Tracing and Explaining the Execution of CLP(FD) Programs in SICStus Prolog

Magnus Ågren
magren@sics.se

July 2002

SICS
Lägerhyddsvägen 18
SE-752 37 UPPSALA
Sweden

Abstract

The increasing interest in Constraint Programming (CP) we now witness gives rise to a demand for new and improved debugging techniques. Graphical tools, such as constraint- and search-tree visualizers, seem to be appropriate to get a general understanding of the complex process of constraint solving. However, many such tools have been built in an ad hoc way, forcing the developer to, for each new tool, provide relevant information from the constraint solver.

In this thesis, we present a solution to the problem, limiting ourselves to Constraint Logic Programming over Finite Domains (CLP(FD)). In order to do this, we come up with a trace structure for describing the execution of CLP(FD) programs in detail. The trace structure consists of various trace events, each trace event containing different information depending on when in the solving process it is created. Among other things, the trace structure contains information about constraint posting, constraint awakening and domain narrowing. We also 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.

An implementation of the trace structure in SICStus Prolog is also presented, as well as a tool using the trace, an extension to the ordinary Prolog debugger.

Keywords: Constraint, Finite Domain, Debugging, Event Trace, SICStus, Prolog.
## Contents

1 Introduction .......................... 1

2 Background .................................. 1
   2.1 Constraint Logic Programming over Finite Domains (CLP(FD)) .................. 1
   2.2 Debugging CLP(FD) .................. 2

3 Our Work: An Overview ................. 3
   3.1 Problem Statement .................. 3
   3.2 Related Work .................. 4
   3.3 Our Approach .................. 4

4 A Trace Format for CLP(FD) ............ 5
   4.1 An Accompanying Example .................. 5
   4.2 A CLP(FD) Kernel .................. 5
   4.3 Defining Types in Prolog .................. 9
   4.4 Basic Types .................. 9
   4.5 Trace Structure .................. 10
   4.6 Identifying Constraints and Variables .................. 10
   4.7 Representing Constraints .................. 10
   4.8 Location of Variables .................. 10
   4.9 Complex Types .................. 12
   4.10 Events .................. 14
      4.10.1 begin_new ctr...end_new ctr .................. 15
      4.10.2 new_demon .................. 15
      4.10.3 push_demon .................. 15
      4.10.4 begin_wake_demon...end_wake_demon .................. 16
      4.10.5 begin_prune...end_prune .................. 16
      4.10.6 prune .................. 17
      4.10.7 before_prune, after_prune .................. 17
      4.10.8 fail .................. 17
      4.10.9 begin_method...end_method .................. 17
      4.10.10 info_method .................. 18
      4.10.11 labeling_start .................. 18
      4.10.12 labeling_step .................. 18
      4.10.13 labeling_fail .................. 18

5 Explanations .......................... 19
   5.1 Usage .................. 19
   5.2 Defining e .................. 19
   5.3 An Example .................. 20
   5.4 Constructing Explanations .................. 21
   5.5 Explained Actions .................. 21
   5.6 Explanation Events .................. 21
      5.6.1 push_demon because .................. 21
      5.6.2 new_ctr because .................. 21
      5.6.3 prune because .................. 22
      5.6.4 fail because .................. 22
   5.7 Explaining all different/1 (naive algorithm) .................. 22
   5.8 Explaining all_distinct/1 .................. 23
   5.9 Explaining assignment/2 .................. 26
   5.10 Explaining circuit/[1,2] .................. 27
   5.11 Explaining min/2 .................. 28
6 Implementation
   6.1 C Part ................................................................. 31
   6.2 Prolog Part ............................................................ 33

7 An Extension to the Prolog Debugger .................................. 33
   7.1 Pretty Printing of Events ........................................... 33
   7.2 Added Commands ..................................................... 34

8 Discussion ........................................................................ 36
   8.1 Trace Format ............................................................ 36
   8.2 Explanations ............................................................. 37
   8.3 Implementation .......................................................... 37

9 Conclusion .................................................................. 37

A Graph Terminology ......................................................... 39

B Trace Structure (BNF) ....................................................... 40

C Tracing the trace_me Example ........................................... 41
   C.1 The Program ............................................................. 41
   C.2 The Trace ............................................................... 41
1 Introduction

Constraint Logic Programming, CLP, is a relatively young programming paradigm which has proved to be useful for solving complex industrial problems. Until recently, there has been a serious lack of debugging tools for developing CLP programs. Traditional tools, usually presenting a changing state and source code position, do not fit well with CLP. This is mainly due to the complexity of constraint solving and the number of variables typically involved in the problem. Furthermore, CLP is very close to the black box way of thinking, meaning that the user specifies a problem and the system solves it for him. This usually involves running complex solving algorithms written in some imperative programming language in order to gain speed.

A basic property of any debugging tool for any programming language is to provide useful information by describing what is going on during execution of the program. In CLP, one way of doing this is to have the system generate a low-level trace of events where each event describes some action taking place. Based on this trace, different tools may extract and present different sorts of relevant information. Constructing such a trace gives rise to several questions. How much and how detailed information do we need to provide in order to satisfy ‘all’ possible debugging tools? What kind of information format should we use and how hard will it be to provide the information?

This thesis describes an implemented trace format for the CLP(FD) solver in SICStus Prolog. Different actions performed by the solver are expressed by different types of events. These are actions like posting new constraints to the system, waking up constraints for propagation, and reducing domains of variables. Another problem that has been addressed is how to explain why actions like these occur.

In the following, Section 2 gives some background information on the subject. Section 3 provides a general overview of what the thesis contains. We discuss the approach, our contribution and compares it with related work. Section 4 presents the trace format. First, a very basic CLP(FD) kernel model is presented; this is useful in order to be able to present the trace format in its context. Following that, we discuss different design issues before presenting the events one by one. Section 5 introduces explanation events. First of all, we mention some possible areas of usage. Following that, the format of the explanations used in the trace is defined. An example is also given before presenting explanations for specific actions. Section 6 gives a rough description of the implementation. Section 7 presents a tool using the trace, an extension to the Prolog debugger in SICStus. Finally, Section 9 discusses results and possible future work.

2 Background

2.1 Constraint Logic Programming over Finite Domains (CLP(FD))

Constraint Programming [Hen 89], CP, is a relatively new programming paradigm that has gained increasing popularity over the past years. Some of the fields of usage are resource allocation, scheduling and optimization problems. Constraints are common in everyday life, one example is a time constraint like “I must be at the airport before 9 o’clock”. We use this kind of constraints to plan our everyday life with different types of appointments etc. More formally, a constraint is a relation between some variables, restricting them to taking certain values in a given domain. While in this document, we will consider variables with finite integer domains, a constraint relation may be defined between variables of any domain such as rational or real numbers.

As a first example, assume that we have a set of variables \( \{X, Y, Z\} \) with the domains \( D_X = \{1, 2, 3, 4\} \), \( D_Y = \{2, 3\} \), \( D_Z = \{2, 5, 8\} \). One constraint between these variables is:

\[
X < Y \land Y = Z.
\]

In the given domains, this constraint has only one solution, \( \{X \mapsto 1, Y \mapsto 2, Z \mapsto 2\} \). One technique, called generate and test, for finding a solution is to simply search for it. This means assigning a value to each variable from their respective domains and check if the constraint holds. If it holds, a solution has been found. If it doesn’t hold, some other values are tried until all combinations have been tested. Another technique is to use the property of the constraint to, a priori, exclude certain values from the domains of the variables. By looking at the above example, we can conclude from \( Y = Z \) that all values not in the intersection
of \( Y \) and \( Z \) should be removed from these variables. This will narrow the domains of these variables to \( \{2\} \).
\( X < Y \) tells us that all values in \( X \) must be less than the maximum value of \( Y \) which will narrow the domain of \( X \) to \( \{1\} \), and a solution is found. This is called constraint propagation. A third technique, which is used by most constraint systems, is a combination of the two previous ones.

Constraint Programming over Finite Domains, CP(FD), is possibly the most useful instance of constraint programming. Many industrial problems like scheduling, routing, and timetabling can naturally be modeled with finite integer domain variables and constraints on these. As an example, consider the timetabling problem of an airline where different flights are assigned to different air crews. An optimized solution to this is to have minimal number of air crews working at the same time and minimal number of air crew seats on the planes, i.e. seats allocated by crew members on their way home or to some job flight. CP(FD) libraries have been written for several existing programming languages. Examples include ILOG\(^*\) solver for C++, Choco\(^{\dagger}\) constraint programming system for Claire and FaCLe\(^{\ddagger}\) constraint library for OCaml. There are also constraint libraries for the Java language. Bringing CP(FD) to popular industry languages like C++ has greatly improved its usage.

Early constraint programming languages of the 80's were extensions to logic programming languages, and it has shown to be a successful combination. Two main reasons for this is the declarative property of logic programming and its backtracking feature. The declarative property closely relates to the idea of constraints, where you can say that the emphasis lies on stating what kind of problem to solve instead of how to solve it. The backtracking feature of logic programming languages is very useful when performing search. Some logic programming systems with an integrated constraint solver is SICStus\(^{\$}\) Prolog, ECL\(^{\$}\)PS\(^{\$}\)

Constraint Logic Programming System and CHIP\(^{\|$}\).

A CLP(FD) system is a Prolog system extended with finite domain variables and some built-in primitive and global constraints that can be posted on the variables. Furthermore, there is usually some way of adding user-defined constraints, both primitive and global. A primitive constraint is a relation between a fixed number of domain variables or integers. One example is the ‘\(<\)’ relation. A global constraint is a relation between a non-fixed number of domain variables or integers. One example of a global constraint is all_different(Xs) where Xs is a list of domain variables or integers. all_different(Xs) holds if all elements in Xs take unique values. Some built-in predicate for labeling, searching for a solution by assigning values to variables, is usually also present in a CLP(FD) system.

A typical CLP(FD) example is the SEND + MORE = MONEY problem which constrains the variables S, E, N, D, M, O, R, Y to take different values in the range 0..9 such that the relation SEND + MORE = MONEY holds and that the leftmost digits of each number is different from 0. One way of solving this in SICStus Prolog is shown in Example 1.

Line (1) loads the finite domain constraint library; lines (2) - (3) declare and initialize the domains of the variables; lines (4) - (9) post the constraints and line (10) calls the predefined predicate labeling/2 to assign a value to each variable of L. A call to sm(\([S,E,N,D,M,0,R,Y]\)) gives the unique solution

\[
S=9, \ E=5, \ N=6, \ D=7, \ M=1, \ O=0, \ R=8, \ Y=2.
\]

For more information about constraint programming, see [MS 98]. There are also many online resources; one good starting point is the Constraints Archive located at http://www.cs.unh.edu/ccc/archive/.

### 2.2 Debugging CLP(FD)

Debugging a CLP(FD) program requires an approach different from traditional debugging techniques, mainly due to the complexity of constraint solving and the number of variables typically involved in the problem. Displaying a changing state and source code position as is commonly done for imperative programming languages is often not practical for a CLP(FD) program, since the number of variables will make it very

---

\(^{\ast}\) http://www.iilog.com

\(^{\dagger}\) http://www.choco-constraints.net

\(^{\ddagger}\) http://www.recherche.enac.fr/opti/facile

\(^{\$}\) http://www.sicstus.com

\(^{\|$}\) http://www.icparc.doc.ic.ac.uk/eclipse

\(^{\|$}\) http://www.candytec.com, CHIP is also available as a C/C++ library.
Example 1 The SEND MORE MONEY problem.

:- use_module(library(clpfd)).

\[ smm(L):- \]
\[ L = [S,E,N,D,M,O,R,Y], \]
\[ domain(L, 0, 9), \]
\[ S \#> 0, \]
\[ M \#> 0, \]
\[ all\_different(L), \]
\[ 1000 \times S + 100 \times E + 10 \times N + D \]
\[ + 1000 \times M + 100 \times O + 10 \times R + E \]
\[ \#= 10000 \times M + 1000 \times O + 100 \times N + 10 \times E + Y, \]
\[ labeling([], L). \]  

hard to filter out relevant information. Also, the complexity of the solving algorithms may make it hard to understand certain state changes.

Debugging can be divided in two different areas: correctness- and performance debugging. Correctness debugging concerns everything that has to do with the correctness of a program. This may be further split up into static source code checking and run-time checking of results produced by the program. Performance debugging concerns the problem of finding the performance bottlenecks in a program. In CLP(FD), performance debugging plays a major role since different modeling techniques and search strategies may influence the performance drastically.

The main task for any debugging tool, whether used for finding errors or for improving performance, is to increase the understanding of what is going on inside the program. For CLP(FD), this concerns things like displaying the control of execution and the different entities in the constraint store in a way easily overviewed by a human. Different users may have various demands resulting in several tools displaying similar things from different viewpoints.

3 Our Work: An Overview

This section starts with stating the problems we deal with in the thesis. After that, we mention some previous and other ongoing work in the same area. Finally, our approach is described, different choices we made are discussed and compared to other work.

3.1 Problem Statement

We address the problem of providing relevant information for building CLP(FD) debugging tools. Many different debugging tools that aid people in developing C(L)P programs have been proposed and implemented. Some examples include the Oz Explorer [Sch 97] in Mozart*, different constraint- and search-tree visualizers in CHIP [DHM 00], and the Grace debugging tool [Mei 95] in ECLIPS*. A common problem when developing such tools is that information from within the constraint solver must be made accessible. This concerns information about domain variables, constraints on these variables and actions taken by the solver such as constraint awakening and variable domain narrowing. If such information could be made accessible in some general way, implementing debugging tools would be less costly. This is the first problem we deal with in the thesis, limiting ourselves to CLP(FD) systems1. Another important issue when it comes to debugging is that of understanding the execution of a program, why certain things happen. This can be very hard to achieve in C(L)P since usually, the filtering algorithms of the constraints are not known in full detail to the users. This is the second problem we address. That is, in addition to providing information about what happens inside the solver, to also provide explanations about why certain actions occur.

* http://www.mozart-oz.org

1Mozart is for example a non Prolog CP system.
3.2 Related Work

The DiSCIPI\* project investigated different ways of debugging CP programs. Several tools for various systems were implemented as a result of the project. However, many debugging tools have been built in an ad hoc way, forcing the developers to, for each new tool, provide the information from the constraint solver. This has the effect that implementing each tool is very time-consuming. Things would be less complicated if one could provide the needed information in a format acceptable by ‘any’ possible tool. We have chosen to do this through a low-level trace of events, where each event provides relevant information about the actions taken by the constraint solver.

This idea is not entirely new; the search-tree visualization tool in CHIP, presented in [SA 00], uses a trace log to be able to display propagation events. More current work is carried out within the OADyMPaC\* project. Langevine et al. [LDDJ 01b] presented a trace model for CLP(FD) as well as an implementation of an interpreter and a trace-analyzer in ECL\*PS\* for experimenting with the model. Their trace model consists of eight different event data structures, each containing information about specific actions in a CLP(FD) kernel. The implementation of the trace-analyzer is naturally influenced by the work of Ducassé [Duc 99] on Opium.

Jussien and Barichard [JB 00] introduced the PaLI system, a CP system based on explanations. These explanations are reasons for why certain actions occur in the constraint solver. They provide contradiction- and elimination explanations, i.e. reasons for a ‘no solution’ answer and reasons for removing values from domain variables. This is achieved by modifying the Choco constraint solver with added code that provides explanations within the solving algorithms. However, there is nothing published about explanations for global constraints. Furthermore, they have proposed several usages of explanation-based constraint solving. These include using explanations to achieve dynamic backtracking [JDB 00] and dynamic problem solving [JO 01].

3.3 Our Approach

Our idea is to provide the information needed to build CLP(FD) debugging tools through a low-level trace of events. The trace of events is a list of Prolog terms containing different information depending on where in the constraint solver they are created. As an introductory example, consider the pruning of the variable \( X \) in the constraint \( \text{element}(X, [V1, V2, V3], Y) \). Assume that the value 2 is removed. In our model, this will be expressed by something like:

\[
\text{prune}(11, \text{ctr-element}_3, [1], \text{remove-value}(2), \text{element}(x, [v1, v2, v3], y), \text{succeed}).
\]

The \text{prune/6} term contains information about the constraint responsible for the pruning, the variable that is pruned, the value that is removed, and the result of the pruning. The different events are presented in detail in Section 4.

We had the following goals when designing the trace format:

- The trace format should contain enough detailed information to satisfy the demands of most CLP(FD) debugging tools.
- Navigating through the trace and finding relevant information should be easy and fast.
- The memory usage of the trace should be as small as possible.

We have chosen to store an exhaustive trace, i.e. all events are stored and never deleted during program execution. This may be a problem when tracing large applications. However, it increases the capabilities of doing post-execution analysis as well as the possibilities for going backward in the trace history.

The trace is built up like a block structure. Several of the solver actions give rise to a set of events enclosed in an opening- and a closing event. All events inside such a block correspond to the action. An example of this is the pruning of a domain variable which creates a set of events enclosed in a \text{begin-prune} event and an \text{end-prune} event. The \text{prune/6} event above will always appear between two events like these. There are

\*DiSCIPI is a two-and-a-half-year long European collaboration between universities and industry. See [DHM 00] or http://discipl.inria.fr for details.

\*OADyMPaC is a French research project with members from both industry and the research community.
several benefits from this approach. First of all, the whole trace will be structured and easy to overview even for a human. 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. A negative consequence of building the trace as a block structure is the increased number of events this means.

Regarding explaining certain solver actions, we have chosen to do this by adding extra events in the trace. An explanation event is an event located before some other event or block of events. This explanation event provides some information about why the following event(s) exist. We provide explanations not only for domain pruning and failure but also for constraint awakening and constraint adding. We extend the work of Jussien et al. by presenting explanations for some global constraints. Section 5 presents explanations in detail.

4 A Trace Format for CLP(FD)

In order to create any debugging tool, information of what is going on inside the constraint solver must be made accessible at the user-level. In the case of the SICStus Prolog constraint solver, we provide this through a trace of events that are generated at different stages in the solving process. The trace is a list of Prolog terms containing different information depending on when in the solving process they are created. Relevant information for a debugging tool can later be extracted from this list.

We start this section by introducing an example that will be used throughout the report. Following that, we present 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 events in a relevant context. After that, some of the information needed to understand the format of the trace events is presented. Last of all, the events are listed and explained.

4.1 An Accompanying Example

An example of a simple CLP(FD) program in SICStus Prolog is shown in Example 2. It will be used throughout the paper.

Example 2 A simple CLP(FD) Program.

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

Line (2) initializes the domains of the included variables. Lines (3) and (4) post the well-known global constraints all_different/1 and element/3. Line (5) calls SICStus's builtin search predicate with a leftmost search strategy.

4.2 A CLP(FD) Kernel

A CLP(FD) kernel manages a constraint store containing domain variables and constraints on these variables. The different constraints wake up when changes in the domains of their variables are noticed. This activates filtering algorithms which, according to consistency rules of the constraints, remove values from the domains of the variables that cannot be in a solution to the constraints. If, at any time during this process, the domain of any variable gets empty, a failure is reported by the filtering algorithm. The filtering algorithms

---

*all_different(Xs), where Xs is a list of domain variables or integers, is true if each element in Xs take distinct values.

*element(X,Xs,Y), where X, Y and the elements of Xs are domain variables or integers, is true if X is an index in Xs and Y is the value on that position.

*leftmost variable, smallest value first.
achieve this by generating propagation events. A propagation event contains information about narrowing the domain of a variable. After a filtering algorithm has run, the kernel performs the actions specified by the generated propagation events. If this results in domain changes for any variable, the constraints that that variable is associated with may be scheduled for waking up. The rest of this section presents the different parts of the kernel in more depth, and also how they interact with each other. The model is influenced by the Choco [Lab 00] and the SICStus Prolog [COC 97] constraint kernels.

Constraints

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. Furthermore, each constraint contains a status attribute, one of the following:

- **active**, at least one of the constraint’s demons is active, i.e. its filtering algorithm has been activated.
- **suspend**, the constraint is in the system but currently no domain narrowing is possible. (According to the implemented consistency rules.)
- **entail**, the constraint is true whatever values its variables may take.
- **fail**, at least one variable in the constraint has an empty domain.

Variables

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.

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.

Propagation Events

The propagation events in our kernel model have the form

\[ X \text{ in } \text{set } S \]

and constrains \( X \) to be a member of the set, \( S \), of integers. A real system would probably have more propagation events, expressing special cases of the above. An example is the propagation event \( X = V \) which constrains \( X \) to take the value \( V \). This is equivalent to \( X \text{ in set } \{V\} \). It may then be useful to perform the narrowings described by such events before the ones of the type above due to efficiency reasons.

Data Structures

The data structures listed below are used by the kernel to store and organize constraints, variables, demons and propagation events.

- **VariableList**, list containing all domain variables currently in the system.
- **ConstraintList**, list containing all constraints in the system.
Figure 1: Kernel state for Ex. 2. Just before the execution of line (4).

- ReadyQueue, queue containing demons that are about to wake up.
- PropagationQueue, queue containing propagation events created by woken demons.

Figure 1 shows the different data structures and how they are connected to each other. The state is taken from Example 2, just before the execution of line (4), i.e. the element/3 constraint is to be added to the system. C1 and C2 are the all_diferent/1 and element/3 constraints respectively. Detailed information about variables X and Y as well as about constraint C2 is shown in three boxes.

These data structures are manipulated through different services or functions described below. Everything in this model is kept very simple. In a real system, there would probably exist different queues for different types of demons and propagation events. Also, for simplicity, the queues use first in first out functionality when enqueuing and dequeuing elements. For a more efficient and detailed kernel model, see [Lab 00].

Services
The kernel provides the following services or functions:

- 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 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 about 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.
The function `dispatchEvent()` has higher priority than the function `dispatchCtr()` in the sense that if both are possible to execute, `dispatchEvent()` will be the first one executed.

**Labeling**

When both queues are empty, the kernel cannot perform any more propagation. Does this mean that if a solution to the problem has not been found, it will never be? No, this is where labeling or search takes over if it is implemented by the program.

Labeling means assigning a value to each variable in `VariableList` from their respective domains until a solution has been found or all possible combinations have been tested. An assignment like this is performed by adding a propagation event `\( \langle X \text{ in } \text{set} \{V\} \rangle \) ` where \( V \in \text{dom}(X) \) to `PropagationQueue`. The narrowing specified is carried out by the kernel, and this will cause all constraints’ demons that \( X \) is associated with to wake up. If a failure is detected during labeling and there are still values that have not been tried, the narrowings that have been done are rolled back to some previous state from which the search can continue. In order to do this, some of the data structures need to be backtrackable*. The different backtrackable data structures in our kernel model are:

- **ConstraintList**, it must be possible to remove constraints that have been added and vice versa.
- **VariableList**, it must be possible to remove variables that have been added and vice versa.
- **Domain**, it must be possible to add values that have been removed.
- **Constraint**, it must be possible to restore old values in the status attribute.

The other data structures only contain temporary data which does not need to be restored.

**The Active Kernel: 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 Figure 1. There is no propagation event to be performed, but the demon D2 of constraint C2, the \( \text{element}/3 \) 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'. 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.

**SICStus Prolog Notes**

A feature of the kernel model described above not present in SICStus Prolog is for the constraints to have more than one demon. Since all examples are created and executed within SICStus Prolog, more than one demon will never be shown.

---

* A data structure being backtrackable simply means that earlier states of it can be restored.

† Since the value 8 on position 4 in the list is not in the domain of Y.
4.3 Defining Types in Prolog

Even though Prolog is a programming language where you don't have to declare any types in the code, it is useful to be able to declare types in some way for documentation and specification issues. We will use the notation below when we describe the contents of the different trace events.

A type can be thought of as defining a set of values. This set may be finite or infinite. One example is the constant type integer which defines the set \{\ldots, -2, -1, 0, 1, 2, \ldots\}. The other constant types used in Prolog are float and atom. All these denote infinite sets\(^*\). In order to define new types, we will use the following rule:

\[
:~:~\text{type } <\text{name}> \equiv <\text{contents}>.  
\]

(1)

where <name> is an atom or term and <contents> is a set expression defining the new type. <contents> may be

a. a constant type, integer, float or atom.

b. another defined type using rule (1).

c. a variant of the type <name> (recursive type).

d. a term with any of the above as arguments (compound type) prefixed by a back-quote sign. (Both prefix and infix notation of term functors may be used, infix terms are enclosed in parenthesis.)

e. any member of the above types prefixed by a back-quote sign.

f. any union of the above using ; as union symbol.

Assume for instance that we want to define the type containing the terms t(1,1), t(1,2),\ldots, t(3,2), t(3,3). We do this with the following rules:

\[
:~:~\text{type oneTwoThree} \equiv '1; '2; '3.
\]

\[
:~:~\text{type t_two_123} \equiv 't(oneTwoThree, oneTwoThree).
\]

As another example we define the type containing integers and the atoms inf and sup standing for minus and plus infinity:

\[
:~:~\text{type integer_inf} \equiv \text{integer}; 'inf; 'sup.
\]

Recursive types are defined using type variables. A legal type variable is any Prolog variable. As an example, we define Prolog lists using the following rule:

\[
:~:~\text{type list(E)} \equiv '[E|list(E)]; ['\].
\]

where E is a member of any type. Type variables are useful not only when defining recursive types. One more example of the use of type variables is

\[
:~:~\text{type pair(A,B)} \equiv '('A-B').
\]

where A and B are members of any type.

4.4 Basic Types

We define the following basic types:

- integer, the set of integers.

- atom, the set of valid Prolog atoms.

- ground, the set of all compound Prolog terms with no variable argument.

- any, the union of all valid Prolog types.

More complex types will be introduced later in order to present the trace events.

\(^*\) Actually, this is true in theory only. In practice, all types are finite.
4.5 Trace Structure

The trace has been designed as a nested block structure with some exceptions. Most actions taken by the kernel give rise to a block of events having an opening begin event and a closing end 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 pruning 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. For example, a tool that visualizes the trace may have the functionality of collapsing and expanding entire blocks. 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.

The full definition of the structure of the trace, i.e. what events may appear where, can be found in Appendix B available as a BNF grammar.

4.6 Identifying Constraints and Variables

There are mainly two entities appearing in a CLP(FD) program that we are interested in, constraints and domain variables. Each of these must be uniquely identifiable.

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

\[
\begin{align*}
\{X \mapsto \text{fdvar}_1, Y \mapsto \text{fdvar}_2, V1 \mapsto \text{fdvar}_3, V2 \mapsto \text{fdvar}_4, \\
C0 \mapsto \text{ctr_domain}, C1 \mapsto \text{ctr_all_diff}, C2 \mapsto \text{ctr_element}\}.
\end{align*}
\]

It may also be useful to have user-defined identifiers. However, this is more an implementation issue.

4.7 Representing Constraints

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 Example 2 for instance, the constraint

\[
\text{all_different([X,Y,3,V1,8,V2]})
\]

is represented as

\[
\text{all_different([fdvar}_1, \text{fdvar}_2, 3, \text{fdvar}_3, 8, \text{fdvar}_4]).
\]

4.8 Location of Variables

Which Variable is Which?

It is not enough to be able to identify variables by their unique names. While this makes it possible 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. For instance, consider the constraint

\[
\text{element}(X, [X1, X2], X)
\]

which is represented as

\[
\text{element}(\text{fdvar}_1, [\text{fdvar}_2, \text{fdvar}_3], \text{fdvar}_1)
\]

assuming that X, X1, X2 is assigned the names fdvar_1, fdvar_2, fdvar_3 respectively. Given the identifier fdvar_1 we cannot decide whether that refers to the first or the second occurrence of X in element/3.

Also, using only identifiers will not make it possible to identify other entities such as integers or Prolog lists, since these are not given any identifiers.

*Unique for the specific type of constraint.
A Solution - the path Type

A member of the path type is a Prolog term that uniquely determines a position or a set of positions in a constraint. When referencing a variable, for example when displaying a variable’s domain, a path term referring to that variable is given instead of its id. This term must always have a context, a corresponding constraint, to be interpreted in. The path type is defined by the following rules:

\[
\begin{align*}
\text{:- type path} & \text{ == ppath; } \text{’(ppath\backslash ppath).} \quad (1) \\
\text{:- type ppath} & \text{ == list(selector).} \quad (2) \\
\text{:- type selector} & \text{ == sselector; } \text{’#sselector.} \quad (3) \\
\text{:- type sselector} & \text{ == integer; list(integer); } \text{’[star].} \quad (4) \\
\text{:- type star} & \text{ == ’*.} \quad (5)
\end{align*}
\]

A member of the path type is represented as a Prolog list containing (lists of) fixed values or other path terms. The first element of a path term identifies the topmost position in the corresponding constraint. For example, any path term referring to some position(s) in the element/3 constraint in Ex. 2 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 path terms with only one element and the position(s) they refer to.

The i+1:st element of a path term 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 set of entities 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 type 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*. Line d in Table 1 illustrates this.

<table>
<thead>
<tr>
<th>Path</th>
<th>Position(s)</th>
<th>Path</th>
<th>Position(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 3]</td>
<td>element(X, [2, 4, 3, 8], Y)</td>
<td>[2, #[1, 2]]</td>
<td>element(X, [2, #3, 8], Y)</td>
</tr>
<tr>
<td>[2]</td>
<td>element(X, [2, 4, 3, 8], Y)</td>
<td>[2, #[*]] \ [2, #1]</td>
<td>element(X, [2, #3, 8], Y)</td>
</tr>
</tbody>
</table>

Extending the path Type

It is useful to be able to designate certain properties of the entities a path term can refer to. The path_function type extends the path type with a fixed set of functions that are applied to whatever a path term may refer to. It is defined by the following rules:

\[
\begin{align*}
\text{:- type function} & \text{ == ’min; ’max; ’range; ’size; ’length; ’arity.} \quad (6) \\
\text{:- type path_function} & \text{ == path; } \text{’(path/function).} \quad (7)
\end{align*}
\]

where min, max, range, and size are restricted to path terms referring to a set of domain variables or integers; length is restricted to path terms referring to a set of Prolog lists; and arity is restricted to path terms referring to a set of non-variable terms. If the path term refers to a set of n entities, the path_function term will return a set of n values. The different members of the function type have the following meaning:

- **min**, minimum value of a domain variable.
- **max**, maximum value of a domain variable.
- **range**, the difference between the maximum and the minimum values of a domain variable.

*In our case, the sets are lists.*
• size, the number of possible values for a domain variable.

• length, the length of a list.

• arity, the number of arguments in a compound term.

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

<table>
<thead>
<tr>
<th>PathFunction</th>
<th>Constraint</th>
<th>Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. [1]/length</td>
<td>all_different([X,Y,3,Y1,8,Y2])</td>
<td>6</td>
</tr>
<tr>
<td>b. [1]/min</td>
<td>element(X,[2,4,3,8],Y)</td>
<td>1</td>
</tr>
<tr>
<td>c. [[1,3]]/max</td>
<td>element(X,[2,4,3,8],Y)</td>
<td>[6,6]</td>
</tr>
</tbody>
</table>

4.9 Complex Types

In order to be able to express the different trace events, in this section, we define the types of their different arguments. It may be useful to keep this section as a reference when reading Section 4.10.

Prolog Data Structures

First of all, we define some common Prolog data structures.

```prolog
:- type list(T) == '[T|list(T)]; '[].
:- type pair(A,B) == '(A-B).
:- type opt(T) == T; 'none.
:- type value_path == pair(opt(path_function), any); path_function.
```

• list(T) is the Prolog list containing elements of any type T.

• pair(A,B) is the pair of elements of any types A and B with the separator ‘-‘.

• opt(T) is the option type which contains an element of any type T or the atom none.

• value_path contains either a path_function term or an any term, or both. In the last case, the any term is the entity referred to by the path_function term.

Finite Domains

Now, we define the types range and fd_set which are used to describe domains of variables.

```prolog
:- type range == '[integer|integer].
:- type fd_set == list(range).
```

A range term, [L|U], where L and U are members of the integer type contain all values in the range L..U. A domain, fd_set term, is simply a list of ranges. Domains can be represented in many ways, we have chosen this one mainly because it is how they are represented in SICStus Prolog.

Results

We go on by defining the different result types we will use: result, constraint_result, and demon_result. They are used to describe the result of different actions in the kernel.

```prolog
:- type constraint_result == 'fail; 'delay; 'entail.
:- type demon_result == constraint_result; 'succeed.
:- type result == 'fail; 'succeed.
```
The *constraint_result* type expresses the result of adding a new constraint to the kernel; the different members have the following meaning:

- *fail*, adding the constraint was not successful.
- *delay*, adding the constraint was successful, but the constraint is not entailed and therefore more propagation is possible in the future.
- *entail*, adding the constraint was successful, no more propagation is possible and the constraint exits from the system.

The *demon_result* type expresses the result of waking up a demon; the different members have the following meaning:

- *fail*, some propagation performed by the demon led to a failure.
- *delay*, more propagation is possible for this demon.
- *entail*, no more propagation is possible and this is the last demon alive for the constraint it is associated with.
- *succeed*, no more propagation is possible for this demon but there are still demons left for the associated constraint.

The *result* type expresses success or failure for single events or for entire blocks of events. The different members have the following meaning:

- *fail*, for a single event this means that the action it describes was not successful. For a block of events it means that at least one action within the block was not successful.
- *succeed*, for a single event this means that the action it describes was successful. For a block of events it means that all actions within the block were successful.

### Demon Waking Conditions

The next two types define the different conditions for waking up a delayed demon.

```prolog
:- type wake_property == ‘min; ‘max; ‘minmax; ‘val; ‘dom.
:- type wake_condition == pair(path, list(wake_property)).
```

The *wake_property* type is defined by the different atoms describing properties of variables that must change in order to wake up their associated demons. The *wake_condition* type associates a set of variables referred to by a *path* term with a list of *wake_property* terms. A change to any of the properties for any of the variables will cause the demon to wake up. The different members of the *wake_property* type have the following meaning for some variable *X*:

- *min*, the demon is woken if the minimum value of *X* is changed.
- *max*, the demon is woken if the maximum value of *X* is changed.
- *minmax*, the demon is woken if any of the minimum or the maximum value of *X* is changed.
- *val*, the demon is woken if *X* is pruned to a singleton domain.
- *dom*, the demon is woken if there is any change in the domain of *X*.
Domain Narrowings

The following types describe different kinds of domain narrowings.

```prolog
:- type pruning == 'remove_interval(integer, integer);
    'remove_value(integer);
    'remove_values(fd_set).
:- type intention == pruning;
    'unify_vars;
    'adjust_min(integer);
    'adjust_max(integer);
    'fix_to_value(integer);
    'force_in_interval(integer, integer);
    'force_in_values(fd_set).
```

The `pruning` type is a subset of the `intention` type. The different members of the types have the following meaning; \( L, U \) and \( V \) are members of the integer type; \( S \) is a member of the \( fd_set \) type:

- **remove_interval** \((L, U)\), remove the interval \( L..U \).
- **remove_value** \((V)\), remove the value \( V \).
- **remove_values** \((S)\), remove the set of values in \( S \).
- **unify_vars**, unify two domain variables.
- **adjust_min** \((V)\), adjust the minimum value to \( V \).
- **adjust_max** \((V)\), adjust the maximum value to \( V \).
- **fix_to_value** \((V)\), fix value to \( V \).
- **force_in_interval** \((L, U)\), remove the set of values not in the interval \( L..U \).
- **force_in_values** \((S)\), remove the set of values not in the set \( S \).

A pruning term describes what values are actually removed from the domain while an intention term describes more naturally what is done to the domain. As an illustration, consider the constraint

\[ X < Y \]

and assume that, initially, \( X \) and \( Y \) both have the domain \( 1..3 \). The consistency rules of the constraint tells us that the maximum value of \( X \) and the minimum value of \( Y \) should be adjusted to 2. So, the intention of the pruning is \( \text{adjust\_max}(2) \) and \( \text{adjust\_min}(2) \) for \( X \) and \( Y \) respectively. However, what actually happens is that the value 1 is removed from \( X \) and that the value 3 is removed from \( Y \). This gives us the pruning terms \( \text{remove\_value}(1) \) and \( \text{remove\_value}(3) \) for \( X \) and \( Y \) respectively.

### 4.10 Events

Listed below are almost all the events that may appear in the trace. The exceptions are explanation events, which are presented in Section 5. Blocks' opening and closing events are presented together. All event attributes are followed by a double colon, `::`, and a type term, denoting the type of the attribute. For example, `VarName::atom` denotes that `VarName` is of type `atom`. The kernel model of Section 4.2 will be referenced to present the events in their correct context.

Appendix C contains a complete trace of Example 2. It may be useful to keep that as a reference when reading this section.

14
4.10.1  begin_new_ctr...end_new_ctr
This block contains information about what actions occur when posting a new constraint to the constraint
kernel. It is created when connectCtr(C) is called by the kernel for some new constraint C. It contains events
describing the posting of demon(s) for the new constraint, and possibly events describing initial propagation.
- begin_new_ctr(EventId::integer,ConstraintId::atom,Constraint::ground)
This is the starting event of the block. A begin_new_ctr event is created when a new constraint is to be
added to the system, i.e. when entering the connectCtr(C) function.
  EventId, the unique event number of this event.
  ConstraintId, the unique id* of the new constraint.
  Constraint, the constraint†, with variables substituted by their given names.
- end_new_ctr(EventId::integer,Result::constraint_result)
The end_new_ctr event is created when demons for the constraint have been created and possible initial
propagation has finished. It is created just before the function connectCtr(C) exits.
  EventId, the unique event number of this event.
  Result, the result of adding the new constraint as described in Section 4.9.

4.10.2  new_demon
- new_demon(EventId::integer,DemonId::atom,DemonType::atom,
  WakeCond::list(wake_condition),Constraint::ground)
This event is created inside a new_ctr block. There might be several demons attached to one constraint and
in that case several new_demon events will appear inside the block.
  EventId, the unique event number of this event.
  DemonId, the unique id of the new demon. In the examples we show this will be the same as the id of the
  associated constraint. This is a simplification in our implementation since SICStus only creates one
demon for each constraint.
  DemonType, a Prolog atom with a descriptive name.
  WakeCond, a list of wake_condition terms holding the properties that must change for each of its variables
  in order to wake up the demon for propagation.
  Constraint, the constraint due to which this demon is created.

4.10.3  push_demon
- push_demon(EventId::integer,DemonId::atom,DemonType::atom)
A push_demon event is generated when a wake_property of at least one variable in a constraint has been
changed and the demons that should wake up due to this are put in ReadyQueue. This event corresponds
to the function enqueueCtr(C,E) where C is the constraint that the demon is associated with and E is the
propagation event responsible for pushing the demon.
  EventId, the unique event number of this event.
  DemonId, the unique id of the pushed demon.
  DemonType, the type of the pushed demon.

* As specified in Section 4.6.
† As specified in Section 4.7.
4.10.4 begin_wake_demon...end_wake_demon

This block contains information about what actions occur when a demon is woken up for propagation, i.e. when its filtering algorithm is activated. If any actual pruning is possible this block will contain prune blocks, possibly surrounded by method blocks. This block is created when the function dispatchCtr() is called.

- begin_wake_demon(EventId::integer,DemonId::atom,DemonType::atom)

This is the opening event of the block. It contains information about the demon that is waking up. It is generated when entering the dispatchCtr() function of the kernel, after dequeuing the topmost demon from ReadyQueue.

eventId, the unique event number of this event.

DemonId, the unique id of the demon that is waking up.

DemonType, the type of the demon that is waking up.

- end_wake_demon(EventId::integer,Result::demon_result)

This is the closing event of the block. It contains the result of any pruning inside the block. It is generated before exiting the dispatchCtr() function, after the filtering algorithm has finished.

eventId, the unique event number of this event.

Result, the result of waking up the demon.

4.10.5 begin_prune...end_prune

This block appears inside a wake demon or a new_ctr block, possibly surrounded by a method block. It contains information about variable domain narrowings and corresponds to pruning performed by a demon’s filtering algorithm. It is created inside the function dispatchCtr().

- begin_prune(EventId::integer,Intention::intention,Constraint::ground)

The opening event contains information about what pruning is intended to take place inside the block. This may be different from what values are actually removed due to the current state of the domains of the variables.

eventId, the unique event number of this event.

Intention, information about what pruning is intended to take place inside the block.

Constraint, the constraint responsible for the pruning.

- end_prune(EventId::integer,Result::result)

The closing event contains the result of the prune block.

eventId, the unique event number of this event.

Result, the result of the prune block.
4.10.6 prune

- prune(EventId::integer,PrunedVars::path,Pruning::pruning,
  Constraint::ground,Result::result)

The prune event contains information about what pruning is actually taking place; i.e. what values are
effectively removed from which variables. It is created inside a prune block and is usually surrounded by a
before_prune and an after_prune event.

EventId, the unique event number of this event.
PrunedVars, a path term referring to one or more variables, the variables that are pruned.
Pruning, the actual pruning, which values are removed.
Constraint, the constraint responsible for the pruning.
Result, the result of the pruning.

4.10.7 before_prune, after_prune

- before_prune(EventId::integer,PrunedVars::path,Domains::list(fd_set),
  Constraint::ground)
- after_prune(EventId::integer,PrunedVars::path,Domains::list(fd_set),
  Constraint::ground)

These events occur before and after a prune event and contain domain information before and after value
removal for the variables that are pruned.
EventId, the unique event number of the event.
PrunedVars, a path term referring to one or more variables.
Domains, a list of domains for the above variables.
Constraint, The constraint responsible for the pruning.

4.10.8 fail

- fail(EventId::integer,Constraint::ground)

This event is generated when the solver discovers a failure due to a necessary condition that is not fulfilled.
It is created inside a prune block and usually surrounded by a method block.
EventId, the unique event number of this event.
Constraint, the constraint that fails.

4.10.9 begin_method...end_method

A prune block or fail event is usually surrounded by a method block. It provides information about which
method of the filtering algorithm is used.

- begin_method(EventId::integer,MethodName::atom)

This is the opening event of the block. It is generated when the filtering algorithm activates a specific method.
It contains a descriptive atom name corresponding to the associated consistency rule of the constraint.
EventId, the unique event number of this event.
MethodName, the name of this method.
- end_method(EventId::integer, Result::result)

The `end_method` event closes the block. It contains the result of the domain narrowings that have been done inside it.

EventId, the unique event number of this event.

Result, the result of the method block.

4.10.10 info_method
- info_method(EventId::integer, InfoName::atom, Info::value_path, Constraint::ground)

This event appears inside a method block, just before a `begin_prune` event. It provides some more detailed information about the active method.

EventId, the unique event number of this event.

InfoName, an atom with a descriptive name for the extra information.

Info, a path term referring to some entities in the constraint and possibly the value of some of their properties.

Constraint, the constraint this method is associated with.

4.10.11 labeling_start
- labeling_start(EventId::integer, VarName::atom, Domain::fd_set, Mode::ground)

This event is generated when labeling of a new variable starts. This event, as well as `labeling_step` and `labeling_fail`, is SICStus specific, describing its builtin `labeling/2` predicate.

EventId, the unique event number of this event.

VarName, id of the variable that is labeled.

Domain, the domain of the variable before narrowing it.

Mode, the mode of the labeling, i.e. the chosen search strategy.

4.10.12 labeling_step
- labeling_step(EventId::integer, VarName::atom, Domain::fd_set, Pruning::intention)

This event is generated when a variable is fixed to a value due to labeling.

EventId, the unique event number of this event.

VarName, name of the variable that are labeled.

Domain, the domain of the variable at the last `labeling_start` event.

Pruning, the update done on the variable.

4.10.13 labeling_fail
- labeling_fail(EventId::integer, VarName::atom, Domain::fd_set)

This event is generated when labeling of a variable fails, i.e. when there are no more values in its domain to try.

EventId, the unique event number of this event.

VarName, name of the variable that are labeled.

Domain, the domain of the variable before labeling started.
5 Explanations

An explanation is a Prolog term with some information about why a specific action is taken by the solver. These are actions such as pruning a variable, failing, adding a new constraint to the kernel, and waking up a constraint for propagation. In our trace format, explanations appear in so called explanation events which are located just before the events they explain.

5.1 Usage

Listed below are some possible areas where explanations are useful.

**Debugging.** Why cannot variable $x$ take value $v$? Why are there no solution(s) to a problem? Questions like these are answered by our explanations. This is of course useful for application developers, helping them to find bugs in their programs. It may also be used by constraint developers for finding missing propagation in new constraints.

**Dynamic Problem Solving.** Adding and removing constraints dynamically during execution can be of great help to application developers. Explanations give the developer guidance in this process, i.e. which constraints to add or remove.

**Dynamic Backtracking.** Introduced by [Gin 93], the idea is to detect choice-points in the search that will always fail due to some earlier bad choices*. Such choice-points can be skipped when backtracking, obtaining a more efficient search. Explanations come in handy in the following way: When a failure is noticed, the explanation for that failure is due to some previous action taken by the solver. If this action is further back than the last choice-point, it may be possible to backtrack to an earlier one.

Jussien and Ouis [JO 01] introduced dynamic problem solving using explanations within the PaLM system. Jussien, Debruyne and Boizumault [JDB 00] introduced dynamic backtracking using explanations. Our extension to the SICStus Prolog debugger, presented in Section 7, provides commands for unfolding, i.e. finding responsible constraints for certain actions.

5.2 Defining $e$

An explanation $e$ for a solver 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 according to the following type definition:

```prolog
:- type explanation == pair(det, list(condition)).
:- type det == integer; 'all.
:- type condition == 'cond(det, path_function, properties).
:- type properties == property; list(property).
:- type property == integer; pproperty.
:- type pproperty == 'dom; 'range; 'min; 'max; 'ground;
  'length; 'arity; 'less(integer); 'lesseq(integer);
  'greater(integer); 'greateq(integer);
  'in(integer, integer); 'notin(integer, integer);
  'eq(integer); 'neq(integer);
  'inset(fd_set); 'notinset(fd_set).
```

Hence, an explanation is a pair, $\langle N,\text{-}\text{CondList} \rangle$, where $N$ is an integer and $\text{CondList}$ is a list of $\langle \text{cond}(M, Pa, PL) \rangle$ terms. An explanation $e = \langle N,\text{-}\text{CondList} \rangle$ holds iff at least $N$ of the $\text{cond}/3$ terms in $\text{CondList}$ hold. Each $\text{cond}/3$ term contains an integer $M$, a path_function term $Pa$ 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,

* This behavior is usually called thrashing.
assume that \( Pa = [1, 3] \), \( PL = [\text{eq}(2), \text{neq}(3)] \) and that \( Pa \) is associated with the \texttt{element/3} constraint in Ex. 2. Then \( X \) is associated with \texttt{eq}(2) \) and \( Y \) is associated with \texttt{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 the single property. A \texttt{cond/3} term holds iff at least \( N \) of the entities referred to by \( Pa \) fulfill their respective properties. Listed below are the possible properties in \( PL \) for some entity \( E \), and their meaning. \( L, U \) and \( V \) are members of the \texttt{integer type}; \( S \) is a member of the \texttt{fd_set type}:

- \texttt{dom}, the domain of \( E \).
- \texttt{range}, the range of \( E \), i.e. the difference between the maximum and the minimum values.
- \texttt{min}, the minimum value of \( E \).
- \texttt{max}, the maximum value of \( E \).
- \texttt{ground}, \( E \) is ground.
- \texttt{length}, the length of \( E \).
- \texttt{arity}, the number of arguments in \( E \).
- \texttt{less(V)}, \( E \) is less than \( V \).
- \texttt{lesseq(V)}, \( E \) is less than or equal to \( V \).
- \texttt{greater(V)}, \( E \) is greater than \( V \).
- \texttt{greatereq(V)}, \( E \) is greater than or equal to \( V \).
- \texttt{in(L,U)}, \( E \) is in the range \( L..U \).
- \texttt{notin(L,U)}, \( E \) is not in the range \( L..U \).
- \texttt{eq(V)}, \( E \) is equal to \( V \).
- \texttt{neq(V)}, \( E \) is not equal to \( V \).
- \texttt{inset(S)}, \( E \) is in the set \( S \).
- \texttt{notinset(S)}, \( E \) is not in the set \( S \).

The different properties are not all applicable to everything a \texttt{path_function} term may refer to. The following restrictions apply:

- The properties \texttt{dom}, \texttt{range}, \texttt{min}, \texttt{max} and \texttt{ground} only apply when \( E \) is a domain variable.
- The property \texttt{length} only applies when \( E \) is a list.
- The property \texttt{arity} only applies when \( E \) is a compound term.
- All other properties apply when \( E \) is a domain variable or an integer value.

### 5.3 An Example

Now, let us give an illustrative example. Assume that

\[
e = 1 - \text{cond}(1, [1, \#3], \text{eq}(3))
\]

explains the solver action \( a \). Also, assume that \( e \) is associated with a pruning generated due to the \texttt{all différent/1} constraint in Example 2. This gives us the information that \( a \) occurs since at least one of the \texttt{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, \texttt{eq}(3). In our case, this is the position marked with grey in

\[
\text{all différent}([X, Y, 3, V1, 8, V2]).
\]

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.
5.4 Constructing Explanations

When we construct an explanation e for some action a, we first describe a, possibly with some example. After that, we try to find out under what condition(s) a may not occur. By negating this, we derive an explanation for why a occurs. One should aim at creating an explanation that is minimal with respect to the number of variables it contains. We denote the explanation that contains the least possible number of variables the sharpest explanation. The least sharp explanation is always simple to construct. It will contain all variables in the constraint corresponding to the action and depend on all of their domains.

5.5 Explained Actions

We provide explanations for the following different solver actions:

**Constraint adding, non top-level.** A new constraint is added to the system by an existing one. One example of this is when an existing constraint is replaced by a new, simpler one or when some constraint is defined by several simpler ones. This corresponds to the `connectCtr(C)` function in our kernel model, given in Section 4.2.

**Demon enqueuing.** A demon of some existing constraint is to be woken up. This action corresponds to the `enqueueCtr(C)` function in Section 4.2, where some demon in C is put in `ReadyQueue`.

**Domain pruning.** Some domain variable is to be pruned. This action corresponds to the function `enqueueEvent(E)` in Section 4.2.

**Failing.** A necessary condition of a constraint is not fulfilled.

We will focus on providing explanations for the last two actions which occur during the filtering algorithms of the constraints.

5.6 Explanation Events

We extend the trace format presented in Section 4 with events containing explanations. They appear just before the events they explain and have a similar name ending with `because`.

5.6.1 `push_demon_because`

- `push_demon_because(EventId::integer, Variables::path, Condition::wake_property, Constraint::ground)`

The `push_demon_because` event occurs just before a `push_demon` event and contains a reason for why the demon is enqueued to `ReadyQueue`.

  EventId, the unique event number of this event.

  Variables, path term referring to the variable(s) responsible for enqueuing the demon.

  Condition, the property of the variable(s) that has changed.

  Constraint, the constraint the enqueued demon is associated with.

5.6.2 `new_ctr_because`

- `new_ctr_because(EventId::integer, ConstraintId::atom, Explanation::explanation, Constraint::ground)`

This event occurs just before a `new_ctr` block and contains a reason for why the new constraint is posted.

  EventId, the unique event number of this event.
ConstraintId, the id of the constraint that creates the new one.

Explanation, the explanation term of this event.

Constraint, the constraint that creates the new one.

5.6.3 prune because

- prune because(EventId::integer,ConstraintId::atom,Explanation::explanation, Constraint:::ground)

This event appears just before a before prune event and contains a reason for why some specific pruning occurs.

EventId, the unique event number of this event.

ConstraintId, the id of the constraint that is responsible for creating the prune event.

Explanation, the explanation term of this event.

Constraint, the constraint that is responsible for creating the prune event.

5.6.4 fail because

- fail because(EventId::integer,ConstraintId::atom,Explanation::explanation, Constraint:::ground)

This event appears just before a fail event and contains a reason for why the failure occurred.

EventId, the unique event number of this event.

ConstraintId, the id of the constraint that fails.

Explanation, the explanation term of this event.

Constraint, the constraint that fails.

5.7 Explaining all different/1 (naive algorithm)

The all different(Variables) constraint where Variables is a list of domain variables or integers is true iff each variable Var ∈ Variables takes a unique value. From a declarative point of view, this is the same as an inequality constraint between each pair of variables and the naive algorithm achieves the same pruning. A more powerful algorithm is given in [Rég 94] which is used for the all distinct constraint below.

Propagation Rule - Ground Variable

Assume that the posted constraint is all different([X₁,...,Xₙ]). If any variable Xᵢ is pruned to a singleton value V, V will be removed from the domains of the variables X_k, k ≠ i.

Assume that a variable Xᵢ is pruned to the singleton value V. Furthermore, assume that there is some set of variables PVars, PVars ⊆ {X₁,...,Xₙ}, such that V ∈ dom(Xᵢ) for all X ∈ PVars. This will wake up the constraint and V will be removed from the domains of all variables in PVars.

How can we avoid this pruning? Well, if Xᵢ had not been made ground to V, exactly this pruning would not have occurred. This gives us the following explanation:

\[ 1-[\text{cond}(1,P,\text{eq}(V))] \]  

where P is the path term referring to variable Xᵢ.
5.8 Explaining all distinct/1

The all distinct(+Variables) constraint where Variables is a list of domain variables or integers is true iff each variable Var ∈ Variables takes a unique value. Declaratively, this is the same as the all different/1 constraint mentioned above. However, this version uses the complete algorithm of [Rég 94] based on bipartite graph matching. All graph terminology used in this section is described in Appendix A.

Propagation rule - |values| < |variables|

One necessary condition for the constraint is that the number of values* is greater than or equal to the number of variables. If this is not the case, a failure will occur.

Assume that the posted constraint is all distinct([X₁,...,Xₙ]), and let U_dom be the union of the domains of all variables Xᵢ. Now, assume that |U_dom| < n which will cause a failure.

How can we avoid this failure? The reason for it to occur is that |U_dom| < n; hence we need to increase the size of U_dom. If any Xᵢ takes a value not in U_dom, a new set of values U_dom' |U_dom' > |U_dom|, is obtained. It may be true that |U_dom'| ≥ n, so the failure may not occur. This gives us the following explanation:

\[ 1 - \text{cond}(all, P, \text{in}(U)) \]  

where P is the path term referring to the list [X₁,...,Xₙ].

This propagation rule will not take care of the case where

Variables = [X₁,X₂,X₃,X₄],
\[ \text{dom}(X₁) = \text{dom}(X₂) = \text{dom}(X₃) = 1..2 \text{ and } \text{dom}(X₄) = 4..8 \]

since length(Variables) = 4 and |U_dom| = 7. However, this is still a failure since \( \bigcup_{i=1}^{n} \text{dom}(Xᵢ) \) < 3. Failures like this are taken care of by another, more powerful, propagation rule involving bipartite graph matching. The above propagation rule is actually redundant and all failures it detects will also be detected by the stronger rule. Moreover, the explanation mentioned above is not a very sharp one since it contains all variables in the constraint. For this, it may be useful to not apply the above propagation rule when creating explanations. It will be caught later anyway and the explanation for the stronger rule will be sharper.

Propagation Rule - Maximal Bipartite Matching

The bipartite graph has variables, X₁,...,Xₙ, on one side and values, \( \bigcup_{i=1}^{n} \text{dom}(Xᵢ) \), on the other side. An edge between any two vertices (Var,Val) denotes that Val ∈ dom(Var). The first action for the algorithm is to calculate a maximal matching in the bipartite graph. A solution to the constraint must have a maximal matching where no variable vertex is unmatched. The bipartite graph for the variables A, B, C, D with domains dom(A) = dom(B) = dom(C) = {1,2} and dom(D) = {3,4,5} is shown in Figure 2(i). There is no solution to all distinct([A,B,C,D]) since \( |\bigcup_{\text{V} \in \{A,B,C\}} \text{dom}(V)| < 3 \). This is noticed when calculating the maximal matching. One maximal matching is shown in Figure 2(ii), where C is left unmatched.

How can we avoid this failure? Well, if the size of the union of the domains of the variables A, B and C is larger than 2 it will not occur. To make it larger, at least one of the variables must have a value V \( \notin \{1,2\} \) in its domain. Generally, this gives us the following explanation:

\[ 1 - \text{cond}(all, P, \text{in}(U)) \]  

where P is the path term referring to a set of variables for which the size of the union of their domains is too small, and U is the union of these variables. These variables are the ones accessible by any number of jumps Var₁ → Val₁ → Var₂, i ≠ k, from the first unmatched variable vertex Var₁ in the graph.

Note that there might exist several of these subsets of variables such that each on its own leads to a contradiction. The first subset encountered will be the one reported as the cause of the failure.

* The union of the domains of all variables.
Figure 2: (i) Bipartite graph for the variables A, B, C, D. (ii) One maximal matching, member edges are marked bold.

Propagation Rule - Different Components

By extending the graph in Figure 2(i) with the edge (C, 3), we derive a graph that has a maximal bipartite matching including all variable vertices; see Figure 3(i). We also add one variable, E, with the domain {8, 9}. After the matching has been generated, the algorithm continues with splitting up the graph in different components. First of all, the matched edges are kept undirected and all other edges are made directional from right to left; see Figure 3(ii). After that, the different components are generated as follows:

1. Every set of vertices that constitutes a strongly connected component and that is not involved in any alternating path is marked as a component.

2. The set of all vertices that are involved in alternating paths is marked as a component.

The different components in our example are shown in Figure 3(iii). The sets of vertices \{A, B, 1, 2\} and \{C, 3\} are components of type 1, and the sets of vertices \{D, 4, 5\} and \{E, 8, 9\} are members of the component of type 2. When all components of the graph have been identified, all edges that are between vertices in different components are removed, since they cannot be in a solution to the constraint. In our example, there are two edges that will be removed, (2, C) and (3, D).

In order to come up with an explanation for this pruning we need to look at the property of the graph. At this time, the graph consists of at least two components, \(C_m\) and \(C_n\), with an edge, \(e_{Val\rightarrow Var}\), between them. The edge points from a value vertex, Val, in \(C_m\) to a variable vertex, Var, in \(C_n\) with the meaning that \(Val \in \text{dom}(Var)\). If we want \(e_{Val\rightarrow Var}\) not to be removed, we need to make the vertices Val and Var members of the same component.

The component \(C_m\) is of type 1. It cannot be an alternating path, since in that case Val would either be the matched value vertex of some variable vertex Var and there would exist an alternating path Val_{exp} \rightarrow \ldots \rightarrow Var_{exp} \rightarrow Val, or Val would be an exposed value vertex. Both of these cases imply that \(e_{Val\rightarrow Var}\) is a member of an alternating path ending in the value vertex matched with Var.

First, let’s see how we can make the vertices Val and Var members of an alternating path, and thus members of the component of type 2. This can be achieved by adding an edge from any value vertex that is a member of an alternating path to any variable vertex in \(C_n\). This means adding a value to the domain of a variable. In order to see this, assume that \(C_{path}\) is an alternating path and that Val_{exp} is a value vertex of \(C_{path}\). Val_{exp} is either an exposed value vertex, the start of the alternating path, or a matched value vertex, ending an alternating path. By adding the edge (Val_{exp}, VarCMP), where VarCMP is a member of \(C_m\), to the graph, a new alternating path is created which goes through Val and Var and ends up in the matched value vertex of Var, Val_{match}. This is correct, since we create it by adding a path containing an odd number of edges (the path from VarCMP to Val_{match}) to an alternating path or an exposed vertex with one edge, (Val_{exp}, VarCMP).

Second, let’s see how we can make the vertices Val and Var members of a component of type 1. A component of type 1 is a strongly connected component, meaning that all vertices in the component are reachable from each other. Since it is possible to reach Var from any vertex in \(C_m\), via the edge (Val, Var)
Figure 3: (i) One maximal matching (bold edges) involving all variable vertices. (ii) Edges involving exposed value vertices are made directional during the execution of the filtering algorithm. (iii) The different components of the graph. Light-grey denotes a component of type 1 and dark-grey denotes alternating paths that are members of the component of type 2. (iv) Adding any of the dotted edges has the effect that the edge (C, 3) is not removed.

and from the fact that $c_m$ is a strongly connected component, all we need to do is to add an edge from the matched value vertex of $Var_{val_{match}}$ to any variable vertex in $c_m$.

In our example, assume that we don't want the edge $(3, D)$ to be removed. We need to make the two vertices members of the same component. By following the reasoning above, we add an edge $(4, C)$ to the graph. This will create a new component of type 1 including the vertices $(C, D, 3, 4)$, and the edge $(3, D)$ will not be removed. We can also add the edge $(5, C)$ which will create a new alternating path including the vertices $(C, D, 3, 4, 5)$. The possible edges we can add in order to keep the edge $(3, D)$ are shown dashed in Figure 3(iv). Note that anyone of them is enough to avoid the pruning.

This gives us the following explanation for removing the value $Val$ from $Var$:

$$
1 - \text{cond(all, P, notinset(S))}
$$

(3)

where $P$ is the path term referring to all variables in $c_m$ and $S$ is the union of all values involved in any alternating path of the graph and the matched value vertex of $Var$.

This explanation is not totally correct. It is possible to extend $P$ to refer to more variables and $S$ to contain more values. In order to sort this out we take one more look at the graph.

First, we try to extend the path term $P$. The component $c_m$, containing $Val$, may be connected to another component $c_1$, of type 1, through some edge $(Val_{src}, Val_{dst})$, $Val_{src} \in c_1$ and $Val_{dst} \in c_m$. If this is the case, it is possible to reach $Val$ from any variable vertex in $c_1$ as well. This is true, since we can reach $Val$ from any variable vertex in $c_m$; at least one of these variable vertices can be reached from $c_1$ and $c_1$ is a strongly connected component. Due to this, we extend $P$ to also refer to all variables in $c_1$. Furthermore, $c_1$ may be connected to some component $c_4$ in the same way, which means that $Val$ is reachable from all variable vertices in $c_4$ too. This can be extended even more since $c_4$ may also be connected to some component from which it is possible to reach $Val$ and so on. In each step there may exist more than one component like $c_1$, which means that we have a tree of components with $c_m$ as the root. All variable vertices in this tree can reach $Val$. This is illustrated in the light-grey part of Figure 4.

Second, we try to add more values to $S$. Since all value vertices in components of type 2 are already in $S$, we need to look at components of type 1. Any value vertex from which we can add an edge to any variable vertex in $P$ and derive a strongly connected component that includes the vertices $Val$ and $Var$ should be in $S$. Up until now, the only value vertex explicitly added to $S$ due to this property is the matched value vertex of $Var_{val_{match}}$. In order to add more of these value vertices to $S$, assume that there are some variable vertices $Var_{j_1}, Var_{j_1+1}, \ldots, Var_{j_1+p}$ such that $Val_{match} \in \text{dom}(Var_{j_i})$ for all $i \in \{j, j + 1, \ldots, j + p\}$. 

25
5.9 Explaining assignment/2

The assignment(\(\{x_s, y_s\}\)) constraint where \(x_s = [x_1, \ldots, x_n], y_s = [y_1, \ldots, y_n]\) are lists of domain variables or integers is true if all \(x_i, y_i\) are in the range \(1..n\) and \(x_i = j \Leftrightarrow y_i = j\). This implies the constraints all_distinct(\(x_s\)), and all_distinct(\(y_s\)) and part of the algorithm for assignment/2 is the same as for all_distinct/1. The main difference is that the bipartite graph mentioned in the all_distinct/1 constraint will contain variable vertices on both sides. The explanations mentioned for all_distinct/1 will be the same for assignment/2 but for the path terms to the variables. Listed below are the added propagation rules.

**Propagation Rule -** \(x_i = j \Leftrightarrow y_j = i\)

The condition \(x_i = j \Leftrightarrow y_j = i\) gives rise to two different propagation rules. The first one handles the case where some variable \(x_i\) is bound to a value \(j\). This means that \(y_j\) must equal \(i\). The second one handles the case where some value \(j\) is removed from \(\text{dom}(x_i)\). This means that variable \(y_j\) cannot take the value \(i\).

For the first rule, assume that \(x_i\) is fixed to the value \(j\). The corresponding \(y_j\) is fixed to \(i\). The explanation for this is quite obvious. If \(x_i\) had not been fixed to \(j\) the pruning had not occurred. Hence, we get the following explanation:

\[
1 - [\text{cond}(1, P, \text{eq}(V))] 
\] (1)
Figure 5: The propagation rule $x_i = j \Leftrightarrow y_j = i$. Fixing a variable $x_i$ to some value $j$, bold edges, leads to fixing the corresponding $y_j$ to $i$. This triggers value removals, dashed edges, possibly leading to more ground variables and so on.

where $P$ is the path term referring to $x_i$ and $V = j$. Of course this rule also applies to the opposite case where any $y_j$ is bound to a value $j$. The only difference is the path term, $P$, which refers to $Y_i$ instead.

For the second rule, assume that the value $j$ is removed from variable $x_i$. Variable $y_j$ cannot take the value $i$. The explanation for this value removal is:

$$1-\left[\text{cond}(1, P, \text{neq}(V))\right]$$

where $P$ is the path term referring to $x_i$ and $V = j$. This rule also applies to the opposite case where a value $j$ is removed from the domain of a variable $x_i$. The only difference is the path term, $P$, which refers to $x_i$ instead. These propagation rules are displayed in Figure 5.

### 5.10 Explaining circuit/[1,2]

The circuit constraint can be thought of as constraining $n$ vertices in a graph to form a Hamiltonian circuit. circuit/1 is a simplified version of circuit/2; the second argument is created internally. Because of this only circuit/2 is discussed here.

The circuit (+Succ, +Pred) constraint where Succ = $[x_1, \ldots, x_n]$ and Pred = $[y_1, \ldots, y_n]$ are lists of domain variables or integers is true iff all $x_i$, $y_i$ are in the range $1..n$, $x_i = j$ $\Leftrightarrow$ $y_j = i$, and Succ, Pred build Hamiltonian circuits. $x_i$ ($y_i$) is the successor (predecessor) of $i$ in the graph. All propagation rules that apply to assignment apply to circuit too. The extra rules are for assuring the Hamiltonian circuit property. All $x_i$ mentioned below are assumed to be members of the Succ list. Similar reasoning holds for the Pred list.

**Propagation Rule - Forbid/Infer Edge**

Assume that there exist some paths in the graph

- $a$: $x_{i+1} \rightarrow \cdots \rightarrow x_{i+m}$
- $b$: $x_{j+1} \rightarrow \cdots \rightarrow x_{j+n}$

and an inferred edge $x_{i+m} \rightarrow x_{j+1}$. If the number of vertices in the graph is $m + n$, the edge $c$: $x_{j+n} \rightarrow x_{i+1}$ should be inferred; if not it should be forbidden. Inferring the edge $c$ means fixing $x_{i+1}$ to $j + n$. Forbidding the edge $c$ means removing $j + n$ from the domain of $x_{i+1}$.

For the first case, what is the explanation for inferring edge $c$? There are actually two conditions for why this happens. If the number of variables is larger than $m + n$, the edge $c$ will not be inferred. This holds since in that case, there exist some variables not in any of the paths a and b that should be members of the complete circuit. Neither will $c$ be inferred if at least one variable $x_k$, $k \neq i + 1$, has a non-singleton domain. This holds, since in that case, either a or b would not exist. This gives us the following explanation:

$$\text{all-} \left[\text{cond}(1, P_{\text{start}}, \text{eq}(L)), \text{cond}(\text{all}, P_{\text{end}}, \text{dom})\right]$$

---

*See Appendix A for graph terminology.
Figure 6: An example of forbidding an edge, dotted, due to an inferred edge, dashed, between two existing paths, a and b.

where $P_{\text{int}}$ is a path function term referring to the length of $\text{Succ}$; $L$ is the length of $\text{Succ}$; and $P_{\text{vars}}$ is a path term referring to the set of variables \{${X}_{i+2}, \ldots, {X}_{i+m}, {X}_{j+1}, \ldots, {X}_{j+n}$\}.

For the second case, what is the explanation for forbidding edge c? There are two conditions even for this case corresponding closely to the previous one. First, if the number of variables, i.e. the length of Succ list, is equal to $m + n$, c will not be forbidden. Second, if any variable ${X}_k$, $k \neq i + 1$, has a non-singleton domain, c will not be forbidden. This gives us the following explanation:

$$\text{all-}[\text{cond}(1, P_{\text{int}}, \text{greater}(L)), \text{cond}(\text{all}, P_{\text{vars}}, \text{dom})]$$

(2)

where $P_{\text{int}}$ is a path function term referring to the length of $\text{Succ}$; $L = m + n$; and $P_{\text{vars}}$ is a path term referring to the set of variables \{${X}_{i+2}, \ldots, {X}_{i+m}, {X}_{j+1}, \ldots, {X}_{j+n}$\}.

5.11 Explaining min/2

The $\text{min}(\text{Min}, \text{Variables})$ constraint where Variables is a list of domain variables or integers and Min is a domain variable or an integer is true if $\text{Min}$ is equal to the minimum of Variables.

Propagation Rule - Pruning Min

The domain of $\text{Min}$ must be between the smallest minimum and the smallest maximum of the elements in Variables. We define the following:

$$S_{\text{min}} = \min({X}_i) \forall i: \min(X_i) \leq \min(X_i),$$

$$S_{\text{max}} = \max({X}_i) \forall i: \max(X_i) \leq \max(X_i).$$

Assume that $\text{Min}$ has some value(s) not in the range $S_{\text{min}}..S_{\text{max}}$. This will lead to a pruning of $\text{Min}$. In order to come up with an explanation for this, we need to know under what circumstances this actual pruning will not occur. Since $\text{Min}$ is pruned according to the range given by the values $S_{\text{min}}$ and $S_{\text{max}}$, we need to see under what conditions $S_{\text{min}}$ and $S_{\text{max}}$ are smaller respective larger. For $S_{\text{min}}$, we see from its definition that it is the smallest minimum value of all the variables in Variables. If any of the variables with the minimum value $S_{\text{min}}$ had a smaller minimum value, the value of $S_{\text{min}}$ would also be smaller. For $S_{\text{max}}$, we see from its definition that it is the smallest maximum value of all the variables in Variables. If all variables with the maximum value $S_{\text{max}}$ had a larger value, $S_{\text{max}}$ would also be larger. This is expressed in the following explanation:

$$\text{all-}[\text{cond}(1, P_{\text{min}}, \text{min}), \text{cond}(\text{all}, P_{\text{max}}, \text{max})]$$

(1)

where $P_{\text{min}}$ is the path term referring to all variables with $S_{\text{min}}$ as their minimum value and $P_{\text{max}}$ is the path term referring to all variables with $S_{\text{max}}$ as their maximum value.

Furthermore, $\text{Min}$ cannot have a value in its domain that is not in the union of the elements in Variables. We denote this union $U$. Assume that there is some set of values, $U'$, in the domain of $\text{Min}$ such that $U' \cap U = \emptyset$. 28
All values in \( U' \) will be removed from \( \text{Min} \). How can we explain this pruning? Well, if the set \( U \) contains some value \( V \in U' \), exactly this pruning will not occur. Since \( U \) is the union of all elements in \( \text{Variables} \), we derive the following explanation:

\[
1-\text{[cond(all,P,notinset(U'))]} \\
\text{(2)}
\]

where \( P \) is the path term referring to \( \text{Variables} \).

**Propagation Rule - Pruning Variables**

The elements in \( \text{Variables} \) cannot have a smaller value than \( \min(\text{Min}) \) in their domains. Each \( V \in \text{Variables} \) will be forced in the interval \( \min(\text{Min}) \ldots \max(V) \). This pruning depends only on the value of \( \min(\text{Min}) \); if this value is decreased, exactly this pruning will not occur. This gives us the following explanation:

\[
1-\text{[cond(1,P,greatereq(M))]} \\
\text{(3)}
\]

where \( P \) is the path term referring to the \( \text{Min} \) variable and \( M \) is the minimum value of \( \text{Min} \).

**Propagation Rule - Unifying Variables**

If there is only one element, \( V \in \text{Variables} \), that has an intersection with the range \( R = \min(\text{Min}) \ldots \max(\text{Min}) \), \( V \) will be unified with \( \text{Min} \).

Assume that this is the case and that the single variable that has an intersection with \( R \) is \( X_k \). \( \text{Min} \) and \( X_k \) are unified. How can we explain this? If any other variable, \( X_i \in \text{Variables} \) with \( i \neq k \), has an intersection with \( R \), the unification will not occur. The following explanation expresses this:

\[
1-\text{[cond(all,P,notinset(L,U))]} \\
\text{(4)}
\]

where \( P \) is the path term referring to all variables in \( \text{Variables} \) but \( X_k \) and \( L, U \) are the values \( \min(\text{Min}) \), \( \max(\text{Min}) \) respectively.

**Propagation Rule - One Intersection**

If there is only one element, \( V \in \text{Variables} \), that has an intersection with \( S = \text{dom}(\text{Min}) \), \( V \) will be pruned according to \( S \). This is quite similar to the previous rule, except that there is at least one element \( V', V' \neq V \), that intersects the range \( \min(\text{Min}) \ldots \max(\text{Min}) \) but not the set \( S \). In order to come up with an explanation for this, we define \( \text{Variables}_2 \) to be the set of all variables in \( \text{Variables} \) but \( X_k \). Assume that the single variable which domain intersects with \( S \) is \( X_k \). All values not in \( S \) will be removed from the domain of \( X_k \). The explanation for this pruning has two conditions, both needed for it to occur. The first involves all elements in \( \text{Variables}_2 \) and the second the \( \text{Min} \) variable. For the first one, if any of the domains of the variables in \( \text{Variables}_2 \) intersects \( S \), exactly this pruning will not occur, since in that case \( X_k \) would not be the only intersecting variable. For the second one, neither will it occur if the domain of \( \text{Min} \) intersects with the union of the domains of the variables in \( \text{Variables}_2 \). The two conditions are of course related to each other. This gives us the following explanation:

\[
\text{all-}[\text{cond(all,P1,notinset(S))},\text{cond(1,P2,notinset(U))}] \\
\text{(5)}
\]

where \( P_1 \) is a path term referring to \( \text{Variables}_2 \); \( S = \text{dom}(\text{Min}) \); \( P_2 \) is the path term referring to the \( \text{Min} \) variable; and \( U \) is the union of the domains of the variables in \( \text{Variables}_2 \).

**Propagation Rule - No Intersection**

If \( \text{Min} \) does not intersect with the domain of any element in \( \text{Variables} \), a failure will occur. This depends on two related conditions. For the first one, if the domain of any variable \( V \in \text{Variables} \) intersects with \( \text{dom}(\text{Min}) \), the failure will not occur. For the second one, if \( \text{dom}(\text{Min}) \) intersects with the union of the domains of all variables, the failure will not occur. The two conditions imply each other. This gives us the following explanation:
all-[\text{cond}(\text{all}, P_1, \text{notinset}(S)), \text{cond}(1, P_2, \text{notinset}(U))]$

where $P_1$ is the path term referring to Variables; $S$ is the domain of $\min$; $P_2$ is the path term referring to $\min$; and $U$ is the union of the domains of the elements in Variables.

5.12 Explaining $\text{scalar\_product}/4$

The $\text{scalar\_product}(\text{Coeffs}, \text{Xs}, \text{RelOp}, \text{Value})$ constraint, where $\text{Coeffs} = [C_1, \ldots, C_n]$ is a list of integers, $\text{Xs} = [X_1, \ldots, X_n]$ is a list of domain variables or integers, $\text{RelOp} \in \{\&\&=, \&\&\neq, \&\&<, \&\&>, \&\&\geq\}$ and $\text{Value}$ is a domain variable or an integer, is true iff $C_1 * X_1 + \cdots + C_n * X_n \text{RelOp} \text{Value}$. For simplicity, we assume that $\text{RelOp} = '\&\&\neq'$ from now on; all operators share similar propagation rules. Also for simplicity, we rewrite this to $C_1 * X_1 + \cdots + C_n * X_n + (-1) * \text{Value} \neq 0$. The equation is simplified during the solving process; terms with ground variables are moved to the right hand side. We denote the current value of the right hand side RHS. Furthermore, we define the following in order to describe and explain the propagation rules.

$$\text{RHS}_x^{\max} = -\sum_{i=1, i \neq k, C_i > 0}^{n+1} C_i * \min(X_i) - \sum_{i=1, i \neq k, C_i < 0}^{n+1} C_i * \max(X_i),$$

where $n+1$ is the number of terms, $C_i * X_i$, including the term $(-1) * \text{Value}$.

**Propagation Rule - Pruning Upper Bound**

Let $x_k^{\max} = \left\lfloor \frac{\text{RHS}_x^{\max}}{C_k} \right\rfloor$ for all terms $C_k * X_k$ such that $C_k > 0$. It must be true that $V \leq X_k^{\max}$ for all $V \in \text{dom}(X_k)$.

Assume that we have a term $C_k * X_k$ for which this does not hold; a pruning of $X_k$ will take place and all values $V \in \text{dom}(X_k)$ such that $V > X_k^{\max}$ will be removed. An explanation for this pruning will contain all variables $X_i$, $i \neq k$, since they all affect $x_k^{\max}$. In order to provide an explanation we consider under what conditions we have less or no pruning. Less pruning is the case if the value $x_k^{\max}$ is larger. Therefore, we need to think about how to make $x_k^{\max}$ larger or actually how to make $\text{RHS}_x^{\max}$ larger. (Since $C_k$ is a constant, that cannot be changed). By looking at the definition of $\text{RHS}_x^{\max}$ we see that by decreasing $\min(X_i)$ for any term $C_i * X_i$ such that $C_i > 0$ and/or increasing $\max(X_i)$ for any term $C_i * X_i$ such that $C_i < 0$, we will have a larger $\text{RHS}_x^{\max}$. This gives us the following explanation containing two different conditions:

$$\text{all-[\text{cond}(\text{all}, P_{pos}, \text{min}), \text{cond}(\text{all}, P_{neg}, \text{max})]}$$

(1)

where $P_{pos}$ is a path term referring to all variables with positive coefficients and $P_{neg}$ is a path term referring to all variables with negative coefficients.

**Propagation Rule - Pruning Lower Bound**

Let $x_k^{\min} = \left\lceil \frac{\text{RHS}_x^{\min}}{C_k} \right\rceil$ for all terms $C_k * X_k$ such that $C_k < 0$. It must be true that $V \geq X_k^{\min}$ for all $V \in \text{dom}(X_k)$.

Assume that we have a term $C_k * X_k$ for which this does not hold; a pruning of $X_k$ will take place and all values $V \in \text{dom}(X_k)$ such that $V < X_k^{\min}$ will be removed. An explanation for this pruning will contain all variables $X_i$, $i \neq k$, since they all affect $x_k^{\min}$. In order to provide an explanation, we consider under what conditions we have less or no pruning. Less pruning is the case if the value $x_k^{\min}$ is smaller. Therefore, we need to think about how to make $x_k^{\min}$ smaller or actually how to make $\text{RHS}_x^{\min}$ larger. (Since $C_k$ is a negative constant, that cannot be changed). Hence, the explanation for this propagation rule is the same as for the previous one.

6 Implementation

The trace described in this thesis has been implemented in a research branch of the CLP(FD) solver in SICStus Prolog version 3.9. The trace events are stored as C structures in a growing buffer during execution. Everything regarding actual creation, retrieval and deletion of events is done in C. The trace is accessible from the Prolog side through some predicates that translate C structures to Prolog terms.
<table>
<thead>
<tr>
<th>new_demon</th>
<th>push_demon</th>
</tr>
</thead>
</table>
| struct {
  EventType type;
  size_t id;
  char *demonId;
  char *demonName;
  Event *creator;
  char *wakeConds;
}; | struct {
  EventType type;
  size_t id;
  NewDemon *demon;
}; |

Figure 7: C definitions of new_demon and push_demon events. Complex information is represented using C strings. A push_demon event contains a pointer to its corresponding new_demon event. The demonId and demonName fields are accessed through this pointer when a push_demon event is converted to its Prolog representation.

6.1 C Part

The C part defines the event structures and primitives for creating, deleting and manipulating these.

Events

The events are all quite similar in their structure. All events have

- a type, this is just to be able to cast the different events to a ‘super’ type and back. Useful for storing the events in a buffer, and
- an id, an increasing number giving a chronological order to the events.

All block events, events starting with begin or end, have

- a startBlock pointer, a pointer to the opening event of the block, or
- an endBlock pointer, a pointer to the closing event of the block.

These pointers are useful for navigating through the trace. One example is when scanning forward or backward, searching for some specific information. With these pointers, one whole irrelevant block may be skipped leading to faster scans.

The information in the different events is represented using C enum types when possible, i.e. when the corresponding Prolog representation is of type atom or integer. Anything more complex such as Prolog compound terms or Prolog lists are represented using C strings.

Many of the events are related to each other, meaning that they contain the same information on the Prolog side. On the C side, however, we don’t want to waste memory, which means that memory consuming information is never duplicated. Instead, this is solved with pointers between such events. As an example, the new_demon and push_demon events both contain the id and type of the created demon. These are both represented with strings, stored in the new_demon event. Instead of allocating memory for that when creating a new push_demon event, a pointer to the corresponding new_demon event is added. When the push_demon event is later converted to its Prolog equivalent, the information is fetched through that pointer. The C definitions of the new_demon and push_demon events are shown in Figure 7.

All events implement the following functions:

- create<event_name>Event(Arguments), allocates memory for the event, fills in the different fields and inserts the event in the buffer of events.

- destroy<event_name>Event(Event), deallocates all memory previously allocated to Event.
• `buildProlog<event_name>(PrologEvent, Event)`, translates the C representation of Event to its corresponding Prolog representation using SICStus's C interface functions*. The result is returned in PrologEvent.

Furthermore, there are primitives for accessing information in other events as mentioned above. One example is the `getCtrEvent(Event)` function which retrieves the `begin_new_ctr` event corresponding to Event. The `begin_new_ctr` event contains information that is needed by several other events such as the Prolog term representation and the identifier of the constraint.

**Storage**

The events are stored in an ‘infinite’ global buffer. Infinite in the sense that events are never removed, (although there are some exceptions, mentioned below). This buffer provides the usual functions for inserting, retrieving, updating, and destroying entries.

**Stack**

Since the trace is built up like a block structure and there are pointers between a block’s opening- and closing events, there must be a way of accessing some events created at earlier stages. This is what the event stack is used for. When a new block is opened by creating a `begin` event, that event is pushed on the stack. When its corresponding `end` event is created later, it is popped from the stack and the pointers between them are updated.

**State**

The different blocks’ `end` events usually contain the result of the blocks. For example, an `end_prune` event contains the result, fail or succeed, of a prune block. This result is `succeed` iff all prune events inside the block have a successful result. In order to keep track of this, a state-flag is maintained containing the current state for each block. Whenever a block is entered, the state for that block type is reset. Any action occurring inside the block might update the flag. When a block’s closing event is created, the flag is examined, and the correct result is retrieved.

**Merging and Filtering Events**

Since a CLP(FD) program might generate a very long trace, it is important to at least try to not generate unnecessary events. If two events are similar enough, it might be possible to merge them into one. One example is the `prune` event with the following contents:

```prolog
prune(Id, PrunedVars, Pruning, Constraint, Result).
```

Two consecutive `prune` events can be merged if the `Pruning` and `Result` fields are the same; the `Constraint` fields are the same by default. Assume that we have two consecutive `prune` events with this property. The latter event is deleted and its `PrunedVar` field is merged with the `PrunedVar` field of the first one.

It may also be possible to merge two consecutive blocks into one. One example when this is possible is when there are no events between two `prune` blocks and the two `begin_prune` events can be merged. A `begin_prune` event looks like:

```prolog
begin_prune(Id, Intention, Constraint).
```

Two `begin_prune` events can be merged if the `Intention` fields are the same. The `Constraint` fields are the same by default. Assume that we have two consecutive `begin_prune` events with this property which is noticed during the creation of the second one. The latter `begin_prune` event is deleted as well as the `end_prune` event corresponding to the first one. Also, the state flag is updated to have the result of the removed `end_prune` event.

---

* See [Car95] for details.
Events that contain no real information should be removed since they will only make the trace unnecessarily big. One example is a prune event which describes no actual narrowing on any variable. Such events are never created. Furthermore, it may be that an entire block does not contain any useful information. Such blocks are also removed. One example is the method block; if it is noticed during the creation of the end_method event that no domain narrowing or failure occurred inside the block, the block is removed.

6.2 Prolog Part

We must be able to create most of the events from the Prolog side. The only events that are never created in Prolog are push_demon_because and push_demon. They are created at central places within the C part of the CLP(FD) solver. All other events have constructor predicates in Prolog. A constructor predicate typically takes the different fields of the event as arguments. Some initial processing on these is performed in order to comply with the format the C interface expects. After that, the corresponding C function is called which does the actual creation of the event and insertion in the buffer.

Furthermore, it must be possible to retrieve the events from Prolog after they have been created. This is taken care of by the interface predicate

\[
get\_events(+Selector::selector,-Events::list(ground))
\]

where Selector is a ground term specifying a set of events in the trace and Events is a list containing those events. The selector type is defined by:

\[
:- \ \text{type selector == integer; pair(selector, selector);}
\]

\[
\text{‘first; ‘last; ‘all; ‘next(selector); ‘prev(selector).}
\]

Each selector has the following meaning:

- I, the single event with event number I.
- \(\text{pair(S1, S2)}\), the range of events between S1 and S2 inclusive.
- \(\text{first}\), the first event in the trace.
- \(\text{last}\), the last event in the trace.
- \(\text{all}\), all events in the trace.
- \(\text{next(S1)}\), the single event following S1.
- \(\text{prev(S1)}\), the single event preceding S1.

where I is an integer, and S1 and S2 are selector terms that determine single events.

Moreover, there are some support predicates available. One example is the predicate \text{assign_name/2} which assigns an atom name as the identifier of a domain variable.

7 An Extension to the Prolog Debugger

This section presents an implemented tool that uses the trace, a CLP(FD) extension to the SICStus Prolog debugger. It has been implemented on-top of the trace, without changing any SICStus internals.

7.1 Pretty Printing of Events

In order to get a nice overview of the trace, it must be presented in a readable form. Displaying the trace in its raw format is not very nice. The implemented trace visualizer provides the following:
• The information in each event is presented in an easy-to-understand format. For example, path terms are presented in a nicer format along with the entity they refer to; domains are presented in the SICStus Prolog range* format.

• The structure of the trace is shown by increasing (decreasing) indentation when entering (exiting) blocks. Furthermore, before the information on each line within a block is presented, a vertical '{' sign is printed at each level making it easy to see how deep inside the current tree you are. At the start of each new non top-level event, the '{' sign for the current tree is changed to a '+' sign.

Figure 8 shows the pretty printing of the first 22 trace events from Appendix C.

7.2 Added Commands

The added commands to the Prolog Debugger include commands for navigating through the trace, printing single or sets of events and retrieving information about different entities in the trace. Currently, it is not possible to step through events at the exact time they are generated. Instead, the trace can be viewed at all Prolog breakpoints. As an example, assume that a Prolog top-level is running and that we issue the call

```prolog
:- trace, trace_me.
```

where the predicate `trace_me` corresponds to Example 2. Assuming that we have breakpoints on every single predicate called by `trace_me/0`, we can step our way, using the ordinary debugger commands, to the line:

```prolog
+ 2 2 Exit: domain([2482, 2879, 3276, 3674], 1, 6)
```

At this point, the variables have been initialized to have the domain `1..6`, and some events expressing this have been created. All commands listed below can now be used to explore these events. When that is done, it is possible to continue to the next breakpoint, and so on.

Commands for Navigating Through and Printing Events

In order to be able to navigate through the trace and print only the events we are interested in, we need to keep track of where we are in some way. This is solved by a global pointer pointing to the “current event”, simply an integer containing the `EventId` attribute of that event. All commands work with this as a base. The current event pointer is updated when the program is executed, so that it always points to the last non `end` event when the execution is stopped at a breakpoint.

Even though the trace is stored in a linear buffer, it is built up by a set of nested trees. The following commands should be interpreted having that in mind. First of all, the commands for navigating through the trace are:

• `F`, move forward in the trace. If the current event pointer points to a non-block¹ event, it is increased by 1. Else, it is set to point to the first event following the corresponding block.

• `B`, move backward in the trace. If the current event pointer points to a non-block event, it is decreased by 1. Else, it is set to point to the first event preceding the corresponding block.

• `I`, go inside a block. This command is applicable when the current event is an opening- or a closing event. The current event pointer is updated to contain the id of the first event in the block, i.e. the one that follows the block’s opening event.

• `O`, go out of a block. This command is applicable when the current event is inside a block. The current event pointer is updated to contain the id of the opening event of the current block.

• `C [S::selector]`, set the current event pointer. `S` can be any of the `selector` terms presented in Section 6 with the constraint that it must specify a unique event. If `S` is not given, this command has no effect.

---

*See [Car95] for details.

¹ All events starting with `begin` or `end` are block events.
begin new ctr  
Event no.: 1  
ID: ctr_domain_1  
Constraint: domain([fdvar_1, fdvar_2, fdvar_3, fdvar_4], 1, 6)  
++begin_prune  
  Event no.: 2  
  Intention: force_in_interval(1, 6)  
  Constraint: domain([fdvar_1, fdvar_2, fdvar_3, fdvar_4], 1, 6)  
++prune because  
  Event no.: 3  
  ID: ctr_domain_1  
  No explanation provided.  
  Constraint: domain([fdvar_1, fdvar_2, fdvar_3, fdvar_4], 1, 6)  
++before_prune  
  Event no.: 4  
  Path: domain[1].#[*]:[fdvar_1, fdvar_2, fdvar_3, fdvar_4]  
  Ranges:  
  inf., sup  
  inf., sup  
  inf., sup  
  Constraint: domain([fdvar_1, fdvar_2, fdvar_3, fdvar_4], 1, 6)  
++prune  
  Event no.: 5  
  ID: ctr_domain_1  
  Path: domain[1],#[*]:[fdvar_1, fdvar_2, fdvar_3, fdvar_4]  
  Intention: remove_values([inf[1], [7], sup])  
  Constraint: domain([fdvar_1, fdvar_2, fdvar_3, fdvar_4], 1, 6)  
  Result: succeed  
++after_prune  
  Event no.: 6  
  Path: domain[1],#[*]:[fdvar_1, fdvar_2, fdvar_3, fdvar_4]  
  Ranges:  
  1, 6  
  1, 6  
  1, 6  
  Constraint: domain([fdvar_1, fdvar_2, fdvar_3, fdvar_4], 1, 6)  
end_prune  
Event no.: 7  
Result: succeed  
end new ctr  
Event no.: 8  
Result: entail  
begin new ctr  
Event no.: 9  
ID: ctr_all_different_1  
Constraint: all_different([fdvar_1, fdvar_2, fdvar_3, fdvar_4])  
++new_domain  
Event no.: 10  
ID: ctr_all_different_1  
Type: all_different_1  
Suspensions:  
all_different[1],#[*] = val  
Constraint: all_different([fdvar_1, fdvar_2, fdvar_3, fdvar_4])  
++begin new domain  
  Event no.: 11

Figure 8: Example of pretty printing some trace events.
Navigating through the trace is of no interest if we cannot retrieve any useful information from it. The following commands are used for printing events:

- P [S::selector], print the set of events specified by S if present; otherwise, print the current event. The selector S can be any of the selector terms presented in Section 6. If S points to an opening- or closing event, the whole block of events is printed.

- Depth [D::integer], set the printing depth to D if specified; reset to 0 otherwise. The printing depth controls how deep information to print. A depth of 0 will result in printing all events, no matter how deeply nested they are. A depth of 1 prints only the top-level events; a depth of 2 prints top-level events and events directly inside top-level blocks but no more. As an example, consider the trace in Figure 8, assume that the current event pointer has the value 9, and that the current depth is 0. Now assume that we execute the command P with no argument; all events between 9 and 22 inclusive will be printed. If we change the depth to 1, the only events that will be printed are the events 9 and 22. If we increase the depth to 2, the events 9, 10, 11, 21 and 22 are printed and so on.

- X [D::integer], print the current event or block with printing depth D, overriding the current setting. If no depth is given, 0 is assumed.

Other Commands

The commands presented here are used for retrieving domain information about variables, searching for specific variables and finding the cause of a pruning or a failure.

- S [+|-] VPath::path|VName::atom, search for the pruning of a variable. The optional `+` or `-` is used to specify the direction of the search. The default is backward, specified by a `-`. The variable is given as a path term or as an atom name. If a path term is used, there must be a current context to interpret it in, i.e. the current event must be an event containing a constraint term. The current event is not included in the search and the result, if the search is successful, will be printing and changing the current event pointer to the previous or the following event that updates the domain of the variable. This event will be a prune- or a labeling_step event.

- Dom VPath::path|VName::atom, print the current domain of a variable. If the information is not in the current event, this means searching backward until such an event is found. If no domain information about the given variable is found, inf..sup will be reported as the domain of the variable. The variable is specified in the same format as for the S command, i.e. a path term or an atom name.

- U, find responsible constraint(s) for a certain pruning, failure or new-constraint creation. This command uses the explanations provided to find responsible constraint(s) for certain actions. For example, if the pruning of a variable X has the explanation that some other variable Y has a singleton domain, the constraint responsible for the narrowing of Y is added to the list of responsible constraints. Furthermore, the reason for pruning Y is looked up and any constraint(s) responsible for that is added to the list and so on.

8 Discussion

8.1 Trace Format

We believe that the trace format presented in this thesis contains enough information in order to implement several existing debugging tools. This comes from the fact that every propagation event that has any effect on any entity in the constraint store is present in the trace. Furthermore, all information concerning which constraint is responsible for some pruning, which variable(s) are pruned and so on is present in the trace. Also, explanation events and method events provide the tool developers with additional useful information about the program execution. Of course, the information in our trace model is influenced by what SICStus Prolog provides but we have tried to be general when designing it. One example of this is that it is possible for constraints to have more than one attached demon which is not the case in SICStus Prolog.
8.2 Explanations

Up until now, we have only mentioned benefits when adding explanations to the event trace. Are there no negative effects?

Providing explanations may be expensive, regarding both space- and time complexity. However, there is always a choice between how much we are willing to pay in efficiency loss and how good results we want. A non-sharp explanation can always be constructed with a small amount of work. As an example, consider the pruning of some variable \( Y \) due to some constraint \( C \). The explanation \( 1 \rightarrow [\text{cond}(\text{all}, \text{P}, \text{dom})] \), \( \text{P} \) is the path term referring to all variables in \( C \), can always be applied even though it is the least sharp explanation. If such an explanation is used in an unfolding* process, it may create a very large set of responsible constraints. Explanation \( 3 \) for the \text{all}\	ext{distinct}/1 constraint is an example of a sharp explanation that may be computationally hard to construct, since it involves extra processing of the bipartite graph.

8.3 Implementation

It turned out that adapting a high end CLP(FD) system like SICStus Prolog to produce a trace of its actions is not an easy task. Particularly, going through the different global constraints implemented in C was very time consuming. During our project, only a subset of the constraints in SICStus was adapted to create trace events. One reason for why it is time consuming to add trace creation code to global constraints is of course the complexity of the filtering algorithms. One must completely understand the algorithm in order to not miss reporting any propagation. Another reason, at least in our implementation, is the complex Prolog entities such as lists and compound terms one must create using C strings. C is unfortunately not a perfect language when it comes to string manipulation. Perhaps one could think of some other way of storing such information in order to improve this.

Using an infinite buffer for storing the trace events might be a problem for large and complex applications. However, thorough testing is needed to sort this out. As a remark though, a traveling salesman problem modeled with 41 domain variables creates a trace log with slightly more than 50,000 events allocating about 1 MB of RAM. One solution to a possible memory resource problem is to only keep a subset of the trace in main memory and to store the main part on disk. This may of course be impractical when it comes to speed. Another solution is to not store the complete trace, but to process the events on the fly. This is the approach used by Langevine et al. [LDDJ 01b].

Another problem worth mentioning is that a variable's id is lost when it is bound to a singleton value. This is due to how SICStus handles ground variables and currently there is no workaround for the problem. It is particularly bad if you lose the name of a variable that is involved in some explanation(s). This will make it impossible to use that variable in an unfolding process. The problem can partly be solved by using the path term of a variable to retrieve its name. However, this will in some cases need unnatural restructuring of the source code.

9 Conclusion

In this thesis, we presented a new trace format for tracing CLP(FD) programs, in particular programs written in SICStus Prolog. We also presented some new ideas about explaining propagation events in global constraints. Furthermore, an implementation of the trace format in SICStus Prolog was described as well as the functionality of an implemented debugging tool built on-top of the trace. Our contributions include new ideas regarding the design of a CLP(FD) trace format. Another contribution is the implementation of such a trace format in a real CLP(FD) system. Yet another contribution is the provided explanations for domain pruning and failure caused by global constraints. Also, the ideas regarding describing the filtering algorithms of global constraints are new.

Of course, more work is needed in order for the trace to be useful. First of all, the complete set of global constraints in SICStus Prolog must be adapted so that their filtering algorithms produce trace events. Second, a thorough evaluation of some possible problems must be carried out. These are problems like the infinite buffer we currently use, the loss of id's regarding ground variables and efficiency problems due to

* See Section 7 for details.
creating explanations. Also, a set of debugging tools should be implemented in order to evaluate the design of the trace format. Possible lack of information in the trace format must be discovered.

Acknowledgments

First of all, I would like to thank Tamás Széredi for working with me on the project. The implementation of the debugging tool mentioned in Section 7 is completely his work. Besides that, we have worked in parallel on most parts regarding the design and implementation of the trace format.

I would also like to thank my two supervisors Mats Carlsson and Nicolas Beldiceanu for giving me the opportunity to take part in the project. Mats’s knowledge about SICStus Prolog and its CLP(FD) solver was of course a necessity for our work. Nicolas work regarding initial design of the trace format and his sharp ideas concerning explanations for global constraints has been invaluable. Thank you also both for reading early drafts of this document and giving fruitful comments.

I also want to thank my examiner Konstantinos Sagonas for proofreading almost finished versions of this document and finally for accepting it as my thesis.

Finally, I want to thank my good friend Harald Hammarström for proofreading the thesis, and my girlfriend Anna for being there.

References


A Graph Terminology

An undirected graph, $G = (V, E)$, consists of a set of vertices $V$, and a set of edges, $E$. An edge between two vertices $v_1$ and $v_2$ is denoted by a pair $(v_1, v_2)$. The difference between an undirected and a directed graph is how to interpret the edges in $E$. In an undirected graph, $(v_1, v_2)$ and $(v_2, v_1)$ denote the same edge. In a directed graph, $(v_1, v_2)$ denotes the directed edge from $v_1$ to $v_2$ while $(v_2, v_1)$ denotes the edge in the opposite direction. A bipartite graph is an undirected graph, $G = (V, E)$, where $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$. Furthermore, each edge $(v_1, v_2) \in E$ implies either $v_1 \in V_1$ and $v_2 \in V_2$ or $v_2 \in V_1$ and $v_1 \in V_2$. An example of a bipartite graph is shown in Figure 9(i). A matching in a bipartite graph, $G = (V, E)$, is a set, $E' \subseteq E$, of edges with no vertices in common. A maximal matching in a bipartite graph is a matching involving as many edges as possible. In Figure 9(ii), the set of edges $\{(a, g), (d, h)\}$ and $\{(a, e), (b, j), (c, f), (d, g)\}$ are both examples of matchings, but only the latter one is a maximal matching. A matched vertex is a vertex that is involved in a matching. An exposed vertex is a vertex that is not involved in a matching. For the maximal matching shown by bold edges in Figure 9(ii), $\{h, i\}$ is a set of exposed vertices while $\{a, b, c, d, e, f, g, j\}$ is a set of matched vertices. An alternating path in a graph is an even number of edges $(v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \ldots, (v_{i+(n-2)}, v_{n-1}), (v_{n-1}, v_n)$ where $v_i$ is exposed and $v_n$ is matched. The
Figure 9: (i) shows a bipartite graph, there are no edges between the two sets of vertices V1 and V2. (ii) shows a maximal matching for the same bipartite graph. (iii) shows a directed bipartite graph containing one strongly connected component and some alternating paths. (iv) shows a graph with a Hamiltonian circuit, bold edges.

sets of edges \{\{j,d\},\{d,i\}\} and \{\{j,d\},\{d,g\},\{g,a\},\{a,e\}\} in Figure 9(iii) are both examples of alternating paths. A **strongly connected component** in a directed graph, \(G = (V,E)\), is a set, \(V' \subseteq V\), of vertices and a set, \(E' \subseteq E\), of edges such that all vertices in \(V'\) are reachable from each other through the edges in \(E'\). In Figure 9(iii), the set of edges \{(a,e),(e,b),(b,f),(f,c),(c,g),(g,a)\} with its corresponding set of vertices is an example of a strongly connected component. Finally, a **Hamiltonian circuit** in a graph, \(G = (V,E)\), is a cycle that includes exactly the vertices in \(V\). An example of a graph with such a cycle is shown in Figure 9(iv). Note that for a bipartite graph, \(G = (V1 \cup V2, E)\), to be covered by a Hamiltonian circuit it, must be true that \(|V1| = |V2|\).

### B Trace Structure (BNF)

A BNF-like syntax extended with the following is used:

- **typewriter font** denotes terminal.
- **italic font** enclosed in ‘‘ and ‘’ denotes non-terminal.
- ‘*’ denotes zero or more repetitions, ‘+’ denotes one or more repetitions.
- ‘[’ and ‘]’ denote at most one occurrence.
- ‘(‘ and ‘)’ denote grouping.

\[
\langle \text{trace} \rangle ::= \langle \text{new\_ctr\_block} \rangle^+ \langle \text{top\_level} \rangle \\
\langle \text{top\_level} \rangle ::= (\langle \text{push\_demon\_because\_push\_demon} \rangle^+ \langle \text{wake\_demon\_block} \rangle^*)^+ \\
| \langle \text{labeling} \rangle \\
\langle \text{labeling} \rangle ::= \langle \text{labeling\_start} \rangle \langle \text{labeling\_step} \rangle [\langle \text{top\_level} \rangle] \\
| \langle \text{labeling\_step} \rangle [\langle \text{top\_level} \rangle] \\
| \langle \text{labeling\_fail} \rangle [\langle \text{top\_level} \rangle] \\
\langle \text{new\_ctr\_block} \rangle ::= \langle \text{begin\_new\_ctr} \rangle \langle \text{new\_ctr\_block\_internal} \rangle^+ \langle \text{end\_new\_ctr} \rangle \\
\langle \text{new\_ctr\_block\_internal} \rangle ::= \langle \text{new\_demon} \rangle [\langle \text{wake\_demon\_block} \rangle] \\
| \langle \text{method\_block} \rangle \\
| \langle \text{prune\_block} \rangle
\]

40
\[ \langle \text{method\_block} \rangle \ := \begin{multline*} \text{begin method} \ 	ext{info\_method}^* \ \langle \text{method\_block\_internal} \rangle^+ \\ \text{end method} \end{multline*} \]

\[ \langle \text{method\_block\_internal} \rangle \ := \begin{multline*} \langle \text{prune\_block} \rangle \\ \text{fail because fail} \\ \text{new\_ctr because (new\_ctr\_block)} \end{multline*} \]

\[ \langle \text{prune\_block} \rangle \ := \begin{multline*} \text{begin prune} \ \langle \text{prune\_block\_internal} \rangle^+ \ \text{end prune} \end{multline*} \]

\[ \langle \text{prune\_block\_internal} \rangle \ := \begin{multline*} \text{prune because before\_prune prune after\_prune} \end{multline*} \]

\[ \langle \text{wake\_demon\_block} \rangle \ := \begin{multline*} \text{begin wake\_demon} \ \langle \text{wake\_demon\_block\_internal} \rangle^* \\ \text{push\_demon because push\_demon}^* \ \text{end wake\_demon} \end{multline*} \]

\[ \langle \text{wake\_demon\_block\_internal} \rangle \ := \begin{multline*} \langle \text{method\_block} \rangle \ | \ \langle \text{prune\_block} \rangle \end{multline*} \]

\section{C Tracing the \texttt{trace\_me} Example}

Listed below is the full trace in raw format of the \texttt{trace\_me} example.

\subsection{C.1 The Program}

\begin{verbatim}
trace\_me :-
  domain([X,Y,V1,V2],1,6),
  all\_different([X,Y,3,V1,8,V2]),
  element(X,[2,4,3,8],Y),
  labeling([leftmost],[X,Y,V1,V2]).
\end{verbatim}

\subsection{C.2 The Trace}

\begin{verbatim}
begin\_new\_ctr(1,ctr\_domain,1,domain([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],1,6))
begin\_prune(2,force\_in\_interval(1,6),domain([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],1,6))
prune\_because(3,ctr\_domain\_1\_none\_domain([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],1,6))
before\_prune(4,[1,#[-]],[[inf|sup]],[[inf|sup]],[[inf|sup]],domain([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],1,6))
prune(5,ctr\_domain\_1,[1,#[*]],remove\_values([[inf|sup]],domain([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],1,6),success)
after\_prune(6,[1,#[*]],[[1|6]],[[1|6]],[[1|6]],[[1|6]],
domain([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],1,6))
end\_prune(7,succeed)
end\_new\_ctr(8,entail)
begin\_new\_ctr(9,ctr\_all\_different,1,all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8))
new\_demon(10,ctr\_all\_different,1,all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8))
begin\_wake\_demon(11,ctr\_all\_different,1,all\_different,1)
begin\_method(12,propagate\_ground\_variable)
  info\_method(13,ground\_variable,[1,#[3]-3,all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8)])
begin\_prune(14,remove\_value(3),all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8))
prune\_because(15,ctr\_all\_different,1,-[cond(1,[1,#3],eq(3))],
all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8))
before\_prune(16,[1,#[1,2,4,6]],all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8))
prune(17,ctr\_all\_different,[1,#[1,2,4,6],remove\_value(3),
all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8),success)
after\_prune(18,[1,#[1,2,4,6]],all\_different([fdvar\_1,fdvar\_2,fdvar\_3,fdvar\_4],3,8))
end\_prune(19,succeed)
end\_method(20,succeed)
end\_wake\_demon(21,delay)
\end{verbatim}
end_new_ctr(22, delay)
begin_new_ctr(23, ctr_element_1, element(fdvar_1, [2, 4, 3, 8], fdvar_2))
new_ctr = because (24, ctr_element_1, 1, cond(1, [2], length(eq(4))),
  element(fdvar_1, [2, 4, 3, 8], fdvar_2))
end_new_ctr(25, ctr_in_1, fdvar_1 in 1.4)
begin_prune(26, force_in_values([[1|4]]), fdvar_1 in 1.4)
prune = because (27, ctr_in_1, none, fdvar_1 in 1.4)
before_prune(28, 3, [[1|2], [4|6]], fdvar_1 in 1.4)
prune(29, ctr_in_1, 1, remove_interval(5, 6), fdvar_1 in 1.4, succeed)
after_prune(30, 1, [[1|2], [4|6]], fdvar_1 in 1.4)
end_prune(31, succeed)
end_new_ctr(32, entail)
new_demon(33, ctr_element_1, element_3, [[1]] - dom, [2, #*] - minmax, [3] - minmax,
  element(fdvar_1, [2, 4, 3, 8], fdvar_2))
begin_wake_demon(34, ctr_element_1, element_3)
begin_method(35, prune_y)
begin_prune(36, adjust_min(4), element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune = because (37, ctr_element_1, 1, element(fdvar_1, [2, 4, 3, 8], fdvar_2))
before_prune(38, 3, [[1|2], [4|6]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune(39, ctr_element_1, [3], remove_value(1),
  element(fdvar_1, [2, 4, 3, 8], fdvar_2), succeed)
after_prune(40, 3, [[1|2], [4|6]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
end_prune(41, succeed)
begin_prune(42, adjust_max(4), element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune = because (43, ctr_element_1, 1, element(fdvar_1, [2, 4, 3, 8], fdvar_2))
before_prune(44, 3, [[1|2], [4|6]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune(45, ctr_element_1, [3], remove_interval(5, 6), element(fdvar_1, [2, 4, 3, 8], fdvar_2), succeed)
after_prune(46, 3, [[1|2], [4|6]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
end_prune(47, succeed)
end_method(48, succeed)
begin_method(49, prune_x)
begin_prune(50, remove_values([[1|4]]), element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune = because (51, ctr_element_1, 1, cond(1, [3], notinset([[1|8]]))),
  element(fdvar_1, [2, 4, 3, 8], fdvar_2))
before_prune(52, 1, [[1|2], [4|6]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune(53, ctr_element_1, [3], remove_value(4), element(fdvar_1, [2, 4, 3, 8], fdvar_2), succeed)
after_prune(54, 1, [[1|2]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
end_prune(55, succeed)
end_method(56, succeed)
end_wake_demon(57, delay)
end_new_ctr(58, delay)
labeling_start(59, fdvar_1, [[1|2]], step(ap))
labeling_step(60, fdvar_1, [[1|2]], fix_to_value(1))
push_demon = because (61, [1, #1], val_all_different([fdvar_1, fdvar_2, 3, fdvar_3, 8], fdvar_4))
push_demon(62, ctr_all_different_1, all_different_1)
push_demon = because (63, [1], dom, element(fdvar_1, [2, 4, 3, 8], fdvar_2))
push_demon(64, ctr_element_1, element_3)
begin_wake_demon(65, ctr_element_1, element_3)
begin_method(66, prune_y)
begin_prune(67, adjust_max(2), element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune = because (68, ctr_element_1, 1, cond(1, [2], neg(2))),
  element(fdvar_1, [2, 4, 3, 8], fdvar_2))
before_prune(69, 3, [[1|2], [4|6]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
prune(70, ctr_element_1, [3], remove_value(4), element(fdvar_1, [2, 4, 3, 8], fdvar_2), succeed)
after_prune(71, 3, [[1|2]], element(fdvar_1, [2, 4, 3, 8], fdvar_2))
end_prune(72, succeed)
end_method(73, succeed)
end_wake_demon(74, entail)
begin_wake_demon(75, ctr_all_different_1, all_different_1)
begin_method(76, propagate_ground_variable)
info_method(77, ground_variable, [1, #2], 2, all_different([fdvar_1, fdvar_2, 3, fdvar_3, 8], fdvar_4))
begin_prune(78, remove_value(2), all_different([fdvar_1, fdvar_2, 3, fdvar_3, 8], fdvar_4))
prune = because (79, ctr_all_different_1, 1, cond(1, [1, #2], eq(2)),
  all_different([fdvar_1, fdvar_2, 3, fdvar_3, 8], fdvar_4))
before_prune(80, 1, [[1|2], [4|6]], [[1|2], [4|6]],
  all_different([fdvar_1, fdvar_2, 3, fdvar_3, 8], fdvar_4))
prune(81, ctr_all_different_1, [1, #4|6], remove_value(2),
  all_different([fdvar_1, fdvar_2, 3, fdvar_3, 8], fdvar_4), succeed)
after_prune(82,[1,#[4],[6]],[[1],[1],[4]],[[1],[1],[4]],[[1],[1],[4]],[[1],[1],[4]],[[1],[1],[4]],[[1],[1],[4]]),
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
end_prune(83,succeed)
info_method(84,ground_variable,[1,#1]-1,
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
begin_prune(85,remove_value(1),all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
prune_because(86,ctr_all_different_1,1-[cond(1,[1,#1],eq(1))],
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
before_prune(87,[1,#[4],[6]],[[1],[1],[4]],[[1],[1],[4]],[[1],[1],[4]]),
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
prune(88,ctr_all_different_1,[1,#[4],[6]],remove_value(1),
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4])),succeed)
end_prune(89,[1,#[4],[6]],[[1],[1],[4]],[[1],[1],[4]]),
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
end_prune(90,succeed)
end_method(91,succeed)
end_wake_demon(92,delay)
labeling_start(93,fdvar_3,[4],[6]),step(up))
labeling_step(94,fdvar_3,[4],[6]),fix_to_value(4))
push_demon_because(96,[1,#4],val,all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
push_demon(96,ctr_all_different_1,all_different_1)
begin_wake_demon(97,ctr_all_different_1,all_different_1)
begin_method(98,propagate_ground_variable)
info_method(99,ground_variable,[1,#4]-4,all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
begin_prune(100,remove_value(4),all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
prune_because(101,ctr_all_different_1,1-[cond(1,[1,#4],eq(4))],
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
before_prune(102,[1,#6],[[4],[6]],all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
prune(103,ctr_all_different_1,[1,#6],remove_value(4),
all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4])),succeed)
end_prune(104,[1,#6],[[5],[6]],all_different([fdvar_1,fdvar_2,3,fdvar_3,8,fdvar_4]))
end_prune(105,succeed)
end_method(106,succeed)
end_wake_demon(107,entail)
labeling_start(108,fdvar_4,[5],[6]),step(up))
labeling_step(109,fdvar_4,[5],[6]),fix_to_value(6))

43