A Generic Trace Model for Finite Domain Solvers*

Pierre Deransart[1], Mireille Ducassé[2], and Ludovic Langevine[1]

[1] INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
{Pierre.Deransart, Ludovic.Langevine}@inria.fr
[2] IRISA/INSA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
Mireille.Ducasse@irisa.fr

Abstract. Debugging tools are essential to help tune constraint solving programs. However, at present, these tools have to be redesigned and re-implemented for each constraint solver whereas much could be factorized. This paper sets the foundations to enable debugging tools to be defined almost independently of finite domain solvers. We propose an "observational semantics" which formalizes relevant aspects of constraint programming and solving. Based on this semantics tracers can then build "generic traces". We show that at least two representative families of finite domain solvers can be "observed" by this semantics: CLP (Gnu-Prolog) and CSP (Ilog solver and Choco). Debugging tools can then consider the generic trace and do not have to take into account all the details of the solvers. We show in the following that many existing debugging tools find in the generic trace all the information they require.

1 Introduction

Many interactive debugging tools have been developed in order to debug applications based on finite domain constraints resolution, for example, Grace [20], the Oz Explorer [23], the Oz Constraint Investigator [21], the CHIP global constraint visualizer [25], the S-box model [15], the Christmas Tree visualizer [3].

Despite their very interesting functionalities these tools have two major drawbacks: they are hard to implement (a specific solver instrumentation is often needed) and they have a low degree of portability (they are implemented on a specific platform and the work to port them from one platform to another may require a tremendous effort). From an industrial point of view, one would like to easily implement new observation tools without modifying the solver. Moreover, industrials often develop applications with different solvers. It is thus for them of primary importance to minimize the efforts of adapting their debugging tools from one platform to another. From a research point of view, one would like to experiment with different tools on different platforms without too much re-engineering effort.

The previous drawbacks are all the more daunting since most of the tools are general purpose oriented, i.e. they do not depend on a particular application

* This work is partly supported by OADymPPaC, a French RNTL project.

$$S_i \xrightarrow[\text{Section 3}]{\alpha_i} OS \xrightarrow[\text{Section 2}]{\beta} GT \xrightarrow[\text{Section 4}]{\delta_j} DT_j$$

Fig. 1. From Constraint Solvers to Debugging Tools through a Generic Model

and they use quite the same kind of basic information collected in the course of the program execution. From this same basic information, more or less complex views are generated.

This paper investigates this common ground. It specifies a common *observational semantics* such that most of the propagation based solvers over finite domains can be viewed as a specialization of such semantics. This semantics is then used to systematically define a *generic trace* model of this family of solvers, such that most of the information needed by the debugging tools can be derived from the generic trace.

Figure 1 illustrates the relationships between a family I of solvers with semantics $\{S_i\}_{i \in I}$ and the input data $\{DT_j\}_{j \in J}$ of a family J of debugging tools. In the middle of the diagram are the observational semantics (OS) and the derived (by the mapping β) generic trace (GT). The observational semantics is relative to a family of solvers and debugging tools for which the mappings α_i and δ_j respectively exist. This semantics specifies possible transitions of *observable states*. It does not specify how the transitions should be combined. In this sense it is not and cannot be a complete specification describing constraint resolution. It must be viewed as a reflect of each solver of the family. With such a semantics and trace, it becomes possible to implement separately a tracer for each solver (the tracer is thus the composition of α_i and β) and a debugging tool can be developed on the basis of the generic trace.

Note that the trace analysis approach has been proved both powerful and efficient for Prolog and Mercury. Indeed, from a fine-grained execution trace, a number of interesting views about program executions could be efficiently generated by trace analysis [11, 17]. Note also that no general formal model of finite domain constraint solving can be used as a basis for a formal specification of generic trace model. Most of the general solver models [22, 5, 26, 2, 13] are far from solvers implementations or cover specific aspects only. Formal specifications exist for some standardized programming languages, but there is no existing consensus for constraint programming.

The contribution of this paper is thus to define a common observational semantics and a *generic trace* model for finite domain solvers. It sets the foundations to enable debugging tools to be defined almost independently of finite domain solvers. In the following, we show that at least two representative families of finite domain solvers can be "observed" by this semantics: CLP (Gnu-Prolog) and CSP (Ilog solver and Choco). We also show that many existing debugging tools find in the generic trace the whole information they require.

The paper is organized following Fig. 1. Each section describes one element of the diagram. We first present the observational semantics and the derived trace (Section 2), then we informally show how different kinds of known constraint languages fit with this semantics (Section 3); finally the family of potential debugging tools is characterized by classes of input data, called *observables* and it is shown that these data can be inferred from the generic trace analysis (Section 4).

2 Observational Semantics and Generic Trace

Here we present an observational semantics and its derived generic trace.
The observational semantics is defined with a set of transition rules between *observed states.*

Definition 1. *(Observed States)*
An observed state is a 9-tuple: $\mathcal{S} = (T_s, \mathcal{C}, \mathcal{V}, \mathcal{D}, A, S_c, S_e, E, R)$
where: T_s *is the time stamp of the state;* $\mathcal{C}$ *is the whole set of constraints declared until this state;* $\mathcal{V}$ *is the set of variables declared until this state;* $\mathcal{D}$ *is the set of domains of the variables of* $\mathcal{V}$*;* A *is the set of active couples of the form (constraint, solver event*[3]*);* S_c *is the set of sleeping constraints;* S_e *is the set of sleeping solver events;* E *is the set of entailed constraints;* R *is the set of unsatisfiable (or false) constraints.*

Some remarks The time stamp corresponds to the relative time (counted from the first state of execution) when the state is reached. Then all states are ordered by their time stamp. The set of all "previous" states (i.e. states with a time stamp less than the time stamp of the current state $\mathcal{S}$) is called $\Sigma(\mathcal{S})$.

It must be noticed that $\mathcal{C}$ may include constraints which have been withdrawn from the store. The couples (constraint, solver event) are sometimes called "running propagators" [21].

The *store of constraints* is actually the set of constraints of $\mathcal{S}$, it is called σ in the following and it is defined as follows:

$$\sigma = \text{constraints\_of}(A) \cup S_c \cup E \cup R$$

The set of variables of a given constraint c is noted $\mathbf{var}(c)$. The domain of a given variable x is noted D_x. The predicate $success(\mathcal{S})$ (resp. $failed(\mathcal{S})$) holds, when a state $\mathcal{S}$ is considered as a solution (resp. failure) by the solver. The predicate *choice-point*$(\mathcal{S})$ holds when the state $\mathcal{S}$ is considered as a choice point by the solver. The predicate $false(c)$ (resp. $entailed(c)$) holds when the constraint c is considered as unsatisfiable (resp. entailed) by the store of the current state. *no-event*(c) ensures that there is no pending event involving c.

[3] This work inherits from two areas, constraint solving and debugging, which both use the word "event" in correlated but different meanings: a *solver event* is produced by the solver and has an impact on the constraint propagation (for example, the addition of a new constraint or the reduction of the domain of a given variable); a *trace event* corresponds to an element of the trace.

$$\textbf{new variable}\ \frac{x \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{x\},\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(x, D_x)\}}\ \{D_x \text{ is the initial domain of } x\}$$

$$\textbf{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \forall x \in \mathbf{var}(c) \cdot (x \in \mathcal{V})}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

$$\textbf{post}\ \frac{c \in \mathcal{C}\ \wedge\ c \notin \sigma}{A \leftarrow A \cup \{(c, \bot)\}}$$

$$\textbf{deactivate}\ \frac{c \in \sigma\ \wedge\ \textit{no-event}(c)}{\sigma \leftarrow \sigma - \{c\}}$$

$$\textbf{restore}\ \frac{x \in \mathcal{V} \wedge \Delta_x \cap D_x = \emptyset}{D_x \leftarrow D_x \cup \Delta_x}\ \{\Delta_x \text{ is a subset of the initial } D_x\}$$

$$\textbf{new child}\ \frac{\textit{choice-point}(\mathcal{S})}{}$$

$$\textbf{jump to}\ \frac{\mathcal{S}' \in \Sigma(\mathcal{S}) \wedge \mathcal{S}' \neq \mathcal{S}}{\mathcal{S} \leftarrow \mathcal{S}'}$$

$$\textbf{solution}\ \frac{\textit{success}(\mathcal{S})}{}$$

$$\textbf{failure}\ \frac{\textit{failed}(\mathcal{S})}{}$$

Fig. 2. Observational Semantics: Control

The observational semantics is specified by rules which characterize a possible state transition. In the initial observed state all sets are empty. Each rule is of the form

$$\textbf{name}\ \frac{\textit{conditions\_on\_current\_observed\_state}\ \mathcal{S}_1}{\textit{new\_observed\_state}\ \mathcal{S}_2}\ \{\textit{definitions}\}$$

In most of the rules, only the changes wrt $\mathcal{S}_1$ are noted, the unchanged items are not reproduced. Notice that in some rules the state remains totally unchanged.

The observational semantics is divided into two parts: the first one is devoted to the control of the execution (variables and constraints management, search handling), the second one is devoted to the propagation details (domain filtering and constraint awakenings).

2.1 Control

The rules of Fig. 2 describe the control part of the observational semantics. The control part handles the constraint store and drives the search. Rule **new variable** specifies that a new variable x is introduced in $\mathcal{V}$ and that its initial domain is D_x. Rule **new constraint** specifies that the solver introduces a new constraint c in $\mathcal{C}$. Note that this constraint does not yet belong to the store σ. The activation of a constraint c is specified by the rule **post**: the constraint is

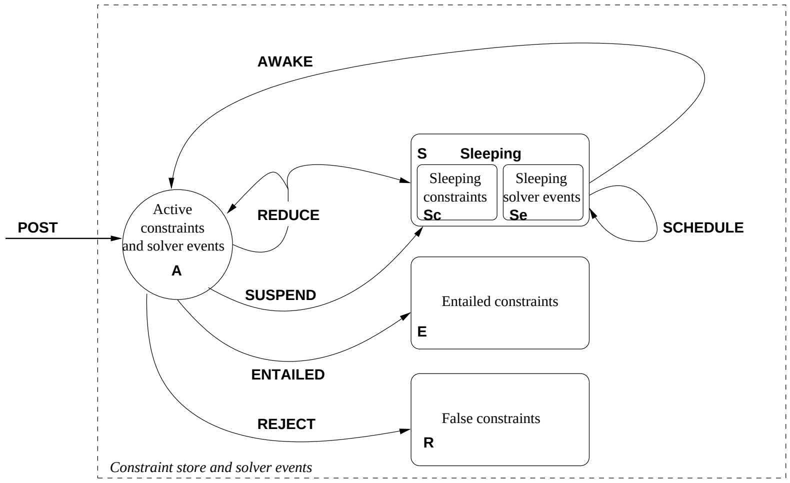

Fig. 3. Illustration of the Transition Described by the Propagation Rules

entered in the store and ready to make first domain reductions. Rule **deactivate** removes a constraint c from the store σ. It still belongs to $\mathcal{C}$. The rule **restore** specifies that the solver is restoring some values Δ_x in the domain of a variable x.

The following four rules describe the construction of a search-tree (see Section 4). Rule **new child** specifies that the current observed state corresponds to a new node of the search-tree which is candidate as choice-point, i.e. the solver may jump back to this state later in order to make a different search. Such a state is called a *choice point*. Jumping from the current observed state to a previous choice point is specified by the rule **jump to**. Finally, the rule **solution** specifies that in the current observed state, the store is a solution. For the rule **failure**, if there is a false constraint in the store, the store is inconsistent - in which case a rule **reject** has previously been applied-, otherwise the state is simply declared as failed, which corresponds to the fact that the search is definitively stopped at this point and will likely continue by application of a **jump to** rule from another state.

2.2 Propagation

The propagation can be described by state transition rules acting in the structured observed state, as illustrated by Fig. 3. These rules are formalized by Fig. 4. A, S_c, S_e, E and R are parts of the current state $\mathcal{S}$ previously defined. The active pairs in A can reduce some domains because of the solver events. Rule **reduce** specifies that the solver reduces the domain of a set of the variables of an active constraint attached to a solver event. Several domains can be reduced in a single step. $\bar{e}$ is a set of solver events that characterize the reduction(s). Examples of

$$\textbf{reduce}\ \frac{(c,e) \in A \ \wedge\ \bar{x} \neq \emptyset}{\forall x \in \bar{x}.(D_x \leftarrow D_x - \Delta_x^c),\quad S_e \leftarrow S_e \cup \bar{e}'} \quad \begin{cases} \bar{x} = \{x \in \mathbf{var}(c) \mid \Delta_x^c \neq \emptyset\} \\ \Delta_x^c \text{ is a subset of } D_x \text{ to remove} \\ \bar{e}' \text{ is a set of solver events on } \bar{x} \end{cases}$$

$$\textbf{suspend}\ \frac{(c,e) \in A \ \wedge\ \forall x \in \mathbf{var}(c) \cdot \Delta_x^c = \emptyset \ \wedge\ \neg entailed(c) \ \wedge\ \neg false(c)}{A \leftarrow A - \{(c,e)\},\quad S_c \leftarrow S_c \cup \{c\}} \quad \{\Delta_x^c \text{ is the set of values to remove from } D_x\}$$

$$\textbf{entail}\ \frac{(c,e) \in A \ \wedge\ entailed(c)}{A \leftarrow A - \{(c,e)\},\quad E \leftarrow E \cup \{c\}}$$

$$\textbf{reject}\ \frac{(c,e) \in A \ \wedge\ false(c)}{A \leftarrow A - \{(c,e)\},\quad R \leftarrow R \cup \{c\}}$$

$$\textbf{awake}\ \frac{c \in S_c \ \wedge\ e \in S_e \ \wedge\ awakecond(c,e)}{A \leftarrow A \cup \{(c,e)\}}$$

$$\textbf{schedule}\ \frac{c \in S_c \ \wedge\ e \in S_e \ \wedge\ action(c,e)}{S_c \leftarrow S'_c,\quad S_e \leftarrow S'_e} \quad \{S'_c \text{ and } S'_e \text{ are solver dependant}\}$$

Fig. 4. Observational Semantics: Propagation

solver events are "increasing of the lower bound of the domain of x" or "instantiation of the variable y". Each domain reduction generates new solver events that are recorded in S_e. When an active constraint cannot reduce any domain at the moment, it is suspended in S_c (rule **suspend**). An active constraint that is true is said to be *entailed* and put in E (rule **entail**). An active constraint that is unsatisfiable is said to be *false* and put in R by the **reject** rule. Solver events in S_e are waiting to be propagated through sleeping constraints in S_c. A solver dependent strategy combines a sleeping constraint and a solver event together (**schedule**). If the condition $awakcond(c,e)$ holds, such a pair (c: *constraint to awake*, e: *awakening cause*) can then be activated by an **awake** transition. This may lead to new domain reductions.

2.3 Generic Trace

The two sets of rules presented by Fig. 2 and Fig. 4 specify the observational semantics. The generic trace derives from this semantics. Each trace event corresponds to the application of a semantics rule. Therefore each semantics rule gives its name to a type of trace event: a *port*.

Definition 2 (Generic Trace). *The generic trace is a sequence of trace events consisting of:*

- *a sequential event number;*
- *the port (the name of one of the semantics rules);*
- *the observed state of the solver;*
- *some specific attributes depending on the port.*

Control Ports		Propagation Ports	
new variable	$x, \quad D_x$	**reduce**	$c, \quad e, \quad \bar{x}, \quad \bar{e},$ $\Delta^c_{\bar{x}} = \{\Delta^c_y \mid y \in \bar{x}\}$
new constraint	c	**suspend, entail**	c
post, deactivate	c	**reject, awake**	$c, \quad e$
restore	$x, \quad \Delta^c_x$	**schedule**	$c, \quad e$
new child	*none*		
jump to	A reference to $(\mathcal{S}')$		
solution, failure	*none*		

Table 1. Specific Attributes of the Ports

The specific attributes correspond to conditions and characteristics of specific actions performed by each rule. For example the port **reduce** has additional attributes corresponding to the variables $\bar{x}$ that have been reduced, the value withdrawals $\Delta^c_{\bar{x}}$ and the solver events that these reductions generate $\bar{e}$. Table 1 presents the list of the specific attributes for each rule, using the notations of Fig. 2 and Fig. 4.In [6] a DTD XML is presented with a concrete format for all ports and attributes. It is worth noticing that an additional port (not described here) is defined, called **newannotation**. It is used to declare more complex objects to be traced. This allows to design more user and application oriented debugging tools.

3 The Observational Semantics vs Two Solver Families

This section confronts the observational semantics (OS) described in the previous section to several real solvers from two families: CLP (GNU-Prolog) and CSP (Choco and Ilog Solver). It shows that there exists a mapping from the known operational semantics of each real solver into OS. This is obtained by showing firstly that there is a one-to-one correspondence between the observational states on OS and some real states in the real solver and secondly that there are transitions between these states in the real solver which correspond to rules of OS. These real state transitions can be viewed as a specialization of rules of OS. As the operational semantics of the real solvers is almost informal (the sources of information come from manuals or paper description) the correspondence is presented informally. Five of the most relevant rules are used for the comparison: **reduce**, **schedule** and **awake** for propagation rules, and **new child** and **jump to** for control rules.

3.1 A clp(FD) Solver: GNU-Prolog

GNU-Prolog is a free Prolog compiler with an embedded finite domain solver developed by Diaz [14]. Its fd-solver is not currently traced. GNU-Prolog is based on

```
x_plus_y_eq_z(fdv X,fdv Y,fdv Z)
{
      start X in min(Z) - max(Y) .. max(Z) - min(Y)
      start Y in min(Z) - max(X) .. max(Z) - min(X)
      start Z in min(X) + min(Y) .. max(X) + max(Y)
}
```

Fig. 5. The implementation of the constraint $x + y = z$ with a bound consistency algorithm in GNU-Prolog (excerpt of the GNU-Prolog distribution)

$$\textbf{reduce}\ \frac{A = \{(x\ \textbf{in}\ r, \bot)\}}{D_x \leftarrow D_x \cap r(\mathcal{D}),\quad S_e \leftarrow \bar{e}} \left\{ \begin{array}{l} r(\mathcal{D}) \text{ is the evaluation of } r \text{ wrt } \mathcal{D} \\ \bar{e} \text{ is a set of solver events involving } x \end{array} \right\}$$

$$\textbf{schedule}\ \frac{e \in S_e \wedge (x\ \textbf{in}\ r) \in S_s \wedge r \text{ depends on } e}{S_s \leftarrow S_s - \{x\ \textbf{in}\ r\},\quad S_q \leftarrow S_q \cup \{x\ \textbf{in}\ r\}}$$

$$\textbf{awake}\ \frac{A = \emptyset \wedge x\ \textbf{in}\ r \in S_q}{S_q \leftarrow S_q - \{x\ \textbf{in}\ r\}, A \leftarrow \{(x\ \textbf{in}\ r, \bot)\}}$$

Fig. 6. Specialization of three propagation rules for GNU-Prolog.

the so-called "RISC approach" [5] which consists in translating at compile-time all complex constraints into simple constraints of the form X **in** r where X is a finite domain variable and r is a *range*. A range represents a non empty finite set of values. It can be *constant* (e.g. 1..10) or *indexical*. An indexical is an expression which uses the following operators: `dom(Y)`, `min(Y)`, `max(Y)` or `val(Y)` which respectively denote the whole domain of a variable `Y`, its lower bound, its upper bound and the value `Y` is assigned to. Such an expression can also use union, intersection or integer arithmetic. Figure 5 shows the code to compile the built-in constraint $x + y = z$ in GNU-Prolog with three simple constraints. The three variables `X`, `Y` and `Z` are declared as finite domain variables (`fdv`). Function `start` posts a simple constraint. A constraint such as $x + y = z$ is represented by 3 simple constraints in the store. Each one constraints the domain of one of `X`, `Y` or `Z`.

Observed State. The simple constraints are the elementary entities the constraint engine handles. Therefore the store σ is the set of all posted simple constraints. The set of **active** constraints and events contains the single constraint under examination, $A = \{(x\ \textbf{in}\ r, \bot)\}$. The set of **entailed** constraints consists of simple constraints which are recognized as entailed by GNU-Prolog. The set of **rejected** constraints R has at most one element X **in** r. $false(X\ \textbf{in}\ r)$ holds iff the domain of X is empty. S_c is a partition, $S_c = S_s \cup S_q$. S_s is the set of **suspended** constraints which have not been scheduled yet. S_q is the **propagation queue**, the set of constraints waiting for their awakening. S_e is the set of solver events to propagate represented by pairs (v, u) where v is a fd-variable and u is a type of update of D_v, for instance the update of its lower bound or its grounding. When the solver event is unknown, $\bot$ is used.

Propagation. The three rules **reduce**, **schedule** and **awake** are specialized for GNU-Prolog as shown by Fig. 6. The rule **reduce** concerns the active constraint associated with the bottom event ($A = \{(x \textbf{ in } r, \bot)\}$). It withdraws from D_x the values Δ_x that are not in the current evaluation of r (denoted by $r(\mathcal{D})$): $\Delta_x = D_x - r(\mathcal{D})$. The solver compares the former D_x to the new one in order to generate one or several solver events about x: $\bar{e}$. These events are put in S_e and immediately processed. The rule **schedule** selects a constraint in S_s whose range may have changed and enqueues it in S_q. A range may have changed if it depends on an event in S_e. For example, if an event in S_e is "update of the lower bound of the domain of `Y`", every sleeping constraint whose range depends on `min(Y)` (the lower bound) or `dom(Y)` (the whole domain of `Y`) is put in S_q. When every solver event has been processed, S_e is emptied: this action is not formalized by these rules. After all this enqueuing, the active constraint can be suspended or entailed. When no constraint is active and some constraints are in S_q, **awake** dequeues a constraint from S_q and activates it with the bottom event $\bot$.

Control. In GNU-Prolog, the control part inherits from logic programming and standard Prolog. The search tree corresponds to the usual SLD-tree [8]. Each node of the SLD-tree can be considered as a choice-point and results from one application of the **new child** rule. Backtracking from a leaf to a former choice-point is then clearly modeled by the **jump to** rule. A top-level success corresponds to a state which *success* and thus to an application of a **solution** rule. A failure *due to the FD solver*, namely a domain emptying out, corresponds to a **reject** rule. In Prolog a failure may also be due to the fact that no clause head is unifiable with the current goal. In both cases the corresponding state will satisfy the *failure* condition of the **failure** rule. The attributes of the **restore** and **deactivate** ports can be deduced from both origin and target states. However, due to the simple search strategy of GNU-Prolog (by backtracking to a parent state), the ports new child and jump to provide the sufficient information to display the search-tree

3.2 CSP Solvers: Choco and Ilog Solver

This section studies two solvers whose computation is based on solver event reaction: Ilog Solver and Choco. In these solvers, a constraint is defined by two sets of rules: static and dynamic. Static rules are used to reduce domains without knowing what happens before, for example when the constraint is posted. Dynamic rules are used to propagate specific domain updates (solver events). Due to space limitation, Choco is briefly presented. A more comprehensive study can be found in [7].

Ilog Solver. Ilog Solver [16] is a widely used commercial solver with an existing *ad hoc* tracer. Ilog Solver functionning is, at the observer level, comparable to Choco. The so-called *constraint propagation queue* is actually a queue of variables that have been updated. When a constraint variable is updated, it is put

$$\textbf{reduce}\ \frac{(c,p) \in A \wedge x \in \mathbf{var}(c) - P \wedge \Delta_x \neq \emptyset}{D_x \leftarrow D_x - \Delta_x, \quad S_e \leftarrow S_e \cup \{x\}}\ \{\Delta_x \text{ is a set of inconsistent values}\}$$

$$\textbf{schedule}\ \frac{x \in S_e \wedge P = \emptyset}{P \leftarrow \{x\}}\ \{p \text{ becomes the variable } \textit{in process}\}$$

$$\textbf{awake}\ \frac{P = \{p\} \wedge c \in S_c \wedge p \in \mathbf{var}(C)}{Q \leftarrow Q - \{x\}, \quad A \leftarrow A \cup \{(c,p)\}}$$

Fig. 7. Specialization of three propagation rules for Ilog Solver

in this queue if it is not already. As long as there are variables in the queue, the algorithm selects the first one which is said *in process*. Some constraints posted on the processed variable are activated in turn. The method `propagate` is the only one performing propagation for a given constraint. It uses elementary domain reduction functions (for instance update of the upper bound of a domain, removal of a given value).

Observed State. The store σ is the set of all the constraints taken into account by the solver. A contains the constraint whose method `propagate` is called, attached to the variable *in process*, $A = \{(c,p)\}$. E is always empty. R contains at most one constraint which has detected its unsatisfiability. S_c is the set of all the other constraints in σ. S_e is the queue of updated variables. P is a subset of S_e that contains at most one variable: the one *in process*

Propagation. Figure 7 formalizes the specialization of **reduce**, **schedule** and **awake**. **reduce** is an elementary domain reduction of a variable x due to a constraint c with p in process. The reduced variable must not be in process as assumed by the Ilog Solver specifications. The reduced variable is stored in S_e. When a variable has been completely processed, P is emptyied out. Then **schedule** selects a new variable in S_e and puts it in P: this is the new variable *in process*. **awake** activates a constraint depending on the variable in process.

Control. The mapping between our model and the control part of Ilog Solver can be shown by its existing trace mechanism. In fact, a *search monitor* (class `IlcSearchMonitorI`)) watches and reports control events in a search. For instance, the `whenCreate` method tells that the current state is a choice point which can be restored later: this is the (**new child**) transition. The `whenEndMove` method is called when the solver has just restored a previous state: this is exactly the **jump to** transition. Finally the `whenFail` and `whenSolution` methods are called repectively on each failure leaf and each solution leaf (**failure** and **solution** rules).

Choco. Choco is an object oriented library of constraint programming designed by Laburthe et al. [18]. Each constraint over finite domains is an instance of the *IntConstraint* class. This class has several methods to encode the reduction rules of a constraint. Method `propagate` performs the first domain reductions by applying the static reduction rules. The dynamic rules are implemented by five methods called when a relevant event occurs: `awakeOnInf` (reacts to lower bound updates), `awakeOnSup` (upper bound), `awakeOnInst` (instantiation of a variable), `awakeOnRem` (removal of a single internal value) and `awakeOnVar` (removal of several internal values). Such methods are the elementary entities the solver handles to perform the propagation. They call domain reduction methods. For instance, `updateSup` ensures that the upper bound of a domain is lower or equal to a given integer. If it is not, a reduction is made. How our generic model fits Choco is detailed in [7].

4 Observables

This section confronts the generic trace of section 2 to families of debugging tools. It is requested that the input data of the tools can be recognized in the sequence of events of the generic trace. Instead of analyzing one after the other the tools of a large collection, we identify and classify the basic data structures used in most of the known tools. These data structures are called *observables* and can be classified in three main classes: observables related to the problem, observables related to the search space and observables related to the propagation. We limit this presentation to these classes, a more comprehensive study can be found in [10]. For each class of observables we show that all required objects can be retrieved from the generic trace.

4.1 Observables related to the problem

A typical program with finite domain constraints consists of, firstly, the definition of a set of variables, their domains and a set of constraints, i.e relations between the variables; and secondly, a labeling phase. Constraints can be simple relations, they can also be complex combinations of basic constraints or, more generally, global constraints which encapsulate specific resolution algorithms. New variables and constraints can be added or removed in the course of the search for solutions.

For debugging tools, variables, constraints, complex constraints must be known with their different possible forms (external, as declared by the user, or internal, as represented in the solver) in order to make friendly usable tools. With such information a debugging tool will have the whole information needed to elaborate complex display like constraint graphs, hierarchical incidence matrix, or, when combined with control of execution, variable update views.

A generic trace contains at every event the information about variables and constraints. Furthermore the ports **new variable**, **reduce** and **restore** keep track of the evolution of the variables. The ports **new constraint**, **post** and **deactivate** keep track of the evolution of the constraint store.

In many cases the user would prefer the tools to understand the constraint abstractions that he is using inside his program, and not their detailed unfolding into primitive constraints. A better handling of variables is also needed. A trace annotation system is being designed to enable such abstractions, as mentioned

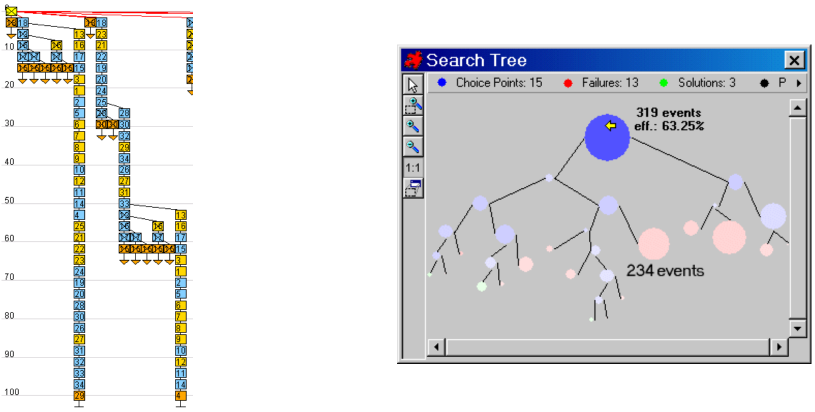

Fig. 8. A labeling tree of `SEARCH-TREE TOOL` and the OPL studio Christmas-tree

in Section 2.3. For example in GNU-Prolog it is necessary to relate a simple constraint with the corresponding user constraint. This enable specific views for global constraints or subproblems as proposed in [25].

4.2 Observables related to the search space

Many debugging tools allow users to analyze the complexity of the search space by different means. Most of them represent and display it as a tree. In the Oz Explorer [23], the parent nodes of the tree correspond to search choices (splitting of domain of some variable for example). There are zooming and collapsing facilities. A tree visualized in the search-tree tool of Cosytec (see Figure 8) [24] corresponds to the labeling phase of the constraints resolution. Each node corresponds to a variable choice with domain value assignment. There are also several zooming and collapsing facilities. The user can display different information associated to each node in the form of side tables. A slightly different view, called "Christmas-tree", is provided by the OPL-studio debugger of ILOG [3] (see Figure 8). Here the tree corresponds to a search-tree like in Oz-explorer, but with each node a disk is displayed whose surface is related to some measure (like the number of domain reductions). In the Calypso debugger [1] the whole Prolog search-tree may be displayed. There is also a language to define different representations of the search-tree.

The structure of the search-tree can be built from the two ports **new child** and **jump to**. In fact the **new child** port corresponds to a state identified by the solver as a choice point and the debugging tool can thus create a new node as a new child of the last visited choice point. The tool must know which is the last visited choice point. As a matter of fact, we have been able to easily implement the display of the search tree, this is reported in [19]

4.3 Observables related to Propagation

Propagation corresponds to fine grained actions performed by the solvers in order to reduce the variable domains and to converge to a solution. In Grace [20], users can get information on domain updates and constraint awakenings. This system thus provides the ability to evaluate expressions using the current domain of the variables which is part of the attributes of the ports. The CHIP tool [24] also provides update views in which, for example, useless awakenings are visible. In this tool as in the OPL-studio tool specific explorer windows allow propagation steps to be observed in a structured manner. The visualization tool by Carro and Hermenegildo [4] traces the constraint propagation in 3 dimensions according to time, variable, and size of the domain. The S-boxes of Goualard and Benhamou [15] allow to structure the propagation in the store and to display the constraint store. In the "constraint Investigator" [21] a debugger, complementary to the Oz-explorer, is dedicated to the propagation steps; the solver events, called "propagators" are treated as first class objects.

All these tools provide somehow equivalent information even if they display it in slightly different ways. This information can be obtained from the ports **suspend**, **schedule**, **awake**, **reduce**, **restore** whose attributes contain information relative to solver actions and variable domain updates. As a matter of fact, we have been able to easily implement the 3D view of [4] on top of our generic trace, this is reported in [19].

With the information provided by the generic trace a debugging tool is also able to produce profiling tables. Data which can be collected from the generic trace include, for example, the total number of events in one run (domain updates, awakenings, updates of the propagation queue); the total number of events related to the control of the store (backtracking, solutions, failures); the evolution of the variable domain (e.g. size, kind of update, speed of domain evolution); the influence of a constraint (e.g. number of awakenings, number of reductions provoked by some awakening).

5 Discussion and Conclusion

We have presented an "observational semantics" which formalizes relevant aspects of propagation based finite domain constraint programming and solving. Based on the semantics, tracers can then build "generic traces". We have shown that at least two representative families of finite domain solvers can be "observed" by this semantics: constraint logic programming (Gnu-Prolog) and CSP (Choco and Ilog solver). We have also shown that many existing interactive debugging tools find in the generic trace the whole information they require. In interactive debugging tools there are two streams of data. In this paper we only focused on the execution information provided by the solvers to the tools. A generic communication from the tools to the solvers is another research topic we want to investigate in further work.

We would like to emphasize that there is no restriction on the number and on the size of the attributes of the generic trace. Such a trace (and even more with a generic trace which may include redundant ports) may slow down considerably any tracer. This problem can be partly overcome as there exists mechanisms in order to tell tracers to produce only what is needed for a given analysis. Hence the actual flow of information does not have to be larger than what a debugging tool requires [12].

We started to experiment the approach in the project OADymPPaC [4] by designing an experimental constraint programming language in the form of a meta-interpreter and a meta-solver written in ECLiPSe [27]. This implementation follows exactly the rules of the observational semantics and a generic trace is produced with a concrete syntax based on a DTD XML. Several visualizers have been easily connected to the generic tracer and are used for experimental purposes. See [19] for more details. Industrial and academic partners are currently implementing the generic tracer on Chip, GNU-Prolog and PalM platforms.

References

1. C. Aillaud and P. Deransart. Towards a Language for CLP Choice-Tree Visualisation. In Deransart et al. [9], chapter 8.
2. K. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
3. C. Bracchi, C. Gefflot, and F. Paulin. Combining propagation information and search-tree visualization using opl studio. In A. Kusalik, M. Ducassé, and G. Puebla, editors, *Proceedings of WLPE'01*, pages 27–39, Cyprus, Décembre 2001. Cyprus University.
4. M. Carro and M. Hermenegildo. Tools for constraint visualisation: the vifid/trifid tool. In Deransart et al. [9], chapter 10.
5. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, June 1996.
6. R. Debruyne, J.-D. Fekete, N. Jussien, M. Gonhiem, and et al. Proposition de format concret pour les traces générées par les solveurs de contraintes. Delivrable OADymPPaCD2.2.2.1, EMN, Nantes (France), November 2001. *(in French)*.
7. P. Deransart, Mireille Ducassé, and Ludovic Langevine. A generic trace model for finite domain solvers. A new version of d1.2.1, INRIA, Rocquencourt (France), July 2002. http://contraintes.inria.fr/OADymPPaC/Public/delivrables.html.
8. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog, The Standard; Reference Manual*. Springer Verlag, April 1996.
9. P. Deransart, M. Hermenegildo, and J. Małuszyński, editors. *Analysis and Visualisation Tools for Constraint Programming*. Number 1870 in LNCS. Springer Verlag, 2000.
10. Pierre Deransart, Francois Fages, Narendra Jussien, Ludovic Langevine, and Raphaël Martin. Débogage dynamique de programmes avec contraintes sur les domaines finis: état et perspectives. Delivrable OADymPPaCD2.1.1.1, INRIA, Rocquencourt (France), February 2002. http://contraintes.inria.fr/OADymPPaC/Public/delivrables.html, *(in French)*.

[4] Information and deliverables about the OADymPPaC project can be found at http://contraintes.inria.fr/OADymPPaC/

11. M. Ducassé. Abstract views of Prolog executions with Opium. In P. Brna, B. du Boulay, and H. Pain, editors, *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, Cognitive Science and Technology, chapter 10, pages 223–243. Ablex, 1999.
12. M. Ducassé and L. Langevine. Automated analysis of CLP(FD) program execution traces. In P. Stuckey, editor, *Proceedings of the International Conference on Logic Programming*. Lecture Notes in Computer Science, Springer-Verlag, July 2002. Poster. Extended version available at http://www.irisa.fr/lande/ducasse/.
13. G. Ferrand, W. Lesaint, and A. Tessier. A model of constraint solvers by chaotic iteration adapted to value withdrawal explanation. Réalisation OADymPPaCd1.1.1, LIFO, Orléans (France), Juillet 2001.
14. GNU-Prolog. A clp(fd) system based on Standard Prolog (ISO) developed by D. Diaz, 2001. http://gprolog.sourceforge.net/ Distributed under the GNU license.
15. F. Goualard and F. Benhamou. Debugging Constraint Programs by Store Inspection. In Deransart et al. [9], chapter 11.
16. Ilog. SOLVER 5.1 reference manual, 2001.
17. E. Jahier and M. Ducassé. Generic program monitoring by trace analysis. *Theory and Practice of Logic Programming*, 2002.
18. François Laburthe and the OCRE research group. *CHOCO, a Constraint Programming kernel for solving combinatorial optimization problems*, September 2001. Available at `http://www.choco-constraints`.
19. L. Langevine, P. Deransart, Mireille Ducassé, and Erwan Jahier. Tracing execution of clp(fd) programs, a trace model and an experimental validation environment. RR 4342, INRIA, Rocquencourt (France), Décembre 2001. Also in post-ICLP'01 workshop: 11th Workshop on Logic Programming Environments.
20. M. Meier. Debugging constraint programs. In U. Montanari and F. Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, number 976 in LNCS, pages 204–221. Springer Verlag, 1995.
21. T. Müller. Practical investigation of constraints with graph views. In *Principles and Practice of Constraint Programming – CP 2000*, number 1894 in LNCS. Springer-Verlag, 2000.
22. Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.
23. C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 286–300, Leuven, Belgium, June 1997. The MIT Press.
24. H. Simonis and A. Aggoun. Search-Tree Visualisation. In Deransart et al. [9], chapter 7.
25. Helmut Simonis, Abder Aggoun, Nicolas Beldiceanu, and Eric Bourreau. Complex constraint abstraction : Global constraint visualisation. In Deransart et al. [9], chapter 12.
26. G. Smolka. The Oz programming model. *Lecture Notes in Computer Science*, 1000, 1995.
27. M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. Technical report, IC-Parc, August 1997.