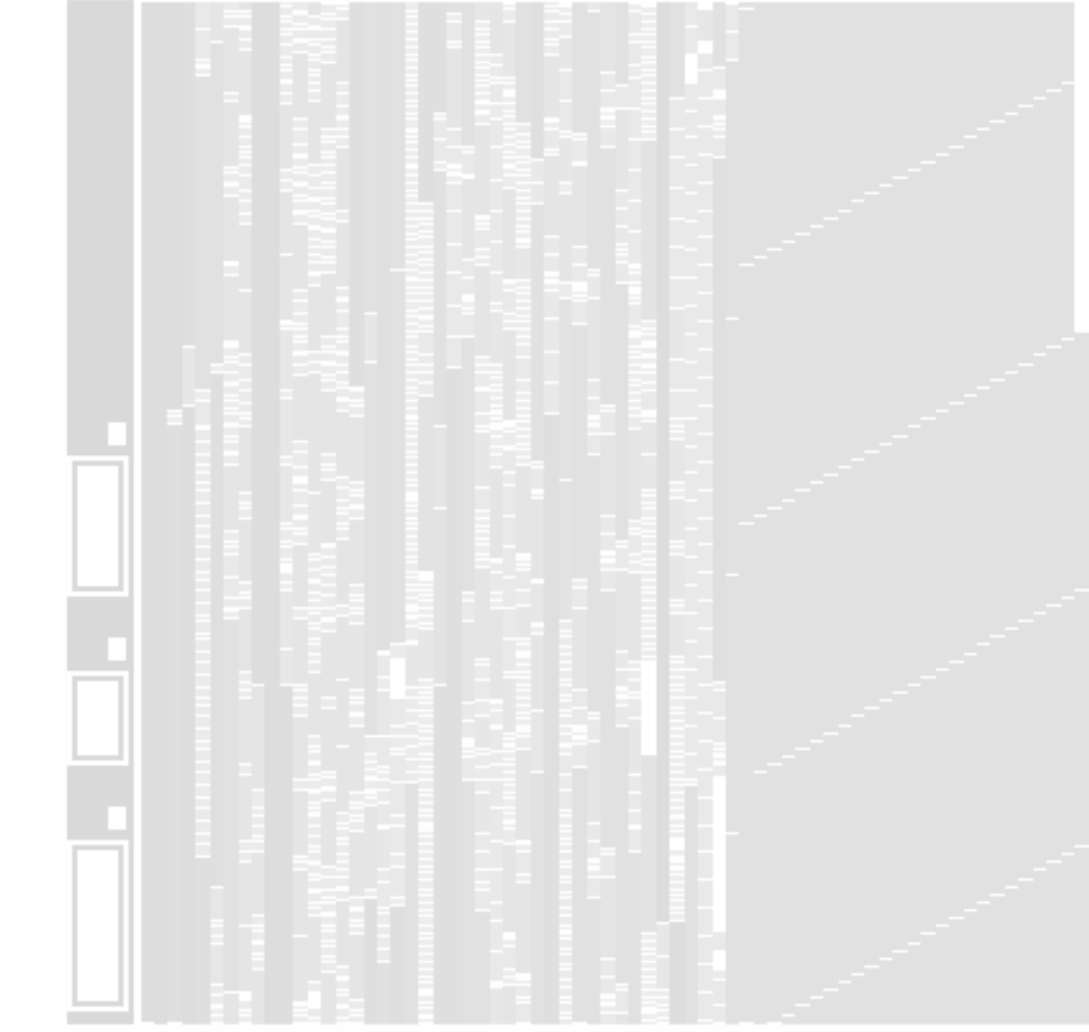


➡ **Low overhead monitoring**

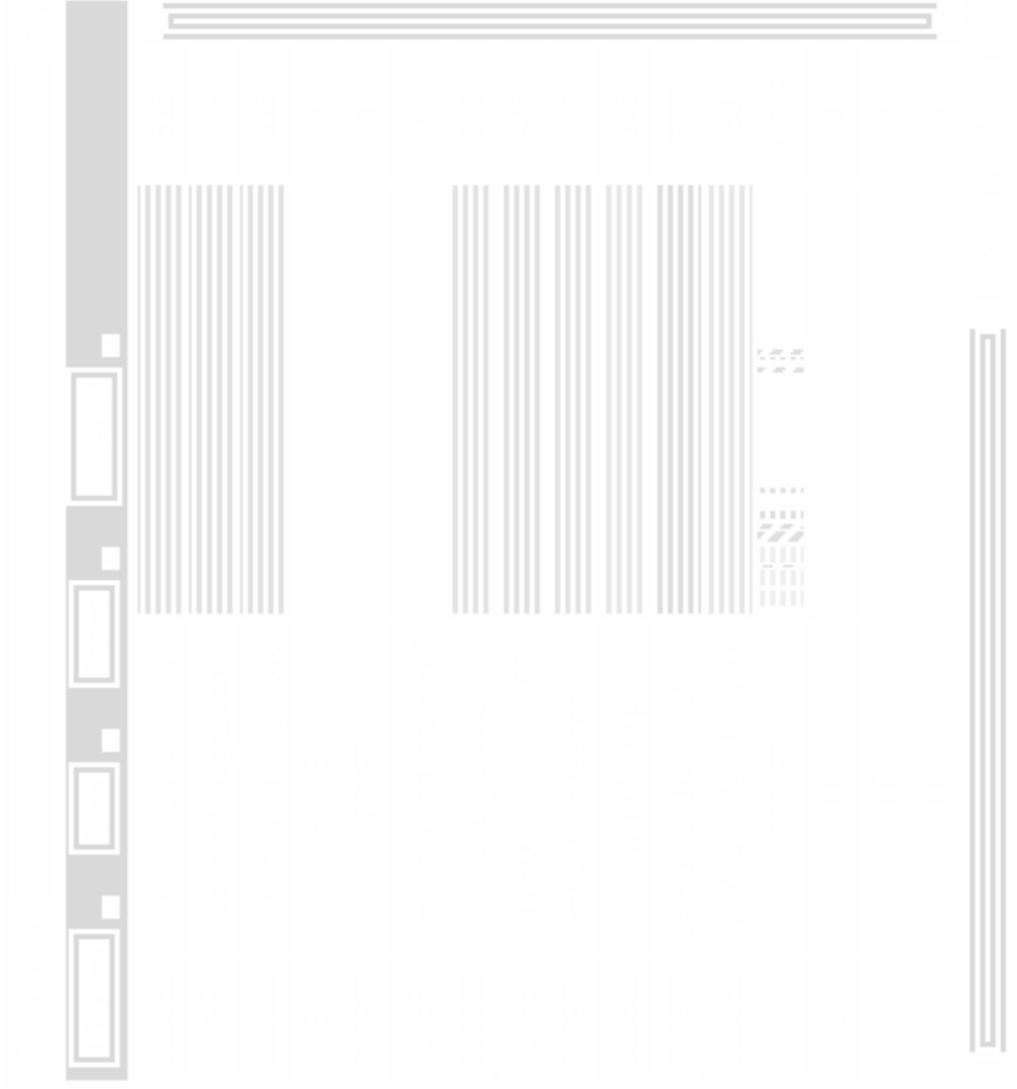
➡ **Monitor**

- **Memory usage (malloc, free)**
- **File I/O (open, close, read, write, ...)**
- **Performance (getrusage)**

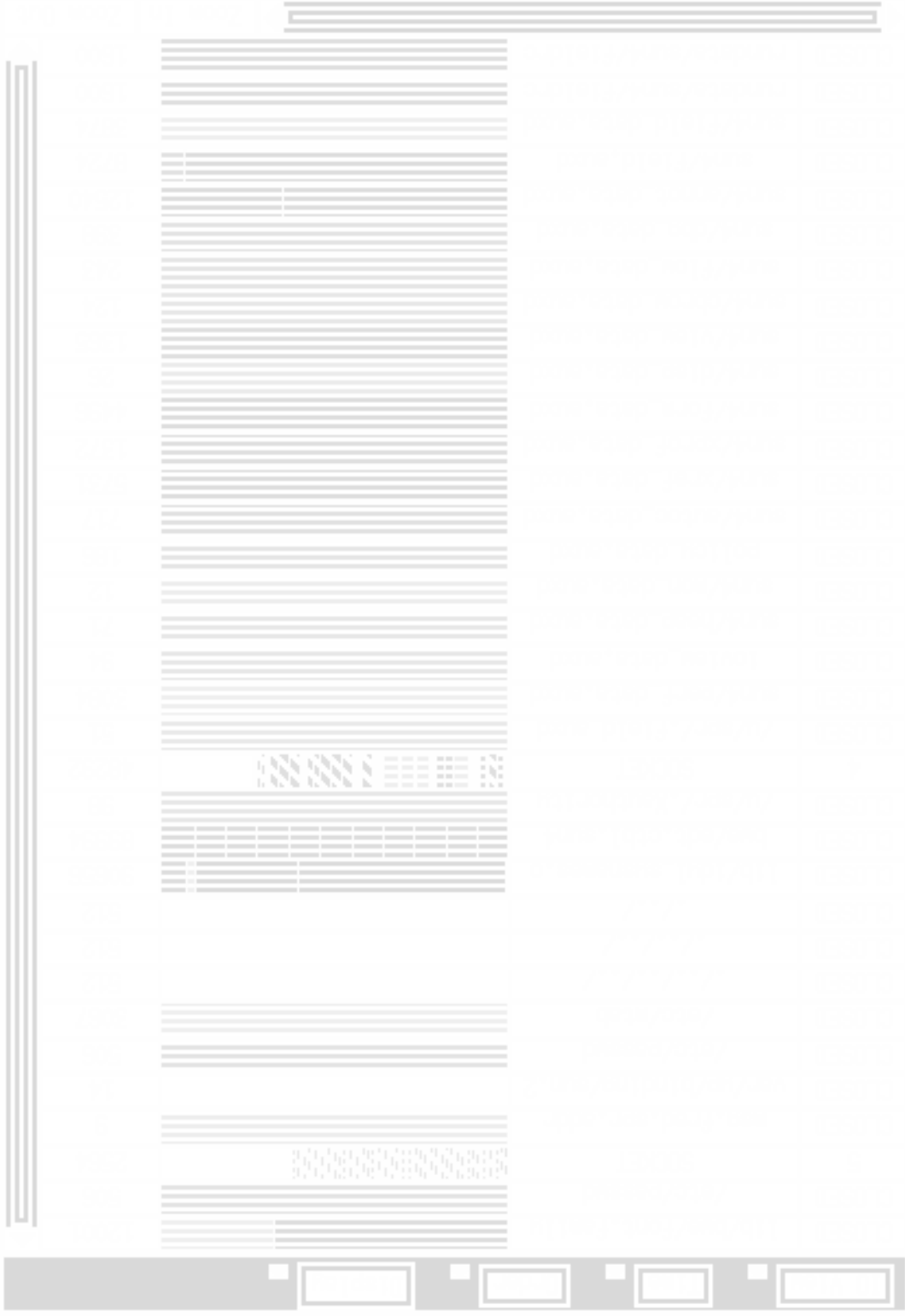
HEAPVIEW Example



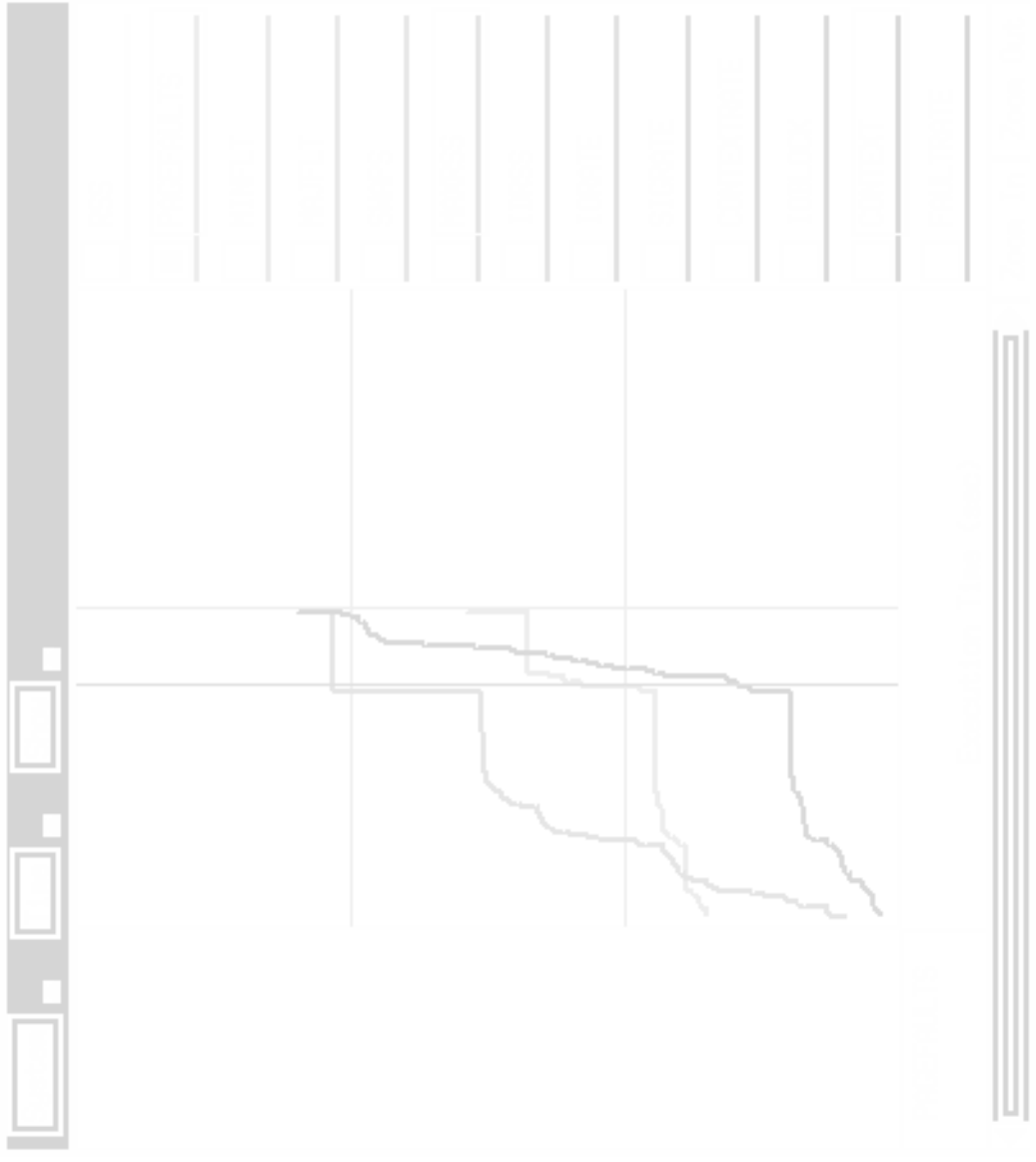
IOVIEW Example



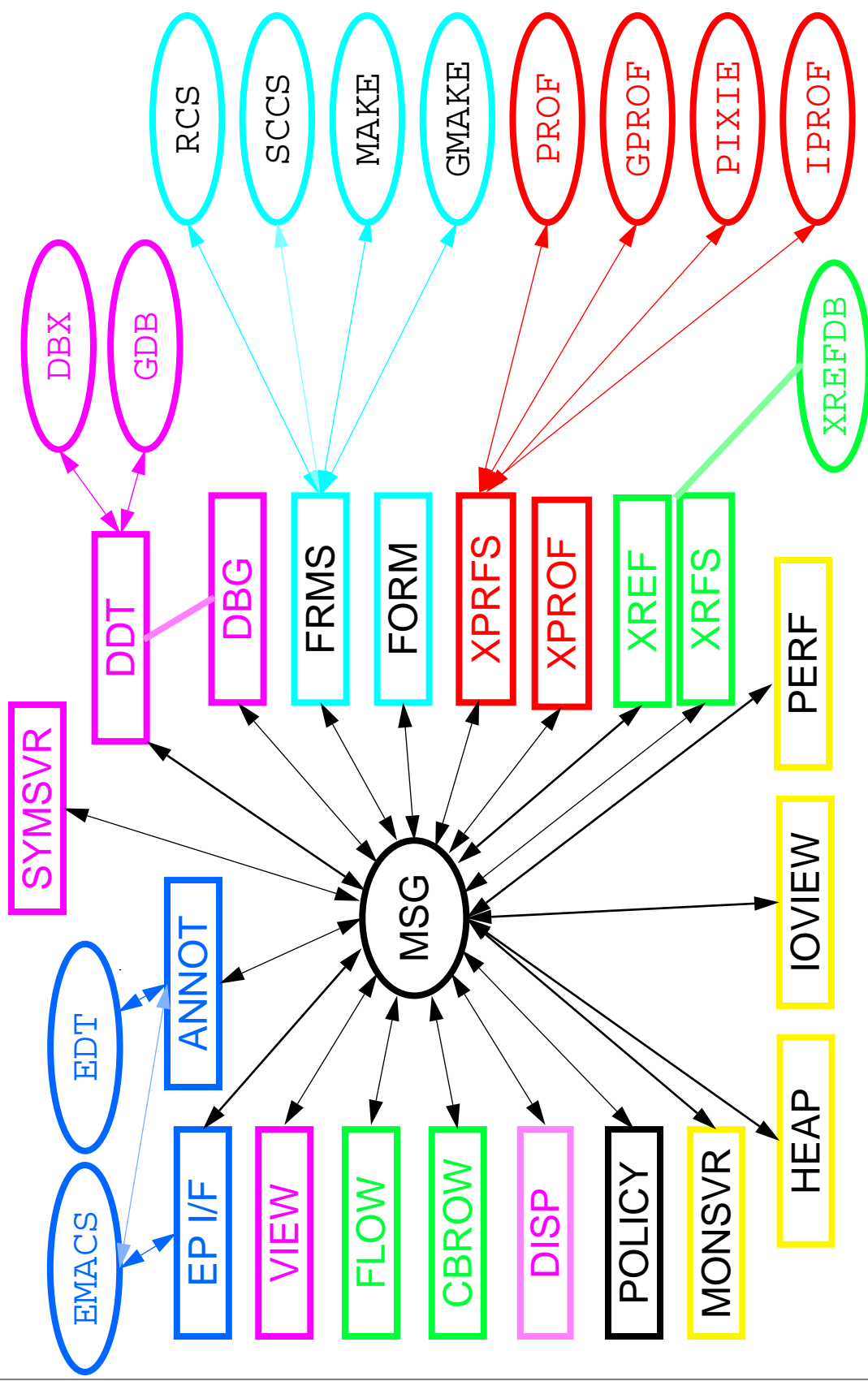
IOVIEW Example



PERFVIEW Example



FIELD Architecture



MEADOW

- 👉 **FIELD is an adaptable system**
 - **Resource files used extensively**
 - **Ability to dynamically load code**

- 👉 **MEADOW is the student version**
 - **Simplified menus**
 - **Single file Pascal programs**
 - **Help and documentation**

MEADOW Example

The screenshot displays the MEADOW IDE interface, which is divided into several panes:

- Code Editor:** Contains Pascal-like code for a bouncing ball simulation. The code includes procedures for drawing segments, updating the ball's position, and erasing the tail.
- Flowchart:** A control flow diagram showing the execution sequence of the code, with boxes representing procedure calls and decision points.
- Game Window:** A window titled 'bounce.p' showing a ball with a trail of segments on a black background. The trail consists of several white circles connected by lines, following a parabolic path.
- IDE Controls:** A top toolbar with buttons for 'File', 'Edit', 'Compile', 'Run', 'Stop', 'ClearStops', 'Print It', and 'Open Insert'.

```
<EJECT>
procedure DrawSegment (b: ball;
  seg: integer;
  playPen: Rect);
begin
  with b do
    with coords[seg] do
      begin
        SetPattern(pattern);
        x := right - radius
        else if x = radius < left then
          x := left + radius;
        if y + radius > bottom then
          y := bottom - radius
        else if y = radius < top then
          y := top + radius;
        Paint_Circle(x, y, radius);
      end;
    end;
end;

procedure EraseTail (b: ball; lastTail: integer; playPen: Rect);
var
  tail: integer;
begin
  tail := b.head + 1;
  if tail > lastTail then
    tail := 1;
  if b.coords[tail].x <> NoTail then
    DrawSegment(b, tail, playPen);
end;

<EJECT>
procedure UpdateBall (var b: ball;
  lastTail, gravity: integer;
  playPen: Rect);
begin
  with b do
    with playPen do
      with coords[head] do
        begin
          dx := -dx;
          if (x + radius > right) or (x - radius < left) then
            dx := -dx;
          if (y + radius > bottom) or (y - radius < top) then
            dy := -dy;
          else
            dy := dy + gravity;
          if dy > MAX_SPEED then dy := MAX_SPEED
          else if dy < -MAX_SPEED then dy := -MAX_SPEED;
          head := head + 1;
          if head > lastTail then
            EraseTail(b, lastTail, playPen);
          UpdateBall(b, lastTail, gravity, playPen);
          DrawSegment(b, head, playPen);
        end;
      end;
    end;
  end;
end;

begin
  with b do
    with coords[1] do
      begin
        x := right - radius;
        y := top + radius;
        dx := 0;
        dy := 0;
        head := 1;
        lastTail := 0;
      end;
    end;
  end;
  UpdateBall(b, lastTail, gravity, playPen);
end;
```

Status of FIELD

 Used extensively at Brown

- For research (C++ debugging)
- For instruction

 Commercialized

- DEC/FUSE
- Basis for Softbench, Tooltalk, ...

 Being distributed from Brown

Current Research

- 👉 **Better programming tools**
 - **Intelligent debuggers**
 - **Intelligent documentation**
 - **Configuration management**

👉 **Abstraction visualization**

👉 **Design-level programming**

Abstractions

- 👉 **Model how system is seen**
- 👉 **Provide basis for visualization**
- 👉 **Four basic types:**
 - **Syntactic**
 - **Semantic**
 - **Event-based**
 - **Data-based**

Syntactic Abstractions

 **Based on program syntax**

 **Examples:**

- **Call graph**
- **Class hierarchy display**
- **Design language**

Semantic Abstractions

- 👉 **Based on program analysis**
 - **More complex than syntactic**

- 👉 **Examples:**
 - **Program slices**
 - **Data flow graphs**
 - **Library-based (e.g. threads)**
 - **Potential class groupings**

Event Abstraction

 **Based on dynamic events**

 **Examples:**

- **Dynamic call graph**
- **Synchronization viewer**
- **Algorithm animation**

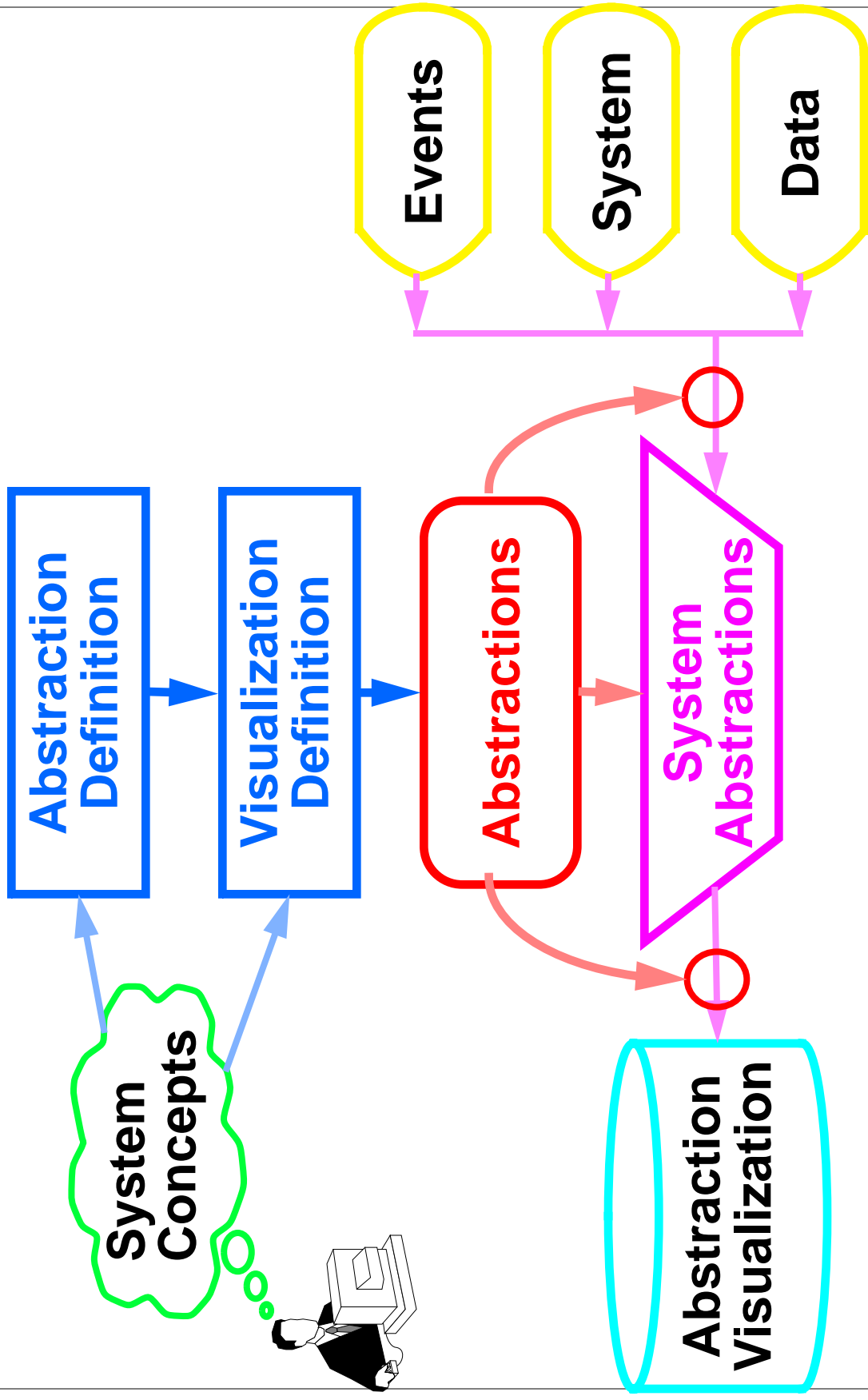
Data-based Abstractions

 **Based on program data**

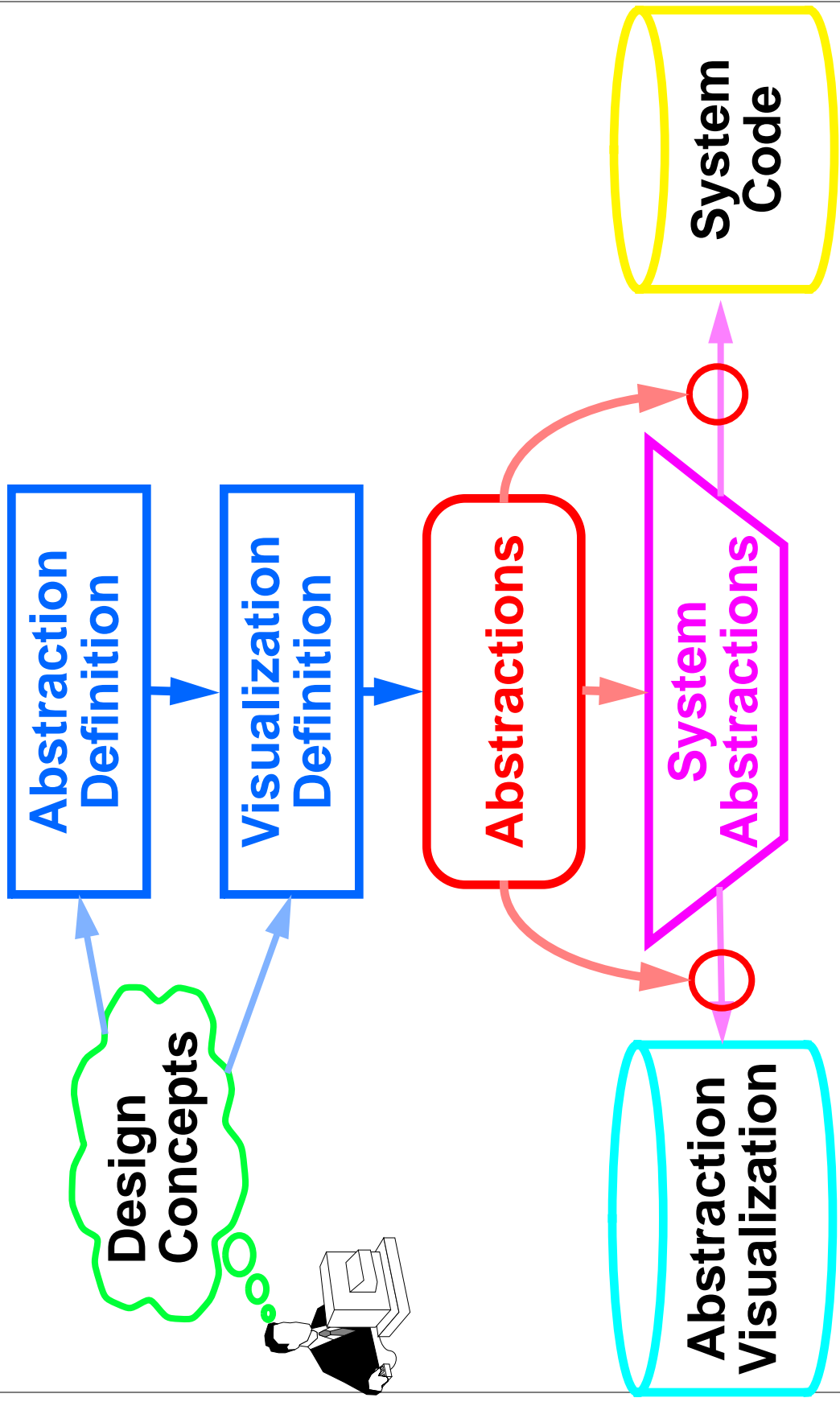
 **Examples:**

- **Data structure visualization**
- **Make dependency graph**

Abstraction Visualization



Abstraction Editing



Editing Abstractions

