FDBG, the CLP($\mathcal{FD}$) Debugger Library of SICStus Prolog

Péter Szeredi, Dávid Hanák, Tamás Szeredi
Budapest University of Technology and Economics
szeredi@cs.bme.hu, {dhanak,tszeredi}@inf.bme.hu

September 7, 2004

Contents

Introduction

CLP($\mathcal{FD}$) and Debugging

- CLP($\mathcal{FD}$) implementations:
  - Designated development environments (e.g. OPL Studio);
  - Embedded into a host language – Prolog is a logical choice (backtracking, logic variables).
- CLP($\mathcal{FD}$) and debugging in SICStus Prolog:
  - Extensive CLP($\mathcal{X}$) libraries, including CLP($\mathcal{FD}$);
  - An excellent, flexible, extensible debugger for Prolog;
  - minimal support for CLP($\mathcal{X}$) debugging (until FDBG).
- Possible approaches to observe a CLP($\mathcal{FD}$) run:
  - interactive tools (e.g. step-by-step debuggers);
  - assertion based methods;
  - *trace generation* and analysis – ideal for nonlinear program execution, like in the case of CLP($\mathcal{FD}$).

FDBG and its Event Trace

- FDBG = Finite domain DeBuGger
- Main purpose: enable CLP($\mathcal{FD}$) programmers to gather information about constraints and variables possibly even without modifying the observed program.
- FDBG translates the run of a CLP($\mathcal{FD}$) program into an event trace:
  - ⋆ a sequence of log entries;
  - ⋆ each entry corresponds to a *CLP($\mathcal{FD}$) event;*
  - ⋆ an event represents:
    - ∗ the activity of a *constraint* and its effect on variables;
    - ∗ a *labeling* decision while exploring the search tree.
  - ⋆ appearance of entries is fully customizable.

A Sample Session with FDBG

```
| ?-
```

A Sample Session with FDBG

```
| ?- use_module(library(clpfd)), use_module(library(fdbg)),
     fdbg_on.
```

A Sample Session with FDBG

```
| ?- use_module(library(clpfd)), use_module(library(fdbg)),
     fdbg_on.
% The clp(fd) debugger is switched on
yes

| ?-
```

A Sample Session with FDBG

```
| ?- use_module(library(clpfd)), use_module(library(fdbg)),
     fdbg_on.
% The clp(fd) debugger is switched on
yes

| ?- X #< 6, X #> 3, labeling([], [X]).
```

A Sample Session with FDBG

```
| ?- use_module(library(clpfd)), use_module(library(fdbg)),
     fdbg_on.
% The clp(fd) debugger is switched on
yes

| ?- X #< 6, X #> 3, labeling([], [X]).
<fdvar_1> in inf..5
    fdvar_1 = inf..sup -> inf..5
    Constraint exited.

<fdvar_1> in 4..sup
    fdvar_1 = inf..5 -> 4..5
    Constraint exited.
```

A Sample Session with FDBG

```
| ?- use_module(library(clpfd)), use_module(library(fdbg)),
     fdbg_on.
% The clp(fd) debugger is switched on
yes

| ?- X #< 6, X #> 3, labeling([], [X]).
<fdvar_1> in inf..5
    fdvar_1 = inf..sup -> inf..5
    Constraint exited.

<fdvar_1> in 4..sup
    fdvar_1 = inf..5 -> 4..5
    Constraint exited.

Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Basic Concepts

CLP($\mathcal{FD}$) Events

- CLP($\mathcal{FD}$) problem solving consists of two repeated phases:
  - ⋆ narrowing a variable domain due to constraint propagation;
  - ⋆ narrowing a variable domain due to labeling.
- Observation: with *two classes of events* we can describe the behavior of a CLP($\mathcal{FD}$) program:
  - ⋆ constraint events
    - ∗ a constraint is woken up and performs propagation;
  - ⋆ labeling events
    - ∗ a choicepoint is created or exhausted (through failure);
    - ∗ the domain of a variable is narrowed.
- Events are intercepted and dispatched to *visualizers* by the FDBG core.

Visualizers

- Predicates responsible for handling CLP($\mathcal{FD}$) events;
- Usually display trace information;
- In general can do any kind of processing (like checking invariants);
- Analogously to event classes, there are two types:
  - ⋆ constraint visualizers;
  - ⋆ labeling visualizers.
- FDBG provides default built-in visualizers for both types;
- Utility predicates support writing custom visualizers.

The User Interface

An Example – The N-queens Problem

```
nqueens(N, Queens) :-
        bb_put(board_size, N),
        length(Queens, N),
        fdbg_assign_name(Queens, queen),
        domain(Queens, 1, N),
        constrain_all(Queens),
%       asiymmetric(N, Queens),           % break symmetry
        labeling([ff], Queens).
...
no_threat(X, Y, I) :-
        fd_global(no_threat(X,Y,I), 1, [val(X),val(Y)]).

:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(no_threat(X,Y,I), S, S, Actions) :-
        (   integer(X) ->  no_threat_prop(Y, X, I, Actions)
        ;   integer(Y) ->  no_threat_prop(X, Y, I, Actions)
        ;                  Actions = []
        ).
...
```

Built-in Visualizers

- FDBG uses built-in visualizers by default;
- Built-in visualizers can work without any program modification.

One block of output of the constraint visualizer

```
no_threat(2,<queen_3>,2)
    queen_3: 1..2 -> {1}
    Constraint exited.
```

Built-in Visualizers

- FDBG uses built-in visualizers by default;
- Built-in visualizers can work without any program modification.

One block of output of the constraint visualizer

`no_threat(2,<queen_3>,2)`	The constraint itself;
`    queen_3: 1..2 -> {1}`	*Legend:* domain narrowings;
`    Constraint exited.`	constraint behavior.

Built-in Visualizers

- FDBG uses built-in visualizers by default;
- Built-in visualizers can work without any program modification.

One block of output of the constraint visualizer

```
no_threat(2,<queen_3>,2)
    queen_3: 1..2 -> {1}
    Constraint exited.
```

The constraint itself;
Legend: domain narrowings;
constraint behavior.

Details:

- The legend lists *all* the variables of the constraint;
- Most common behaviors are entailment (above) and failure.
- Variable names:
  - ⋆ all variables are assigned a name for clarity;
  - ⋆ needed because name in source is not preserved in Prolog;
  - ⋆ usually displayed between angle brackets (<queen_3>).

Built-in Visualizers – cont.

Output of the labeling visualizer

```
Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Built-in Visualizers – cont.

Output of the labeling visualizer

```
Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Details:

- Each event results in one line of output;

Built-in Visualizers – cont.

Output of the labeling visualizer

```
Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Details:

- Each event results in one line of output;
- The number in brackets (3 here) identifies the choicepoint;

Built-in Visualizers – cont.

Output of the labeling visualizer

```
Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Details:

- Each event results in one line of output;
- The number in brackets (3 here) identifies the choicepoint;
- Name after number is the variable being labeled in choicepoint;

Built-in Visualizers – cont.

Output of the labeling visualizer

```
Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Details:

- Each event results in one line of output;
- The number in brackets (3 here) identifies the choicepoint;
- Name after number is the variable being labeled in choicepoint;
- This is followed by a specification of the event:
  - choicepoint creation (`start`);

Built-in Visualizers – cont.

Output of the labeling visualizer

```
Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Details:

- Each event results in one line of output;
- The number in brackets (3 here) identifies the choicepoint;
- Name after number is the variable being labeled in choicepoint;
- This is followed by a specification of the event:
  - choicepoint creation (`start`);
  - labeling choice (here `step`);

Built-in Visualizers – cont.

Output of the labeling visualizer

```
Labeling [3, <fdvar_1>]: starting in range 4..5.
Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4
X = 4 ? ;
Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5
X = 5 ? ;
Labeling [3, <fdvar_1>]: failed.
```

Details:

- Each event results in one line of output;
- The number in brackets (3 here) identifies the choicepoint;
- Name after number is the variable being labeled in choicepoint;
- This is followed by a specification of the event:
  - choicepoint creation (`start`);
  - labeling choice (here `step`);
  - failure.

Variable Naming

- Variables are assigned a name:
  - manually by calling `fdbg_assign_name/2`;
  - automatically when calling `fdbg_annotate/2,3`;
  - auto-assigned names look like `fdvar_`N (N unique counter).
- Names are primarily used to refer to variables in the trace;
- This done via *annotation:* each variable in a term is replaced by a term containing its name, itself, and its narrowed domain;
- Convenience service: assign names to an entire term and each variable in it with a single call:

term/variable	selector	name	
`bar(A, [B, C])`	`[]`	`foo`	assigned name

Variable Naming

- Variables are assigned a name:
  - ⋆ manually by calling `fdbg_assign_name/2`;
  - ⋆ automatically when calling `fdbg_annotate/2,3`;
  - ⋆ auto-assigned names look like `fdvar_`N (N unique counter).
- Names are primarily used to refer to variables in the trace;
- This done via *annotation:* each variable in a term is replaced by a term containing its name, itself, and its narrowed domain;
- Convenience service: assign names to an entire term and each variable in it with a single call:

term/variable	selector	name	
`bar(A, [B, C])`	`[]`	`foo`	assigned name
`A`	`[1]`	`foo_1`	implicit derived names
`B`	`[2,#1]`	`foo_2_1`	
`C`	`[2,#2]`	`foo_2_2`	

Customizing Visualizers

- Built-in visualizers have no knowledge of the problem structure;
- Customized visualization can exploit this additional knowledge;
- Customization is possible on two levels:
  - ⋆ slight modification of the output of the built-in visualizers by defining hook predicates;
  - ⋆ writing custom visualizers.
- Use a hook predicate to modify the legend of *N-queens*:
  - ⋆
    ```
    no_threat(2,<queen_3>,2)
         queen_3: [ X - . . ]
         Constraint exited.
    ```
- Or a custom visualizer to completely redefine it:
  - ⋆
    ```
    no_threat(4,<queen_3>,1)
         [ X X . . ]
         [ . . . X ]
         [ X X - - ]
         [ X X X X ]
    ```

Legend Portray Hook

```
fdbg:legend_portray(Name, Var, After) :-
        write(Name), write(': '),
        print_row(Var, After).

print_row(Var, After) :-
        bb_get(board_size, N),
        fd_set(Var, Now),
        write('['), print_fields(1, N, Now, After), write(']').

print_fields(I, N, _, _) :-
        I > N, !, write(' ').
print_fields(I, N, Now, After) :-
        write(' '),
        (   fdset_member(I, After) -> write('X') % allowed
        ;   fdset_member(I, Now)   -> write('-') % being pruned
        ;                             write('.') % pruned
        ),
        I1 is I+1,
        print_fields(I1, N, Now, After).
```

Custom Visualizer

```
nqueens_show(Constraint, Actions) :-
        fdbg_current_name(Queens, queen),
        fdbg_annotate(Constraint, AConst, _),
        fdbg_annotate(Queens, Actions, AQueens, _),
        print(AConst), nl,
        print_board(AQueens).

print_board([]) :- nl.
print_board([fdvar(_,Var,After)|Qs]) :- !,
        write('    '),
        print_row(Var, After), nl,
        print_board(Qs).
print_board([V|Qs]) :-
        write('    '),
        fdset_singleton(Set, V),
        print_row(V, Set), nl,
        print_board(Qs).
```

Implementation Issues

Event detection

- SICStus debugger provides *advice points:* programmable breakpoints;
- FDBG places advice points on all constraint handling predicates;
- Limitation: only *global constraints* are handled, *indexicals* are ignored;
- Workaround: when FDBG is turned on, constraints otherwise compiled as indexicals translate into global constraints (through *goal expansion*);
- Consequences:
  - ⋆ FDBG should be consulted before the program to be traced;
  - ⋆ Negligible effect on performace compared to overhead of FDBG in general;
  - ⋆ Minor behavioral changes (slightly different propagation);
  - ⋆ Original form of constraints is lost in the process.

Variable Naming Revisited

Reminder

- Names can be assigned by the user to any term or variable;
- Visualizers refer to variables with names exploiting *annotation;*
- *Annotation* is the process of replacing variables in a term with descriptive compound terms.

Variable Naming Revisited

Reminder

- Names can be assigned by the user to any term or variable;
- Visualizers refer to variables with names exploiting *annotation;*
- *Annotation* is the process of replacing variables in a term with descriptive compound terms.

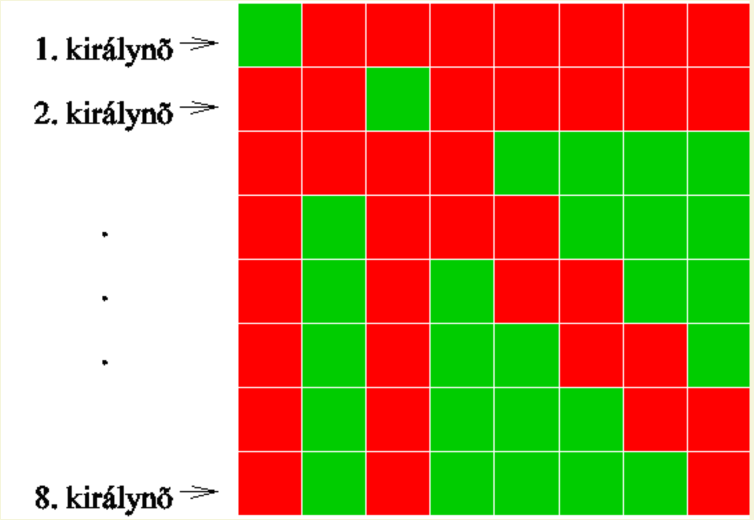
Implementation

- Names are stored in an AVL tree in the *global private* field;
- As a result, the name store is *volatile* (fresh for each query);
- Consequences:
  - ⋆ No need to clear the name store after each query (good);
  - ⋆ Need to assign names in each query (seemingly inconvenient but unavoidable: different variables!);
  - ⋆ The best place to do this is within the program itself.

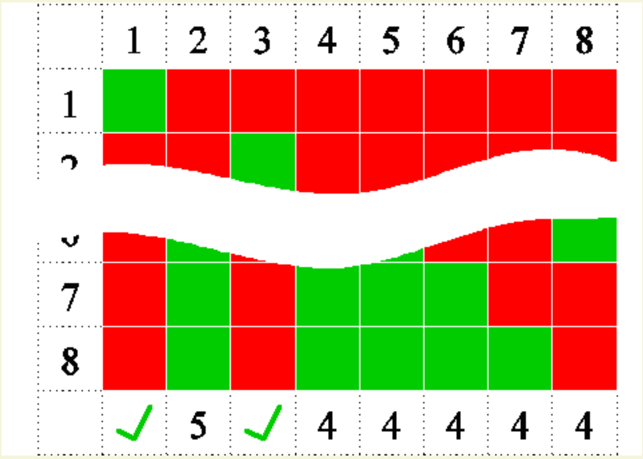
Future Work and Conclusions

Future Work

- Indexical to global constraint redirection fails to preserve original form of constraints – need to find an acceptible solution;
- When labeling fails, there is no information in the log about what state the domains of variables are *restored* to through backtracking;
- A generic, configurable graphical visualizer – plans have already been proposed, still need to implement it.



Basic 2-dimensional visualization



Embellished version

Conclusions

- Presented basic trasing scheme for CLP($\mathcal{FD}$) programs written in SICStus Prolog;
- Introduced events and visualizers;
- Showed how variable naming and output customization can help to clarify the trace log;
- Given examples to visualizer customization;
- Covered a few implementational details;
- Scetched possible directions of future development.

Conclusions

- Presented basic trasing scheme for CLP($\mathcal{FD}$) programs written in SICStus Prolog;
- Introduced events and visualizers;
- Showed how variable naming and output customization can help to clarify the trace log;
- Given examples to visualizer customization;
- Covered a few implementational details;
- Scetched possible directions of future development.

Thanks for your attention!