A Generic Trace Schema for the Portability of CP(FD) Debugging Tools*

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

[1] Inria Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
{Ludovic.Langevine, Pierre.Deransart}@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 and each platform has its environment tools. However, at present, these tools are specific and have to be redesigned and re-implemented for each constraint solver whereas much could be factorized. This article sets the foundations to enable debugging tools to be defined almost independently from finite domain solvers, and conversely, tracers to be built independently from these tools. We propose a *generic trace schema* based on a *generic observational semantics* which formalizes relevant aspects of constraint programming and solving. We illustrate the generecity of the schema on three representative families of finite domain solvers: CLP (Gnu-Prolog), CSP (Choco) and explanation based CP (PaLM). Debugging tools can use the generic trace and do not have to take into account all the details of the solvers. We experimented several combinations of new debugging tools with the above mentioned solvers. We show that they all find in the generic trace the whole information they require.

1 Introduction

There exists a number of useful debugging tools for finite domain solvers, for example, Grace [1], the Oz Explorer [2], the Oz Constraint Investigator [3], the CHIP global constraint visualizer [4], the Christmas Tree visualizer [5] or Clpgui [6]. However, not all solvers benefit from all the existing tools. Indeed, as illustrated by Fig. 1, at present if a debugging tool is to be integrated into a solver environment, the entire connection between the solver and the debugging tool has to be built, resulting in a set of one-to-one specialized connections between a solver and its tools. In general, hooks have to be added into the solver code in order to be able to send execution information to the debugging tools. Furthermore, the types of basic information required by a given debugging tool is not often made explicit and may have to be reverse-engineered. This is a non neglectable part of the cost of porting debugging tool from one constraint

* This work is partly supported by OADymPPaC, a French RNTL project: http://contraintes.inria.fr/OADymPPaC.

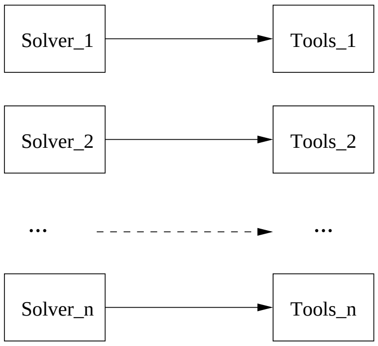

Fig. 1. Current situation: each solver has to be connected the hard way to a set of debugging tools

programming platform to another. The connection between a solver and a debugging tool in itself is rather tedious to achieve and it has to be done for each solver and each debugging tool.

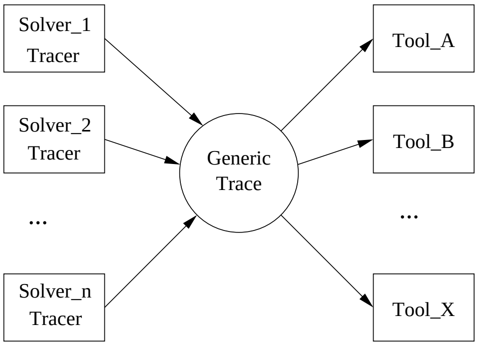

Fig. 2. Ideal situation: a generic trace schema enables all debugging tools to work straightforwardly with all solver tracers

In order to solve the above mentioned problems, we advocate to make the debugging information explicit in terms of an execution trace schema adapted to finite domain solvers. Traces conforming to this schema can be generated by solver tracers and used by debugging tools[1]. As a matter of fact, even if the finite domain solvers have different implementations and different optimizations, their operational semantics is not so different. Hence, in an ideal world, as illustrated by Fig. 2, we would aim at specifying a generic trace schema so that all solver

[1] Note that there exists mechanisms to avoid generating irrelevant pieces of trace [7].

tracers generate traces following this schema and the generated traces would be straightforwardly understood by all debugging tools.

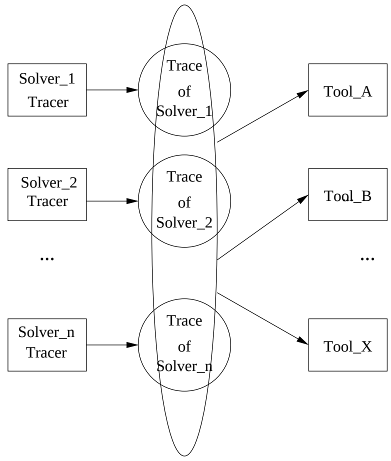

Fig. 3. Our approach: all debugging tools work for all solvers, using the generic part of the solver traces

However, experience shows that the situation is slightly more complicated. As illustrated by Fig. 3, all tracers cannot generate exactly the same trace. Therefore the debugging tools, in order to work for different solvers, have to have a specialized front end. We show in the following that this front end can be considerably small for three independently developed debugging tools. The presented trace schema is indeed rich enough to cover the needs of at least all these tools. It should be noted that the tedious work of adding hooks into the solvers is done only once for each solver, while building its tracer. Connecting a debugging tool does not cost anything on the solver side.

In this article we specify a *generic trace schema*. The traces generated by a given tracer will follow a specialized version of this generic trace schema. In a first stage, we formally specify a *generic observational semantics* by a set of state transition rules. Each rule identifies a type of execution events which can be useful for debugging. The rules also define what part of the execution state is of interest for each event type. An observational semantics can be viewed as a sampling of a complete formal semantics.

Then as illustrated in Fig. 4 this generic observational semantics (GOS) is specialized for each solver S_k into a specialized observational semantics (OS_k). From the observational semantics the trace schemata are designed by abstracting the state information. Traditionally, a tracer does not explicitly give information about states when this information can be deduced. Note that, therefore, the

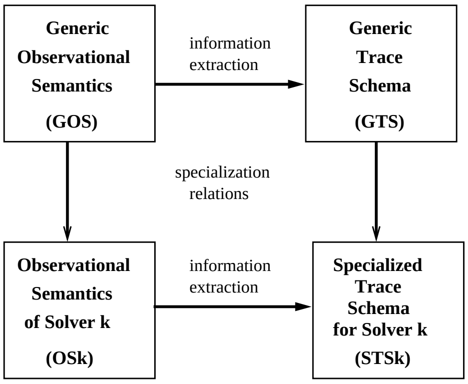

Fig. 4. From Observational Semantics to Trace Schemata (Top: *generic*, Bottom: *specialized*)

rules are very important to interpret a trace, it is unfortunate that they usually do not explicitly appear in the documentation of tracers.

In the remaining of this article, the "arrows" of Fig. 4 are informally discussed and the "boxes" are formally described: from the generic observational semantics to three specialized observational semantics for GNU-Prolog, Choco and PaLM, and from the generic trace schema to the three specialized trace schemata for GNU-Prolog, Choco and PaLM. Three solvers and three debugging tools have been connected together by different people. The generic trace schema has proven very effective for these experiment and has indeed enabled different partners of a national project to share data and tools. The front ends for the experiments have been easily implemented by the developers of each tool.

In the following, Section 2 gives the formal specification of the generic observational semantics. Section 3 specifies the generic trace schema. Section 4 gives the specializations of the semantics and of the schema for three different kinds of solvers. Section 5 describes the experimentations with three solvers anda meta-interpreter, and three debugging tools.

2 A Generic Observational Semantics for CLP(FD)

A trace is a sequence of execution events. Each event is an instance of a finite set of event types. An event-type corresponds to a transition from an execution state to another. The execution state is formalized as an *observed state* and each event type is defined by a state-transition rule between two observed states.

An observed state consists of two parts: the solver state and the search-tree state. In the remaining of this section, we describe in detail the components of the solver state, we present the search-tree and then give all the rules that formally describe the events. Events are divided into two classes: control and propagation. Control events are related to the management of variables and constraints, as well as the management of the search. Propagation events are related to the reduction of domains and the awakening process of constraints.

2.1 Solver State and Search-Tree

Definition 1. *(Solver State)*

A solver state is a 8-tuple: $\mathbb{S} = (\mathcal{V}, \mathcal{C}, \mathcal{D}, A, E, R, S_c, S_e)$
where: $\mathcal{V}$ *is the set of already declared variables;* $\mathcal{C}$ *is the set of already declared constraints;* $\mathcal{D}$ *is the function that assigns to each variable in* $\mathcal{V}$ *its domain (a set of values in the finite set* $\mathbb{D}$*);* A *is the set of active couples of the form (constraint, solver event[2]);* E *is the set of entailed constraints;* R *is the set of unsatisfiable (or rejected) constraints.* S_c *is the set of sleeping constraints;* S_e *is the set of solver events to propagate ("sleeping events").*

A, S_c, E and R are used to describe four specific states of a constraint during the propagation stage: active, sleeping, entailed or rejected. These states are consistent with the states Müller defines to describe the *propagators* in the Oz system [3]. The *store of constraints* is actually the set of all the constraints taken into account. The store is called σ in the following and defined as the partition $\sigma = \{c \mid \exists (c,e) \in A\} \cup S_c \cup E \cup R$. All the constraints in σ are defined, thus $\sigma \subseteq \mathcal{C}$. The set of variables involved in the c constraint is denoted by $\mathbf{var}(c)$. The predicate $false(c, \mathcal{D})$ (resp. $entailed(c, \mathcal{D})$) holds when the constraint c is considered as unsatisfiable (resp. entailed) by the domains in $\mathcal{D}$.

The search is often described as the construction of a search-tree.

Definition 2. *(Search-Tree State)*

The search-tree is formalized by a set of labeled nodes $\mathcal{N}$ *and a function* Σ *which assigns to each node a solver state. The nodes in* $\mathcal{N}$ *are ordered by the construction. Three kinds of nodes are defined and characterized by three predicates: failure leave (*$failed(\mathbb{S})$*), solution leave (*$solution(\mathbb{S})$*), and choice-point node (*$choice\text{-}point(\mathbb{S})$*). The last visited node is called* current node *and is denoted* ν*. The usual notion of* depth *is associated to the search-tree: the depth is increased by one between a node and its children. The function* δ *assigns to a node* n *its depth* $\delta(n)$*. Therefore, the state of the search-tree is a quadruple:* $\mathbb{T} = (\mathcal{N}, \Sigma, \delta, \nu)$*.*

In the initial solver state, ν denotes the root of the search-tree and all the sets that are part of $\mathbb{S}$ and $\mathbb{T}$ are empty.

---

[2] 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 to be propagated (e.g. the update of the domain bounds of a variable); a *trace event* corresponds to an execution step which is worth reporting about.

2.2 Transition between Observed States

The observational semantics is specified by rules which characterize possible state transitions. The top of each rule describes the side conditions on the observed state $(\mathbb{S}, \mathbb{T})$ required to use the rule. The bottom of the rule details the change of the observed state. Some definitions are added on the right-hand side.

$$\text{name}\ \frac{\mathit{conditions\_on\_current\_state}(\mathbb{S}, \mathbb{T})}{\mathit{actions\_on\_the\_observed\_state}(\mathbb{S}, \mathbb{T})}\ \{\mathit{definitions}\}$$

In the following Sec. 2.3 describes the control events and Sec. 2.4 the propagation events.

2.3 Control

$$\text{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v : \text{initial domain of } v\}$$

$$\text{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

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

$$\text{choice point}\ \frac{\mathit{choice\text{-}point}(\mathbb{S})\ \wedge\ n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{back to}\ \frac{n \in \mathcal{N} \wedge \mathit{choice\text{-}point}(\Sigma(n))}{\mathbb{S} \leftarrow \Sigma(n),\quad \nu \leftarrow n}$$

$$\text{solution}\ \frac{\mathit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{failure}\ \frac{\mathit{failure}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{remove}\ \frac{c \in \sigma}{\sigma \leftarrow \sigma - \{c\}}$$

$$\text{restore}\ \frac{v \in \mathcal{V}\quad \wedge\quad \Delta_v \cap \mathcal{D}(v) = \emptyset}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) \cup \Delta_v}\ \{\Delta_v \text{ is a subset of the initial } D_v\}$$

Fig. 5. Control rules of the generic observational semantics

The rules of Fig. 5 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 v is introduced in $\mathcal{V}$ and that its initial domain is D_v. Rule new constraint specifies that the solver introduces a new constraint c in $\mathcal{C}$, having checked that all variables involved in c are already defined. This constraint is declared without being posted: it does not yet belong to the store σ. The activation of a declared constraint c is specified by the rule post: the constraint is entered in the store as an active constraint and is ready to make domain reductions. It is attached to the event $\perp$ which denotes the activation of the constraint.

The following four rules describe the construction of the search-tree. Rule choice point specifies that the current solver 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. This state is recorded in Σ. Jumping back from the current solver state to a previous choice-point is specified by the rule back to: a whole former state is restored in $\mathbb{S}$. Finally, two rules are used to declare the leaves of the search-tree. solution specifies that the current solver state corresponds to a solution leaf. failure specifies that the current solver state is a failure leaf.

Two additional rules are used to describe search strategies that are not based on a search-tree, such as the *path-repair* technique presented by Jussien and Lhomme [8]. Those strategies enable the removal of any constraint from the store σ and the cancellation of any previous domain reductions, leading to new parts of the search space. Rule remove removes a constraint c from the store σ. The rule restore specifies that the solver is restoring some values Δ_v in the domain of variable v.

2.4 Propagation

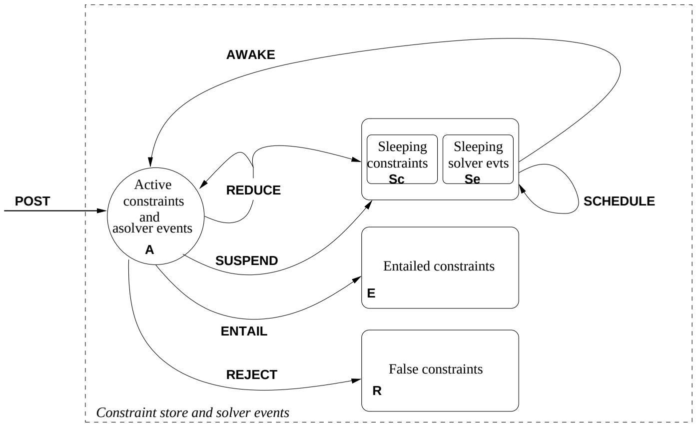

Fig. 6. Generic Observational Semantics: illustration of the transitions described by the propagation rules

The propagation can be described by state transition rules acting in the solver state, as illustrated by Fig. 6. These rules are formalized by Fig. 7. A, S_c, S_e, E and R are parts of the current state $\mathbb{S}$ previously defined. The active pairs in A can reduce some domains because of their solver events. Rule reduce

$$\textsf{reduce}\ \frac{(c,e) \in A\ \wedge\ v \in \mathbf{var}(c)}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) - \Delta_v^c,\quad S_e \leftarrow S_e \cup \bar{e}'}\ \left\{\begin{array}{l}\Delta_v^c \text{ is a subset of } \mathcal{D}(v) \text{ to remove}\\ \bar{e}' \text{ is a set of solver events on } v\end{array}\right\}$$

$$\textsf{suspend}\ \frac{(c,e) \in A}{A \leftarrow A - \{(c,e)\},\quad S_c \leftarrow S_c \cup \{c\}}$$

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

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

$$\textsf{awake}\ \frac{c \in S_c\ \wedge\ e \in S_e \cup \{\bot\}\ \wedge\ awakecond(c,e)}{A \leftarrow A \cup \{(c,e)\},\quad S_c \leftarrow S_c - \{c\}}$$

$$\textsf{schedule}\ \frac{c \in S_c\ \wedge\ e \in S_e\ \wedge\ action(c,e)}{S_c \leftarrow S_c',\quad S_e \leftarrow S_e'}\ \{S_c' \text{ and } S_e' \text{ are solver dependent}\}$$

Fig. 7. Propagation rules of the generic observational semantics

specifies that the solver reduces the domain of a variable of an active constraint attached to a solver event. A single domain can be reduced by this rule. $\bar{e}'$ is a set of solver events that characterize the reduction. Examples of solver events are "increase 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 pair cannot reduce any domain at the moment, the e event is said to have been propagated by c and the c constraint 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 action schedules the propagation by acting on a sleeping constraint and a sleeping event ($action(c,e)$ in schedule): this leads to a modification of the internal structure of S_c and S_e. If the solver dependent condition $awakecond(c,e)$ 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. Notice that e can be $\bot$: the awakening is then due to the sole activation of the constraint.

3 Generic Trace Schema

The two sets of rules presented by Fig. 5 and Fig. 7 specify an observational semantics. A generic trace schema derives from this semantics. As already mentioned, a trace is a sequence of events. Each trace event corresponds to the application of a semantic rule. Therefore each semantic rule gives its name to a type of trace event: a *port*.

Definition 3 (Generic Trace). *A generic trace is a sequence of trace events consisting of a set of attributes: a sequential event number; the depth in the search tree; the port (the type of the corresponding semantic rule); the whole*

Control Ports		Propagation Ports	
new variable	v, D_v	reduce	c, e, v, $\bar{e}'$, Δ_v^c
new constraint	c	suspend, entail	c
post, remove	c	reject	c
restore	v, Δ_v	awake	c, e
choice point	ν after the state transition	schedule	c, e
back to	ν before and after the state transition		
solution, failure	ν after the state transition		

Table 1. Specific Attributes of the Ports

observed state of the solver after the transition; some specific attributes depending on the port.

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 concerned constraint and solver event, the variable v that is being reduced, the value withdrawal Δ_v^c 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. 5 and Fig. 7.

4 Specialized Observational Semantics

This section specializes the generic observational semantics for three solvers: the constraint logic programming platform GNU-Prolog, the constraint programming library Choco and the explanation-based platform PaLM. It shows how the generic observational semantics fits those different platforms in such a way that each specialization satisfies the semantics of a platform. In this section, the specialized observed state of each platform is detailed, as well as its dedicated control and propagation rules.

4.1 Specialized Observational Semantics for GNU-Prolog

GNU-Prolog is a Prolog platform with an embedded FD solver developed by Diaz [9]. A self-contained description of the specialized observational semantics, as well as the description of the actual tracer implementation, can be found in [10].

Solver State for GNU-Prolog. A GNU-Prolog primitive constraint is defined by several indexicals (constraints of the form $x \in f(\mathcal{D}_{|\mathbf{var}(c)})$). The constraints considered in GNU-Prolog are the primitive constraints whereas the indexicals are *propagators*. The solver state of GNU-Prolog is formalized as $\mathbb{S} = (\mathcal{V}, \mathcal{C}, \mathcal{D}, A_c, A_e, E, R, S_c, Q)$. A_c contains at most a single active constraint while A_e contains at most the current event to propagate. The generic set A is thus defined as $A = A_c \times A_e$. The set S_e of the generic model is refined as $S_e = Q \cup A_e$ where Q is the tail of the propagation queue that drives the constraint awakenings, and A_e is its head.

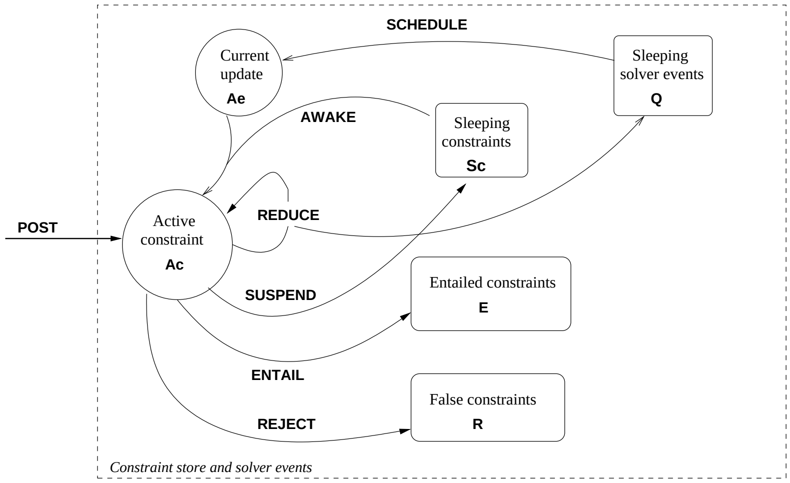

Fig. 8. GNU-Prolog Specialized Observational Semantics: illustration of the transitions described by the propagation rules

Control of GNU*-Prolog executions.* As defined in Standard Prolog [11], the search strategy of GNU-Prolog is modeled as the depth-first left-to-right traversal of a tree. From a CP point of view, the search-tree consists of a subset of nodes of the SLD tree, the traversal order being preserved. Therefore the search is described by the four events choice point, back to, solution and failure only. Since the search is completely described as a tree, the remove and restore rules are useless. Notice that, in GNU-Prolog, failure denotes FD constraint rejections as well as classical Prolog failures. new variable denotes the creation of a FD variable (a logic variable that appears in a constraint and gets a domain). new constraint followed by post denotes the call of a GNU-Prolog goal that is a primitive constraint. The only different rules with respect to the control part of the generic semantics are post (because of $A = A_c \times A_e$) and solution (there can be no solution once R is not empty).

Propagation. The propagation in GNU-Prolog is formalized by the rules of Fig. 10 and illustrated in Fig. 8. The constraint propagation is driven by the Q

$$\text{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v : \text{initial domain of } v\}$$

$$\text{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

$$\textit{post}\ \frac{c \in \mathcal{C}\ \wedge\ c \notin \sigma}{A_c \leftarrow \{c\},\quad A_e \leftarrow \{\bot\}}$$

$$\text{choice point}\ \frac{\textit{choice-point}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{back to}\ \frac{n \in \mathcal{N} \wedge \textit{choice-point}(\Sigma(n))}{\mathbb{S} \leftarrow \Sigma(n),\quad \nu \leftarrow n}$$

$$\textit{solution}\ \frac{\textit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N} \wedge R = \emptyset}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{failure}\ \frac{\textit{failure}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

Fig. 9. Specialized control rules for GNU-Prolog

queue: each time a domain is reduced, the corresponding domain updates are recorded in Q (rule reduce). Updates in Q are dequeued one by one and put into A_e (rule schedule). The solver event in A_e awakes each constraint that it potentially impacts (rule awake). In GNU-Prolog, the potential impact of e on c is modeled by dependency lists denoted by the relation *dependence*(c, e).

A solver event e is represented by a couple (v, t) where v is the reduced variable and t is the type of reduction. The types of reduction are: *min* (increase of the lower bound), *max* (decrease of the upper bound), *minmax* (either of the two previous ones), *val* (instantiation of the variable) and *dom* (any domain reduction). A single reduce can then lead to several solver events. For example, when $\mathcal{D}(v)$ is reduced from $\{1, 2, 3\}$ to $\{1, 2\}$ the resulting solver events are $\bar{e} = \{(v, min), (v, minmax), (v, dom)\}$.

GNU-Prolog tests constraint entailment only when a constraint is posted ($A_e = \{\bot\}$) (entail).

A constraint is said to be rejected when the computed inconsistent values of a variable v (Δ_v^c) is the whole domain of v (reject). Notice that, once a constraint is rejected, no propagation rule can be applied anymore.

4.2 Specialized Observational Semantics for Choco

Choco is an object-oriented library of constraint programming designed by Laburthe et al. [12].

Solver State for Choco. The state of the Choco solver can be mapped onto the generic model with the refinement $S_e = Q_h \cup Q_t$ and $E = \emptyset$ (Choco does not detect constraint entailment). Thus, $\mathbb{S} = (\mathcal{V}, \mathcal{C}, \mathcal{D}, A, R, S_c, Q_h, Q_t)$. During the propagation, A contains at most a pair. This active pair consists of the constraint of which one filtering algorithm is running and the domain update

$$\textsf{reduce}\ \frac{A_c = \{c\}\ \wedge\ v \in \mathbf{var}(c)\ \wedge\ R = \emptyset}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) - \Delta_v^c,\ \ Q \leftarrow Q \cup \bar{e}'}\ \left\{\begin{array}{l}\Delta_v^c \text{ is a subset of } \mathcal{D}(v) \text{ to remove}\\ \bar{e}' \text{ is a set of solver events on } v\end{array}\right\}$$

$$\textsf{suspend}\ \frac{A_c = \{c\}\ \wedge\ R = \emptyset}{A_c \leftarrow \emptyset,\ \ S_c \leftarrow S_c \cup \{c\}}$$

$$\textsf{entail}\ \frac{A_c = \{c\}\ \wedge A_e = \{\bot\}\ \wedge\ \mathit{entailed}(c, \mathcal{D})\ \wedge\ R = \emptyset}{A_c \leftarrow \emptyset,\ \ E \leftarrow E \cup \{c\}}$$

$$\textsf{reject}\ \frac{A_c = \{c\}\ \wedge\ v \in \mathbf{var}(c) \wedge \Delta_v^c = \mathcal{D}(v)}{A_c \leftarrow \emptyset,\ \ R \leftarrow \{c\}}$$

$$\textsf{awake}\ \frac{A_c = \emptyset\ \wedge\ A_e = \{e\} \wedge c \in S_c\ \wedge\ \mathit{dependence}(c, e)\ \wedge\ R = \emptyset}{A_c \leftarrow \{c\},\ \ S_c \leftarrow S_c - \{c\}}$$

$$\textsf{schedule}\ \frac{A_c = \emptyset \wedge\ e \in Q\ \wedge\ R = \emptyset}{A_e \leftarrow \{e\},\ \ Q \leftarrow Q - \{e\}}$$

Fig. 10. Specialized propagation rules for GNU-Prolog

that is propagated, or $\bot$. As in GNU-Prolog, S_e (the set of all solver events to process) is a queue that is refined as the head Q_h (event currently processed) and the tail Q_t. These events are processed in turn in order to schedule the constraint awakenings.

Control. The control part of the specialized semantics is shown in Fig. 12. Choco search strategy is modeled by a tree where any previous choice point can be restored at backtracking. This is taken into account by the generic back to rule as soon as each choice point has been declared by the generic choice point rule. The failure leaves are always due to a constraint rejection, thus $R \neq \emptyset$. Since the search can be described as a tree, the remove and restore rules are useless.

Propagation. Choco's propagation engine is illustrated in Fig. 11 and formalized in Fig. 13. The filtering algorithms of a given constraint are defined by several methods (in the sense of object programming). For instance, `awakeOnInf` propagates a lower bound increase and `awakeOnInst` propagates the instantiation of a variable. The awakening of a constraint is the invocation of one of these methods. The propagation is driven by the queue of the solver events to propagate. schedule focuses on an event e in the queue Q_t and puts it in Q_h as formalized in Fig. 13. The event in Q_h is processed by awakening every constraint that can propagate it, according to the specialized relation $\mathit{dependence}(c, e)$ (awake). Notice that a constraint can be awaken with the $\bot$ event: it denotes the invocation of a specific filtering algorithm that does not take into account any domain update but only the current domains. The value removals are computed knowing e (Δ_v^c becomes $\Delta_v^{c_e}$ in the rule reduce to express that). A constraint is rejected only when it has just emptied out a domain $\mathcal{D}(v) = \emptyset$ (reject). Once the processing of an event e is complete, another event can be selected, until Q_t is empty.

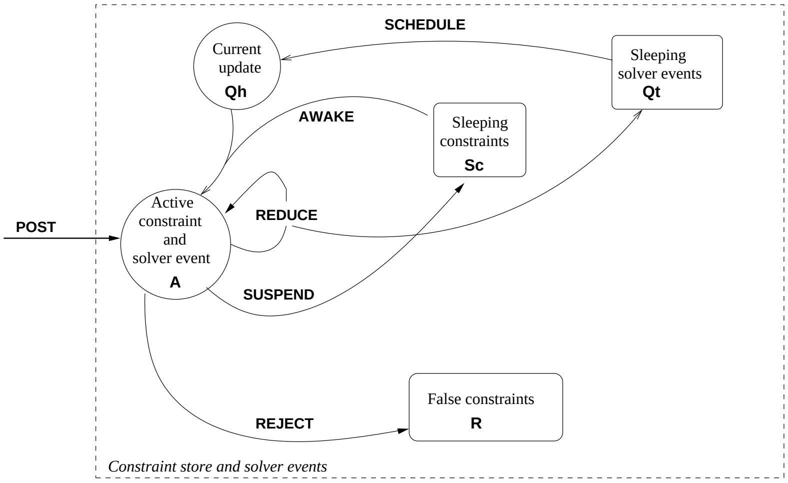

Fig. 11. Choco Specialized Observational Semantics: illustration of the transitions described by the propagation rules

4.3 Specialized Observational Semantics for PaLM

PaLM is an explanation-based constraint programming platform by Jussien and Barichard [13]. It is a major extension of the Choco library presented above. Each value removal is labeled with a set of constraints that makes the value inconsistent. Such a set of constraints is called an *explanation* of the removal. The function that assigns to each value removal (v, d) (removal of d from $\mathcal{D}(v)$) its explanation is denoted $\mathcal{E}(v, d)$. The explanations are computed by the filtering algorithms, knowing the explanations of the previous value removals. This implementation of explanations is consistent with the theoretical framework developed by Debruyne et al. [14]. Explanations enable the solver to relax a constraint while undoing the value removals it is responsible for. When a domain has just been wiped out, the resulting failure can be diagnosed thanks to the recorded explanations. PaLM is then able to move to another point of the search space by relaxing well-chosen constraints.

PaLM specialization of the propagation part is presented before the control part, because the first one introduces several details about explanations that are worth knowing to understand the control well.

Solver State for PaLM. The solver state of PaLM is modeled as the state of Choco, with the additional explanation function $\mathcal{E}$. Thus $\mathbb{S} =< \mathcal{V}, \mathcal{D}, \mathcal{C}, A, R, S_c, Q_h, Q_t, \mathcal{E} >$, we have again $S_e = Q_h \cup Q_t$ with Q_h a singleton. In PaLM there is no detection of constraint entailment, therefore the set of the entailed constraints E is not used and the entail rule is irrelevant. $\mathcal{E}$ formalizes the recorded explanations. As presented above, it is a partial function $\mathcal{E} : \mathcal{V} \times \mathbb{D} \longrightarrow \mathcal{P}(\sigma)$ which assigns to each value removal (v, d) a set of non relaxed

$$\textsf{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v : \text{initial domain of } v\}$$

$$\textsf{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

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

$$\textsf{choice point}\ \frac{\textit{choice-point}(\mathbb{S})\ \wedge\ n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\textsf{back to}\ \frac{n \in \mathcal{N} \wedge \textit{choice-point}(\Sigma(n))}{\mathbb{S} \leftarrow \Sigma(n),\quad \nu \leftarrow n}$$

$$\textsf{\textit{solution}}\ \frac{R = \emptyset \wedge \textit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\textsf{\textit{failure}}\ \frac{R \neq \emptyset \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

Fig. 12. Specialized control rules for Choco.

$$\textsf{\textit{reduce}}\ \frac{A = \{(c, e)\} \wedge v \in \mathbf{var}(c) \wedge \Delta_v^{c_e} \neq \emptyset \wedge R = \emptyset}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) - \Delta_v^{c_e},\quad Q_t \leftarrow Q_t \cup \{e'\}}\ \left\{\begin{array}{l}\Delta_v^{c_e}\text{: values that } c \text{ has to re-}\\ \text{move from } \mathcal{D}(v) \text{ because of}\\ e\text{. The reduction generates}\\ e'.\end{array}\right\}$$

$$\textsf{\textit{suspend}}\ \frac{A = \{(c, e)\}\ \wedge\ R = \emptyset}{A \leftarrow \emptyset,\quad S_c \leftarrow S_c \cup \{c\}}$$

$$\textsf{\textit{reject}}\ \frac{A = \{(c, e)\}\ \wedge\ v \in \mathbf{var}(c) \wedge \mathcal{D}(v) = \emptyset}{A \leftarrow \emptyset,\quad R \leftarrow \{c\}}$$

$$\textsf{\textit{awake}}\ \frac{A = \emptyset \wedge c \in S_c \wedge e \in Q_h \cup \{\bot\} \wedge \textit{dependence}(c, e) \wedge R = \emptyset}{S_c \leftarrow S_c - \{c\},\quad A \leftarrow \{(c, e)\}}$$

$$\textsf{\textit{schedule}}\ \frac{A = \emptyset \wedge e \in Q_t \wedge R = \emptyset}{Q_h \leftarrow \{e\},\quad Q_t \leftarrow Q_t - \{e\}}$$

Fig. 13. Specialized propagation rules for Choco.

constraints that explains the removal. This partial function is updated only by the reduce and restore rules.

Domain reductions and propagation. The specialized propagation rules are presented in Fig. 14. Rules suspend, reject, awake and schedule are inherited from Choco. Explanations are introduced in the reduce rule. An elementary reduction is called *filtering decision* in PaLM terminology (for instance updating a bound or removing a specific value). As in Choco, it produces a single solver event e. This decision is taken by (one of) the filtering algorithm(s) of the active constraint in A and is explained by a single set of constraints C. C is computed by the filtering algorithm. While the inconsistent values are removed, the new solver event e is put in the propagation queue and the C explanation is recorded in $\mathcal{E}$ (for each removed value).

$$\textsf{\textit{reduce}}\ \frac{A = \{(c,e)\} \ \wedge \ v \in \mathbf{var}(c) \ \wedge \ \Delta_v^{c_e} \neq \emptyset \ \wedge \ C \subseteq \sigma \ \wedge \ R = \emptyset}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) - \Delta_v^{c_e}, \quad Q_t \leftarrow Q_t \cup \{e'\}, \quad \mathcal{E} \leftarrow \mathcal{E} \cup \{(v,d,C) \mid d \in \Delta_v^{c_e}\}}$$

$$\left\{ \begin{array}{l} \Delta_v^{c_e} \text{ set of inconsistent values for } v \\ C \text{ explains the removal of } \Delta_v^{c_e} \text{ from } \mathcal{D}(v) \\ \text{The reduction generates } e' \end{array} \right\}$$

$$\textsf{\textit{suspend}}\ \frac{A = \{(c,e)\} \ \wedge \ R = \emptyset}{A \leftarrow \emptyset, \quad S_c \leftarrow S_c \cup \{c\}}$$

$$\textsf{\textit{reject}}\ \frac{A = \{(c,e)\} \ \wedge \ v \in \mathbf{var}(c) \wedge \mathcal{D}(v) = \emptyset}{A \leftarrow \emptyset, \quad R \leftarrow \{c\}}$$

$$\textsf{\textit{awake}}\ \frac{A = \emptyset \wedge c \in S_c \wedge e \in Q_h \cup \{\bot\} \wedge \mathit{dependence}(c,e) \wedge R = \emptyset}{S_c \leftarrow S_c - \{c\}, \quad A \leftarrow \{(c,e)\}}$$

$$\textsf{\textit{schedule}}\ \frac{A = \emptyset \wedge e \in Q_t \wedge R = \emptyset}{Q_h \leftarrow \{e\}, \quad Q_t \leftarrow Q_t - \{e\}}$$

Fig. 14. Specialized propagation rules for PaLM

The reduction described by the reduce rule is made by c and is likely a consequence of some previous reductions of some variables involved in c. Therefore, C consists of c and the union of k other explanations ($k \geq 0$) about the variables involved in c (restriction denoted $\mathcal{E}_{|\mathbf{var}(c)}$). This additional description of C is formalized as follows:

$$\exists (C_1, \ldots, C_k) \subseteq \mathcal{E}^k_{|\mathbf{var}(c)} \text{ such that } C = \{c\} \cup \bigcup_{i=1}^{k} C_i$$

Control. The search in PaLM is not based on standard backtracking and there is no notion of search-tree. The rule back to is thus useless. The navigation in the search-space is fully described by the remove and restore rules specialized in Fig. 15. The choice point rule is used to tag specific points of the search-space where decisions have to be taken by the search-procedure. This is similar to the notion of choice-point in a search-tree. As a result, $\mathcal{N}$ denotes the explored search-space, that is not necessarily a tree. Rules new variable, new constraint, post, choice point, solution and failure are inherited from Choco.

PaLM uses explanations when a failure occurs. The C recorded explanation of the empty domain is a *contradiction*. At least one constraint in C has to be relaxed. $\mathcal{E}$ enables PaLM to relax a constraint while undoing its previous effects: each value removal whose explanation contains the relaxed constraint have to be undone. The event consisting in the relaxation of a single constraint c is remove, as defined in the generic model. The restoration of the values R_v in $\mathcal{D}(v)$ is described by the rule restore. The removal of the R_v values from $\mathcal{D}(v)$ is explained by (at least) one removed constraint. As proven in [14], after domain restorations, a consistency-checking has to be done so as to get back to a locally coherent state as if the removed constraints never appeared in σ. This consistency-checking is done by re-propagating all the constraints involving the restored variable: each restoration $\bar{e}$ is then recorded into Q_t. A stage of domain

$$\text{new variable}\ \frac{v \notin \mathcal{V}}{\mathcal{V} \leftarrow \mathcal{V} \cup \{v\},\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(v, D_v)\}}\ \{D_v : \text{initial domain of } v\}$$

$$\text{new constraint}\ \frac{c \notin \mathcal{C}\ \wedge\ \mathbf{var}(c) \subseteq \mathcal{V}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}}$$

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

$$\text{choice point}\ \frac{\mathit{choice\text{-}point}(\mathbb{S})\ \wedge\ n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\textit{solution}\ \frac{R = \emptyset \wedge \mathit{solution}(\mathbb{S}) \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\textit{failure}\ \frac{R \neq \emptyset \wedge n \notin \mathcal{N}}{\mathcal{N} \leftarrow \mathcal{N} \cup \{n\},\quad \Sigma \leftarrow \Sigma \cup \{(n, \mathbb{S})\},\quad \nu \leftarrow n}$$

$$\text{remove}\ \frac{c \in \sigma}{\sigma \leftarrow \sigma - \{c\}}$$

$$\textit{restore}\ \frac{v \in \mathcal{V} \wedge R_v \subseteq \{d \in \mathbb{D} | \mathcal{E}(v, d) - \sigma \neq \emptyset\} \wedge E = \{\mathcal{E}(v, d) | d \in R_v\}}{\mathcal{D}(v) \leftarrow \mathcal{D}(v) \cup R_v,\quad Q_t \leftarrow Q_t \cup \bar{e},\quad \mathcal{E} \leftarrow \mathcal{E} - E}$$

$$\{\bar{e} \text{ describes the restoration of } \mathcal{D}(v)\}$$

Fig. 15. Specialized control rules for PaLM

restoration is thus followed by a sequence of propagation events until Q_h is empty or a new contradiction is found.

5 Experimental Validation

We started to validate the generic trace schema by connecting three independently developed debugging tools to four independently developed solvers with tracers.

The finite domain solvers which have been considered up to now are the three solvers discussed so far (Gnu-Prolog, Choco and PaLM) and an instrumented meta-interpreter (Mescaline). Their tracers have been developed by two different teams: Gnu-Prolog and Mescaline at INRIA, and the others at Ecole des Mines de Nantes (EMN).

The debugging tools offer several kinds of static or dynamic views of the traces. Three tools have been considered: PAVOT, a simple search-tree/propagation visualizer, VISADJ, a tool for adjacency matrix visualization, and Discovery a data analysis tool. All these tools allow large problems with hundreds of variables and constraints to be analyzed. All have been developed by different teams (resp. INRIA, EMN and ILOG).

All combinations are possible, as illustrated by Fig. 16. As explained below these tools have a general interface which can be parametrized by the user according to the origin of the trace.

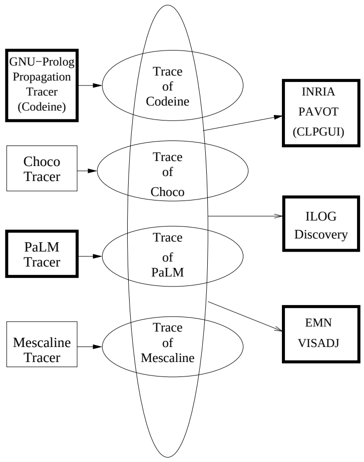

Fig. 16. Experimentation with 4 Solvers and 3 Tools (bold boxes are used in the illustration of Sec. 5.3)

The specialized traces produced by each tracer contain the ports (event types) and attributes defined in the generic trace schema[3]. In practice, tracers allow a trace to be stored in a file such that a post-mortem analysis of the trace is possible with different tools. The traces contain all the ports relevant to the corresponding solver but may sometime differ on the attributes. For example some events of the traces of PaLM contain an explanation attribute, when traces of other platform do not. A simple parametrization of the tools (selecting the events and attributes to be considered) allows specialized ports and attributes to be taken into account. In any case no modification of the generic trace scheme was necessary.

5.1 The Tracers

GNU-Prolog (Codeine) Codeine [16, 10] is an extension of GNU-Prolog [9] with the search-tree control and propagation tracer developed at INRIA. It is efficient and it has all the ports of the generic trace schema except remove and restore, according to the specialisation described in Sec. 4.1.

Choco [12], whose tracer has been implemented at EMN, contains similar finite domain solvers using usual backtracking which can be visualised by search-tree. Its trace has the same ports as above (cf. Sec. 4.2).

PaLM [13] is an explanation based solver developed at EMN. The control in PaLM is based on the manipulation of "explanations" instead of search-tree.

[3] The trace is broadcasted according to an XML DTD defined in [15].

Even if a search-tree can be theoretically derived from the solver behaviour, it is not explicitly described in the trace. Therefore there is no back to port. As PaLM does not check constraint entailment, there is no entail port either (cf. Sec. 4.3).

Mescaline [17] is a meta-interpreter embedded in Eclipse [18]. It is a generic tracer prototype written in Prolog and only small constraint programs can be executed. Implemented at INRIA in 2001 it has been used at the beginning of the OADymPPaC project to help specify the generic trace schema. It has all the ports of the generic trace schema except remove and restore.

5.2 The Debugging Tools

We have experimented with two categories of debugging tools: one is specifically designed to observe constraint solvers search-tree and propagation (PAVOT), two are general purpose data analysis tools whose interface has been adapted to handle a generic trace as a particular data entry (Discovery, VISADJ). Our purpose here is not to describe the tools, their efficiency or their capabilities, but to focus on their connectivity. The figures are intended to illustrate one of the functionalities of the tools.

PAVOT [19] is a propagation viewer currently developed at INRIA following the ideas of the Christmas-tree view of [5] and of CLPGUI [6] (in particular its 3-D viewer). A view represents a search-tree whose nodes are weighted by the number of selected events induced by the constraints posted just before the choice node. The weight is displayed in the form of circles whose radius is proportional to the weight. PAVOT uses the architecture described below for VISADJ, and supports all kinds of traces. However, to be used on PaLM traces a small adaptation was necessary: as PaLM does not have a search-tree, new nodes are created each time a constraint is declared (port new constraint). The resulting tree is simply linear. PAVOT allows post-mortem trace analysis as well as on the fly analysis, using a communication by sockets.

Fig. 17 shows the complete search-tree corresponding to the resolution of the bridge scheduling problem, minimizing the time to build a 5-segment bridge [20]. The picture shows the first solution reached quickly (top left part of the tree) with few propagations, then a large tree corresponding to the proof of optimality.

Discovery ^{TM}[21] is a general purpose data analysis tool based on data manipulation and visualization techniques developed by ILOG. It has a special entry to read a trace conforming to the generic trace schema [15], and facilities to display predefined views (starfield display, parallel histograms, table lenses, treemaps, and user defined displays). At the moment, only post-mortem trace analysis is possible. The user selects which ports and attributes must be selected from the trace: event number, port, constraints, variables, path of the search-tree, min/max domain value, domain size. Explanations are not handled yet. The

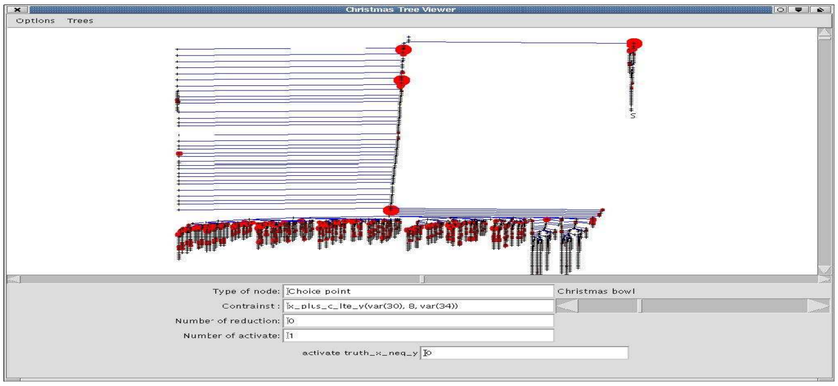

Fig. 17. Bridge scheduling problem: a 2-D view of the search space with PAVOT

tool has been developed with the sole knowledge of the generic trace schema and could be used on all traces including those generated by PaLM. By selecting the data to analyze, the users make themselves the adaptation of the tool to the considered solver.

Fig. 18 shows a Discovery parallel histogram window for the bridge scheduling problem. It has seven columns and as many lines as trace events. The events are sorted by their sequential event number (second column). Each column corresponds respectively to event port (one colour per event type), sequential event number, concerned constraint, concerned variable, depth, domain min, and domain size. By the evolution of the depth from top to bottom one recognizes the profile of the search-tree depicted in Fig. 17. Discovery allows display to be manipulated (for example ordering wrt each column) and thus to discover more characteristics of the computation.

VISADJ has been designed at EMN to analyze large graphs, with several graph visualization techniques [22], including animations. It consists of several interconnected viewers, one devoted to showing graphs using adjacency matrix technique. An n-nodes graph is represented by a n-n-matrix whose elements are weighted edges. The tool helps analyze the structure of a graph whose nodes are usually constraints or variables and whose edges are coloured according to their weight (dark colours correspond to higher weight). The colour of an edge is thus an indication of how much a constraint influences another one. Fig. 19 illustrates one of such large graphs whose detailed structure can by observed with the use of a "fish-eye". This view represents the adjacency matrix built with the hundred posted constraints in the sorted(100) problem[4] solver with PaLM. The constraints are ordered in the posting sequence from left to right and from top to bottom. A fish-eye allows the user to magnify a region of interest. Relationship

[4] Hundred variables on domain [1..100] must different and increasingly ordered.

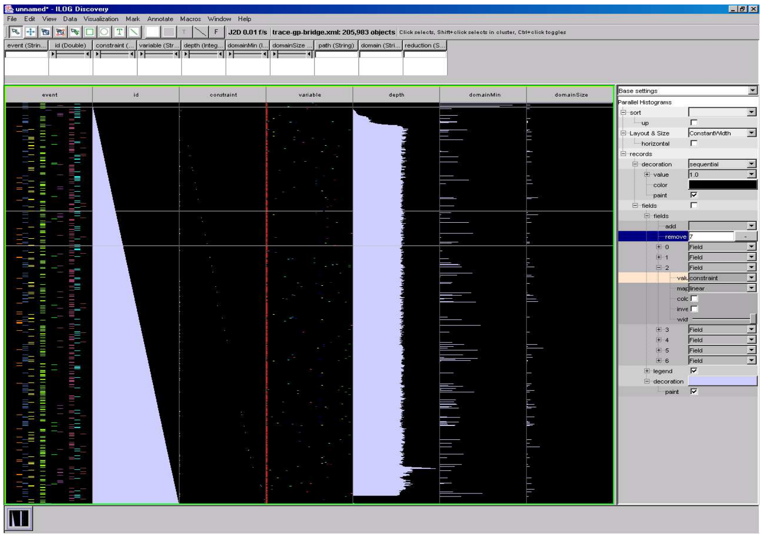

Fig. 18. Bridge scheduling problem: a parallel histogram display by Discovery analysing the trace events

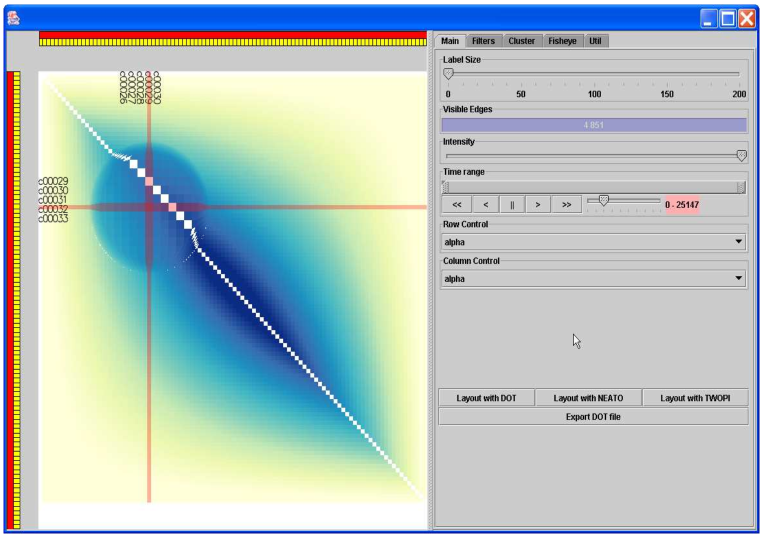

Fig. 19. An adjacency matrix for sorted(100) by the VISADJ tool. A fish-eye allows detailed analysis of a part of the graph

between constraints depends on the number of time a constraint appears in its explanation. Notice that the farther the constraints are from one another (in the posting sequence), the less they interact. Moreover, central constraints seem to interact with each other more often than peripheral ones (darker region in the centre of the view).

The board on the right allows in particular to create animations by defining an animated sequence of views corresponding to slices of the trace (in particular between two solutions). The tool has been originally experimented with traces issued from PaLM, and it has been interesting to see that it was able to handle traces generated by Codeine.

This Java based tool has essentially two components [22]: a library of "listeners" to access the trace and a library of viewers. The first component acts as a general interface able to interpret all events defined in the generic trace schema.

5.3 Using several Tools on several Solvers

We finish this section with some views showing the versatility of our approach. The generic trace schema has been to be sufficiently complete to cover the debugging needs we have encountered so far. In fact all tools can easily be used with all generated traces of all the solvers provided the post mortem trace is sufficiently rich.

The following two series of views correspond to traces of *sorted(100)* respectively issued from Codeine (for GNU-Prolog) and PaLM. Each view is intended to analyze the propagation with respectively PAVOT, Discovery and VISADJ. We just compare GNU-Prolog (Fig. 20) and PaLM (Fig. 21) traces since these solvers are based on very different principles.

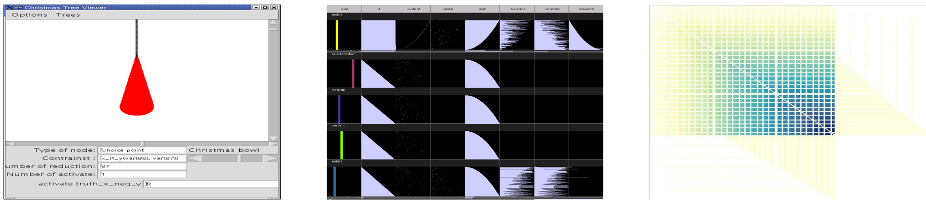

Fig. 20. A GNU-Prolog trace of sorted(100) viewed with 3 tools: PAVOT, Discovery and VISADJ

In GNU-Prolog (Fig. 20-left) some propagation steps happen each time a new constraint is posted and the number of `reduce` events is proportional to the number of already posted constraints. This is clearly revealed by PAVOT where the diameter of the red balls increases regularly.

On Discovery (Fig. 20-middle) the parallel histogram display has been split by event types. Five ports are visible here, from top to bottom: `back to`, `schedule`, `awake`, `suspend` and `reduce`. The columns are the same as in Fig. 18. By the "depth" (column 5) we immediately observe the complementary roles played by the ports back to and reduce: a linear tree is developed, then the whole backtracking occurs at the end (high event numbers in column 2 for back to). We observe the similarities between the events schedule, awake and suspend which are indistinguishable on this problem (same displays). Finally we observe the regularity of the variable domains reductions (column 7).

The presented VISADJ view (Fig. 20-right) is the adjacency matrix of the hundreds posted constraints. As there is no explanation in the traces of Codeine the level of interaction between two constraints corresponds to the number of events, generated by a constraint, which influence another one. We observe that a new posted constraint interacts with all previously posted ones.

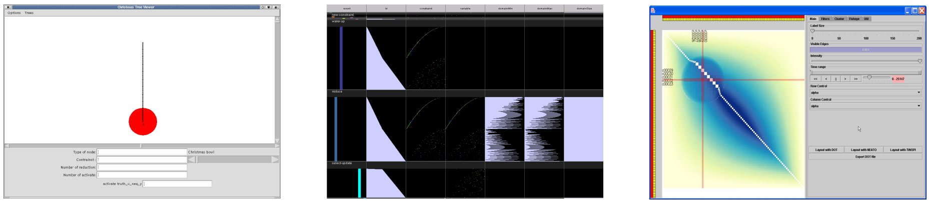

Fig. 21. A PaLM trace of sorted(100) viewed with 3 tools: PAVOT, Discovery and VISADJ

Fig. 21 shows the same views as in Fig. 20 but obtained with traces generated by PaLM instead of Codeine. This allows us to observe some differences between PaLM and GNU-Prolog resolution strategies. In PaLM (Fig. 21-left) all propagation steps happen after all constraints have been posted. In the PAVOT view this is clearly revealed by a single ball after all constraints have been posted.

The Discovery (Fig. 21-middle) split parallel histogram display has only three visible lines corresponding to the events (from top to bottom): awake, reduce and schedule. The number of suspend events is insignificant. We observe two principal phases visible in the reduce display and revealed by the change of declivity of the top line: one when a reduce event concerns one constraint and one variable at once (restriction of the domains after all constraints are posted, low declivity), and a second one where all reduce events are separated by several other events (awake and schedule events), i.e. cycles of reduce and schedule and awake events (stronger declivity).

The VISADJ-view (Fig. 21-right) has been commented above in Sec. 5.2. It shows another aspect of the PaLM strategy (in this view the constraints have been ordered according to the order of posting): closer constraints have more interaction.

The possibility to use several debugging tools to observe several modelling of a problem with different solvers offers richer possibilities to analyze the problem resolution.

6 Related Work

Two event-oriented tracers have already been implemented into real solvers. The first one, a tracer for Ilog Solver, provides some tracing facilities as a class of its solver library [23]. It is fully dedicated to Ilog Solver but we have shown in [24], that it can be seen as a specialisation of our generic trace schema. A few tools have been built upon this execution trace. For example, the Ilog Christmas Tree [5] processes an XML trace performed by the debugger thanks to the overriding of its virtual functions. However adapting a new tool would require a deep understanding of the solver behaviour. Moreover, the amount of data available at each event is very limited compared to the full state that our schema enables. For instance, the set of constraints and variables cannot be traversed.

The second one, a tracer for the FD solver of Sicstus Prolog, has been prototyped by Ågren et al [25]. The trace is dedicated to Sicstus: the schema comes from the platform implementation.

Both tracers provide solver-dependent data about the execution. For instance, some details about the awaken demons or about the methods used to reduce a domain. Those data could be added in the current model as specific extensions: refinement of the constraint identification system, additional attributes of generic events, or specific events inserted into the generic events. The generic kernel of the trace is preserved while enabling solver-dependent analysis.

Some debugging tools enable users to interact with the execution states. Users can act on the current state of the execution to drive the search-tree exploration (Oz Explorer [2]), to add new constraints on a partial solution (CLPGUI [6]), to recompute a former state (both) or to cancel some constraints chosen thanks to failure explanations (PaLM [13]). Such interactions are not taken into account in the generic observational semantics (only the output flow of information is considered), but it would be useful to formalize such interactions in a generic way too. We will consider such extension in a future work.

It must be observed that interactive tools essentially use online tracing and debugging. Our first experimentation used post-mortem traces from different origins. The presented approach is however suitable for both online tracing and post-mortem analysis. Indeed, the contents of the flow of information produced by a tracer can be filtered according to what is requested by the debugging tools [7, 26]. Therefore, no useless information is sent over to the tools. The tools presented in Sec. 5 are currently being adapted to support on the fly analysis too.

7 Conclusion

We have presented a generic trace schema based on a generic observational semantics which formalizes relevant aspects of finite domain constraint programming and solving. Based on this semantics, tracers can then build specialized generic traces in which most of the debugging tools can find the information they need. In this paper the specialization functions have been rigorously but informally presented. In order to facilitate the use of the generic observational semantics on new solvers it is the matter of a current work to define these functions more formally.

We have shown that at least three representative families of finite domain solvers can be observed by this semantics: constraint logic programming (Gnu-Prolog), CSP library (Choco) and explanation based CP (PaLM)[5]. We have also shown that three different debugging tools can find in the specialized traces the whole information they require.

We would like to emphasize that there is no restriction on the number or on the size of the attributes of the generic trace. One may object that such a trace may slow down considerably any tracer. This problem can be overcome as there exist 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 [7, 26]. Such mechanism is currently implemented in Codeine. It will ease the use of the GNU-Prolog tracer by providing a convenient programming interface to implement trace-based debugging tools.

Several tracers based on the generic trace approach have been implemented by different groups. Our experimentation with Codeine, the GNU-prolog propagation tracer, shows that such a tracer is reasonably efficient [10]. Moreover the specialized observational semantics plays the role of a specification and the implementation of the tracer is considerably facilitated [28]. On the other side, several new debugging tools (based on recent visualization techniques) have been developed independently using the generic trace as input data only. They can be used on several solvers without difficult adaptation. The generic observational semantics can be viewed as a standard for the output of solvers and input of the tools. This standard has currently the form of an XML DTD defining the syntax of the generic trace schema [15].

The generic trace schema should not be considered as definitively settled. Although first experimentation has shown its robustness, more information may be needed when new tracers and new tools will be considered. However the presented schema allows extended experimentations with many debugging tools to be started. We intend to pursue such experimentations. We also expect that this approach contributes to facilitate and to stimulate the production of propagation tracers and new debugging tools.

Acknowledgements The authors would like to thank Narendra Jussien and Guillaume Rochart for their technical explanations about the exact behaviour of the PaLM platform, Mohammad Ghoniem and Thomas Baudel for explanations about VISADJ and Discovery respectively. We also thank anonymous referees for their many interesting and useful remarks.

[5] Another tracer is currently being developed for CHIP [27].

References

1. Meier, M.: Debugging constraint programs. In Montanari, U., Rossi, F., eds.: Proceedings of the First International Conference on Principles and Practice of Constraint Programming. Number 976 in LNCS. Springer Verlag (1995) 204–221
2. Schulte, C.: Oz Explorer: A Visual Constraint Programming Tool. In: Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97), Leuven, Belgium, The MIT Press (1997) 286–300
3. Müller, T.: Practical investigation of constraints with graph views. In: Principles and Practice of Constraint Programming – CP 2000. Number 1894 in LNCS, Springer-Verlag (2000)
4. Simonis, H., Aggoun, A., Beldiceanu, N., Bourreau, E.: Complex constraint abstraction : Global constraint visualization. In Deransart, P., Hermenegildo, M., Małuszyński, J., eds.: Analysis and Visualization Tools for Constraint Programming. Number 1870 in LNCS. Springer Verlag (2000)
5. Bracchi, C., Gefflot, C., Paulin, F.: Combining propagation information and search-tree visualization using OPL studio. In Kusalik, A., Ducassé, M., Puebla, G., eds.: Proceedings of Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE'01), Cyprus, Cyprus University (2001) 27–39 Available as CoRR:cs.PL/0111042, `http://xxx.lanl.gov/abs/cs.PL/0111042`.
6. Fages, F.: CLPGUI: a generic graphical user interface for constraint logic programming over finite domains. In Tessier, A., ed.: Proceedings of the 12th Workshop on Logic Programming Environments (WLPE'02), Copenhagen (2002) Available as CoRR:cs.SE/0207048.
7. Ducassé, M.: Opium: an extendable trace analyzer for Prolog. The Journal of Logic Programming, special issue on Synthesis, Transformation and Analysis of Logic Programs **39** (1999) 177–223
8. Jussien, N., Lhomme, O.: Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence **139** (2002) 21–45
9. Diaz, D.: GNU-Prolog, a free Prolog compiler with constraint solving over finite domains (2003) `http://gprolog.sourceforge.net/`, Distributed under the GNU license.
10. Langevine, L., Deransart, P., Ducassé, M.: A propagation tracer for GNU-Prolog: from formal definition to efficient implementation. In Palamidessi, C., ed.: Proc. of 19th International Conference on Logic Programming (ICLP 2003). Volume 2916., Springer Verlag (2003) 269–283
11. Deransart, P., Ed-Dbali, A., Cervoni, L.: Prolog, The Standard; Reference Manual. Springer Verlag (1996)
12. Laburthe, F., the OCRE research group: CHOCO, a Constraint Programming kernel for solving combinatorial optimization problems. (2001) Available at `http://www.choco-constraints`.
13. Jussien, N., Barichard, V.: The PaLM system: explanation-based constraint programming. In: Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000, Singapore (2000) 118–133
14. Debruyne, R., Ferrand, G., Jussien, N., Lesaint, W., Ouis, S., Tessier, A.: Correctness of constraint retraction algorithms. In: FLAIRS'03: Sixteenth International Florida Artificial Intelligence Research Society Conference, St. Augustine, Florida, USA, AAAI press (2003) 172–176

15. Fekete, J.D., al.: Proposition de format concret pour les traces générées par les solveurs de contraintes. Public deliverable d2.2.2.1, INRIA, INRIA (France) (2001) http://contraintes.inria.fr/OADymPPaC/Public/delivrables.html.
16. Langevine, L.: Codeine, a propagation tracer for GNU-Prolog (2003) http://contraintes.inria.fr/~langevin/codeine.
17. Langevine, L., Deransart, P., Ducassé, M., Jahier, E.: Tracing execution of CLP(FD) programs, a trace model and an experimental validation environment. RR 4342, INRIA, Rocquencourt (France) (2001) Also in post-ICLP'01 workshop: 11th Workshop on Logic Programming Environments.
18. Aggoun, A., Chan, D., Dufresne, P., Falvey, E., Grant, H., Herold, A., Macartney, G., Meier, M., Miller, D., Mudambi, S., Perez, B., van Rossum, E., Schimpf, J., Tsahageas, P., de Villeneuve, D.: ECLIPSE 3.5. User Manual. European Computer Industry Research Centre (ECRC), Munich. (1995)
19. G. Arnaud: PAVOT Reference Manual (2004) http://contraintes.inria.fr/~arnaud/pavot/.
20. Van Hentenryck, P.: Constraint satisfaction in logic programming. The MIT Press (1989)
21. Baudel, T., et al: Discovery reference manual (2003) Manufactured and distributed by Ilog, http://www2.ilog.com/preview/Discovery/.
22. Gonhiem, M., Jussien, N., Fekete, J.D.: Visualizing explanations to exhibit dynamic structure in constraint problem satisfaction. In O'Sullivan, B., ed.: Proceedings of the third International Workshop on User Interaction in Constraint Satisfaction (UICS'03), Cork University (Ireland) (2003) Available at http://www.cs.ucc.ie/~osullb/UICS-03/.
23. Ilog: Solver 5.1 reference manual (2001) http://www.ilog.com/.
24. Deransart, P., Ducassé, M., Langevine, L.: A generic trace model for finite domain solvers. In O'Sullivan, B., ed.: Proceedings of the second International Workshop on User Interaction in Constraint Satisfaction (UICS'02), Cornell University (USA) (2002) Available at http://www.cs.ucc.ie/~osullb/UICS-02/papers/deransart_et_al-uics02.ps.
25. Agren, M., Szeredi, T., Beldiceanu, N., Carlsson, M.: Tracing and explaining execution of CLP(FD) programs. In Tessier, A., ed.: Proceedings of the 12th Workshop on Logic Programming Environments, Copenhagen, Denmark (2002) Available as CoRR:cs.SE/0207047.
26. Ducassé, M., Langevine, L.: Automated analysis of CLP(FD) program execution traces. In Stuckey, P., ed.: Proceedings of the International Conference on Logic Programming, Lecture Notes in Computer Science, Springer-Verlag (2002) Poster. Extended version available at http://www.irisa.fr/lande/ducasse/.
27. Cosytec: CHIP++ Version 5.2. documentation volume 6. http://www.cosytec.com (1998)
28. Ducassé, M., Langevine, L., Deransart, P.: Rigourous design of tracers: an experiment for constraint logic programming. In Ronsse, M., ed.: Proc. of AADEBUG'03, Gent University (2003) Available as CoRR:cs.SE/6309027.
29. Deransart, P., Hermenegildo, M., Małuszyński, J., eds.: Analysis and Visualization Tools for Constraint Programming. Number 1870 in LNCS. Springer Verlag (2000)