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

Dávid Hanák, Tamás Szeredi, and Péter Szeredi

Budapest University of Technology and Economics
{dhanak,tszeredi}@inf.bme.hu, szeredi@cs.bme.hu

1 Introduction

Debugging tools serve an important role in software development. This also holds for constraint programming and CLP($\mathcal{FD}$) in particular, where it is often necessary to observe how the domains of various variables behave, how different constraints linked by shared variables affect each other, etc. There are numerous projects for implementing debuggers for CLP($\mathcal{FD}$) systems. Some have committed themselves to interactive tools, others have chosen assertion based methods, and a large number of publications deal with trace generation and analysis.

We have decided to develop a trace based debugger for the CLP($\mathcal{FD}$) library of SICStus Prolog, a library which neatly embeds the theory of finite domain constraints into Prolog. The SICStus environment has an advanced and extensible debugger for the base language, but until recently it has lacked direct support to observe the run of constraint programs. The goal of FDBG (which is short for Finite domain DeBuGger) is to fill in this gap.

FDBG was written almost entirely in Prolog, as user space code, no native routines were used directly. The library (along with its source code) is part of the SICStus Prolog distribution versions 3.9 and upwards, and is documented in detail in the SICStus User's Manual.

2 Debugger Services

FDBG consists of two loosely coupled parts. The core is responsible for making the run of a CLP($\mathcal{FD}$) program observable by translating it into a sequence of *events*. The outer layer consists of a collection of configurable and extensible *visualizers* and utility predicates which process and display the events according to the needs of the user. The two are linked together through a simple interface.

CLP($\mathcal{FD}$) Events. By observing the process of CLP($\mathcal{FD}$) problem solving we can conclude that events can belong to two classes. The events of the first class called *constraint events* occur when a constraint does some propagation on the domains of its variables. Events belonging to the second class are *labeling events,*

* The subject is presented in full detail at the WLPE'04 workshop.

representing decisions made during the exploration of the search space. Events of these two classes appear interleaved in a trace log, as labeling triggers additional propagation, after which labeling is resumed, etc.

Every event intercepted by the FDBG core is described with a Prolog term and then dispatched to the appropriate visualizers. Most visualizers, such as the default ones provided by FDBG, display the event in the trace log. Consequently, a log usually contains a block of lines for each event. However, in general a visualizer can do any kind of processing or analysis.

Basic Services. To make FDBG produce a verbose text log of the trace events using its built-in visualizers, a CLP($\mathcal{FD}$) program needs no modification. All the user needs to do is to turn on the debugger before invoking the main program.

For constraint events, the default log entry consists of the name and actual arguments of the constraint, and the list of variables narrowed in that particular step, showing their domains *before* and *after* the propagation took place. Variables are identified by their names, which are assigned either implicitly by the debugger core, or explicitly by the user. A sample of such a trace with explicitly assigned names can be seen on Fig 1.

For labeling events, an entry contains the name of the variable being labeled, and describes the way its domain is divided. This can either be the selection of a single value or narrowing to a subset of its domain. Alternative choices branching off the same choicepoint can be recognized as such by a unique identifier included in the log.

Advanced Features. A useful service of FDBG is the *naming of variables* and terms in general. An advantage of naming is that built-in visualizers will use the specified name to refer to variables wherever they appear in the log. The user can also easily identify these variables anywhere from the program by using the *annotation* service, which replaces variables with descriptive Prolog terms.

If the user is unsatisfied with the output of the default visualizers or finds it hard to understand, he has the opportunity to customize them, or to write his own visualizers. FDBG provides an easy way to switch between the built-in and custom visualizers, and they may also be used simultaneously. An example custom log can be seen on Fig. 2.

Writing visualizers also provides a simple means to filter trace events, or to silently wait for the occurrence of an event and start the Prolog debugger at that point.

The figures below show two snippets from two traces of the 4-queen problem. The first one was printed by the built-in visualizer, while the second one, showing the entire checkerboard, was created by a custom visualizer.

```
no_threat(4,<queen_3>,1)
    queen_3: {1}\/{3} -> {1}
    Constraint exited.
```

Fig. 1. Basic log entry

```
no_threat(4,<queen_3>,1)
    [ . X . . ]
    [ . . . X ]
    [ X . - . ]
    [ X . X X ]
```

Fig. 2. Custom log entry