Tracing and Explaining Execution of CLP(FD) Programs

M. Ågren, T. Szeredi, N. Beldiceanu, and M. Carlsson

SICS, Lägerhyddsv. 18, SE-752 37 UPPSALA, Sweden
{magren,tszeredi,nicolas,matsc}@sics.se

Abstract. Previous work in the area of tracing CLP(FD) programs mainly focuses on providing information about control of execution and domain modification. In this paper, we present a trace structure that provides information about additional important aspects. We incorporate explanations in the trace structure, i.e. reasons for why certain solver actions occur. Furthermore, we come up with a format for describing the execution of the filtering algorithms of global constraints. Some new ideas about the design of the trace are also presented. For example, we have modeled our trace as a nested block structure in order to achieve a hierarchical view. Also, new ways about how to represent and identify different entities such as constraints and domain variables are presented.

1 Introduction

In this paper, we present new ideas in the area of tracing CLP(FD) programs. New trends such as tracing global constraints and explaining solver actions are investigated. Since these issues are still in their development stages and not systematically used, we have not tried to create a universal trace model. Instead, we focus on CLP(FD) and in particular the CLP(FD) solver of SICStus Prolog.

The idea of tracing constraint programs originates from Meier [9]. Also, the search-tree visualization tool in CHIP [13] uses a trace log to be able to display propagation events. More current work is carried out by Langevine et al. who presented a CLP(FD) trace model in [8]. Their work is naturally influenced by the work of Ducassé [3] on Opium.

Previous work such as the above mainly focuses on providing information about control of execution and domain modification. Of course, this information is needed, but if nothing else is provided, important information about the execution is lost. We have looked at the execution of a CLP(FD) program from the following perspectives:

1. Control of Execution; information about posting new constraints, waking up delayed constraints, constraint entailment/failure.
2. Domain Modification; information about variable domain narrowing (pruning).
3. Declarative Aspect; information about why certain actions are taken by the solver and in which context they occur.

4. Procedural Aspect; information about filtering algorithms and their entities.

We build the trace of events as a nested block structure. This means that most of the trace events appear inside a block surrounded by a starting event and an ending event. One example of this is the posting of a new constraint, which gives rise to a block surrounded by the trace events `begin_new_ctr` and `end_new_ctr`. Inside this block, trace events describing the posting of demon(s) for the new constraint and initial prunings are generated. Building the trace like this has several advantages. First of all, it makes it easier to get a structured overview of the trace. Second, similar actions may be grouped together inside the same block. Furthermore, extra information about the action(s) taking place may be added to the opening and closing events as well as through added events inside the block.

We identify variables and other entities by their context, i.e. their positions in the constraints, and not only by some identifier. This is useful when one needs to distinguish a specific occurrence among several of the same variable. In addition to that, it makes it possible to also clearly identify entities such as integers, atoms and compound terms.

Also, corresponding to point 3 above, we provide *explanations* [5] for why certain actions are taken by the solver. These explanations are incorporated in the trace by specific explanation trace events. Actions for which explanations are provided include domain narrowing, failure, demon enqueuing and internal constraint posting. We extend previous work by also providing compact explanations for global constraints. By doing this, we obtain a stronger link between the consistency rules of the global constraints and the trace.

Moreover, corresponding to point 4 above, we provide trace events containing detailed information about the filtering algorithms of global constraints. This concerns identifying and presenting the different entities inside the algorithms. This too increases the link between the global constraints and the trace. However, this is more related to the procedural aspect.

An example of a simple CLP(FD) program in SICStus Prolog is shown in Ex. 1. It will be used throughout the paper. Line `(2)` initializes the domains of the included variables. Lines `(3)` and `(4)` post the well-known constraints `all_different/1`[1] and `element/3`. Line `(5)` calls SICStus's builtin search predicate with a `leftmost`[2] search strategy.

In the following, Sect. 2 presents a CLP(FD) execution model. This is needed in order to be able to present the trace structure in its relevant context. Sect. 3 presents the different trace events. First, some design choices concerning how to represent different entities such as constraints and variables are introduced. Following that, the trace events are presented in chunks; trace events corresponding to the same perspective, as noted above, are presented together. After that, a short description of the implementation made in SICStus Prolog is given. Sect. 4 discusses possible areas of usage. Finally, Sect. 5 concludes the paper.

[1] This is the naive version of `all_different`, achieving the same pruning as a set of pairwise inequality constraints.

[2] Leftmost variable, smallest value first.

Ex. 1 A simple CLP(FD) Program.

```
trace_me :-                                          (1)
        domain([X,Y,V1,V2],1,6),                     (2)
        all_different([X,Y,3,V1,8,V2]),              (3)
        element(X,[2,4,3,8],Y),                      (4)
        labeling([leftmost],[X,Y,V1,V2]).            (5)
```

2 Execution Model

This section presents an execution model of a CLP(FD) kernel; the different data structures it contains, and how these interact. By doing this, we can later present the trace structure in its relevant context. The model is influenced by the constraint solvers of SICStus Prolog [1] and Choco [7].

Data Structures. The data structures contain lists and queues for storing information regarding constraints, variables, demons and propagation events.

Each *constraint* is associated with a list of domain variables, `VList`, and a list of demons, `DList`. `VList` contains the variables that occur in the constraint. `DList` contains the demons that are responsible for removing inconsistent values from the variables in `VList`.

Each *variable* is associated with a domain, `Domain`, containing the values the variable may take. These domains are narrowed by propagation events specified below. Furthermore, each variable is associated with a list of constraints, `CList`, which it is involved in. As soon as the domain of a variable changes, one or more of these constraints may wake up and activate their demons.

Each *demon* contains a *wake-up condition*. When the wake-up condition is fulfilled, the demon's *filtering algorithm* is activated. This filtering algorithm may exclude some values from the domains of some variables. A filtering algorithm consists of a set of *methods*, $\{M_1, \ldots, M_n\}$, where each M_i corresponds to some consistency rule of the constraint that the demon belongs to. If some method, M_i, notices that the domains of some variables are not consistent with the constraint, M_i generates propagation events which specify domain narrowings on these variables.

The *propagation events* have the form `X in_set S`, which constrains the variable `X` to be a member of the set `S` of integers.

Two *global lists* are used for storing current constraints and variables in the system; `ConstraintList` and `VariableList`. Furthermore, two *global queues* are needed to manage the actions taken by the kernel. `ReadyQueue` contains the demons that are about to wake up. `PropagationQueue` contains the propagation events that have been created by active demons.

Fig. 1 shows the different data structures and how they are connected to each other. The state is taken from Ex. 1, just before the execution of line `(4)`. `C1` and `C2` are the `all_different/1` and `element/3` constraints respectively. Detailed

information about variables X and Y as well as about constraint C2 is shown in three boxes.

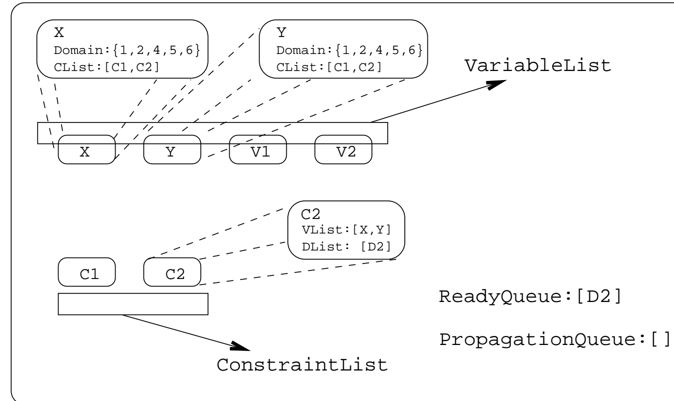

Fig. 1. Kernel state for Ex. 1. Just before the execution of line (4).

Services. Our CLP(FD) kernel provides the following services:

- connectCtr(C), adds the constraint C to the kernel. This includes adding C to ConstraintList, adding the variables in C to VariableList, adding C to the CList of all its variables, creating the demons[3] associated with C and running each demon's filtering algorithm for initial propagation.
- disconnectCtr(C), removes the constraint C from the kernel. This is done when the constraint is entailed, i.e. when C is true no matter what values its variables take.
- enqueueCtr(C,E), puts last in ReadyQueue the demons associated with the constraint C that are to wake up due to the propagation event E. If any of these demons are already in ReadyQueue, it will not be enqueued again.
- dispatchCtr(), dequeues the demon that is first in ReadyQueue and activates its filtering algorithm.
- enqueueEvent(E), puts E last in PropagationQueue.
- dispatchEvent(), dequeues the event E that is first in PropagationQueue, performs the narrowing on some domain variable X it specifies, and calls enqueueCtr(C,E) for all constraints in X's constraint list.

If the functions dispatchEvent() and dispatchCtr() can both be executed, the former will have higher priority.

[3] Our trace structure supports the use of more than one demon for each constraint. However, since the examples in this paper are created using SICStus Prolog, and it currently only associates one demon with each constraint, more than one demon per constraint will never be shown.

The Active System: An Example. The easiest way to see how the above described data structures interact is by looking at an example. Assume that the state of the kernel is as in Fig. 1. There is no propagation event to be performed, but the demon `D2` of constraint `C2`, the `element/3` constraint, should be woken up. The kernel calls the function `dispatchCtr()`, which removes `D2` from `ReadyQueue` and activates its filtering algorithm. A first method, `M1`, generates a propagation event, `E1`, which restricts `X` to take a value inside the range `1..4`. Equally, a second method, `M2`, generates a propagation event, `E2`, which restricts `Y` to take a value inside the set `{2,3,4}`. Also, a third method, `M3`, generates a propagation event, `E3`, which restricts `X` from taking the value 4[4]. When the filtering algorithm of `D2` has finished, the kernel notices that `PropagationQueue` has changed to contain the propagation events `E1`, `E2` and `E3`. The kernel calls the function `dispatchEvent()` three times which does the following each time:

- The first propagation event in `PropagationQueue` is dequeued.
- The narrowing of the variable specified by the propagation event is performed by the kernel.
- The demons of the constraints `C1` and `C2`, that are supposed to wake up due to the narrowing performed, are put in `ReadyQueue` by calling the function `enqueueCtr()` twice.

When `PropagationQueue` is empty, the kernel starts over by calling the function `dispatchCtr()` again. This is repeated until both queues are empty. When this is the case, the control of execution is given back to the Prolog level. In our case, this means that labeling takes place which will fix a variable and therefore trigger more propagation.

3 Trace Structure

In this section, the trace structure and the information it contains is presented. We start by introducing our ideas about how to identify the different entities in a CLP(FD) program such as constraints and variables. Following that, the trace events are presented together with some examples. Finally, our implementation in SICStus Prolog is described.

3.1 Representing and Identifying Entities in CLP(FD) Programs

Identifiers. The *id of a constraint* is a Prolog atom created by prepending the atom `ctr_` to the source code functor and appending a unique[5] number to this. The *id of a variable* is a Prolog atom of the form `fdvar_N` where `N` is a unique number. For example, the following identifiers are produced by Ex. 1 where `C0`, `C1` and `C2` are the constraints posted on lines `(2)`, `(3)` and `(4)` respectively:

$$\{\texttt{X} \mapsto \texttt{fdvar_1}, \texttt{Y} \mapsto \texttt{fdvar_2}, \texttt{V1} \mapsto \texttt{fdvar_3}, \texttt{V2} \mapsto \texttt{fdvar_4},$$
$$\texttt{C0} \mapsto \texttt{ctr_domain_1}, \texttt{C1} \mapsto \texttt{ctr_all_different_1}, \texttt{C2} \mapsto \texttt{ctr_element_1}\}$$

[4] Since the value 8 on position 4 in the list is not in the domain of `Y`.
[5] Unique for the specific type of constraint.

Constraint Representation. A constraint is represented similarly to its source code representation. It has the same functor atom and structure, with all variables replaced by their given identifiers. In Ex. 1 for instance, the constraint

```
all_different([X,Y,3,V1,8,V2])
```

is represented as

```
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]).
```

Location of Variables. It is not enough to be able to identify variables by their identifiers. While this allows to distinguish different variables, it will not make it possible to, within one constraint, uniquely determine a specific occurrence among several of the same variable.

We have solved this with what we call a *path*. A path uniquely determines a position or a list of positions in a constraint. When referencing arguments of a constraint, for example when displaying a variable's domain, a path referring to that variable is given instead of its id. This path must always have a context, a corresponding constraint, to be interpreted in.

A path is represented as a Prolog list containing (lists of) fixed values or other paths. The first element of a path identifies the topmost position in the corresponding constraint. For example, any path referring to some position(s) in the `element/3` constraint in Ex. 1 has one of the integers 1, 2, 3 or one of the lists `[1,2]`, `[1,3]`, `[2,3]`, `[1,2,3]` as first element. Lines a and b in Table 1 show two of these in paths with only one element and the position(s) they refer to.

The i+1:st element of a path refers to some position(s) **inside** whatever the i:th element refers to. This means that the position(s) the i:th element refers to must be a compound term or a list of compound terms. Line c in Table 1 illustrates this. This example also introduces one more symbol in the path term; the `#` sign. It is used whenever the term(s) we refer to is inside a list (a non-list compound term would not have a preceding `#` sign).

Some more syntax is needed in order for the path to be more expressive and compact. First of all, we will use `[*]` to express "all positions" of a compound term. It may also be useful to identify **almost** all positions in a compound term. We will use the \ sign for this purpose, denoting set subtraction[6]. Line d in Table 1 illustrates this.

In addition to the above, we may apply functions to the entities referred to by a path. `P/F` has the meaning that the function `F` is applied to each entity referred to by the path `P`. Possible values for `F` are, among others, `min`, `max` and `length`. Table 2 shows some examples of this. If `P` refers to a list of positions of size n, `P/F` returns a list of values of size n. However, one element lists are simplified as shown on lines a and b in Table 2.

---

[6] In our case, the sets are lists.

Table 1. The positions marked with gray are the positions referred to by the corresponding path.

Path	Position(s)	Path	Position(s)
a. [[1,3]]	element(X,[2,4,3,8],Y)	c. [2,#[1,2]]	element(X,[2,4,3,8],Y)
b. [2]	element(X,[2,4,3,8],Y)	d. [2,#[*]]\[2,#1]	element(X,[2,4,3,8],Y)

Table 2. Functions applied to paths. The state of the variables is as in Fig. 1, i.e. before the execution of line (4) in Ex. 1.

P/F	Constraint	Value(s)
a. [1]/length	all_different([X,Y,3,V1,8,V2])	6
b. [1]/min	element(X,[2,4,3,8],Y)	1
c. [[1,3]]/max	element(X,[2,4,3,8],Y)	[6,6]

3.2 Trace Events

Now, it is time to present the trace events. They are organized in chunks, each chunk containing trace events for a specific purpose. The chunks of trace events are presented in tables followed by examples generated by Ex. 1.

All trace events have an `EventId` attribute, an increasing number. We will often identify them by the value of this attribute and simply call it *id*. The `EventId` is always the first attribute of the Prolog fact. Furthermore, when we talk about a whole block of trace events, we mean the opening and closing trace events of the block and all trace events in between. For example, a `prune` block includes the trace events `begin_prune` and `end_prune`, and all trace events between these.

Within our illustrations of Ex. 1, since the constraint representations occur very frequently, we will use the following short notations:

- `ALLDIFF = all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4])`
- `ELEMENT = element(fdvar_1,[2,4,3,8],fdvar_2)`

Also, when mentioning variables, we will use the variable name from the source code of Ex. 1, even if the trace events contain the given identifier.

Control of Execution. In this section, trace events describing the control of execution are presented. This concerns things like new constraint posting, demon enqueuing and demon wakening. The trace events that describe these actions are presented in Table 3.

Lines (3) and (4) in Ex. 1 post two constraints to the system. This gives rise to the following trace events to be created[7]:

[7] There are several gaps, identified by '...', since we only present the trace events relevant to the control of execution at this stage.

Table 3. Control of execution trace events.

Trace Event	Attributes	Meaning
`begin_new_ctr`	`EventId`, `ConstraintId`, `Constraint`	Generated when a new constraint is to be added to the system, i.e. when entering the `connectCtr()` function. `ConstraintId` and `Constraint` are respectively the constraint's id and internal representation as explained above.
`end_new_ctr`	`EventId`, `Result`	Generated when demons for the constraint have been created and possible initial propagation has finished, i.e. just before exiting the `connectCtr()` function. `Result` denotes the result of adding the new constraint. `Result` $\in$ {`fail`, `delay`, `entail`}
`new_demon`	`EventId`, `DemonId`[a], `DemonType`, `WakeConds`, `Constraint`	Generated when a new demon is added to the system. This happens inside the `connectCtr()` function. `DemonType` is a descriptive atom of the demon. `WakeConds` is a list of pairs (`VPath`,`Condition`) where `VPath` is a path to some variable(s) and `Condition` $\in$ {`min`, `max`, `minmax`, `val`, `dom`}.
`push_demon`	`EventId`, `DemonId`, `DemonType`	Generated when a wake-up condition for at least one variable has been fulfilled. This corresponds to the function `enqueueCtr()`.
`begin_wake_demon`	`EventId`, `DemonId`, `DemonType`	Generated when a demon is to wake up for propagation. This corresponds to entering the `dispatchCtr()` function, after dequeuing the topmost demon from `ReadyQueue`.
`end_wake_demon`	`EventId`, `Result`	Generated when a demon has finished all propagation. This corresponds to exiting the `dispatchCtr()` function, after the filtering algorithm has finished.

[a] Identifiers for demons are generated similar to identifiers for constraints.

```
begin_new_ctr(9,ctr_all_different_1,ALLDIFF)
  new_demon(10,ctr_all_different_1,all_different_1,[[1,#[*]]-val],
            ALLDIFF)
  begin_wake_demon(11,ctr_all_different_1,all_different_1)
  ...
  end_wake_demon(21,delay)
end_new_ctr(22,delay)
begin_new_ctr(23,ctr_element_1,ELEMENT)
  ...
  begin_new_ctr(25,ctr_in_1,fdvar_1 in 1..4)
  ...
  end_new_ctr(32,entail)
  new_demon(33,ctr_element_1,element_3,
```

```
          [[1]-dom,[2,#[*]]-minmax,[3]-minmax],ELEMENT)
  begin_wake_demon(34,ctr_element_1,element_3)
  ...
  end_wake_demon(57,delay)
end_new_ctr(58,delay)
```

The first five trace events are created due to the posting of the constraint `all_different/1`. This gives rise to the creation of a new demon (id `10`) and the wakening of that demon (id:s `11` and `21`). The last one of these trace events (id `22`) contains the result `delay`, meaning that the constraint will not wake up again until the state of at least one of its variables changes. Moving on, the `element/3` constraint is introduced in much the same way. The difference is the creation of an internal `in/1` constraint (id:s `25` and `32`) to ensure that `X` takes a value in the range `1..4`.

Domain Modification. Now, we introduce trace events describing domain modification, i.e. pruning of variables. Such trace events are presented in Table 4.

For Ex. 1, the following trace events are added due to the posting of the `all_different/1` constraint on line `(3)`:

```
begin_prune(14,remove_value(3),ALLDIFF)
  ...
  before_prune(16,[1,#[1,2,4,6]],[[[1|6]],[[1|6]],[[1|6]],[[1|6]]],
              ALLDIFF)
  prune(17,ctr_all_different_1,[1,#[1,2,4,6]],remove_value(3),ALLDIFF,
        succeed)
  after_prune(18,[1,#[1,2,4,6]],[[[1|2],[4|6]],[[1|2],[4|6]],
                                 [[1|2],[4|6]],[[1|2],[4|6]]],ALLDIFF)
end_prune(19,succeed)
```

These will occur between the trace events with id:s `11` and `21`, the wakening of the `all_different/1` demon. The first trace event contains information about the intended domain narrowing. In this case the value `3` is to be removed from some variables. The path to these variables is shown in the following three trace events. The first and last of these (id:s `16` and `18`) contain domain information about the variables before and after pruning. The middle trace event (id `17`) contains information about the actual pruning. Finally, the trace event with id `19` contains the result of the pruning. Since it did not produce any empty domains, the result is `succeed`.

The posting of the `element/3` constraint generates the following trace events containing information about pruning the variable `X`:

```
begin_prune(50,remove_values([[4|4]]),ELEMENT)
  ...
  before_prune(52,[1],[[[1|2],[4|4]]],ELEMENT)
  prune(53,ctr_element_1,[1],remove_value(4),ELEMENT,succeed)
  after_prune(54,[1],[[[1|2]]],ELEMENT)
end_prune(55,succeed)
```

Table 4. Domain modification trace events.

Trace Event	Attributes	Meaning
`begin_prune`	`EventId`, `Intention`, `Constraint`	Generated when some pruning is about to occur, corresponds to actions inside the `dispatchCtr()` function. `Intention` is a Prolog fact with information about the intended pruning[a].
`end_prune`	`EventId`, `Result`	Generated when some pruning has been carried out. `Result` denotes the result of all domain narrowings performed inside the block. `Result` = `succeed` iff all prunings inside the block was successful, otherwise `Result` = `fail`.
`prune`	`EventId`, `PrunedVars`, `Pruning`, `Constraint`, `Result`	Generated when some domain narrowing occurs. This corresponds to the `enqueueEvent()` function. `PrunedVars` is a path referring to the positions of the pruned variables. `Pruning` is a Prolog fact with information about the actual pruning. `Result` denotes the result, `fail` or `succeed`, of the pruning.
`before_prune`, `after_prune`	`EventId`, `PrunedVars`, `Domains`, `Constraint`	These are generated before (after) a `prune` trace event. They contain domain information about the pruned variables before (after) value removal. `Domains` is a list of domains in the SICStus Prolog format[b].
`fail`	`EventId`, `Constraint`	Generated when inconsistency is noticed due to some non-fulfilled necessary condition of a constraint.

[a] Since some values might have been removed already, the actual values removed may be different from the intended pruning.

[b] For example, `[[3|5],[7|7]]` denotes the set `{3,4,5,7}`, see the CLP(FD) section of [4] for details.

Explanations. The trace events presented in this section provide *explanations* [5] for other trace events, i.e. reasons for why certain actions are taken by the solver. These trace events are presented in Table 5.

First of all, we will make clear what we mean with explanations: An explanation e for some kernel action a is a logical formula e such that $e \Rightarrow a$, i.e. if e holds, a will occur. In Prolog, we represent an explanation with a pair, `N-CondList`, where `N` is an integer and `CondList` is a list of `cond(M,Pa,PL)` terms. An explanation e = `N-CondList` holds iff at least `N` of the `cond/3` terms in `CondList` hold. Each `cond/3` term contains an integer `M`, a path `Pa` and a list of properties `PL`. The property on index i in `PL` is associated with the entity on index i in the list of positions referred to by `Pa`. For example, assume that `Pa` = `[[1,3]]`, `PL` = `[eq(2),neq(3)]` and that `Pa` is associated with the `element/3` constraint in Ex. 1. Then `X` is associated with `eq(2)` and `Y` is associated with `neq(3)`. When `PL` contains a single property, it is associated with each entity referred to by `Pa`. Also, the list structure is skipped and `PL` will denote the value of

Table 5. Explanation trace events. All but push_demon_because contain the same type of information.

Trace Event	Attributes	Meaning
push_demon_because	EventId, Variables, Condition, Constraint	Generated before a push_demon trace event. Condition is a property fulfilled by the variables referred to by the path Variables. Condition ∈ {min, max, minmax, val, dom}.
prune_because, fail_because, new_ctr_because	EventId, ConstraintId, Explanation, Constraint	Generated before the corresponding[a] trace events. Explanation is an explanation or a reason for why the corresponding action occurs.

[a] Trivially, a prune_because trace event is for example generated before some trace events describing domain narrowing.

the single property. Possible properties in PL are, among others, eq(I), neq(I), inset(S) and notinset(S) where I is an integer and S is a finite set of integers. A cond/3 term holds iff at least M of the entities referred to by Pa fulfill their respective properties.

Now, let us give an illustrative example. Assume that

$e =$ 1-[cond(1,[1,#3],eq(3))]

explains the solver action a. Also, assume that e is associated with a pruning generated due to the all_different/1 constraint in Ex. 1. This gives us the information that a occurs since at least one of the cond/3 terms holds. For the only one, it holds since at least one of the entities referred to by [1,#3] can take the value 3, eq(3). In our case, this is the position marked with gray in all_different([X,Y,3,V1,8,V2]). In order for a not to occur, the least thing one must do is to change the value on that position to some value distinct from 3. Actually, this explanation occurs in the trace event

```
prune_because(15,ctr_all_different_1,1-[cond(1,[1,#3],eq(3))],ALLDIFF).
```

Hence, e is an explanation for removing the value 3 from all variables in the all_different/1 constraint.

The following are some more explanation trace events from Ex. 1:

```
new_ctr_because(24,ctr_element_1,1-[cond(1,[2]/length,eq(4))],ELEMENT)
...
prune_because(51,ctr_element_1,1-[cond(1,[3],notinset([[8|8]]))],ELEMENT)
```

The first one is generated before the begin_new_ctr trace event with id 25. It contains an explanation for posting the X in 1..4 constraint inside the creation of the element/3 constraint. The explanation is that the length of the list on the second position in the constraint is 4.

Similarly, the id of the last one tells us that it is generated before the before_prune trace event with id 52. It contains an explanation for pruning

the variable X. This too occurs inside the creation of the element/3 constraint. The explanation is that the third argument of the constraint, the variable Y, does not intersect with the set consisting of the single value 8. Due to this, X cannot take the value 4.

For some solver actions, there are several different reasons for why they occur. In order to show this, we need to introduce one more example.

Ex. 2 A failing all_distinct/1 constraint.

```
explain_me :-                                                    (1)
        domain([X1],1,6),                                        (2)
        domain([X2,X3,X4,X5],1,2),                               (3)
        all_distinct([X1,X2,X3,X4,X5]).                          (4)
```

The all_distinct/1 constraint on line (4) in Ex. 2 uses the complete filtering algorithm of Régin [11]. When, during the posting of that constraint, its demon is woken up, it will produce a failure since the variables X2, X3, X4 and X5 are all in the range 1..2. For this failure, we generate the following explanation:

```
1-[cond(all,[1,#[2,3,4,5]],inset([[1|2]]))]
```

where all is a keyword for identifying all entities referred to by the associated path. By looking at this explanation, we conclude that, in order for the failure not to occur, we need to enlarge the domains of any two of the variables X2, X3, X4 and X5 with at least two distinct values. We see this since there are 4 variables taking values in a set of integers of size 2 and $4 - 2 = 2$.

Filtering Algorithms. The trace events presented in this section describe the execution of the filtering algorithms associated with the demons. Since each filtering algorithm is built up by a set of methods, information about each active method is needed. Table 6 presents the relevant trace events.

Each prune block is surrounded by a method block, and possibly preceded by an info_method trace event. For the first example concerning domain modification, the following trace events are added:

```
begin_method(12,propagate_ground_variable)
  info_method(13,ground_variable,[1,#3]-3,ALLDIFF)
  ...
end_method(20,succeed)
```

By looking at the id:s, we see that they are generated before and after the prune block describing the removal of the value 3 from the variables X, Y, V1 and V2. The begin_method trace event contains a name of the active method. In this case, propagate_ground_variable denotes the method that removes a ground value from all non-ground domain variables in the all_different/1 constraint. Following that, the info_method trace event contains more detailed information

Table 6. Filtering algorithms trace events.

Trace Event	Attributes	Meaning
begin_method	EventId, MethodName	Generated when a demon's filtering algorithm activates a specific method. MethodName is a Prolog atom with a name corresponding to the consistency rule of the associated constraint.
end_method	EventId, Result	Generated when a specific method has finished. Result denotes the result of the prunings carried out inside the block. Similar to the Result attribute in the end_prune trace event.
info_method	EventId, InfoName, Info, Constraint	Generated inside a specific method. Contains more detailed information about entities involved in the filtering algorithm. InfoName is a Prolog atom with a descriptive name of the provided information. Info is a pair P-E containing the extra information, P is a path referring to the entity E.

about the actual entities involved in the method. More specifically, it tells us that the ground variable is on the position referred to by [1,#3] and its value is 3. Finally, the end_method trace event contains the result, succeed, of the whole method block. The result will be fail iff at least one failure inside the block is detected, otherwise succeed.

The second example concerning domain modification, the one associated with the element/3 constraint, is surrounded by the following method block:

```
begin_method(49,prune_x)
...
end_method(56,succeed)
```

3.3 Implementation

The described trace structure has been implemented in a research branch of the CLP(FD) solver in SICStus Prolog version 3.9.

Each trace event is represented by an abstract datatype (ADT) in C. Complex information, such as compound terms and lists, is represented using C strings. Each ADT has creation and deletion primitives as well as a primitive for converting the information to a Prolog fact.

On the Prolog side, it is possible to create and access the trace events through some predicates. For the creation, each trace event (with two exceptions[8]) have a corresponding constructor predicate. For retrieving, a predicate get_events(+Selector, -TraceEvents) is available. Selector is an integer, an atom or a compound term specifying a set of trace events. TraceEvents is a list of trace events corresponding to Selector.

[8] The trace events push_demon and push_demon_because are created at a central place within the C part of the kernel.

4 Discussion

Tracing CLP(FD) programs is still an open area. In our case, more testing is needed in order to evaluate the trace model. This includes building debugging tools, using the different kinds of information the trace provides.

One interesting tool to build would be a search-tree visualizer such as the Oz Explorer [12] by Schulte. Such a tool would explore the information in the trace concerning the control of execution aspect. Also, as was done for the Oz Explorer, one could connect to it other tools using a plug-in scheme, displaying various information concerning other aspects at the nodes of the search-tree. An example of such a tool for the Oz Explorer is the Constraint Investigator [10] by Müller.

Another tool, exploring the information concerning the domain modification aspect, would be a variable domain visualizer. Domains could for instance be visualized in a way similar to as in [2].

The information concerning the declarative aspect could be used by a tool for solving over-constrained problems. Explanations would help to point out the set of inconsistent constraints. Jussien and Ouis presented a tool like this in [6].

For educational purposes, a tool that provides information about filtering algorithms would be useful. Such a tool could connect the consistency rules of the constraints with the different methods in the corresponding filtering algorithms.

As a first use of the trace structure, we have implemented an extension to the Prolog debugger in SICStus. At the usual breakpoints of the debugger, it is possible to ask queries about the trace and the entities in it. The user may for example display certain trace events, ask queries about domain variables and generate explanations for specific domain narrowings.

5 Conclusion

In this paper, we presented new ideas in the area of tracing CLP(FD) programs. In addition to providing information about control of execution and domain modification, we also provide information concerning the declarative and procedural aspects. We did this by incorporating the idea of explanations in the trace, as well as coming up with new ideas about tracing filtering algorithms. By doing this, we have obtained a link between the consistency rules and the filtering algorithms in constraint solving.

Furthermore, we presented some new ideas about how to present the trace, both regarding the structure and the information contained. This includes building the trace as a nested block structure, as well as using a path for identifying the different entities in a CLP(FD) program.

Implementing the trace structure in a high-end system like SICStus is not an easy task. This is especially true for the task of adapting filtering algorithms to generate explanations. Sometimes, the added overhead it meant to provide the information needed for sharp explanations was too large. In this case, some *ap-*

proximation explanation[9] has been used. In the long run, providing explanations probably means that filtering algorithms need to be changed in order to, more naturally, provide the necessary information.

References

1. M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag, 1997.
2. M. Carro and M. Hermenegildo. Tools for Constraint Visualization: The VIFID/TRIFID Tool. In P. Deransart, M. Hermenegildo, and J. Małuszyński, editors, *Analysis and Visualisation Tools for Constraint Programming*, number 1870 in LNCS, chapter 10. Springer Verlag, 2000.
3. M. Ducassé. Opium: An Extendable Trace Analyser for Prolog. *The Journal of Logic Programming*, 39:177–223, 1999.
4. M. Carlsson et al. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, release 3 edition, 1995. ISBN 91-630-3648-7.
5. N. Jussien. e-constraints: explanation-based Constraint Programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, 2001.
6. N. Jussien and S. Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on Logic Programming Environments*, Paphos, Cyprus, 1 December 2001.
7. F. Laburthe. CHOCO: Implementing a CP kernel. In *Proc. TRICS: Techniques foR Implementing Constraint programming Systems*, pages 71–85, Singapore, 2000. CP2000 workshop.
8. L. Langevine, P. Deransart, M. Ducassé, and E. Jahier. Prototyping CLP(FD) tracers: a trace model and an experimental validation environment. In A. Kusalik, editor, *Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE'01)*, Paphos, Cyprus, 1 December 2001.
9. M. Meier. Debugging Constraint Programs. In *Principles and Practice of Constraint Programming*, volume 976 of *LNCS*, pages 204–221. Springer-Verlag, 1995.
10. T. Müller. Practical Investigation of Constraints with Graph Views. In R. Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming*, volume 1894 of *Lecture Notes in Computer Science*, pages 320–336, Singapore, 2000. Springer-Verlag.
11. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.
12. C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300. MIT Press, 1997.
13. H. Simonis and A. Aggoun. Search-Tree Visualization. In P. Deransart, M.V. Hermenegildo, and J. Małuszyński, editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *LNCS*, pages 191–208. Springer-Verlag, 2000.

[9] An explanation that is true but not necessarily the one that expresses the least dependency.