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

Dávid Hanák and Tamás Szeredi

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

Abstract. FDBG (short for Finite domain DeBuGger) is a tool addressing the problems of debugging finite domain (FD) constraint programs. Implemented as a SICStus Prolog library, it allows CLP($\mathcal{FD}$) programmers to trace FD variable domain changes. FDBG is capable of detecting the wake-up of constraints as well as the steps of labeling, and it reports the effects of these events on the variables. The simplest way to use FDBG is to produce a wallpaper trace of the run of the constraint program. The trace is analyzed (typically *post-mortem*) either manually or by a tool designed for this purpose. However, program behavior may also be observed in real time exploiting the modular design and the flexible output redirecting capabilities. FDBG was written almost entirely in Prolog as user space code, no native routines were used directly. It is also included (with source) in the official SICStus distribution in versions 3.9 and upwards.

1 Introduction

When writing CLP($\mathcal{FD}$) programs or global constraints in particular, 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 aimed at developing tools to debug or trace applications based on finite domain constraints, such as Grace [1] or the DiSCiPl project [2]. Some projects committed themselves to interactive tools, like the Constraint Investigator for Oz [3], others have chosen an assertion-based approach [4,5], and a large number of publications deal with trace generation and analysis [6,7,8].

A number of CLP($\mathcal{X}$) systems are standalone environments developed specially for writing CLP($\mathcal{X}$) programs. These often have a built-in debugger tool, such as the OPL Studio Visualizer [9]. Others are embedded into a host system or language, like the SICStus Prolog CLP($\mathcal{X}$) libraries [10]. These environments are usually also equipped with a debugger of the base system, but it is not at all straightforward to acquire the necessary trace information on constraints by means of these debuggers. In case of SICStus, constraints are implemented very much like a group of coroutines, running in the background, waking up whenever it is necessary. Therefore single stepping a program is inadequate, because one does not see what happens in the background. In order to observe that, one has no choice but to place breakpoints on the coroutines, which is quite tedious.

The main purpose of writing FDBG (which stands for Finite domain DeBuGger) was to enable CLP($\mathcal{FD}$) programmers to gather all information about constraints and variables without even modifying the program itself, let alone writing predicates specifically for debugging purposes. For regular, sequentially running Prolog programs, a traditional step-by-step debugger is an adequate tool. But because CLP($\mathcal{X}$) programs run nondeterministically, we have chosen to implement a debugger producing wallpaper trace instead. This decision is justified by the large number of similar attempts we have already referred to.

The trace output is a sequence of log entries. Each entry corresponds to a CLP($\mathcal{FD}$) event. One group of events represent the wake-up and activity of constraints. Such events describe which constraint is active currently and how it narrows the domains of variables. Events of the other group inform us about the process of *labeling,* containing data on the structure of the search tree, showing its active and failed branches. The appearance of the log entries can be customized by the programmer who is provided with a set of tools to process the events. Filters may also be applied easily to reduce the size of the log.

Constraints which are kept in the store and woken up regularly to narrow domains are often referred to in the literature as *propagators.* Likewise, the labeling process is also called *distribution* of the search space. In this paper, however, we will follow the conventions of SICStus by using the terms constraints and labeling. It is important to note that FDBG works only for a subset of constraints called *global constraints*, but this seemingly big limitation is mostly overcome by a trick discussed in Section 4.1.

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.

We assume the reader is familiar with constraint programming and knowing details of the SICStus CLP($\{\lceil$) library can also be useful. For an introduction to CLP($\mathcal{X}$) using SICStus Prolog, refer to [11].

This paper is structured as follows. Section 2 introduces the notion of *events* and the related concept of *visualizers* which are essential in understanding the principles of FDBG. Section 3 gives a birds-eye view of the user interface and shows some examples of using the debugger. Section 4 covers the most interesting aspects of the implementation and illustrates how most of the services can be provided by user space Prolog code. Section 5 compares the services of FDBG with those of some other constraint debuggers and Sect. 6 shows some ideas about the directions of future development. Finally, Sect. 7 concludes the paper.

2 Events In CLP($\mathcal{FD}$) Execution

The process of finding a solution satisfying a given set of constraints consists of two repeated phases. In the first phase the CLP($\mathcal{FD}$) solver calls the appropriate constraint dispatchers as long as there are unprocessed variable domain changes. When there are no more constraints to wake, labeling is initiated (or resumed), which—by constraining the domain of a yet uninstantiated variable—reactivates the solver. This iteration of the two phases eventually leads to a solution.

From this observation we can draw the conclusion that there are *two classes of events* that control the behavior of a CLP($\mathcal{FD}$) program. The events of the first class are called *constraint events* in FDBG. These occur when a global constraint is woken up by the solver. The data available on such events are the name and arguments of the constraint, and the so called *action list,* a list of terms describing the domain narrowing decisions made in the given propagation step. The events of the second class are called *labeling events,* and can fall into one of the following two groups:

- A *choicepoint event* is an event associated with the creation or the failure of a choicepoint. Exactly one variable belongs to a choicepoint, but the same variable might be considered in more than one choicepoints. For example, in 'enum' mode labeling, a single choicepoint enumerates all possible values, but in 'step' mode labeling, several binary choicepoints may have to be created for the same variable.
- A *labeling choice* is made when the domain of a variable constrained either through instantiation or by removing a part of the domain. Several choices branch off a single choicepoint.

Available data for a labeling event are the variable under consideration, a term specifying the subtype of the event, and a unique identifier of the choicepoint.

Every event is intercepted and dispatched to the appropriate *visualizers* by the FDBG core. A visualizer is a Prolog predicate that reacts to events, usually by displaying the trace information, but in general it can do any kind of processing, like checking invariants, etc. Analogously to the the two major event classes we can also distinguish two different types of visualizers: *constraint visualizers* handling constraint events and *labeling visualizers* handling labeling events. FDBG provides several built-in visualizers which produce an exhaustive output, but one can also develop user-defined visualizers to filter the information or to organize it in a way that suits the problem at hand better. Both possibilities are introduced in the next section in greater detail.

3 The User Interface

FDBG works out-of-the-box, that is, in order to use the most basic services the user does not have to modify the CLP($\mathcal{FD}$) program at all. When started without any options, FDBG will use its built-in visualizers, which are discussed in the first subsection below. However, the default trace output can often be confusing. In order to make the output more understandable, minor modifications to the program are necessary. Fortunately these modifications are sparse and do not decrease the efficiency significantly. These are described in Sects. 3.2 and 3.3.

If this proves to be insufficient, one can customize the output of the built-in visualizers or even write user-defined visualizers which can exploit any problem-specific knowledge while displaying the trace information. Naturally, these require larger and larger modifications and extensions to the original program. These features are covered by Sect. 3.4.

FDBG also provides several auxiliary services, two of which are introduced in Sect. 3.5, concluding the introduction of the user interface.

For further reading and technical details, please refer to the user manual [10, Sect. FDBG].

3.1 Built-in Visualizers

The library provides a default visualizer for both event classes. `fdbg_show/2` is the default visualizer for constraint events, and `fdbg_label_show/3` is the default for labeling events.

The arguments required by the visualizers are collected and passed by the FDBG core when an event occurs. A sample output produced by these visualizers is shown in Fig. 1. In lines 1–5 FDBG is loaded and switched on. Lines 7–12 show the output of `fdbg_show/2`, and finally, lines 13–18 are produced by `fdbg_label_show/3`. These are discussed in detail below.

```
1  | ?- use_module(library(clpfd)), use_module(library(fdbg)).
2  yes

3  | ?- fdbg_on.
4  % The clp(fd) debugger is switched on
5  yes

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

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

13 Labeling [3, <fdvar_1>]: starting in range 4..5.
14 Labeling [3, <fdvar_1>]: step: <fdvar_1> = 4

15 X = 4 ? ;
16 Labeling [3, <fdvar_1>]: step: <fdvar_1> >= 5

17 X = 5 ? ;
18 Labeling [3, <fdvar_1>]: failed.
```

Fig. 1. A sample session using built-in visualizers

The output of the constraint visualizer. Each block in the output (separated by empty lines) represents a wake of a global constraint. The first, unindented line showing the constraint itself is followed by a *legend*, which informs the user about all the variables appearing in the constraint. Note that in line 7 the `X #< 6` constraint is displayed in a somewhat peculiar way. This is because the CLP($\mathcal{FD}$) library of SICStus transforms built-in constraints before posting them into the store, and this often changes the appearance. However, the meaning is naturally in accordance with the entered constraint.

Variables in the constraint are assigned a name, and this name is displayed between angle brackets instead of the usual underscore notation. In the example, we only have one variable, so it is clear that `fdvar_1` means `X`, but in case of a more realistic problem, where we have tens or even hundreds of variables, these names assigned by FDBG are hard to resolve. Therefore FDBG provides tools to assign names to variables ourselves, as introduced in Sect. 3.2.

Each line in the first part of the legend shows a variable of the constraint. These lines consist of the name of the variable, its domain *before* the constraint was woken, and optionally, following the arrow, the domain *after* the constraint narrowed it. A few remaining lines may inform about the behavior of the constraint, like entailment, failure or side effects. In lines 9 and 12 we can see that both constraints become entailed after the domain of `X` is narrowed.

The appearance of both the constraint and the legend can be customized, see Sect. 3.4 for details.

The output of the labeling visualizer. Each labeling event produces a single line of output, some are followed by an empty separator line, others are immediately followed by another labeling action.

The number in the brackets (3 in this case) uniquely identifies the choicepoint.[1] The name following the number is the name of the variable being labeled. Note that several identification numbers might belong to the same variable, depending on the mode of labeling. The text after the colon specifies the actual labeling event, informing about the creation (line 13) or failure (line 18) of a choicepoint, or of a specific labeling choice that constrains the domain of the labeled variable (lines 14 and 16).

3.2 Naming Terms

Automatically assigned variable names are not very informative. FDBG offers a service to assign names not only to single variables, but also to whole terms for later reference. The typical application is to assign a name to a compound term containing several constraint variables. In this case, all the variables in the compound are also automatically assigned a name. This is derived from the name of the term by appending a *variant of its selector* inside the compound (Table 1). During the calculation of such a selector FDBG treats a list of N items

[1] The number is in fact the standard SICStus debugger invocation number of a predicate which is called just before the labeling of that variable.

as a compound with N arguments rather than a N-deep chain of `./2` structures. This special treatment is indicated by a `#` prefix in the selector column of the table.

Table 1. Derived names

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

Term names are utilized in the process of *annotation,* which is another service of FDBG. Here, each variable appearing in a Prolog term is replaced with a compound term describing it (i.e. containing its name, the variable itself, and some data regarding its domain). In fact, the trace output of `fdbg_show/2` contains the annotated form of the constraints. During annotation, unnamed constraint variables are also given a unique "anonymous" name automatically. These names begin with the `fdvar` prefix.[2]

As an auxiliary service, names can also be used to retrieve terms during execution. A use of this feature is shown in Fig. 3, line 2. It is also exploited by another built-in constraint visualizer called `fdbg_guard/3`, which is not discussed in detail here.

3.3 Labeling

Labeling events are distinguished from constraint events if the user relies on the labeling predicates provided by the CLP($\mathcal{FD}$) library. When one chooses to apply user-defined labeling, domain narrowing is usually performed by the same predicates which are used for constraint propagation (i.e., `X=Y`, `X in RANGE`, etc.), and so FDBG has to take special measures to make a distinction between the two kinds of events. To let FDBG know these calls occur as a part of labeling, one has to insert calls to two predicates provided by FDBG as instructed by the manual [10]. These are `fdbg_start_labeling/2`, which informs FDBG about the creation and failure of a choicepoint, and `fdbg_labeling_step/2`, which denotes a specific choice. These calls do not change the behavior of labeling, and they influence efficiency only when FDBG is turned on.

3.4 Output Customization and Trace Analysis

If one is not fully satisfied with the output of the built-in visualizers one has two options. The output can be modified slightly by defining hook predicates, but for

[2] It would seem natural to name the variable as it appears in the source code, but this information is lost at the first stage of compilation, when the code is parsed. Notice that not even the highly sophisticated Prolog debugger can use the original names.

radical changes in the trace log, one has to implement user-defined visualizers. Such visualizers also serve as a great means for synchronous trace analysis. The advantage of both possibilities is that one can exploit problem specific knowledge which FDBG does not have. Although we don't show an example for a visualizer performing analysis, we would like call attention to an already mentioned built-in visualizer, `fdbg_guard/3`, which produces no output, but watches the domains of a few selected variables, and pauses the execution by firing up the Prolog debugger when a solution is "accidentally" removed from these domains.

Writing hook predicates. By implementing two hook predicates, namely `fdbg:fdvar_portray/3` and `fdbg:legend_portray/3`, one can change the appearance of variables in the annotated constraints and the appearance of legend lines respectively. When called, both are passed three arguments: the variable in the state before the constraint has actually narrowed it, the domain it would have after the propagation, and the name assigned to it.

In case of the well known N-queens problem, for instance, one could display the corresponding row of the chess-board showing possible positions of the queen for each variable by defining `fdbg:legend_portray/3` appropriately. Figure 2 shows such a hook predicate and the output it produces. In the latter, an `X` marks the fields where the queen might still be, a dash (`-`) marks the positions waived in the current propagation step, and a dot (`.`) denotes the positions that have been known to be threatened by other queens before. (The details of the solution and the applied `no_threat/3` constraint are not discussed here.)
A more resourceful application of `fdbg:legend_portray/3` is for problems dealing with symbolic values. Such values must be translated into natural numbers when modeling the problem, but because this translation is often artificial and arbitrary, it is hard to retrace it in one's mind. Using this hook predicate, however, the trace output can be made a lot clearer by replacing the numbers with their symbolic equivalents.

Defining your own visualizers. For more complicated problems one might want to change the general structure of the trace output, which cannot be done by means of the above hook predicates. This can be achieved by writing user-defined visualizers, an approach that enables one to process raw trace information provided by the FDBG core at one's liking. To enable rapid development, FDBG provides several utility predicates which perform term annotation, legend printing and even the *simplification of action lists* in case one would like to process the action list returned by a global constraint (as done by the default legend printer). To use such a visualizer, its name has to be passed as an argument to `fdbg_on/1` when activating the debugger.

Returning to the N-queens problem, one might want to write a visualizer that displays the whole chess-board at each event instead of listing the affected variables only. Such a visualizer and it sample output is shown in Fig. 3 (relying on `print_row/2` from Fig. 2). Note how the naming service is utilized in line 2 to fetch the list of variables representing the whole chess-board.

The hook predicate

```
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, put_code(0' ), !.
print_fields(I, N, Now, After) :-
        put_code(0' ),
        (   fdset_member(I, After) -> put_code(0'X)
        ;   fdset_member(I, Now)   -> put_code(0'-)
        ;                             put_code(0'.)
        ),
        I1 is I+1,
        print_fields(I1, N, Now, After).
```

Sample output

```
no_threat(2,<queen_3>,2)
    queen_3: [ X - . . ]
    Constraint exited.
```

Fig. 2. A portray hook for the N-queens problem

3.5 Auxiliary Tools

Simplification of action lists. When someone writes a user-defined visualizer, one of the problems to tackle is the parsing of the action list. To make this task easier, FDBG offers an auxiliary service to simplify such lists.

An action list returned by a global constraint may contain several domain narrowings in various shapes (`in/2`, `in_set/2` and unification), explicit entailment and failure actions, and calls to Prolog goals. Failure of the constraint can be caused by either a narrowing that removes all values from a domain or by an explicit failure action.

Annotation takes the action list into account when calculating the future domains of annotated variables. After that is done, the corresponding narrowing actions are not relevant any more. The rest of the action list, on the other hand, is too varied to be parsed easily. By calling an auxiliary predicate of FDBG with a list of the annotated variables and the original action list, a much simplified action list is produced:

- all the irrelevant narrowing actions are removed;

The 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).
```

Sample output

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

no_threat(4,<queen_4>,2)
    [ . X . . ]
    [ . . . X ]
    [ X X . . ]
    [ X - X - ]
```

Fig. 3. A user-defined visualizer for the N-queens problem

- narrowing actions causing failure are converted to explicit failures;
- all other narrowings are replaced by compound terms describing the affected variables—these compounds are exactly like the ones replacing variables during annotation.

Debugger command hooks. The Prolog debugger offers a hook predicate to extend its command interface. Using this hook FDBG modifies one command and introduces two new ones to help the programmer to examine the state of finite domain programs also while tracing their execution. The command which lists blocked goals is modified to also print the annotated form of domain variables. One of the two new commands allows one to print the annotated form of a subterm of the current goal, the other lets us assign a name to a subterm.

4 Some Implementation Issues

In this section we are going to discuss certain interesting details of the implementation. First it is explained how events are detected and reported to the visualizers. This is followed by the explanation of how variable naming works in Sect. 4.2.

4.1 Event detection

Global constraints vs. indexicals. As we have mentioned in the Introduction, FDBG is capable of handling global constraints only. To understand the reasons behind this limitation, and to see why is it not too serious after all, let us recollect the main differences between the two kinds of constraints known to the CLP($\mathcal{FD}$) library of SICStus.

Indexicals [12], the simpler but quicker constraints, always work on a fixed number of arguments. Most arithmetical constraints (like `X #> 0`) are translated to indexicals at compile time. One can also define indexicals with a syntax defined specifically for this purpose. Such constraints are handled by the CLP($\mathcal{FD}$) library internally in a very special way.

Constraints of the other kind, called *global constraints* are more generic in the sense that they can handle a variable number of arguments, such as lists of variables. They are represented by clauses of a Prolog predicate which—following certain rules—implements propagation. This predicate is called by the CLP($\mathcal{FD}$) library whenever propagation seems possible.

The fact that indexicals are treated specially, while the propagators of global constraints use traditional Prolog syntax and semantics, is the reason why FDBG ignores indexicals altogether. This means, in other words, that any domain narrowings done by indexicals happen unnoticed, making FDBG output for programs relying on them harder to follow. Since we know that many arithmetical constraints translate to indexicals, this seems to be a big limitation. Fortunately, after the FDBG library is consulted, all arithmetical constraints are translated to global constraints instead, and so their behavior becomes observable. Since translation happens in compile time, it is advisable to load FDBG before any user programs. The process of translation is explained in detail in the next paragraph.

Translation of arithmetical constraints. When SICStus encounters a call to an arithmetical constraint at compile time, it hands it over to the CLP($\mathcal{FD}$) library for *goal expansion,* a reformulation into a call to a regular Prolog predicate. The library simplifies and optimizes the constraint by precalculating constant subexpressions, bringing all the variables to one side of the equation, etc. Under regular circumstances, the resulting equation, depending on its shape, gets expanded into a call to one of the numerous indexicals, each of which represents a very specific arithmetical constraint. When the equation is more complicated than what the indexicals can handle, it is expanded into a call to `scalar_product/4`, a global constraint which can handle linear equations in general, but is of course less efficient than the indexicals.

When FDBG is turned on, the aforementioned optimization phase is skipped, and the arithmetical constraint is always expanded into `scalar_product/4`. Naturally this will have its effect on the performance, but that is negligible compared to the overhead of FDBG in general. It may also modify the behavior of the program slightly, since the propagation of `scalar_product/4` is often weaker than that of the indexicals. Nonetheless, we believe that the chances of a program behaving incorrectly with indexicals but working perfectly with the global constraint are small (unless of course it is one of the built-in constraints which is erroneous).

The translation lacks one important feature, however. In the process of goal expansion the original form of an arithmetical constraint is lost. FDBG does its best to re-transform the scary looking `scalar_product/4` constraint into a more natural form, but this is still different than which it appears in the code and unfortunately it also varies from time to time. An successful attempt was made to preserve the original form for run time by introducing a fifth argument to `scalar_product` storing the source goal, which was supposed to be ignored by all except FDBG. However, this required the overwriting of several CLP($\mathcal{FD}$) library predicates (so they could cope with the change in the argument count), which would have been hard to maintain and was quite understandably considered as undesirable, and so it has been abandoned.

Advice-points. To detect events, we exploit an advanced service of the standard SICStus Prolog debugger. Namely, advice-points[3] are placed on several predicates of the CLP($\mathcal{FD}$) library. These predicates can be divided into two groups, as follows.

1. Predicates generating constraint events:
   - The internal equivalent of `clpfd:dispatch_global/4`, the predicate which is responsible for global constraint propagation.
   - A group of predicates handling goals such as `X in RANGE`, `X in_set FDSET`, `X=Y` and `domain(Xs, Min, Max)`. Strictly speaking, these are not constraints, since they do not perform any propagation, "only" narrow the domain of the affected variable(s) in a single step. Still, for the sake of uniformity, FDBG presents them to the visualizers exactly like global constraints. In order to do this, a fake action list is generated, which contains the narrowings and states that the "constraint" has become entailed.
2. Predicates generating labeling events:
   - Three predicates performing narrowing when the user relies on built-in labeling: `clpfd:labeling_singleton/3`, `clpfd:labeling_min/3`, and `clpfd:labeling_max/3`.

[3] An advice-point is a kind of breakpoint that allows one to carry out some actions (like calling predicates) at certain points of execution, independently of the tracing activity.

– `fdbg_start_labeling/2`, which marks the creation and the failure of a choicepoint, and `fdbg_labeling_step/2`, which notifies FDBG of a labeling choice, both of which were already mentioned in Sect. 3.3.

Both event types are tidied up a bit before being reported to the visualizers. For example, the constraints which had been optimized before posting are now transformed again to at least resemble their original form.

4.2 Variable Naming

As described briefly in Sect. 3.2, FDBG offers a service to assign names to terms and variables in particular. The name serves two purposes: it is used by the visualizers to refer to domain variables and it may be used to retrieve terms at any point in the program. This latter feature was exploited in the example visualizer shown in Fig. 3.

Names are stored in an AVL tree managed by the ASSOC library of SICStus Prolog. The tree itself is stored in a somewhat peculiar place: in a *mutable* value wrapped up in a unary compound, which is an element of an open-ended list stored in the *global private* field of the debugger associated with break level zero. Let us see, why.

– This global private is capable of storing a Prolog term for the lifetime of a toplevel query. It is initiated with a fresh variable for each query, and thereon can be unified with arbitrary terms by calling the appropriate debugger predicate.
– Since there is exactly one global private associated with a query, FDBG recommends a convention to allow multiple simultaneous uses of this resource. Namely, it unifies it with an open-ended list, and allocates a single (first available) element of this list, giving that element a functor which uniquely identifies its use (`fdbg/1` in this case, the name of the module using the element in general).
– This functor carries a mutable value, storing an AVL tree of terms with their associated names as keys. The value is mutable because we need to modify the AVL tree whenever a name is added to the store.

As a consequence, we have an expandable name store which is automatically reset for every toplevel query. This has the advantage that we do not have to clear it after we are done. On the other hand, names must be reassigned in each query.[4] The best way to do this is to insert the call doing the name assignment into the program itself.

5 Related Work

5.1 VIFID

Just like FDBG, the VIFID (Variable Visualization for Finite Domains) program [13] was also aimed at tracing SICStus Prolog CLP($\mathcal{FD}$) programs. VIFID

[4] Since we lose all references to the variables of a query once it exits, this is inevitable anyway.

provides a graphical front-end, written in Tcl/Tk, which displays domains as a series of red and green squares, green ones denoting values *in* the domain, red ones denoting values *outside of* the domain of the variable. It is also capable of showing the history of a variable: a graphical listing of all its earlier domains. A very useful feature is that one may remove values from a domain, or even post a new constraint to the store, and observe the effect on the other variables immediately. Then, before continuing execution, one has to decide whether to keep or discard the changes.

On the other hand, the list of variables to be watched as well as the value interval to be displayed must be specified in advance when the system is initialized. Moreover, the specified interval is the same for all variables. Due to this limitation, it is uncomfortable (or even impossible) to watch variables with different initial domains simultaneously. Furthermore, the representation of variables is fixed, i.e., it is always a series of colored squares, there is no way to customize this look. One also has to insert a call to a VIFID predicate at each point one wishes to examine the state of the program. This limitation results in the user not being able to follow propagation in small steps.

5.2 Generic Trace Model

The scheme of a generic trace model is presented in [6]. This model is based on the formal *observational semantics* of a finite domain program, not unlike FDBG. The observational semantics are used to build a *generic trace,* which is already independent from the specific solver. The trace in turn can be considered by debugging tools which thus do not have to take into account all the details of the solvers.

This separation of the transformation of the observational semantics into a generic trace and the visualization of the trace makes the construction very flexible. It, in fact, resembles the basic structure of FDBG, where the problem of visualization is separated from the question of observing the behavior of the solver. Thanks to this similarity in their approach, one could easily write a visualizer for FDBG which produces a generic trace in the format specified by [6].

5.3 Explaining CLP($\mathcal{FD}$) Execution

The work described by Ågren et al. [14] also produces a log of events, but focuses on providing explanations for the actions during the process of constraint solution. This is achieved by extending the implementation of the global constraints with code for generating the appropriate events (typically for the domain prunings performed by the algorithm). This is in contrast with FDBG, which does not require any modification of the constraint solver, but provides a coarser granularity trace. While Ågren's work can report on any details of the constraint solving algorithm, FDBG only shows the change of the variable domains between the wakeup and subsequent falling asleep of the solver. Another difference is that while Ågren reports on research work in progress, work in this paper has been encapsulated in a library of SICStus Prolog. As such, FDBG contains a number of tools and hooks that enable the user to easily customize the processing of the events.

6 Future Work

In Sect. 4.1 we discussed the translation of arithmetical constraints into global constraints, explained that in this process the original form is not preserved, and mentioned an successful but unsatisfactory experiment to resolve this issue. An acceptable alternative solution is still to be found.

The search tree is represented by labeling events in the trace log. When a new branch is created, the user is properly informed about its parent node, the selected variable and the actual reduction of its domain. When a branch fails, however, the restored state is not contained explicitly, one has to trace it back from earlier events. It would be very useful if the trace log included this information, too. Also, it would be interesting to extend the scope of labeling events to more complicated choices, like nontrivial disjunctive constraints.

A well chosen graphical representation of variable domains can help to grasp the execution state. For this reason we would like to implement a highly configurable, interactive graphical visualizer for FDBG. Plans of such a visualizer have already been proposed in [15], and a rudimentary tool has also been developed.

7 Conclusions

We presented an event based tracing scheme for finite domain constraint programs written SICStus Prolog. First, we introduced the notion and classes of events which give means to observe the execution of a constraint program. Then the concept of visualizers was introduced, which convert events to trace entries. Afterwards we recounted how the trace output can be made clearer by means of naming terms and variables, and by applying annotation, an auxiliary service of FDBG utilizing variable names.

It was also shown how can one modify the look of the trace log to a certain extent by customizing the built-in visualizers, or even fully determine the output by writing user-defined visualizers. Examples were presented for both cases.

Following the introduction of the features offered by FDBG, we covered some of the details of the implementation such as the way of detecting CLP($\mathcal{FD}$) events, or the method of storing variable names.

Finally we compared FDBG with some other approaches targeted at debugging finite domain programs by pointing out the similarities and differences in their services, and we also showed possible directions of future development.

Acknowledgement

The authors would like to thank Mats Carlsson, the main implementer of SICStus Prolog for his invaluable ideas, his continuous support, and his contribution with the modification of the CLP($\mathcal{FD}$) library in order to support FDBG.

References

1. Meier, M.: Debugging constraint programs. In Montanari, U., Rossi, F., eds.: Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings. Volume 976 of Lecture Notes in Computer Science., Springer-Verlag (1995) pp 204–221
2. Deransart, P., Hermenegildo, M.V., Maluszynski, J., eds.: Analysis and Visualization Tools for Constraint Programming. Volume 1870 of Lecture Notes in Computer Science., Springer-Verlag (2000)
3. Müller, T.: Practical investigation of constraints with graph views. In Dechter, R., ed.: Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings. Volume 1894 of Lecture Notes in Computer Science., Springer-Verlag (2000) pp 320–336
4. Puebla, G., Bueno, F., Hermenegildo, M.: An assertion language for constraint logic programs. In Deransart, P., Hermenegildo, M., Maluszynski, J., eds.: Analysis and Visualization Tools for Constraint Programming. Number 1870, Springer-Verlag (2000) pp 23–61
5. Puebla, G., Bueno, F., Hermenegildo, M.: A framework for assertion-based debugging in constraint logic programming. Lecture Notes in Computer Science **1520** (1998) p 472
6. Deransart, P., Ducassé, M., Langevine, L.: A generic trace model for finite domain solvers. In O'Sullivan, B., ed.: Proceedings of User Interaction in Constraint Satisfaction (UICS'02), Cornell University (USA) (2002)
7. Ducassé, M., Langevine, L.: Automated analysis of CLP(FD) program execution traces. In: Proceedings of the International Conference on Logic Programming. Volume 2401 of Lecture Notes in Computer Science., Springer-Verlag (2002)
8. Jahier, E., Ducassé, M.: Generic program monitoring by trace analysis. In: Theory and Practice of Logic Programming. Volume 2 part 4&5 of Special Issue Program Development., Cambridge University Press (2002) pp 613–645
9. Bracchi, C., Gefflot, C., Paulin, F.: Combining propagation information and search tree visualization using ILOG OPL studio. In Kusalik, A., ed.: Proceedings of the Eleventh International Workshop on Logic Programming Environments (WLPE'01), Paphos, Cyprus (2001)
10. Swedish Institute of Computer Science Uppsala, Sweden: SICStus Prolog User's Manual. (2003) http://www.sics.se/isl/sicstuswww/site/documentation.html.
11. Kreuger, P., Bohlin, M.: Introduction to constraint programming. (Lecture notes at http://www.idt.mdh.se/phd/courses/constraints. See slides.)
12. van Hentenryck, P., Saraswat, V., Deville, Y.: Constraint processing in cc(FD). Unpublished report (1992)
13. Carro, M., Hermenegildo, M.: Visualization designs for constraint logic programming. In: Swiss Informatics Societies. Volume 2. (2001) pp 27–34
14. Ågren, M., Szeredi, T., Beldiceanu, N., Carlsson, M.: Tracing and explaining execution of CLP(FD) programs. In Tessier, A., ed.: Proceedings of the Twelfth International Workshop on Logic Programming Environments (WLPE'02), Copenhagen, Denmark (2002) pp 1–16
15. Szeredi, T.: Korlát-logikai programok nyomkövetése (in Hungarian). Master's thesis, Budapest University of Technology and Economics (2001) *Title in English: Tracing Finite Domain Programs.*