Generic Trace Format for Constraint Programming
Version 2.1

The OADymPPaC RNTL Project

July 21, 2004

Abstract

This document describes a generic trace format for finite constraint solvers, including search space and propagation, its syntax and its semantics. This generic trace enables debugging tools to be defined almost independently from finite domain solvers, and conversely, tracers to be built independently from these tools.

The generic trace syntax is represented using an XML DTD, called “gentra4cp.dtd”. A compliant trace is encoded in an XML format according to this DTD and follows the semantics described as the generic trace model.

The trace contains also some elements for communications and synchronization between solvers and debugging tools.

Contributors

This document has been established with contributions of several partners in the French RNTL Project OADymPPaC (November 2000 - May 2004)

Cosytec SA Abderrahmane Aggoun, Mohammed Inelhaj, Raphaël Martin

EMN (Ecole des Mines de Nantes) Romuald Debruyune, Narendra Jussien, Mohammad Ghoniem

ILOG SA Thomas Baudel

INRIA-Rocquencourt **Pierre Deransart (Editor)**, Guillaume Arnaud, Ludovic Langevine, François Fages

INRIA-Futurs Jean-Daniel Fekete

INSA-Rennes Mireille Ducassé

LIFO (University of Orléans) Alexandre Tessier, Gérard Ferrand, Willy Lesaint.

Copyrights

Foreword: History and Future

The idea of having a generic trace format was first considered in the DiSCiPL project [5] which did not follow the idea and thus focused on the development of new debugging tools. The resulting tools were running only on the platforms for which they had been developed.

The motivation for a generic trace format is twofolds: to offer the possibility of interoperability between solvers and tools (to allow tools to be developed for different constraint solvers) and to understand the meaningful aspects of constraints solving. The OADymPPaC Project [18] focused almost on the first aspect. But there is no possibility to fix a generic level of trace without ideas concerning the relevant aspects of constraint solving.

The generic trace format presented in this document covers finite domain solvers and has been experimented with several tracers and debugging tools. The tracers have been developed for GNU-Prolog, PaLM, Choco and CHIP V5. Tools are ILOG Discovery, Cosytec Visual Search-Tree and Visualize, INRIA Pavot and CLP-GUI systems. Most of them can be freely dowloaded from the OADymPPaC public web site.

This generic trace format has been created, developed, experimented and updated during the fourty monthes of the OADymPPaC project. This is far to be sufficient to claim that the presented version is the ultimate one. We hope it will be a useful starting point.

We expect that solver and tool implementors may be willing to join this experience and contribute to experiment, using and improving this format. We would encourage constraint solvers implementors to include this format in their tracers and environment tools builders to use this format as one of their entry.Several extentions need to be considered: better precision in the relevant concepts, inclusion of different constraint domains and search methods, interactions between solver and tools.

To continue this work all the results of the OADymPPaC project will be made public domain on the project site and further experiments and developments will be recorded and coordinated through an international open network.

Contents

List of Figures

Chapter 1

Introduction

This document describes a *generic trace format* for finite domain constraint solvers, including search space and propagation. This format is intended to facilitate adaptation of debugging tools on different finite domain solvers. It enables debugging tools to be defined almost independently from finite domain solvers, and conversely, tracers to be built independently from these tools. For this reason it is qualified “generic”. The generic trace format contains the definitions of the trace events that each tracer should generate when tracing execution of a finite domain constraint solver. The corresponding trace is called for short the *generic trace*, and a generic trace event a *generic event*.

This document defines the syntax and the semantics of the generic trace.

As illustrated by Fig. 1.1 each tracer may generate a *specific trace* with many particular events not taken into account by the generic trace. It is thus requested that each event whose semantics corresponds to a generic event must be represented according to the syntax described in this document. It is also requested that the subsequence of the specific trace which corresponds to the generic trace must be a consistent generic trace, i.e. a trace whose syntax and semantics follows this document and thus can be understood by the debugging tools. Notice that not all solvers may be able to generate all described generic events. Thus the generic trace format describes a superset of the generic events a particular tracer is able to generate.

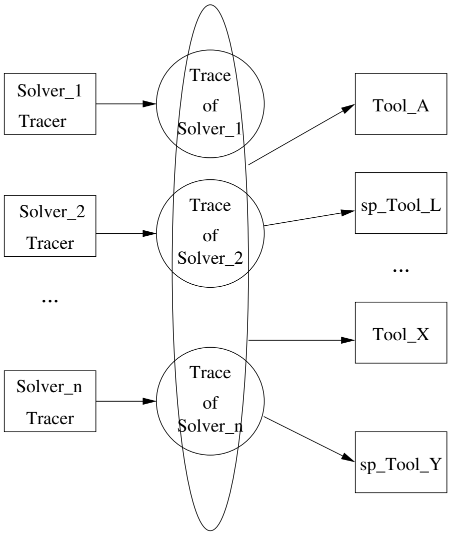

Figure 1.1: Connecting Tracers to Debugging Tools

On the other side a “portable” debugging tool should be able to extract from a specific trace and to understand the sub-flow of events corresponding to the generic trace. The Fig. 1.1 illustrates two cases: portable tools which use the generic trace only, and specific tools which uses also the specific parts of a specific trace. Both situations are acceptable. A specific tool which relies on specific trace events may be more difficult to adapt to another solver.

Traces are encoded in an XML format, using the XML DTD described in this document. A trace must be a valid XML document according to this DTD. A trace with specific tracer events should be a valid XML document too and provide a reference to the corresponding DTD, fully compatible with the one described here.

The long term objective of designing a generic trace format is to fully define the communications between solvers and tools ensuring full compatibility of all possible debugging tools with all possible constraint solvers. This communication includes several flows in both directions (e.g. dynamic parametrisation of the trace, synchronization, re-execution, . . .). More work and experiments are necessary to fully formalize it. Only one part of this communication is considered in this document which includes some simple mechanisms of communication in order to allow further experimentation.

The Fig. 1.2 illustrates the structure of the communication between solver/tracer and debugging tool. It can be described by four flows, two for tracer/tool interactions and two for solver/tool interactions. It shows also that one part of the communication from solver to tool has been included in the generic trace. Notice that this implies that the user application must be able to communicate on demand information to the tracer. The four flows are as follows:

Tracer/tool interactions:

- **generic trace**: It consists of the generic trace events as defined formally in the generic trace model of Chap. 3, plus some informally defined solver-to-tool communication events. The syntax and the semantics of the generic trace is completely described in this document.
- **trace requests**: It consists of the dynamic parametrisation of the flow of trace with some synchronization, requested by the debugging tool. As the format of the request concerning the form of the trace may be related to the syntax of the generic trace, one chapter is devoted to this flow (Chap 8). It is based on the ideas of [6] and may serve as a basis for further standardisation of the communication from tool to tracer.

Solver/tool interactions:

- **solver requests**: It consists of other requests adressed by a debugging tool to a solver and the running application. It may concern store manipulation (adding or relaxing constraints) or control of execution (synchronisation, mode of execution, re-execution, backtracking on demand, . . .). These requests are not considered here, and need more studies. Some tool/tracer synchronisation aspects are considered in Chap 8.
- **tool commands**: It consists of the commands addressed by the solver and the running application to the debugging tool. This includes for example some display commands or specification of views, but also solver-to-tool synchronization. These commands are not specified here and need more studies to be included in the generic trace. However some events have been included in the trace to allow further experimentation and it is recommeneded to use them for such purpose.

The inclusion of elements of solver-to-tool communication in the generic trace does not mean that the whole communication should necessarily go through the generic trace flow, i.e. a single standardized flow. The authors consider that the solver/tool interactions could be studied separatly and, if necessary, be standardized separately.

Execution Overload Related to the Use of XML The generated trace is not intended to be stored in a file, but it is intended to be broadcasted to likely several processes which may analyse it on the fly. The XML format has the advantage to be humanly understandable and can be handled by numerous high quality program libraries. In turn, this format is extremely verbose and it is costy to make on the fly syntactic analysis.

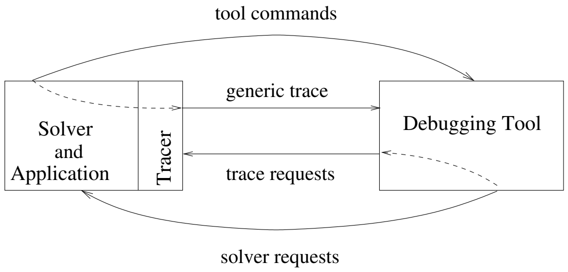

Figure 1.2: Communications between Solver and Debugging Tool

The debugging tools which have been primarily considered include sophisticated visualisation tool which handle predefined data stuctures (like table models[1] or graph models[2]) collecting likely complex items and thus use a front-end pre-encoding unit. Therefore in practice the pre-encoding time is greater than the time requested to generate XML encoded trace and read it. For this reason, although there exist efficient technique to compress XML data (for example *wbxml*) the authors did not consider the use of compression techniques to broadcast a smaller trace flow. More experimentation remains necessary.

Organisation of this Document

- Chap. 2 introduces the trace elements, organized in three modules (metadata, constraints and externals), and some syntactical conventions.
- Chap. 3 presents the formal semantics of the generic trace events of the constraint module. Other elements of the generic trace are informally specified.
- Chap. 4 introduces XML format of the elements of the Metadata and Stream Control Module.
- Chap. 5 and Chap. 6 introduce XML format of the trace event related to search control and constraint propagation in the Constraints Module.
- Chap. 7 introduces XML format of the elements of the Externals Module.
- Chap. 8 introduces the problem of the interactions between tracers and debugging tools. It is not normative, but it clarifies the way tracer and tool may be synchronized and parametrized in order to exchange information.
- Chap. 9 defines what are compliant tracers and tools, and compliant trace extentions.
- The annex contains the gentra4cp DTD (annex A), examples of trace specification (annex B) and examples of trace (annex C) issued from different solvers.

[1]Table model: data structure used as entry of the Discovery tool [2].
[2]Graph table: data structure used as entry of the INFOVIS tool [7].

Chapter 2

Syntax: Overview of the Trace Format

This chapter gives a summary of all the generic trace events and some conventions regarding the syntax of the trace.

2.1 Elements of the Trace

A trace is made of trace events. Each trace event has a name (its type, also called a *port*), a sequential number which identifies the event and several other attributes. However a trace document may contain other kind of information regarding its proper presentation (e.g. meta-data) which do not need to be numbered. This information is also presented in the form of small units of similar form. All these units together with the trace events will be called *trace elements*. For sake of clarity the trace elements are organised in three modules

1. **meta-data and stream control module** all information about the trace, granularity and synchronization events.
2. **constraints module** all events defined in the generic trace model (control and propagation).
3. **externals module** an experimental set of events for several kinds of annotations, display commands or computation phases relative to an application.

2.1.1 Elements of the Meta-data Module

There are five events corresponding to meta-data, i.e. trace identification and control:

header a trace identification (origin, date, . . .) including a specification of the tracer capabilities;

provide level of details of the trace in the trace document (also used for tracer specification) ;

complement a complement of attributes regarding the last trace event;

packet a way to split the trace into smaller encapsulated pieces (also used for synchronization);

breakpoint synchronization of the tracer with a tool: the tracer stops and waits for some request and/or signal to continue.

2.1.2 Elements of the Constraints Module (the Ports of the Generic Trace Schema)

A port (cf. Sec. 3.4) is one of the 15 generic trace event types concerning constraints whose semantics is defined in Sec. 3.3. First nine events are said *control ports* and the six following are *propagation ports*.

- 9 control ports (*constraints module*)

new-variable declaration of a new variable;

new-constraint declaration of a new constraint;

post introduction of a constraint into the store;

choice-point creates a new node in the search-tree (if the search-space can be represented by a search-tree), otherwise an indication that the current state is a choice-point.

back-to declaration of a jump to a previously created node of the search-tree;

solution creates a new succes leaf in the search-tree;

failure creates a new failure leaf in the search-tree;

remove withdrawal of a constraint from the store;

restore restoration of some values in the domain of some variable.

- 6 propagation ports (*constraints module*)

reduce reduction of the domain of some variable;

suspend suspension of a constraint (the satisfiability may not be known);

solved declaration that a constraint is solved (it is not active anymore and it does not influence further reductions either);

reject declaration that a constraint is unsatisfiable;

awake a woken constraint becomes active;

schedule reorganisation of suspended constraints and solver events.

2.1.3 Elements of the Externals Module

There are 7 events to send commands to external processes through the trace.

annotation passing information by an annotation (a way to communicate with the debugging tool);

new-stage declaration of a new stage ;

start-stage beginning of a stage ;

suspend-stage suspension of a stage ;

resume-stage re-start of a suspended stage ;

stop-stage end of a stage.

2.2 Lexical rules, XML Terminology and Syntactic Conditions

A trace document is a valid XML document, according to recomendations [3], compatible with [20].

2.2.1 The “gentra4cp” DTD

A trace document, i.e. a complete trace generated by a compliant tracer, follows the lexical and syntactical conventions of XML and of the DTD named `gentra4cp.dtd`. A trace document is surrouded by the tags `<gentra4cp></gentra4cp>`.

For each element defined in the DTD we describe its function, its attributes[1], its contents, its XML declaration (as it is in the DTD) and a short illustrative document sample as example.

[1] Here “attributes of an XML element”. In this document “attribute” is used in different places with different meanings: attributes of solver events (as defined in Chap. 3) or attributes of XML elements. The right meaning should be clear from the context.

The expressive power of the used DTD (basically a contex free grammar) is very low but sufficient for our purpose. We do not use the whole syntax specification power of XML DTDs. In fact the generic nature of the trace schamata obliges to leave many details open. Furthermore there are some context sensitive conditions (like references to a unique identifier) which are required but informally specified. This is acceptable, since a trace document is supposed to be produced automatically by a compliant tracer and thus should satisfy automatically these context sensitive conditions.

In this DTD we make use of the *entities* (macros) to express types of attributes. These types cannot be validated by a standard processor, but are readable in the DTD. They could be described more precisely using XML Schema. However the proposed format seems sufficient enough at this stage. More experience is needed to make a more precise specification of the generic trace syntax.

2.2.2 Terminology

In XML a pair of tags `<tag></tag>` denotes an *element*. There may be empty elements denoted `<tag />`. Even if some elements are used to represent trace events, they should not be confused. There is no XML event. An *Event* always refer to an execution event.

Similarly a tag may have a sequence of valued fields, for example `<reduce chrono="17" depth="2"/>`. These fields are named *attributes* in the XML terminology. But in Chap 3 as in several other places we refer to attributes of the trace events. If the context is not clear enough we will make the distinction using the notation *XML attribute* or *event attribute*. Notice that XML attributes of XML elements corresponding to trace events are also trace attributes, but a trace event attribute may sometimes be represented by an XML content instead of an XML attribute (e.g. `<state>` is an XML content representing an event attribute).

2.2.3 Codings

XML allows to use extended Unicode character set [19] and defines five reserved characters only: `&`, `<`, `>`, `"` et `'`, respectively named `&`, `<`, `>`, `"` and `'`.

For particular or national characters, XML uses the 7 bits encoding UTF-8 [21].

The trace described in this document are coded in UTF-8. A compliant tracer which makes use of national characters should follow this norm.

2.2.4 Context Sensitive Conditions

`<new variable>`, `<constraint>`, `<annotation>` or `<new-stage>` declarations and some other XML elements are trace events containing an identifier (resp. `vident`, `cident`, `aident`, `sident`) which is unique in the context of the whole trace document, even if it is split in several files. This is not specified in the DTD (they are declared as `CDATA`), but it is a strong requirement and must be guaranteed by the tracer.

Furthermore, there are many XML attributes like `vname`, `cname`, or XML contents which are just declared as `CDATA` or `PCDATA` (usually unformated text) with no detail or very few details on the syntax of the corresponding text. For example we do not specify how the name of a variable (constraint, annotation or stage) which may be used in a debugging tool (`vname, cname, ...`) must be encoded.

2.2.5 Attributes Recomended Values

In many places one will find element attributes defined with a CDATA and with "attribute recomended values" in the text. This holds in particular for the attribute `status` of `<state>`, `type` of `<new-variable>`, or `types` of `<update>`. The values are specified in the text, not in the DTD. The recommended values are normative and the recommended strings should be used with the right semantics. However due to the generic nature of the trace, it may happen that, for some solver, some new value must be introduced. The choice made in the DTD allows to do this without changing the DTD. These additional values are tracer defined and should include the old ones.

This form of specification should facilitate practical refinements or extensions, when needed. In fact there is no sufficient experience to make more precise specification at this time.

Another reason to use CDATA is to facilitate the use of empty attributes in the `<provide>` element. In fact in the case of implied attributes, an XML parser may generate default values for such attributes even if it is empty. This choice simplifies the parsing task in this case.

Chapter 3

Semantics: Generic Trace Model

This chapter gives the semantics of the generic trace events and specifies the generic trace schema (the type of the events and their main attributes). In the generic trace there are three kinds of events: the events related to control (constraint and variable declarations, search-space evolution), events related to propagation (constraints effects and status), and events related to communication between solver and debugging tools. For the two first categories of events the semantics is given formally, based on a so called "observational semantics", defined by rules acting on abstract states. This semantics models most of the solver aspects which are interesting to observe when trying to analyze the behavior of a finite domain solver.

3.1 Finite Domain Solvers and Resolution

We first give a short informal presentation of a unified view of finite domain solvers, independent from the nature of the platform and of the language supporting the solver. General introduction to constraint programming may be found in [16, 1].

3.1.1 Constraint Problem and Solutions

A finite domain constraint problem is specified by the following elements.

- a finite set $\mathcal{V}$ of finite domain variables;
- a finite set D containing all possible values for variables $\mathcal{V}$; in FD-solvers, D is a set of non negative integers ranging over 0 to *maxint*;
- a function which associates to each variable v its current domain (a subset of D), denoted by D_v; min_v and max_v are respectively the lower and upper bounds of D_v;
- a finite set of constraints $\mathcal{C}$ denoted the *store* in the sequel. Each constraint c defines a relation between the variables $var(c)$, a subset of $\mathcal{V}$, the variables of c.

A solution of the constraint problem is an assignment of the variables of $\mathcal{V}$ to values in D, such that all constraints in $\mathcal{C}$ are satisfied. All the constraints are thus *solved* and the problem is said to have a solution.

A constraint is said *solved* (assuming only a subset of its variables have an assigned value) if any assignment of its remaining variables is a solution. For example whatever are the other constraints of the problem, the constraint `0+x#=x` is solved.

For a given problem there may be a lot of solutions, especially in the presence of many symmetries (leading to apparently trivially equivalent solutions like naming conventions or geometrical invariance). There may be also no solution at all. If a constraint is not satisfiable (no assignment can make it true) it is said *false* or equivalently *rejected*.

If there is no solution but a solution is still expected, the problem is said *over-constrained*. The solutions may also be approximated instead of given exactly. A solver guarantees only that, if there is a solution,

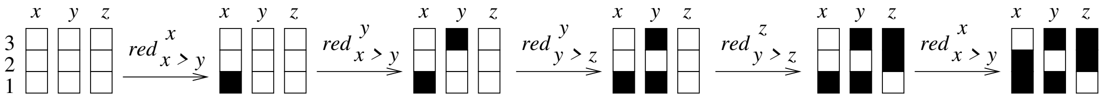

Figure 3.1: Application of reductions to the system $\{x > y; y > z\}$.

then it is in the given approximation. Due to the "incompleteness" of most of the solvers, the existence of a solution may not be guaranteed. Only the absence of solution, if a complete search terminates, is guaranteed.

Most of the constraint problems have non polynomial algorithmic solutions and are considered intractable. There are several ways to try to find solutions using complete (systematic exploration of the search space) or incomplete approaches (using local search techniques or genetic algorithms). All use families of heuristics in order to find a solution as quick as possible.

The process to find solutions, including search algorithms and heuristics, is called *resolution*. The resolution may be conducted in a systematic, but not necessarily exhaustive, way of exploring the search space. In many cases the search space can be formalized by a tree called *search-tree*. The way the search space is explored corresponds to the *control*[1].

The resolution uses also different processes called *propagation*. They consist of computation of solution approximations, narrowing the domains of the variables by repeated withdrawals of proved inconsistent values.

The operational semantics of constraint programming results from the combination of these two paradigms: control and propagation. Different solvers may differ by the control and by the propagation algorithms. The language in which a solver is embedded may contribute to the control, when the propagation is a proper characteristic of the solver.

3.1.2 Generic View of Propagation

Propagation can be performed as soon as there is a constraint in the store. For a given constraint, a set of inconsistent values is withdrawn from the domains of its variables[2]. These values can be determined by local consistency algorithms such as "node consistency", "arc consistency", "hyper-arc consistency" or "bounds consistency" described for example in [16]. Following the approach of Ferrand et al [8], we introduce the *local reduction operator*, as a way to give a generic specification of the propagation.

To every constraint it can be attached a set of *local reduction operators*.

Definition 1 (Local Reduction operator) *A* reduction operator red_c^v *is a function attached to a constraint* c *and a variable* v*. Given the domains of all the variables used in* c*, it returns the domain* D_v *without the values of* v *which are inconsistent with the domains of the other variables. The set of withdrawn values is denoted* Δ_v^c*.*

$red_c^v(D|_{\mathbf{var}(c)}) = D_v - \Delta_v^c.$

There are as many reduction operators attached to c as variables in $\mathbf{var}(c)$. In general, for efficiency reasons, a reduction operator does not withdraw all inconsistent values.

A simple example of reduction operator for $c : x > n$, where n is a given integer, is $red_c^x(D|_{\{x\}}) = D_x - \{u \mid u \in D_x \wedge u \leq n\}$.

The evolution of the domains can be viewed as a sequence of applications of reduction operators attached to the constraints of the store. Each operator can be applied several times until the computation reaches a fix-point [8]. This fix-point is the set of final domain states.

An example of computation with reduction operators is shown in Figure 3.1. There are three variables x, y and z and two constraints, $x > y$ and $y > z$. At the beginning, $D_x = D_y = D_z = \{1, 2, 3\}$, represented

[1]In this document "control" is used in different places with different meanings: control relative to the way of exploring the search space, and control related to the tracer-tool dialog. The right meaning should be clear from the context.

[2]In the generic trace this will result in a sequence of reduce events, one by variable whose domain is modified.

by three columns of white squares. Considering the first constraint, it appears that x cannot take the value "1", because otherwise there would be no value for y such that $x > y$; $\mathrm{red}^x_{x>y}$ withdraws this inconsistent value from D_x. This withdrawal is marked with a black square. In the same way, $\mathrm{red}^y_{x>y}$ withdraws the value 3 from the domain of y. Then, considering the constraint $y > z$, the operators $\mathrm{red}^y_{y>z}$ and $\mathrm{red}^z_{y>z}$ withdraw respectively the sets $\{1\}$ and $\{2, 3\}$ from D_y and D_z. Finally, a second application of $\mathrm{red}^x_{x>y}$ reduces D_x to the singleton $\{3\}$. The fix-point is reached. The final solution is: $\{x = 3, y = 2, z = 1\}$.

3.1.3 Generic Solver Events

In the generic trace, the reduction operators are not characterized, since they are embedded in more general and solver dependent reduction algorithms[3]. Only a solver defined abstraction of the effect of the operator will be notified. It is called *solver event*.

Each successful application of a local reduction operator generates a set of solver events.

Definition 2 (Solver event) *A* solver event *is a couple (v, t) of variable, which characterizes the effect of the application of a reduction operator red^v_c on the domain of a variable v of the constraint c. The effect t is characterized by a* type of update *of the domain of the variable.*

With each predefined constraint of a finite domain solver it can be associated a set of solver events which may be produced when this constraint is handled.

Depending on the solvers the nature or the form of the solver events may differ slightly (see [14] for a study of several finite domain solvers), but all can be abstracted as described.

3.1.4 Generic View of Scheduling Actions and Constraints Awakening

The way a fix-point is computed depends on the solver. The resolution is driven by the constraints. At some moment one constraint is said *active* and its local reduction operators are applied.

For each successful application of an operator, the generated *solver events* are stored in a set of the *sleeping events*. In fact these events will be used later to drive resolution. After all reduction operators of a constraint have been applied, and if the constraint is not solved nor rejected, it becomes "sleeping"; that is to say it is inserted in the set of *sleeping constraints* until it may contribute to some further reduction.

Notice that since all solvers do not recognize always that a constraint is solved, the constraint may be activated for some further reduction even if it cannot contribute anymore to any value withdrawal.

At any time, the sleeping constraints are all the constraints in the store which are neither active, nor solved, nor rejected. The sleeping events are all the solver events which may still contribute to wake up some sleeping constraint. The solver events contribute to the constraint awakening conditions.

Definition 3 (Constraint awakening condition) *The* awakening condition *of a constraint c is a predicate depending on solver events. This condition holds when a new value withdrawal can be made by the local reduction operators of this constraint. The condition is optimal when it holds* if and only if *a value withdrawal can be made.*

The awakening condition of c by the event e is denoted in the generic trace model by $\mathrm{awakecond}(c, e)$.

The awakening conditions are solver dependent, and result from a compromise between the cost of their computation and how many awakenings they prevent.

At each step of the propagation, a constraint is selected according to a given strategy depending on implementation (for example a constraint with more variables first) and one of the reduction operators of the selected constraint is applied. This choice is solver dependent and is formalized by the *scheduling action*.

Definition 4 (Scheduling action) *A* scheduling action *is a modification of the sets of sleeping constraints and events, leading these sets in a different internal state. It depends from at most one constraint and one event.*

The scheduling action defined by the sleeping constraint c and the sleeping event e is denoted by $\mathrm{action}(c, e)$.

[3]These algorithms may be referred in the `reduce`.

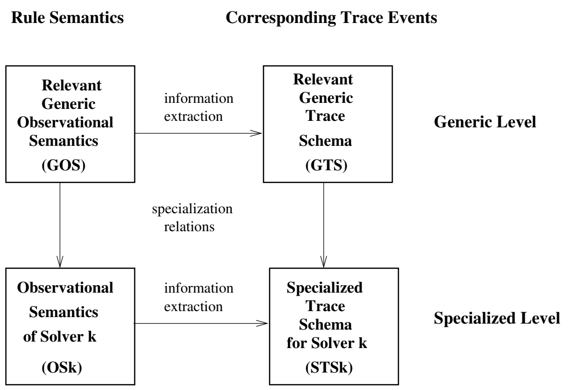

Figure 3.2: From Observational Semantics to Trace Schemata (Top: *generic*, Bottom: *specialized*)

This scheduling action essentially consists in choosing which constraint will be woken and become active.

The propagations ends and a fix-point is reached when no more constraint is active nor can be woken.

To sum up, there are three fundamental operations: *scheduling* of constraint to be woken with the corresponding event in the sets of sleeping constraints and events, *awakening* of constraint (with the event), which become active, and *reduction* of variable domains.

3.2 Generic Trace versus Specialized Trace

A generic trace is a sequence of trace events reflecting the resolution process with control and propagation. Each event is assumed to correspond to a meaningful state transition in every finite domain solver. There is a finite set of such meaningful transitions. These transitions are formally specified by rules at some abstract level and the set of these rules is called *Generic Observational Semantics* (GOS).

Finite domain solvers have operational semantics which may significantly differ from each other; but, the generic observational semantics is defined in such a way that, for any finite domain solver S_k, it is possible to interpret a reasonable subset of its rules (a "relevant subset") by relevant transitions of S_k. It has been shown in [14] that these transitions can be represented by a set of specialized rules in the semantics of solver S_k, leading to a corresponding *(specialized) Observational Semantics of the solver* S_k (OS_k). The relation between these two semantics can be viewed as a specialization relation from relevant GOS to OS_k. It is illustrated in the left part of the Fig. 3.2. This approach guarantees that it should be possible to implement in most finite domain constraint solvers, tracers which generates the events of the generic trace schema with the same semantics, as described in the generic observational semantics.

From the generic observational semantics it is possible to derive a *Generic Trace Schemata* (GTS) which is the description of the generic trace events (relation "information extraction" in Fig. 3.2). 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*. Each port has a set of *attributes* which correspond to the description of the modified elements of the solver abstract state. For a solver S_k, the corresponding relevant subset of trace events leads to a *Specialized Trace Schema for Solver k* (STS_k in Fig. 3.2) with the same ports and same attributes, but a likely specialized meaning.

It has been established in [4, 14] that each rule of the generic observational semantics characterizes aspects of the execution which are relevant for debugging.

3.3 Generic Observational Semantics of CP(FD)

A trace is a sequence of selected execution events. There is a finite set of event types defined in the generic trace schema. Each execution event is an instance of one of the element defined in the trace schema. 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 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 variable domains and the awakening process of constraints.

3.3.1 Solver State and Search-Tree State

Definition 5 *(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 declared variables;* *$\mathcal{C}$ is the set of 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 D);* *A is the set of active couples of the form (constraint, solver event[4]);* *E is the set of solved 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, solved or rejected. These states are consistent with the states Müller defines to describe the *propagators* in the Oz system [17]. 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. $solved(c, \mathcal{D})$) holds when the constraint c is considered as unsatisfiable (resp. solved: it is universally true and does not influence further reductions any more) by the domains in $\mathcal{D}$.

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

Definition 6 *(Search-Tree State) The search-tree is formalized by a set of ordered labeled nodes $\mathcal{N}$ representing a tree, 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 $\mathbb{S}$ are empty.

3.3.2 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. Definitions are on the right-hand side.

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

In the following Sec. 3.3.3 describes the control events and Sec. 3.3.4 the propagation events.

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

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

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

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

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

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

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

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

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

Figure 3.3: **Control rules** of the generic observational semantics

3.3.3 Control

The rules of Fig. 3.3 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 $\bot$ (no event) which denotes the activation of the constraint with no participation of any event.

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 create the leaves of the search-tree. solution specifies that the current solver state corresponds to a successful state and that a new solution leaf is created in the search-tree. failure specifies that the current solver state is failed (this may be due to a rejected constraint or some solver decision) and that a new failure leaf is created in the search-tree.

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 [13]. 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.

3.3.4 Propagation

The propagation can be described by state transition rules acting in the solver state, as illustrated by Fig. 3.4. These rules are formalized by Fig. 3.5. 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

[4]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.

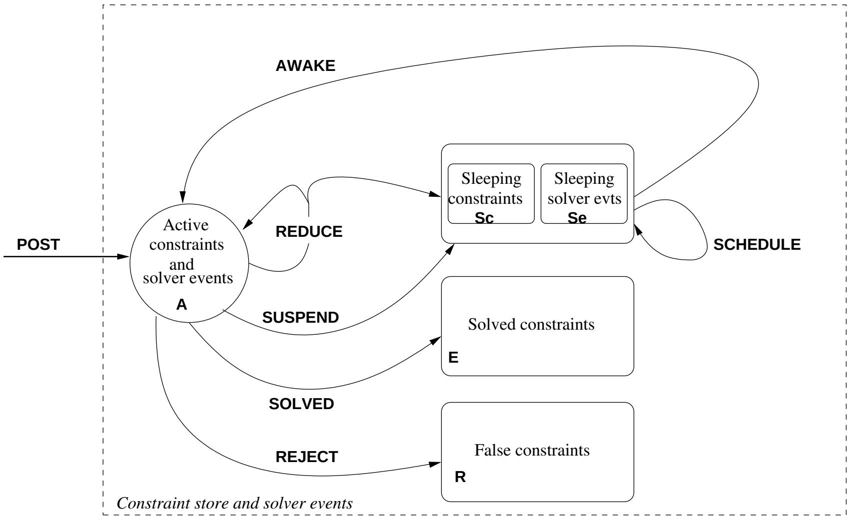

Figure 3.4: Generic Observational Semantics: illustration of the transitions described by the propagation rules

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

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

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

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

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

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

Figure 3.5: **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 solved is put in E (rule solved). 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 constraint.

Control Ports	
new variable	$v, \quad D_v$
new constraint	c
post, remove	c
restore	$v, \quad \Delta_v$
choice point	ν after the state transition
back to	ν before and after the state transition
solution, failure	ν after the state transition

Propagation Ports	
reduce	$c, \quad v, \quad \bar{e}', \quad \Delta_v^c$
suspend, solved	c
reject	c
awake	$c, \quad e$
schedule	$c, \quad e$

Figure 3.6: Formally defined **Specific Attributes of the Ports**

3.4 Generic Trace Schema

The two sets of rules presented by Fig. 3.3 and Fig. 3.5 specify an observational semantics. The 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*. Each port has a set of attributes. Some are common to all ports (the *common attributes*), the others are called *specific attributes*. We quote here the attributes described in the formal semantics. For the others, an informal specification is given in the following chapters.

Definition 7 (Generic Trace Schema) *A generic trace is a sequence of trace events identified by their port (the name of the corresponding semantic rule), and associated with a set of attributes: a sequential event number (monotonic, non necessarily contiguous integers); the depth in the search-tree; the whole 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, $\bar{e}'$. Fig. 3.6 presents the list of the formally described specific attributes for each rule, using the notations of Fig. 3.3 and Fig. 3.5.

3.5 Other Elements of the Trace

We introduce here a new specific attribute of the port reduce called explanation. This information may be used in debugging tools to “explain” the reasons of values withdrawal from the domain of a variable; in particular it allows to observe more accurately the mutual influence of constraints during resolution.

3.5.1 Explanations

`explanation` There are several approaches of explanations ([12, 10]). We use here the generic approach of explanations presented in [10] and its application to the generic trace described in [9].

An explanation is a tree whose root is a value withdrawal (a pair variable-value meaning that the value has been removed from the domain of the variable) and all the successive children are other value withdrawals. Such (possibly very huge tree) can be used to find the origin of an erroneous withdrawal (usually a withdrawal leading to reject unexpectedly a constraint and leading to an unexpected failure). As to each withdrawal it can be associated a local reduction operator, it is thus possible with an appropriate debugging tool to find the constraint which may be responsible for the erroneous root. In practice however there may be several such trees with the same erroneous root.

There are two ways of representing explanations in the generic trace: with constraints or with "causes".

In the first way each value withdrawal of a variable is associated a the tree of constraints that makes the value inconsistent. The tree may also be flatten, in this case it is a set of constraints. This approach is implemented in [12]. In this approach the complete tree of the explication must be built by the tracer. In [12] it is built by the solver, and therefore there is no additional cost for the tracer.

The second approach [10] may be more suitable for solver which do not compute already explanations. However it is necessary to generate in the trace the sufficient elements such that a debugging tool may build either an explanation tree as in [9] or the corresponding tree with constraints. Thses elemeents correspond to what we call the "causes" in the generic trace.

The withdrawal by constraints c of several values from de domain of the variable v, Δ_v^c, may be partitionned into $k > 0$ smaller subsets of withdrawn values (in the best case all subsets are singleton): $0 < i \leq k$: $\Delta_v^c(i) \subseteq \Delta_v^c$. Each part of Δ_v^c admits an explanation which can thus be represented by a "cause".

An explanation for (v, Δ_v^c) is thus a set of k pairs whose first element is $\Delta_v^c(i)$ and the second a set of causes. Each cause is a pair $(v', \Delta_{v'}^{c'}(j))$ (previous withdrawals concerning variables $v' \in \mathbf{var}(c) \setminus \{v\}$[5]).

In the worst case $i = 1$ and $\Delta_v^c(i) = \Delta_v^c$, and in the best case the $\Delta_v^c(i)$ are singleton and there are as many as withdrawn values. This representation allows several levels of precision in the explanations. A structured representation of the causes and/or associated constraints can then be deduced from a trace containing such kind of explanations.

3.5.2 Solver-Tool Communication

The generic trace includes some additional trace events. They correspond to solver-to-tool communication events. They are informally described here.

Annotations There should be a way to allow an application to communicate several kinds of informations to a debugging tool through the trace. It is the purpose of the `<annotation>` event. Its main attribute is a string of textual data sent to the tool.

There may be several usages of such event. It may be used to send display commands to a debugging tool with visualization devices. It may also be used to define new semantical objects based on groups of variables. For example in debugging of scheduling problems the user is not interested in the direct manipulation of the variables of the problem, but in the manipulation of higher level entities like "tasks". A task can thus be defined by a group of three variables like starting time of the task, its duration and on which machine it is applied.

Stages Constraint solvers may be embedded in larger systems with several interleaved phases or *stages*. It may thus be useful to make explicit in the trace the scope of a procedure, the execution of a particular CLP predicate, the use of a specific algorithm, or particular steps of the resolution (typically: initial posting of constraints, propagation, then labelling).

In order to trace the bounds of these stages we introduced five events described in Chap. 7: `<new-stage>`, `<start-stage>`, `<suspend-stage>`, `<resume-stage>` et `<stop-stage>`.

Synchronization To allow interactions between the solver (application) and the debugging tool, the solver must be able to indicate to the tool that it is in a synchrone mode, waiting for some signal from the tool. A breakpoint event (`breakpoint`, Sec. 4.5.1) is thus included in the generic trace schema. Such event may be generated either by the tracer after answering some trace request, or by the running application. A breakpoint has a control attribute which characterizes the kind of breakpoint. The use of such features in a tracer-tool dialog is comprehensively presented in Chap. 8.

[5] v' is a variable of c different from v.

Chapter 4

Trace Structure, Metadata and Stream Control Module

4.1 Trace Structure

A trace document consists of

- a prologue identifying the document (<header> element) ;
- an element characterizing the set of trace events and attributes which will be encountered in the trace document (<provide> element). This element is normally required. If it is not given, the prologue must contain a <provide> element in the prologue.
- a possibly empty sequence of toplevel elements %Toplevel; or packets;
- a packet is a possibly empty encapsulated sequence of toplevel elements (<packet> element).

The <provide> element used in the prologue indicates the maximal level of details in the trace the tracer is able to provide. It allows a debugging tool to recognize at the beginning whether or not it can handle the trace. In the course of the trace a <provide> element specifies dynamically the level of details of the trace.

The <packet> element allows to split the trace into smaller pieces as explained in Sec. 4.5.

4.1.1 XML Declaration of the Toplevel Structure

```
<!ELEMENT gentra4cp (header, (%Toplevel; | packet)*) >
<!ATTLIST gentra4cp
   xmlns CDATA \#FIXED "http://contraintes.inria.fr/OADymPPaC">

<!ENTITY % Toplevel
  "(provide | complement | breakpoint | new-variable | new-constraint | post |
    choice-point | back-to | solution | failure | remove |
    restore | reduce | suspend | solved | reject | awake | schedule |
    annotation | new-stage | start-stage | suspend-stage |
    resume-stage | stop-stage )*" >
```

4.1.2 Example

This example shows a trace beginning with a prologue (<header> element), followed by a <provide> and a <packet> of trace events (here <new-constraint> and <new-variable> elements).

```xml
<gentra4cp>
<header>
  <date>2004-02-04 10:30:00</date>
  <source>queen-gnu.pl</source>
  <provide>
    <!-- ...-->
  </provide>
</header>
<provide>
    <!-- ...-->
</provide>
<packet control="breakpoint">
   <new-variable chrono="1" vident="1" vname="x">
     <vardomain> <values>1 2 3</values> </vardomain> </new-variable>
   <new-variable chrono="2" vident="2" vname="y">
     <vardomain> <values>1 2 3</values> </vardomain> </new-variable>
    <!-- ...-->
   <new-constraint chrono="5" cident="4"  cname="c1" cexternal="X ## Y">
     <update vident="1" types="ground min"/>
     <update vident="y" types="ground max"/>
   </new-constraint>
    <!-- ...-->
</packet>
<!-- ...-->
</gentra4cp>
```

4.2 Prologue header

A trace can be used on-line by some external process or stored into some file before being processed in a post-mortem mode. In any cases it must contain some data about its origin and purposes. This is the reason of the prologue `<header>`. Every new or stored trace should start with a prologue.

The prologue contains useful meta-data such as the date of creation or the traced solver, and gives the level of details of the trace.

The form of the elements of the prologue should follow the recommendations of the *Dublin Core Meta-data Element Set* (http://dublincore.org) which defines the XML elements of the most common data. Only two are mandatory (date and source). Only a subset of the *Dublin Core Meta-data Element Set* is proposed. The four last are new and are tracer defined.

date : `<date>` (req)[1] date and time of the trace creation. It is recommended to follow ISO 8601 (`yyyy-mm-dd hh:mm:ss`);

source : `<source>` (req) name of the program, problem or application whose execution is traced (recommendations to name program or trace files are in Sec 4.2.3);

creator `<creator>` (opt)[2] a person responsible for the generation of the trace;

contributor `<contributor>` (opt) an entity responsible for making contributions to the realization of the trace;

description `<description>` (opt) an account of the content of the trace (problem, instance, version, properties...);

identifier `<identifier>` (opt) a unique identifier of the trace file in the context of the trace production, as recommended in Sec 4.2.3;

[1] required (req).
[2] optional (opt).

rights <rights> (opt) information about the rights held in and over the trace, and to use it. It should probably be free of rights or it is proprietary and one should specify the rights to use it;

solver <solver> (opt) the solver used to generate the trace (official name and version);

parameters <parameters> (opt) additional information regarding the traced execution, for example the options of the program and the used search strategy . . .);

solver-parameters <solver-parameters> (opt) describes some characteristics of the solver/tracer which can help the tool rendering the trace;

model-parameters <model-parameters> (opt) describes some restrictions specified by the user of the solver/tracer characteristics which can help the tool rendering the trace;

checksum <checksum> (opt) allows control of trace file integrity in the case of transmission of a post-mortem trace. The checksum is computed on the file without the prologue using a one-way hash function like MD5; the content of <checksum> is the computed value (see `http://www.ariadne.ac.uk/issue17/biblink/`);

provide : <provide> (opt) the description of the maximal level of trace the tracer is able to generate, as specified in Sec. 4.3. It works as a specification of the used tracer.

Only <date>, <source> <solver>, <checksum> and some elements and attributes of <solver-parameters> may be given automatically.

The elements <solver-parameters> and <model-parameters> have the same contents: <vardomain> and <varenum>. The <vardomain> element is described in Sec. 5.3.1; it means that the union of the variable domains never goes beyond those bounds. The <varenum> element, if present, says that some variable domains may contain enumerated values.

The <solver-parameters> elements has four attributes: `back-to-strategy`, `vident`, `cident` and `nident`. The `back-to-strategy` may have two values anywhere (the search-tree construction may use heuristics and "jump" anywhere) or incremental (the search-tree is constructed in a depth-first left-to-right manner (a la GNU-prolog, for instance) and never jumps to a past choice-point other than one of its immediate ancestors). The three other attributes may have two values: index or string. The first value means that the corresponding identifier is an array index in a global table and therefore the tool can use an array instead of a hashtable to store the corresponding identifiers (respectively variables, constrainst or nodes). The second value indicates that the tool will have to use hashtables instead of arrays to store the corresponding identifiers.

4.2.1 XML Declaration of the header Element

```
<!ELEMENT header ( date, source, creator?, contributor?, description?,
                   identifier?, rights?, solver?, parameters?,
                   solver-parameters?, model-parameters?, checksum?,
                   provide? ) >

<!ELEMENT date (#PCDATA)>
<!ELEMENT source (#PCDATA)>
<!ELEMENT creator (#PCDATA)>
<!ELEMENT contributor (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT identifier (#PCDATA)>
<!ELEMENT rights (#PCDATA)>
<!ELEMENT solver (#PCDATA)>
<!ELEMENT parameters (#PCDATA)>

<!ELEMENT solver-parameters ( vardomain? , varenum? )>
<!ATTLIST solver-parameters
  back-to-strategy  CDATA #IMPLIED
```

```
  vident CDATA #IMPLIED
  cident CDATA #IMPLIED
  nident CDATA #IMPLIED >

<!ELEMENT varenum EMPTY >

<!ELEMENT model-parameters ( vardomain? , varenum? )>
<!ATTLIST model-parameters
   back-to-strategy  CDATA #IMPLIED>

<!ELEMENT checksum (#PCDATA)>
```

4.2.2 Example

A header with a simple tracer specification.

```
<header>
  <date>2004-02-04 10:30:00</date>
  <source>queen-gnu.pl</source>
  <creator>Guillaume Arnaud</creator>
  <contributor>INRIA-Rocquencourt</contributor>
  <description>Trace to display a Propagation Tree with
               PAVOT</description>
  <identifier>queen-150-gnu-firstsol-ff-mv.xml</identifier>
  <rights>Free for academical use </rights>
  <solver>GNU Prolog 1.2.16, traced by Codeine 0.7</solver>
  <parameters>queen(150,_), first solution, first-fail
              middle value first
  </parameters>
  <solver-parameters
     back-to-strategy="anywhere" vident="index" cident="string">
       <vardomain min="0" max="32768"/><varenum/>
  </solver-parameters>
  <model-parameters back-to-strategy="incremental">
       <vardomain min="0" max="12"/>
  </model-parameters>
  <provide>
<!-- see appendix B.1 for examples of tracer specification -->
  </provide>
</header>
```

4.2.3 Name of the Program and Trace Files

In order to facilitate the communication of post-mortem trace files, the following naming conventions are recommended.

Problem Source Code File: `problem[-instance]-solver[-version].xx`
where

problem : (req) a string name characterizing the problem ;

instance : (opt) a string characterizing the instance of the problem ;

solver (req) the name of the solver or tracer used to generate the trace ;

version (opt) the version of the solver or tracer used to generate the trace ;

xx (req) the corresponding suffix of the program file (there may be an additional suffix if compressed) ;

A program file may be an archive with several source files and named with the same convention. Each file in the archive should be named with the same convention.

Examples:

```
openshop-66-gnuprolog.1.2.16.pl
sorted-100-SICStus.3.11.0.pl
```

Trace File: `problemsourcecodefile[-typetrace].xml`
where

problemsourcecodefile : (req) the name of the problem source code file, as described above ;

typetrace : (opt) a string characterizing the kind of trace ;

xml (req) the suffix of the XML trace file (there may be an additional suffix if compressed) ;

Examples:

```
sorted-100-PaLM.xml
openshop-66-gnuprolog.1.2.16-100000-firstevents.xml.gz
```

4.3 Trace Parameters provide

Depending on the problem, a trace may not contain all events and attributes the tracer is able to produce. The purpose of the element `<provide>` is to specify either the maximal level of details of the trace (the whole events and attributes that the tracer is able to provide, the "maximal trace" `<provide>` specified in the prologue), or to declare the level of details of the current trace. In the later case, it must be a sub-pattern of the one declared in the prologue. The `<provide>` element allows a tool to recognize whether or not it can handle the trace (see Chap. 8 for more details on communications between tracer and debugging tool).

The content of the `<provide>` element is a sequence of elements describing the patterns of the trace events. There shall be one pattern only by trace event quoted in the `<provide>` element [3].

There is a small language to describe the patterns: each traced content must occur with all its traced attributes; and a traced attribute is represented with an empty value, the others are just ignored. These two rules are applied recursively (see examples).

The `<provide>` element may occur in several places in a trace: in the `<header>` element or later in the trace. We consider separately theses two cases.

- **The `<provide>` element in the `<header>` element**. It is used to specify the maximal level of the trace, i.e. it is a specification of the tracer. In this case, the content of the `<provide>` element may have a last `<state>` element describing the state that the tracer is able to provide at any time. This avoids to repeat with each trace element description the same state. So in practice there is no need to specify with each trace element description the description of its `<state>` element.

  However if a `<state>` element is given with an event description, it must be interpreted as a restriction: only the specified elements (contents and attributes) described here can be requested just after this event. In particular, if the state is empty (`<state/>`), it means that there is no possibility to provide any state just after this event.

[3] Repetition of patterns for the same event shall be treated as error by the trace analyzers.

- **The <provide> element(s) after the <header> element**. It gives a description of the following trace. It may be given by the tracer as an indication for the tools about the verbosity of the trace. In this case there is no <state> element at the end, but each element of the trace may be described with a particular state. Such state must be a sub-pattern of the <state> element given at the end of the <header> element (if any), otherwise it is an error.

  If a <state> element is given as last element of this <provide> (it must be a sub-pattern of the corresponding element in the <provide> element of the prologue, if any), it means that this state will be displayed with each event, unless a <state> element has been provided for this event.

Notice that the trace/tracer specification language of <provide> is sufficient to specify the tracers, but not to specify any kind of trace. This question is addressed in Chap. 8 and a complete language for trace requests is con sidered in [15].

4.3.1 XML Declaration of the provide Element

```
<!ELEMENT provide (( new-variable | new-constraint | post |
          choice-point | back-to | solution | failure | remove |
          restore | reduce | suspend | solved | reject | awake |
          schedule | annotation |  new-stage | start-stage |
          suspend-stage | resume-stage | stop-stage )* , state? ) >
```

4.3.2 Example

Sufficient specification of a trace to display a search-tree, including labelling and solutions. The <state> given in <new-constraint> allows to have the constraint variables references with this event.

```
<provide>
 <new-variable chrono="" vident="" />
 <new-constraint chrono="" cident="" >
     <state>
        <constraint cident="" >
          <variables/>
        </constraint>
     </state>
 </new-constraint>
 <post chrono="" cident=""/>
 <choice-point chrono="" depth=""/>
 <solution chrono="" depth="" >
    <state>
       <variable vident="">
          <vardomain size=""> <values/> <range from="" to=""/>
          </vardomain>
       </variable>
    </state>
 </solution>
 <choice-point chrono="" depth="" />
 <back-to chrono="" depth="" />
</provide>
```

Other examples of tracer specifications are given in Annex B. Examples of <provide> for several trace specifications are in Annex B.4.

4.4 Comprehensive Event complement

Not all the likely provided attributes of a trace event are displayed in the generated trace. As long as an attribute can be provided by the tracer (as specified by the <provide> element of the <header>

element), the tracer may be requested to generate a trace event including some more attributes than the one specified in the `<provide>` element defining the currently generated trace. This is the purpose of the `<complement>` element. It is particularly useful for example when, after a phase of execution with mute trace, the trace is again generated and the debugging tool need to know the current state, not displayed in the current trace event. This event can thus be regenerated (i.e. with the same chrono attribute) with the additional missing attributes.

The trace event attributes resulting of a traced event followed by one or several `<complement>` elements in the trace are the union of all described trace event attributes.

Notice that a `<complement>` element does not correspond to any trace event. Therefore it has no attribute and in particular no `chrono` attribute.

4.4.1 XML Declaration of the complement Element

```
<!ELEMENT complement ( state | new-variable | new-constraint | post |
    choice-point | back-to | solution | failure | remove |
    restore | reduce | suspend | solved | reject | awake | schedule |
    annotation | new-stage | start-stage | suspend-stage |
    resume-stage | stop-stage ) >
```

4.4.2 Example

```
<reduce chrono="477" cident="c4" >
   <vardomain size="2" min="1" max="3">
      <values> 1 3 </values>
   </vardomain>
</reduce>
<complement>
  <reduce chrono="477" vident="v13" >
    <state>
       <constraint cident="c4"  cexternal="equal(v13,v2)" status="sleeping">
          <variables> v13 v2 </variables>
       </constraint>
    </state>
  </reduce>
</complement>
```

Also,

```
<new-constraint cident="c1" chrono="4" > <variables> v1 v2 </variables>
</new-constraint>
<complement>
  <new-constraint cident="c1" chrono="4" depth="1" >
    <variables> v3 v4 </variables>
  </new-constraint>
</complement>
<complement>
  <new-constraint cident="c1" chrono="4" time="0" >
     <variables> v4 v5 </variables>
  </new-constraint>
</complement>
```

is equivalent to

```
<new-constraint chrono="4" depth="1" time="0" >
   <variables> v1 v2 v3 v4 v5 </variables>
</new-constraint>
```

4.5 Stream Control packet breakpoint

For several reasons it may be useful to split the flow of the trace into smaller pieces. The `<packet>` element may be used for such purpose. A control attribute allows to specify the status of each packet. Recommended but not limited values are breakpoint, continue, stable. If the value is breakpoint, the tracer will stop at the end of the packet and wait for some request from the debugging tool. If the value is continue it means that only a portion of a trace event is contained in the packet. If the value is stable it means that at the two solver states, at the beginning of the packet and reached at the end, correspond to stable meaningful states from the point of view of debugging: all needed information has been transmitted and the tool may operate safely.

To allow interactions between the solver (application) and the debugging tool, the solver must indicate to the tool that it is in a synchrone mode, waiting for some signal from the tool. The `<breakpoint>` element (cf. Sec. 3.5.2) allows the solver to tell the debugging tool that it is such a mode. It has also a control attribute to indicate the kind of synchronization which is requested. By default (no attribute) it is just a break-point.

4.5.1 XML Declaration of the packet and breakpoint Elements

```
<!ELEMENT packet %Toplevel;>
<!ATTLIST packet
   control  CDATA #IMPLIED >

<!ELEMENT breakpoint EMPTY>
<!ATTLIST breakpoint
   control  CDATA #IMPLIED >
```

4.5.2 Examples

both following elements are semantically equivalent.

```
<packet control="breakpoint" />

<breakpoint />
```

Other example:

```
<packet control="continue" >
   <post chrono="6" depth="5" cident="c2" />
   <choice-point chrono="7" depth="6" nident="n6" />
   <back-to chrono="8" depth="5"  node="n5" node-before="n6" />
</packet>
<packet control= "bearkpoint" >
   <failure chrono="11" depth="6" nident="1" />
   <solution chrono="12" depth="1" nident="6" />
</packet>
```

Chapter 5

Constraints Module: Common Attributes and Control

5.1 Common Event, Variable and Constraint Attributes

Each execution event has a set of attached attributes. Some attributes are common to all the ports, some are exclusively related to variables or constraints. An XML entity is used to denote a set of event attributes. These entities are XML attributes of the XML elements representing the ports. The following entities are %eventAttributes; for the common event attributes, and %constraintAttributes; and %variableAttributes; for the common attributes related to constraints and variables respectively.

The common attributes denoted in the DTD by the entity %eventAttributes; are:

chrono `chrono` (req) trace event number (a unique identifier of the event in this trace): a non negative integer. This number indicates the order in which the trace events are generated: it is increasing but not necessarily consecutive (the interval between two consecutive event numbers[1] may by more than one) ;

depth `depth`(opt) depth of th current node in the search-tree, a non negative integer ;

time `time`(opt) the absolute event time: a non negative integer corresponding to the time of the event (microsecond, start time is time zero) ;

context `context`(opt) a chain describing the context, a useful indication for some tools (example in CLP[2]: the calling predication) ;

line `line`(opt) the number of the concerned source code line ;

file `file`(opt) the file name of the file containing the source code line.

The common attributes related to constraints denoted in the DTD by the entity %constraintAttributes; are :

cident `cident` (req) a unique identifier of the constraint in this trace: a tracer dependent string[3] ;

cinternal `cinternal` (opt) an internal representation of the constraint: a tracer defined string ;

cname `cname` (opt) a user representation of the constraint: a string ; this representation may be a short name recognized and used by a debugging tool user ;

[1] Non consecutive, since there may be additional solver specific events or mute parts of the trace.

[2] CLP: Constraint Logic Programming.

[3] If an integer is used as unique identifier, this can be documented in the tracer documentation.

cexternal `cexternal` (opt) the system external representation of the constraint : a string.

The common attributes related to variables denoted in the DTD by the entity %variableAttributes; are :

vident `vident` (req) a unique identifier of the variable in this trace: a tracer dependent string[4] ;

vinternal `vinternal` (opt) the system dependent internal representation of the variable: a string ;

vname `vname` (opt) a user representation of the variable: a string. This attribute can be used to communicate used defined variable names to debugging tools ;

vexternal `vexternal` (opt) the system external representation of the variable: a string.

The `vname` (resp. `cname`) is a name given by the application programmer to a variable (resp. to a constraint)[5]. The `vinternal` (resp. `cinternal`) is a solver dependent representation of the variable (resp. constraint), as it is manipulated by the system when the trace event is generated. It is recommended that the variables of the `cinternal` constraint use the declared identifiers of the variables. This allows to relate variables and arguments of a constraint. The `vexternal` (resp. `cexternal`) is the representation of the variable (resp. constraint) as it is represented at the programmer's level. For example an "element" constraint will be represented by the propagation tracer of GNU-Prolog in the `cinternal` attribute as `fd_element(v3, [1,2,5,7], v4)` where `v3` and `v4` are the variables identifiers. Thus it can be displayed by a tool using the corresponding `vname` instead.

To clarify the DTD, the entity %integer; is used to declare XML attributes wit integer values.

5.1.1 XML Declaration of eventAttributes, constraintAttributes, variableAttributes and integer

```
<!ENTITY % eventAttributes
  "chrono %integer; #REQUIRED
   depth %integer; #IMPLIED
   time  %integer; #IMPLIED
   context CDATA #IMPLIED
   line %integer; #IMPLIED
   file CDATA #IMPLIED" >

<!ENTITY % constraintAttributes
  "cident CDATA #REQUIRED
   cinternal CDATA #IMPLIED
   cname CDATA #IMPLIED
   cexternal CDATA #IMPLIED" >

<!ENTITY % variableAttributes
  "vident CDATA #REQUIRED
   vinternal CDATA #IMPLIED
   vname CDATA #IMPLIED
   vexternal CDATA #IMPLIED" >

<!ENTITY % integer "CDATA" >
```

[4] If an integer is used as unique identifier, this can be documented in the tracer documentation.

[5] This implies that there is a way in the host language to communicate to the tracer a name given to a constraint or a variable.

5.2 Common State Element

The `<state>` element describes the current state (the state reached after the last trace event and likely displayed with the event). It is a common attribute of each trace event, but it is represented by an XML element and therefore it is not listed in the %eventAttributes;.

The `<state>` element may have the following attributes corresponding to solver and search-tree states :

chrono `chrono` (opt) current (the last one) trace event number : a non negative integer ;

depth `depth` (opt) current depth in the search-tree (the depth of the current node), a non negative integer ;

time `time` (opt) the absolute event time: a non negative integer corresponding to the time at the moment of this event (microsecond, start time is zero time) ;

context `context` (opt) a chain describing the context ;

line `line` (opt) the number of the concerned source code line ;

file `file` (opt) the file name of the file containing the source code line.

current-node `current-node` (opt) the number or the identifier referencing the current node ; This identifier has been created by one of the events choice point, failure or solution under the name `nident` in the corresponding event ;

nname `nname` (opt) the name of the current node ;

status `status` (opt) the status of the current node; a node may have the following status (values of the attribute `status`): choice-point, failure, success, or another value which is tracer defined.

choice-constraint `choice-constraint` (opt) the constraints (usually a variable assignment) which are the main causes for the development of this branch since the parent node ; it should be the concatenation of all the constraints declared with the `<choice-constraint>` element of `<new-child>`, `<solution>` or `<failure>`.

next-node `next-node` (opt) a number or an identifier referencing the next node (declared in a future choice-point, if any). This information may be useful to know the next node to which all posted constraint will be attached ;

The `<state>` element may have the following elements as contents :

a sequence of `<constraint>`s the possibly empty set of constraints with their status (see below) ;

a sequence of `<variable>`s the possibly empty set of the variables with their domains ;

a sequence of `<update>`s the possibly empty set of currently sleeping or active solver events ;

`<misc>` (opt) some tracer defined information.

A `<constraint>` element has all the attributes declared with the `<new-constraint>` element except the %eventAttributes;, plus an optional `status` attribute whose recommended values are: active, suspended, rejected, solved, undefined. The last value means that the constraint is not in the store (it has been just declared or removed from the store). Any additional value shall be tracer defined.

A `<constraint>` element has one `<variables>` `<variables>` element and possibly several `<update>` elements as optional contents. The `<variables>` element is intended to indicate the set of the variable identifiers of all the variables of the constraints. This is useful since the syntax of the constraints is not specified in this document. It is recommended to put the variables identifiers consecutively as the values in the `<values>` element. The order is tracer dependent. The `<update>` elements are described below. `<update>` They correspond to the solver events likely generated by a constraint (they may have been declared in the `<new-constraint>` element). The optional attribute `status` is not used here.

A <variable> element has the common variable attributes %variableAttributes; and the type of the variable, as declared in the <new-variable> element. Its content is its current domain <vardomain> <vardomain> which is described in Sec. 5.3.1.

Notice that there is a difference between the absence of content (no content) and an empty domain (represented by the content <values>). This allows for example, to distinguish in a <provide> element whether the domain is provided (empty domain) or not (no content).

The <variable> element has also an optional attribute type as declared in the <new-variable> element.

An <update> element represents one or more solver events. It has no content and three attributes:

vident (req) a variable identifier.

types types (opt) the solver events qualified by their type. The types attribute may contain several values (as many as the number of solver events). They are described in Sec. 6.1.

status status (opt) it is intended, according to the generic model, to reflect the status of the update (active or sleeping). Its recommended values are: active, sleeping. Any additional value shall be tracer defined.

The contents of the <misc> is tracer defined and depends on the running application.

5.2.1 XML Declaration of the Elements: state, constraint, variables, update and misc

```
<!ELEMENT state ((constraint)*, (variable)*, (update)*, misc?) >

<!ATTLIST state
   chrono %integer; #IMPLIED
   depth %integer; #IMPLIED
   time  %integer; #IMPLIED
   context CDATA #IMPLIED
   line CDATA #IMPLIED
   file CDATA #IMPLIED
   current-node CDATA #IMPLIED
   nname CDATA #IMPLIED
   status CDATA #IMPLIED
   choice-constraint CDATA #IMPLIED
   next-node CDATA #IMPLIED >

<!ELEMENT constraint ( variables? , (update)* ) >
<!ATTLIST constraint
  %constraintAttributes;
  orig CDATA #IMPLIED
  status CDATA #IMPLIED >

<!ELEMENT variables (#PCDATA) >

<!ELEMENT variable  ( vardomain? ) >
<!ATTLIST variable
  %variableAttributes;
  type CDATA #IMPLIED >

<!ELEMENT update EMPTY >
<!ATTLIST update
  vident CDATA #REQUIRED
  types CDATA #IMPLIED
  status CDATA #IMPLIED >
```

```
<!ELEMENT misc (#PCDATA) >
```

5.2.2 Example

Considering the following GNU-prolog program (with previous declarations of two stages: constraint posting and labelling):

```
fd_domain(X,1,10), X#<5, X#>2, fd_labeling(X).
```

the following state is reached after the resolution of the two first constraints and the posting of the third one (before its reduce action).

```
<state chrono="100" depth="10" time="3333" current-node="n25" next-node="n26"
        choice-constraint="v1=3">
  <constraint
    cident="c1" cinternal="$fd_domain$(_#0(1..4),1,10)"
    cname="fd1" cexternal="fd_domain(v1,1,10)" orig="user"
    status="solved" > <variables> v1 </variables>
  </constraint>
  <constraint
    cident="c2" cinternal="_#0(0..4)&lt;5"
    cname="inequal1" cexternal="v1#&lt;5" orig="user"
    status="solved" > <variables> v1 </variables>
  </constraint>
  <constraint
    cident="c3" cinternal="_#0(0..4)&gt;2"
    cname="inequal2" cexternal="v1#&gt;2" orig="user"
    status="active" > <variables> v1 </variables>
  </constraint>
  <variable vident="v1" vinternal="_#0(0..4)" vname="var1" vexternal="X"
            type="int">
    <vardomain> <range from="0" to="4"/> </vardomain>
  </variable>
  <update vident="v1" types="max any" status="sleeping"/>
  <misc> last constraint with effect &quot;c2&quot;
    declared stages &quot;constraints posting&quot;  &quot;fdlab&quot;
    current stages &quot;constraints posting&quot;
  </misc>
</state>
```

5.3 new-variable

When a variable is used in a constraint it must have been previously declared by a `<new-variable>` element in the trace which gives it a unique identifier. Such an identifier will be used in the sequel to refer to this variable when it is useful.

The attributes are the common event attributes %eventAttributes;, the common variable attributes %variableAttributes; (see Sec. 5.1) and a type.

type `type` (opt) the type of a variable may be an integer int (default value, also used for finite domain variables represented by integer lists and/or ranges), a real real, enum for an enumeration of different values, or a character string string. No other type is specified at the moment.

The content of the element is a variable domain `<vardomain>`, and an optional state.

Notice that technically, the description allows variable domains which consist in a finite sequence of enumerated values (e.g. colors such as "red green blue"). Yet, the DTD has neither been evaluated nor thought with these domain types in mind. It is thus suggested not to provide enumerated domains, or at least not count on all tools to behave correctly when reading a trace which contains such domains.

5.3.1 Elements vardomain, values and range

The <vardomain> element has three attributes

min min (opt) the lower bound of the variable domain ;

max max (opt) the upper bound of the variable domain ;

size size (opt) the size of the domain (number of elements in the domain) ;

Its content is the variable domain, specified by the XML entity %valueList;.

Variable domain values may be described by extension or by intervals. Then domain values are the union of enumerated values (<values>) and/or ranges (<range>) expressed by their lower (from) and upper bounds (to). If there is a mixture of <values> and <range> elements in the description of the domain, it is recommended to follow the increasing order of the values inside each element and between the sets of values defined in the elements. This order between elements is however tracer defined.

An empty domain is denoted by <values/> in %valueList;.

Notice that the domains of variables which occur in the elements <new-variable>, <explanation>, <reduce>, <delta>, and <restore-delta> are normally nonempty since, when they are introduced, they denote non empty domain definition (<new-variable>), domain portion (<explanation>), or domain modification (<reduce>, <delta>, <restore-delta>). However in the <state> element the declared state of the domain of some variables may be empty (for example after a constraint has been rejected).

5.3.2 XML Declaration of the Elements: new-variable

```
<!ELEMENT new-variable ( vardomain? , state?) >
<!ATTLIST new-variable
  %eventAttributes;
  %variableAttributes;
  type CDATA #IMPLIED >

<!ELEMENT vardomain %valueList; >
<!ATTLIST vardomain
  min %integer; #IMPLIED
  max %integer; #IMPLIED
  size %integer; #IMPLIED >

<!ENTITY % valueList "( values | range )*" >

<!ELEMENT values (#PCDATA) >

<!ELEMENT range EMPTY>
<!ATTLIST range
  from CDATA #REQUIRED
  to CDATA #REQUIRED >
```

5.3.3 Example

Referring to program of Ex. 5.2.2

```
<new-variable chrono="5" depth="1" time="0" context="fd_domain(_#0(1..10),1,10)"
      line="3" file="example.pl"  vident="v1" vinternal="_#0(1..10)"
      vname="var1" vexternal="X" type="int">
    <vardomain min="1" max="10" size="10"> <range from="1" to="10"/>
    </vardomain>
</new-variable>
```

or equivalently

```
<new-variable chrono="5" depth="1" time="0" context="fd_domain(_#0(1..10),1,10)"
           line="3" file="example.pl"  vident="v1" vinternal="_#0(1..10)"
             vname="var1" vexternal="X" type="int">
   <vardomain min="1" max="10" size="10">
      <values>1 2 3 4 5 6 7 8 9 10 </values>
   </vardomain>
</new-variable>
```

5.4 new-constraint

<new-constraint>

Before any use of a constraint in some event, it must be declared in the trace by a new-constraint event, represented by a <new-constraint> element, which gives it a unique identifier. Such an identifier will be used in the sequel to refer to this constraint when it is requested.

The attributes are the common event attributes %eventAttributes;, the common attributes of a constraint %constraintAttributes; (see Sec. 5.1) and an origin `orig`.

orig `orig` (opt) specifies whether the constraint has been posted by the user's program of by the system (for example during a predefined labelling phase). The recommended values are user (constraint occurring in the user's program) and system (other constraints).

The content of the element is a list of solver events which could contribute to the solver awakening conditions, an optional list of <variables> (described in Sec. 5.2), and an optional state.

The solver events (<update> elements) are optional since in some solvers they may not be known at the time of this constraint declaration. They are described by a list of <update> elements defined in Sec. 3.1.3 (semantics) and in Sec. 6.1 (syntax), and are tracer defined. They have only a variable identifier and types (one or more if grouped), characterizing the type of variable domain update.

Remarks:

The solver events declared here are the events which can be generated by an active constraint. There are just declared here and there is no guarantee that all will be used during resolution (e.g. a constraint may never be activated).

The identifiers of the constraint variables may be found in the <state> constraint content (<variables> element).

5.4.1 XML Declaration of the new-constraint Element

```
<!ELEMENT new-constraint ( variables?, (update)*, state?) >
<!ATTLIST new-constraint
  %eventAttributes;
  %constraintAttributes;
  orig CDATA #IMPLIED>
```

5.4.2 Example

Referring to program of Ex. 5.2.2. In this example the <state> is given to get the constraint variables.

```
<new-constraint chrono="4" depth="1" time="0" context="$toplevel$" line="3"
      file="example.pl" cident="c1" cinternal="fd_domain(v1,1,10)"
      cname="fd1" cexternal="fd_domain(_#0,1,10)" orig="user">
  <update vident="v1" types="min max"/>
  <state>
    <constraint cident="c1" >
```

```
      <variables> v1 </variables>
    </constraint>
  </state>
</new-constraint>
```

5.5 post

The introduction of a constraint in the store generates a post event, represented by a <post> element. The attributes are the common event attributes and the common constraint attributes (the required identifier is a reference to a predeclared constraint) and the content is an optional state.

5.5.1 XML Declaration of the post Element

```
<!ELEMENT post (state?) >
<!ATTLIST post
  %eventAttributes;
  cident CDATA #REQUIRED>
```

5.5.2 Example

Referring to program of Ex. 5.2.2

```
<post chrono="6" depth="5" time="1" context="$toplevel$" cident="c2" />
```

5.6 choice-point

When a new choice point node is created in the search-tree or when a choice point is declared (some solvers do not make explicit construction of search-trees) a choice-point event is generated, represented by a <choice-point> element.

The attributes are:

The common %eventAttributes; ;

nident nident (opt) if a new node is explicitly created, a unique node identifier (such an identifier will be used in the sequel to refer to this node when it is requested): a tracer dependent string[6]. Notice that the identifier is optional, as all its references in this DTD, but a tracer must be able to provide it on request ;

nname nname (opt) a user node name: a string. This attribute can be used to communicate used defined node names to debugging tools ;

The contents are a possibly empty set of <choice-constraint> elements and an optional state.

A <choice-constraint> element describes the constraints (usually a single variable assignment) which is the main cause for the development of this branch since the parent node. In the case of a full trace it is redundant since all constraints are declared in the trace by a <new-constraint> (this applies to the variable assignments made during labelling which are considered in the generic model as constraints). However, in the case of a trace limited to the search tree during a labeling phase, this attribute may be useful: it allows to recall some of or all the used constraints.

The <choice-constraint> element has an empty content and three attributes: vident, value and constraints. The two first correspond to the description of a single variable assignment. The second allow to put several other constraints. The exact form of the constraint(s) named in this attribute is tracer defined.

[6] If an integer is used as unique identifier, this can be documented in the tracer documentation.

The general intention is to allow to declare with a set of <choice-constraint> elements, the set (usually a singleton, in the case of system labelling) of the last variable assignments.

Solvers without backtracking mechanism do not generate <choice-point> event. However they may use such event to indicate the restoration of a consistent state after re-introduction of suppressed values in the domains of some variables. In this case, the <depth> attribute may correspond to the number of active constraint.

Notice that a systematic use of <choice-constraint> to quote constraints without declaring them may results in the impossibility to refer to these constraints in a <state> element.

5.6.1 XML Declaration of the Elements choice-point and choice-constraint

```
<!ELEMENT choice-point ( (choice-constraint)* , state? ) >
<!ATTLIST choice-point
   %eventAttributes;
   nident CDATA #IMPLIED
   nname CDATA #IMPLIED >

<!ELEMENT choice-constraint EMPTY >
<!ATTLIST choice-constraint
    vident CDATA #IMPLIED
    value %integer; #IMPLIED
    constraints CDATA #IMPLIED >
```

5.6.2 Example

Referring to program of Ex. 5.2.2

```
<choice-point chrono="10" depth="6" time="2" context="fd_labeling(v1)" nident="n6" >
   <choice-constraint vident="v1" value="2" />
</choice-point>
```

5.7 back-to

A back-to event corresponds to a jump to some previously constructed choice-point in the search-tree. It is represented by a <back-to> element. It is important to observe that a back to event leaves the solver in a previous consistent state which corresponds to an already traced choice point point, even if there is no explicit tree structured search.

The attributes are the common event attributes and two numbers node and node-before denoting respectively the target and origin nodes. Notice that the node-before attribute is redundant for a tool which memoizes the last visited node in the search-tree.

The contents are two optional lists of domain elements, <delta>* and <removed-values>*, describing the variations of the variables domains, and an optional <state>.

The first list of <delta>s corresponds to the added values in the domains of the variables. The second list of <removed-values>s corresponds to the values which must be supressed from the domains of the variables. Both lists allow together to restore the domains of the variables corresponding to the new solver state. Such domains can be found also in the <state> element (if requested). Notice that the second list is useful only in the case of random visits of the search-tree. In the case of solvers with chronological visits (depth first - left to right) of the serach-tree the first list is sufficient.

The idea is that the (possibly empty) lists of <delta>s and <removed-values> describe the modifications of the variable domains between the original and target states, such that the original state may be retrieved from the target state and vice versa. Therefore to find the original variable domains from the current state, the values of the <delta>s must be supressed and the values of the <removed-values>s must be added[7].

An <delta> element has an attribute (vident) and the %valueList; as content, representing the variation of the domain of the referred variable. Although the vident attribute is optional in the DTD, it is here required. The <delta> element is described in Sec. 6.1.1 (<reduce>).

The <removed-values> element has the same structure as the <delta> element.

5.7.1 XML Declaration of the back-to Element

```
<!ELEMENT back-to ( (delta)*, (removed-values)*, state? ) >
<!ATTLIST back-to
  %eventAttributes;
  node CDATA #IMPLIED
  node-before CDATA #IMPLIED>

<!ELEMENT removed-values %valueList;  >
<!ATTLIST removed-values
  vident CDATA #IMPLIED >
```

5.7.2 Example

Assume that in the state of a node n4 the variable v1 has values 1 2 3 in its domain. The node n4 has two children n5 and n6 at which the variable v1 has respectively the values 2 3 and 1 2 in its domains. Then describing the domain variations of this variable when jumping from node

```
<back-to chrono="12" depth="5" time="2" context="fd_labeling(v1)" node="n5"
         node-before="n6" >
   <delta vident="v1"> <values> 3 </values> </delta>
   <removed-values vident="v1"> <values> 1 </values> </removed-values>
</back-to>
```

5.8 solution

A solution event corresponds to a successful state of computation and likely to the creation of a solution leaf in the search-tree. It is represented by a <solution> element.

The attributes are the common event attributes and, if a new node is explicitly created, three optional attributes:

nident (opt) a unique node identifier (such an identifier will be used in the sequel to refer to this node when it is requested): a tracer dependent string[8]. Notice that the identifier is optional, as all its references in this DTD, but a tracer must be able to provide it on request,

nname (opt) a name , a string denoting in a user understandable manner the node,

val (opt) a number (declared using the %number; entity). This number may be a signed integer, a decimal number or a float. It is a measure of the quality of the solution. For example in the case of an optimization problem it may be the value of the objective function.

The contents are a possibly empty list of <choice-constraint> elements and an optional state.

The <choice-constraint> element contains the constraints (usually limited to one variable assignment) which are the main cause for the development of this branch since the parent node (more details in Sec. 5.6).

[7] Notice that the <delta> element is the same element as for the <reduce> element, but used in the opposit direction (in <reduce> to find the new state instead of the previous one).

[8] If an integer is used as unique identifier, this can be documented in the tracer documentation.

5.8.1 XML Declaration of the solution Element

```
<!ENTITY % number "CDATA" >

<!ELEMENT solution ( (choice-constraint)* , state?) >
<!ATTLIST solution
  %eventAttributes;
  nident CDATA #IMPLIED
  nname CDATA #IMPLIED
  val %number; #IMPLIED>
```

5.8.2 Example

Referring to program of Ex. 5.2.2

```
<solution chrono="11" depth="6" time="2" context="fd_labeling(v1)" nident="6" >
  <choice-constraint vident="v1" value="3" />
  <state>
    <variable vident="v1" >
      <vardomain> <values>3</values></vardomain>
    </variable>
  </state>
</solution>
```

5.9 failure

A failure event corresponds to a failed state of computation and likely corresponds to the creation of a failure leaf in the search-tree. It is represented by a `<failure>` element.

The attributes are the common event attributes and, if a new node is explicitly created, three optional attributes:

nident (opt) a unique node identifier (such an identifier will be used in the sequel to refer to this node when it is requested): a tracer dependent string[9]. Notice that the identifier is optional, as all its references in this DTD, but a tracer must be able to provide it on request,

nname (opt) a name , a string denoting in a user understandable manner the node,

The contents are a possibly empty list of `<choice-constraint>` elements and an optional state.

The `<choice-constraint>` element contains the constraints (usually limited to one variable assignment) which are the main cause for the development of this branch since the parent node (more details in Sec. 5.6.1).

5.9.1 XML Declaration of the failure Element

```
<!ELEMENT failure ( (choice-constraint)* , state?) >
<!ATTLIST failure
  %eventAttributes;
  nident CDATA #IMPLIED
  nname CDATA #IMPLIED >
```

5.9.2 Example

```
<failure chrono="15" depth="1" nident="n1" >
   <choice-constraint vident="v1" value="6" />
</failure>
```

[9] If an integer is used as unique identifier, this can be documented in the tracer documentation.

5.10 remove

The withdrawal of a constraint from the store generates a remove event, represented by a <remove> element. The attributes are the common event attributes and a reference to a constraint (the required identifier is a reference to a predeclared constraint), and the content is an optional state.

5.10.1 XML Declaration of the remove Element

```
<!ELEMENT remove (state?) >
<!ATTLIST remove
  %eventAttributes;
  cident CDATA #REQUIRED>
```

5.10.2 Example

```
<remove chrono="6" depth="5" context="fd_labeling(v1)" cident="c4" />
```

5.11 restore

A restore event indicates that one or several values have been restored into a variable domain. It is represented by a <restore> element.

The attributes are the common event attributes and an optional identifier referencing the concerned variable (there is one event per concerned variable and the identifier of the variable may be given as attribute of <reduce> and/or of <delta>).

The vident attribute of the <restore> and the <delta> element are optional. However there must be at least one provided.

The contents are an optional <delta> element, an optional <vardomain> element as defined in 6.1, an optional <update>, and an optional state. The <delta> describes the modifications of the variable domain (restored variable domain values).

Notice that there is one <delta> element only which describes the restored (i.e. re-introduced) values[10]. The possibility to go back to any kind of domain is not considered.

5.11.1 XML Declaration of the restore Element

```
<!ELEMENT restore ( delta?, vardomain?, update?, state? ) >
<!ATTLIST restore
  %eventAttributes;
  vident CDATA #IMPLIED >
```

5.11.2 Example

```
<restore chrono="6" depth="5" context="fd_labeling(v1)" vident="v1" >
   <delta vident="v1">
      <values>3 4</values>
   </delta>
</restore>
```

[10] Notice that the <delta> element is the same element as for the <reduce> element, but used in the opposit direction (in <reduce> it is used to find the new state instead of the old one).

Chapter 6

Constraints Module: Propagation

6.1 reduce

A reduce event corresponds to the reduction of the domain of a variable by an active constraint. It is represented by a `<reduce>` element.

The attributes are the common event attributes %eventAttributes;, an optional reference `cident` to the active constraint, an optional reference `vident` to the variable whose domain is modified (this attribute is used only if there is no update, in order to keep trace of the concerned variable), and `algo`, an optional indication of the used algorithm which implements the used local reduction operator (only one operator is used likely updating one variable domain only, see Sec. 3.1.2).

The optional contents are:

- the description of the withdrawn values (`<delta>` element).

  The attribute of the `<delta>` element is the reference to the concerned variable `vident` and its content is the description of the set of withdrawn values %valueList;.

  Notice that the `vident` attribute is optional in the DTD, but it is required that it occurs at least once either as attibute of `<reduce>` or of `<delta>`. Therefore if there is no `vident` as attribute of `<delta>` there must be such attribute for `<reduce>` and vice versa.

- The new domain of the variable after the withdrawal (`<vardomain>` element). It is described in Sec. 5.3.1. Notice that the same variable has also the same domain in the state element. This information is given at this level in order to avoid a systematic use of the state element but it may be also easier to implement than the `<delta>`.

- The solver events generated by the local operator application and collected in the `<update>` element with a reference to the variable and the list of types of events.

- a possibly empty set of explanations (`<explanation>`, see 6.1.2 below).

- a state.

The `<update>` element has two attributes: the reference to the variable (req) and a list of types for this solver event (opt). This list, which may describe several types, corresponds to the list of solver events generated by a the local reduction operator of the constraint as defined in Sec. 3.1.3[1]. See Sec 5.2.1 for the syntax of the `<update>` element.

The solver events which are generated are solver dependent. The recommended non exclusive values for the attribute `types` are: ground (fixed value), min (update of the domain lower bound), max (update of the domain upper bound), minmax (update of both bounds), val (a removed value which is not a bound), any (some value removed), empty (domain emptied), nothing (domain not modified).

Notice that if the list of solver events is empty, the current reduction will not influence future events.

[1] There is also a third optional attribute (`status`) used in the `<state>` element description only.

6.1.1 XML Declaration of the Elements: reduce and delta

```
<!ELEMENT reduce ( delta?, vardomain?, update?, explanation*, state? ) >
<!ATTLIST reduce
  %eventAttributes;
  cident CDATA #IMPLIED
  vident CDATA #IMPLIED
  algo CDATA #IMPLIED >

<!ELEMENT delta %valueList;  >
<!ATTLIST delta
  vident CDATA #IMPLIED >
```

6.1.2 Elements : explanation, cause and constraints

Explanations are presented in Sec. 3.5.1. They are represented by the element <explanation>. It is either a list of elements <cause> or a <constraints> element, both giving causes of the variable domain values withdrawal.

The element <cause> describes a variable domain update by a reference to a variable (attribute vident) and a domain (%valueList;) which is the "cause" of the variable domain update of the top-level variable, as explained in Sec. 3.5.1. There is an additional optional ctype attribute intended to inform about the type of the cause (it can be like the types in an <update> an indication of the kind of variable domain update). This may be useful in the context of solvers using repair techniques. It is tracer defined.

The element <constraints> is restricted to its attribute cidents, a string denoting a list of constraint references or a tree structured list of constraint references.

6.1.3 XML Declaration of the explanation Element

```
<!ELEMENT explanation ( %valueList; , (cause)*, constraints?) >

<!ELEMENT cause %valueList; >
<!ATTLIST cause
  vident CDATA #REQUIRED
  ctype CDATA #IMPLIED>

<!ELEMENT constraints EMPTY>
<!ATTLIST constraints
  cidents CDATA #REQUIRED >
```

6.1.4 Example

```
<reduce chrono="10" depth="1" cident="c3" algo="r5">
   <delta vident="v2"> <range from="0" to="5"/><values>7 9</values>
   </delta>
   <vardomain min="6" max="10" size="3">
      <values> 6 8 10 </values>
   </vardomain>
   <update vident="v2" types="min any" />
<!-- explanation of the withdrawn values 0,1,2,3-->
   <explanation>
      <range from="0" to="3"/>
      <cause vident="v1">
         <values>0 1 2</values>
      </cause>
```

```
      <cause vident="v2">
         <values>1 2</values>
      </cause>
      <constraints cidents="c1 c2"/>
   </explanation>
<!-- explanation of the withdrawn values 7, 9-->
   <explanation>
      <values> 7 9 </values>
      <cause vident="v1">
         <values>2</values>
      </cause>
      <cause vident="v2">
         <values>10</values>
      </cause>
      <constraints cidents="c3"/>
   </explanation>
</reduce>
```

6.2 suspend

The suspend event indicates the suspension of an active constraint. It is represented by a `<suspend>` element. The attributes are the common event attributes and a reference to a constraint (the required identifier is a reference to a predeclared constraint), and the content is an optional state.

6.2.1 XML Declaration of the suspend Element

```
<!ELEMENT suspend (state?) >
<!ATTLIST suspend
  %eventAttributes;
  cident CDATA #REQUIRED>
```

6.2.2 Example

```
<suspend chrono="20" depth="1" cident="c3"/>
```

6.3 solved

The solved event indicates that a constraint is true (any assignment of remaining variables satisfies the constraint) and does not influence any more the resolution. It is represented by the `<solved>` element. The attributes are the common event attributes and a reference to a constraint (the required identifier is a reference to a predeclared constraint), and the content is an optional state.

Notice that the solvers which do not perform dynamic "entailment" will not be able to generate such event.

6.3.1 XML Declaration of the solved Element

```
<!ELEMENT solved (state?) >
<!ATTLIST solved
  %eventAttributes;
  cident CDATA #REQUIRED>
```

6.3.2 Example

```
<solved chrono="40" depth="1" cident="c2">
    <state>
        <constraint cident="c1" status="active"/>
        <constraint cident="c2" status="solved"/>
        <constraint cident="c3" status="sleeping"/>
    </state>
</solved>
```

6.4 reject

The reject event indicates that a constraint is unsatisfiable. It is represented by the `<reject>` element. The conditions for rejection are verified after a reduce event. Usually this will immediately lead to a failed state and generate a failure event. The attributes are the common event attributes and a reference to a constraint (the required identifier is a reference to a predeclared constraint), and the content is an optional state.

6.4.1 XML Declaration of the reject Element

```
<!ELEMENT reject ( state? ) >
<!ATTLIST reject
  %eventAttributes;
  cident CDATA #REQUIRED>
```

6.4.2 Example

```
<reject chrono="50" cident="c3">
   <state>
      <constraint cident="c1" status="sleeping"/>
      <constraint cident="c2" status="solved"/>
      <constraint cident="c3" status="rejected"/>
   </state>
</reject>
```

6.5 awake

The awake event indicates that a sleeping constraint becomes active (one of the awakening condition is verified). It is represented by the `<awake>` element.

The attributes are the common event attributes and a reference to the woken constraint (the required identifier is a reference to a predeclared constraint).

The contents are a solver event (`<update>` element defined in 6.1) responsible for the awakening, and an optional state. The `<update>` element indicates which solver event is concerned. The form of the `<update>` element is tracer defined, but the choice of the solver event is solver dependent.

6.5.1 XML Declaration of the awake Element

```
<!ELEMENT awake (update?, state?) >
<!ATTLIST awake
  %eventAttributes;
  cident CDATA #REQUIRED>
```

6.5.2 Example

```
<awake chrono="30" cident="c1">
   <update vident="v2" types="min"/>
   <state>
      <update vident="v2" types="min" status="active" />
   </state>
</awake>
```

6.6 schedule

The schedule event indicates a re-organization of the sleeping constraints and solver events which is solver dependent (it depends on the algorithm used for the resolution). The generic trace model imposes only to indicate which constraint and/or which solver event is concerned. It is however possible to indicate some tracer defined scheduling actions which are performed. It is represented by the `<schedule>` element.

The attributes are the common %eventAttributes;, a reference to the concerned constraint (it is a reference to a predeclared constraint) and a list of tracer defined `actions`.

The content is the concerned solver event, described by an `<update>` element (Sec. 6.1), and an optional state.

The `<update>` element indicates the solver event which is concerned. The form of the `<update>` element is tracer defined, but the choice of the solver event is solver dependent.

6.6.1 XML Declaration of the schedule Element

```
<!ELEMENT schedule ( update?, state?) >
<!ATTLIST schedule
  %eventAttributes;
  cident CDATA #IMPLIED
  actions CDATA #IMPLIED>
```

6.6.2 Example

```
<schedule chrono="60" cident="c3" actions="gprolog_last-in-queue" >
   <update vident="v2" types="any"/>
   <state>
      <update vident="v2" types="any" status="sleeping" />
   </state>
</schedule>
```

Chapter 7

Externals Module

This chapter describes the trace events which can be used to communicate some information to the debugging tool.

The qualification "externals" means that such events are issued by the application rather than from the solver. It is assumed that the host language of the solver allows some communication between the application and the tracer.

7.1 annotation

The `<annotation>` trace event allows to communicate an application dependent text to the debugging tool as a piece of text (PCDATA). This offers the possibility of solver-to-tool communication or to define semantical objects, or also to relate low level objects to higher level ones. This may be particularly useful to handle global constraints. It is represented by the `<annotation>` element. The content of the `<acmd>` element is a text (a command or a description of the nature of the object); and an optional state. The attributes are the common event attributes %eventAttributes;, and:

aident `aident` (req) a unique identifier of this annotation in this trace: a tracer dependent string[1]. In the case the annotation contains the description of a new semantical object, this identifier will be used as reference;

type `type` (opt) the type of the annotation characterizing the nature of the string (tool command, display specification, semantical object, ...). Recommended but non exclusive values are: cmd (command) and obj (semantical object);

aname `aname` (opt) In the case the annotation contains the description of a new semantical object, the denotation of this object;

refs `refs` (opt) In the case the annotation contains the description of a new semantical object, the references to other objects used in its description (references to variables, constraints or other semantical objects).

7.1.1 XML Declaration of the annotation Element

```
<!ELEMENT annotation ( acmd, state?) >
<!ATTLIST annotation
  %eventAttributes;
  aident CDATA #REQUIRED
  type CDATA #IMPLIED
  aname CDATA #IMPLIED
  refs CDATA #IMPLIED >
```

[1] If an integer is used as unique identifier, this can be documented in the tracer documentation.

```
<!ELEMENT acmd (#PCDATA) >
```

7.1.2 Example

```
<annotation chrono="100" type="obj" aident="T11" aname="Task 1"
            refs="S11 D11 M11">
   <acmd> task </acmd>
</annotation>
<annotation chrono="200" type="obj" aident="M1" aname="Machine 1"
            refs="T11 T12 T13 T14">
   <acmd> machine </acmd>
</annotation>
<annotation chrono="300" type="cmd" aident="A20">
   <acmd> display what you want </acmd>
</annotation>
```

7.2 new-stage

The <new-stage> trace event declares a new step or a new phase in the application process which will be later referenced. A phase may rely on very different actions like labelling or loading a file. It may be also useful to make apparent some significant steps in the algorithms used for global constraints. It is represented by the <new-stage> element.

The content of the <new-stage> element is an optional comment <scomm> and an optional state.

Its attributes are the common event attributes %eventAttributes; and the common stage attributes %stageAttributes;

The common attributes related to stages are denoted in the DTD by the entity %stageAttributes; :

sident sident (req) a unique identifier which will be used to refer to the stage during its life cycle: a tracer dependent string[2] ;

sname sname (opt) a step or phase user name ;

refs refs (opt) a sequence of references to related entities (objects or other phases) ;

detail detail (opt) some information about this particular phase (if the phase is a file loading, then the name of the file may be indicated). The nature of the detail may be tracer defined.

7.2.1 XML Declaration of the new-stage Element

```
<!ENTITY % stageAttributes
  "sident CDATA #REQUIRED
   sname CDATA #IMPLIED
   refs CDATA #IMPLIED
   detail CDATA #IMPLIED" >

<!ELEMENT new-stage (scomm?, state?) >
<!ATTLIST new-stage
  %eventAttributes;
  %stageAttributes;>

<!ELEMENT scomm (#PCDATA) >
```

[2] If an integer is used as unique identifier, this can be documented in the tracer documentation.

7.2.2 Example

```
<new-stage chrono="100" sident="s1" sname="loading" detail="essai.pl">
    <scomm> loading file essai.pl</scomm>
</new-stage>
<new-stage chrono="1234" sident="s2" sname="labeling" refs="o1 o2 o3"
           detail="fd_labeling([MP,PM,PA,AM,AP,MA])">
    <scomm> labeling of refered objects </scomm>
</new-stage>
```

7.3 start-stage/suspend-stage/resume-stage/stop-stage

The following trace events start-stage, suspend-stage, resume-stage and stop-stage allow to describe the status of a declared stage (resp. started, suspended, resumed or terminated). They allow to preserve the integrity of the XML syntax in the case of embedded phases or non contiguous ones.

They are represented by the elements `<start-stage>`, `<suspend-stage>`, `<resume-stage>` and `<stop-stage>`.

Their content is an optional state.

Their attributes are the common event attributes %eventAttributes; and a unique attribute `sident` (req) which refers to a previously declared stage.

7.3.1 XML Declaration of the Elements: start-stage suspend-stage resume-stage and stop-stage

```
<!ELEMENT start-stage ( state? ) >
<!ATTLIST start-stage
  %eventAttributes;
  sident CDATA #REQUIRED >

<!ELEMENT suspend-stage ( state? ) >
<!ATTLIST suspend-stage
  %eventAttributes;
  sident CDATA #REQUIRED >

<!ELEMENT resume-stage ( state? ) >
<!ATTLIST resume-stage
  %eventAttributes;
  sident CDATA #REQUIRED >

<!ELEMENT stop-stage ( state? ) >
<!ATTLIST stop-stage
  %eventAttributes;
  sident CDATA #REQUIRED >
```

7.3.2 Example

```
<new-stage chrono="100" sident="s1" sname="loading" detail="essai.pl">
    <scomm> loading file essai.pl</scomm>
</new-stage>
<!-- ... -->
<start-stage chrono="100" sident="s1" />
<new-variable chrono="100" vident="v1">
   <vardomain min="1" max="10" size="10"> <values>1 2 3</values>
   </vardomain>
</new-variable>
```

```
<!-- ... -->
<new-constraint chrono="100" cident="c1" >
  <update vident="v1" types="ground"/>
  <update vident="v2" types="ground"/>
</new-constraint>
<!-- ... -->
<stop-stage chrono="100" sident="s1" />
```

Chapter 8

Tracer-Tool Interaction Schema

This chapter addresses the problem of the interactions between tracers and debugging tools. It concerns the generic trace and the trace requests as stated in the introduction (Chap. 1). It is not normative, but it clarifies the way tracer and tool may be synchronized and parametrized in order to exchange information. It completes Sec. 3.5.2 of Chap. 3 on semantics and clarifies the role of the elements `<breakpoint>`, `<provide>` and `<complement>`.

8.1 Tracer-Tool Architecture

The communication between tracer and tool corresponds to a client/server architecture: the client is the tool and the server is the tracer. The tool controls the synchronization process; it means that the tool sends the synchronization commands.

There may be various ways to synchronize both processes. One possibility is to split the trace using the `<packet>` element to define breakpoints in a systematic way (using the breakpoint control attribute value), giving the tool the ability to control frequently the trace flow. Another rather general possibility is to interrupt the flow when a particular state is reached.

This state can be identified by analyzing the trace itself, using some conditions computed on the fly. For example a condition may be that the “chrono” reached a fixed number, that the number of nodes of the search-tree reached a fixed maximum number, or that the resolution reached a solution from which some user must decide how to resume (e.g. removing or adding manually some constraints in the store). A more complex condition may be that a visualization process collected enough data to start a relatively slow drawing, and requires the tracer (and the solver) to wait a moment, likely to wait for a decision how to resume. The conditions may be arbitrarily complex; the sole limitation comes from the expressiveness of the trace and the tracer performance. In practice only efficiently computable conditions can be used in order no to slow down excessively the tracer.

It is not the purpose of this document to define how to specify such conditions, nor how to implement them (a proposal based on sets of event patterns can be found in [15]). We just assume that the tracer has a capability to handle conditions, to interpret commands from the tool and to generate the requested trace until the next breakpoint. `breakpoint`

The Fig. 8.1 illustrates this architecture.

The *tracer driver* has several functions: it must be able to set and update the *parameters of the trace* on request. These parameters are represented by a *synchronization and parameters* box; they concern the synchronization conditions and define the level of detail of the trace as requested by the tool. The driver generates the resulting trace encoded in the XML format.

The debugging tool has a *trace analyzer and filter* which allows it to select in the broadcasted trace the events and attributes of interest in the case the trace is too large[1].

[1] A larger trace may be necessary in the case several tools are used in parallel.

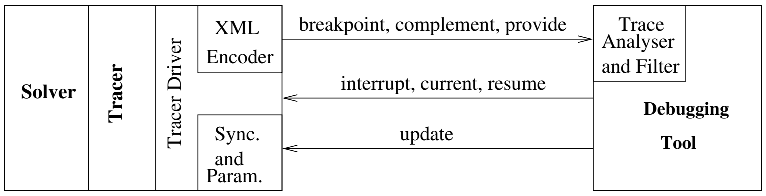

Figure 8.1: Tool/Tracer Architecture and Command Schemata

Notice that it should be possible also to stop the trace flow at any time (for example an unexpected endless propagation step, a too wide search-space, or a too large figure to draw). In this case, before resuming the resolution, the tool may require some information about the current solver state, in order to start again itself in a consistent way.

8.2 Tracer-Tool Interactions and Synchronization

A tracer likely generates a *default trace*. This means that initially there are default trace parameters used by the tracer driver.

However it should be possible to configure on demand a tracer, i.e. to update on demand the trace parameters. These parameters may be given by the programmer when launching the tracer, or provided by some interactive debugging tool. One of the objective of the generic trace is to allow partial automatisation of such interaction: the tool may be able to adjust automatically the right level of trace needed for its execution. This can be done by a negotiation process like the following one:

1. the tracer gives its *maximal trace* in the prologue (`<provide>` element in the `<header>` element) and declares the current level of detail of the trace in the top-level `<provide>` element, then it stops and waits for autorisation to resume (`<breakpoint>`).

2. If the tool is not satisfied with the maximal trace then the trace cannot be handled and the session probably stops. Otherwise if the level of detail of the trace is insufficient or too verbose, the tool may send a request, in order to get the right elements and to optimize the input flow of the trace. In turn a (`<provide>`) element in the trace may inform the tool about the new level of detail of the trace.

3. After the receipt of the autorisation, the tracer resumes and proceeds with the new parameters according to the new configuration.

During this process, the tracer and the tool are synchronized. The synchronization ends with the acknowledgement of the tool. From this moment both processes are asynchronous: the trace flow may be broadcasted according to the trace parameters and the tool uses it on the fly.

As we have seen, as the trace may be modified when some particular state is reached, there is two kinds of parameters: conditions to fix the next breakpoints (*synchronization parameters*) and conditions to fix the level of detail of the trace (*trace level parameters*).

- *synchronization parameters*: they specify that, at specified trace events, the execution is frozen until the tool orders the execution to resume. For example in the dialog above, the tool launches the tracer by specifying that the tracer must stop after the `<provide>` element in the prologue and after the top-level `<provide>` element.

- *trace level parameters*: they specify that, at specified trace events, some trace data are to be sent to the tool without freezing the execution. For example, in the dialog above, the tool starts the uninterrupted flow of the trace after sending to the driver a condition describing “any” trace event and likely a condition identifying the next breakpoint (in the absence of synchronization parameters, the tracer will continue until the end of the execution).

8.3 Command Schemata

All the commands sent to the tracer driver by a tool for trace parameterization correspond to the trace requests. It is not the purpose of this document to specify a language for such requests; therefore we only comment the needed commands in the form of command schemata without specifying any syntax. There are four general commands illustrated on Fig. 8.1. They can be used when the tracer is stopped, waiting for instructions.

- **`current`** This is a request to the tracer for particular additional information. Arguments may have the same form as a `<provide>` element as defined in Chap. 4. In turn, an answer is expected which is a `<complement>` element.
- **`resume`** The tool asks the solver and the tracer to restart. Possible argument is an additional synchronization parameter ("resume until condition").
- **`update`** This updates the trace parameters (synchronization and level of the trace) defining a new level of detail of the trace and the form of the next events where to break execution. Arguments may have the style of contents of the `<provide>` element with boolean combinations of values for attributes. No direct answer is expected.
- **`interrupt`** This interrupts suddenly the solver, the tracer and the generic trace flow and possibly leaves the tool in an inconsistent state. When the execution resumes, the tool may need some information to restore a consistent state which can be obtained using `current`.

In order to illustrate this approach by some example dialogs, we remind here the commands for interaction included in the generic trace are (see Chap. 4):

- **`<breakpoint>`** The tracer indicates that it is synchronized, waiting for some command.
- **`<provide>`** The tracer indicates the new level of detail of the trace.
- **`<complement>`** The tracer sends on request (using `current`) detailed information on the last trace event or the current state.

The use of these command schemata is illustrated by the following dialog. It starts in a synchronous mode.

```
sync   <-- update               specifies new breakpoints and new trace format
       --> provide              sends the new profile of the trace
       <-- resume               the new trace is broadcasted
async                           until
       --> breakpoint           next breakpoint reached
sync   <-- current              information requested
       --> complement           information sent
       <-- update               new specification of breakpoints and trace
       <-- resume               ... until next specified breakpoint
       ...
```

Chapter 9

Compliant Tracer and Tool

9.1 Compliant Tracer

Every tracer providing a compliant trace with all the trace events compatible with the corresponding solver and the meta-data as described in Chap. 4 is compliant.

A compliant trace is a trace which contains a subset of the elements described in this document such that:

1. the syntax follows the XML DTD described in the annex A,
2. the semantics, as described in this document, is respected.

9.2 Compliant Tool

A compliant debugging tool shall

1. accept any compliant trace, ignoring ports and attributes of the generic trace which it is not able to interpret,
2. be able to address trace requests
3. to synchronize according to Chap. 4.

9.3 Compliant Extension of the trace

The generic trace format may be extended.

An extension of the generic trace format is compliant if

1. the generic sub-trace it contains is a compliant trace (same syntax and semantics), and
2. it uses the `Namespace` convention to address the specific extensions.

Bibliography

[1] Krzysztof R. Apt. *Principles of Constraints Programming: An Introduction*. Cambridge University Press, 2003.

[2] T. Baudel and et al. DISCOVERY reference manual, 2003. Manufactured and distributed by Ilog, `http://www2.ilog.com/preview/Discovery/`.

[3] W3C Consortium. Extensible markup language (xml) 1.0 (third edition), w3c recommendation 04 february 2004, 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

[4] P. Deransart, M. Ducassé, and L. Langevine. A generic trace model for finite domain solvers. In Barry O'Sullivan, editor, *Proceedings of the second International Workshop on User Interaction in Constraint Satisfaction (UICS'02)*, Cornell University (USA), Aug 2002. Available at `http://www.cs.ucc.ie/~osullb/UICS-02/papers/deransart_et_al-uics02.ps`.

[5] P. Deransart, M. Hermenegildo, and J. Małuszyński, editors. *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer Verlag, 2000. European Project (1997-2000) http://discipl.inria.fr.

[6] 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*, pages 470–471. Lecture Notes in Computer Science, Springer-Verlag, July 2002. Poster. Extended version available at `http://www.irisa.fr/lande/ducasse/`.

[7] Jean-Daniel Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *Proceedings of IEEE Symposium on Information Visualization 2002 (InfoVis 2002)*, pages 117–124. IEEE Press, October 2002.

[8] G. Ferrand, W. Lesaint, and A. Tessier. Value withdrawal explanation in CSP. In M. Ducassé, editor, *AADEBUG'00 (Fourth International Workshop on Automated Debugging)*, pages 188–201, 2000. The COmputer Research Repository (CORR) cs.SE/0012005.

[9] G. Ferrand, W. Lesaint, and A. Tessier. Explanations to understand the trace of a finite domain constraint solver. In F. Mesnard, editor, *Constraint Programming, Proceedings of JFPLC 2004*. Hermès, June 2004. To appear.

[10] Gérard Ferrand, Willy Lesaint, and Alexandre Tessier. Theoretical foundations of value withdrawal explanations for domain reduction. *Electronic Notes in Theoretical Computer Science*, 76, November 2002. http://www.elsevier.com/gej-ng/31/29/23/126/23/26/76008.pdf.

[11] Mohammad Ghoniem and Jean-Daniel Fekete. Visualisation de graphes de co-activité par matrices d'adjacence. In Cépadues, editor, *Actes de la conférence IHM 2002*, pages 279–284, October 2002.

[12] Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[13] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, July 2002.

[14] Ludovic Langevine, Pierre Deransart, and Mireille Ducassé. A generic trace schema for the portability of cp(fd) debugging tools. In K.R. Apt, F. Fages, F. Rossi, P. Szeredi, and Jozsef Vancza, editors, *Recent Advances in Constraints, 2003*, number 3010 in LNAI. Springer Verlag, May 2004.

[15] Ludovic Langevine and Mireille Ducassé. Un pilote de traceur pour la plc. déboguer, auditer et visualiser une exécution avec un même traceur. In Fred Mesnard, editor, *Programmation en logique avec contraintes, proceedings of JFPLC 2004*, Angers (France), June 2004. Hermès. English version: A tracer driver to enable debugging, monitoring and visualization of CLP executions from a single tracer, at OADymPPaC URL (public deliverables).

[16] K. Marriott and Stuckey P. J. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[17] 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.

[18] OADymPPaC. Tools for dynamic analysis and debugging of constraint programs. French RNTL project (2001-2004) `http://contraintes.inria.fr/OADymPPaC`.

[19] The Unicode Consortium. *The Unicode Standard, Version 3.0*. Addison-Wesley, Reading, MA, USA, 2000. Includes CD-ROM.

[20] Jean Paoli Tim Bray and C. M. Sperberg-McQueen eds. Extensible markup language (xml) 1.0. Technical report, W3 Consortium, 1998. http://www.w3.org/TR/REC-xml.

[21] F. Yergeau. RFC 2279: UTF-8, a transformation format of ISO 10646, January 1998.

Appendix A

gentra4cp DTD

```
<!--
  Copyright (c) 2004
  INRIA, Ecole des Mines de Nantes, INSA-Rennes, University of Orleans,
  Cosytec S.A., ILOG S.A.

  $Id: gentra4cp.dtd v 2.1 2004/05/12 19:00:00 deransart Exp $

  DTD describing the generic trace syntax for finite domain constraint solvers.
  Project OADymPPaC.  All the documentation related to the project can
  be found at the folloging URL:
  http://contraintes.inria.fr/OADymPPaC/

  For the gentra4cp version 2.1:

  Namespace:
    http://contraintes.inria.fr/OADymPPaC

  Public identifier:
    PUBLIC "-//GENTRA4CP//DTD GENTRA4CP 2.0.2//INRIA"

  URI for the DTD:
    http://contraintes.inria.fr/OADymPPaC/Public/Trace/gentra4cp.2.1.dtd

  URI for the documentation (syntax and semantics):
    http://contraintes.inria.fr/OADymPPaC/Public/Trace/gentra4cp-doc.2.1.pdf
-->

<!ENTITY % Toplevel
  "(provide | complement | breakpoint | new-variable | new-constraint | post |
    choice-point | back-to | solution | failure | remove |
    restore | reduce | suspend | solved | reject | awake | schedule |
    annotation | new-stage | start-stage | suspend-stage |
    resume-stage | stop-stage )*" >

<!ELEMENT gentra4cp (header, (%Toplevel; | packet)*) >
<!ATTLIST gentra4cp
   xmlns CDATA #FIXED "http://contraintes.inria.fr/OADymPPaC/Public/Trace">

<!ELEMENT packet %Toplevel;>
<!ATTLIST packet
   control  CDATA #IMPLIED >
```

```
<!ELEMENT breakpoint EMPTY>
<!ATTLIST breakpoint
   control  CDATA #IMPLIED >

<!ELEMENT header ( date, source, creator?, contributor?, description?,
                   identifier?, rights?, solver?, parameters?,
                   solver-parameters?, model-parameters?, checksum?,
                   provide? ) >

<!ELEMENT date (#PCDATA)>
<!ELEMENT source (#PCDATA)>
<!ELEMENT creator (#PCDATA)>
<!ELEMENT contributor (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT identifier (#PCDATA)>
<!ELEMENT rights (#PCDATA)>
<!ELEMENT solver (#PCDATA)>
<!ELEMENT parameters (#PCDATA)>

<!ELEMENT solver-parameters ( vardomain? , varenum? )>
<!ATTLIST solver-parameters
  back-to-strategy  CDATA #IMPLIED
  vident CDATA #IMPLIED
  cident CDATA #IMPLIED
  nident CDATA #IMPLIED >

<!ELEMENT varenum EMPTY >

<!ELEMENT model-parameters ( vardomain? , varenum? )>
<!ATTLIST model-parameters
   back-to-strategy  CDATA #IMPLIED>

<!ELEMENT checksum (#PCDATA)>

<!ELEMENT provide (( new-variable | new-constraint | post |
          choice-point | back-to | solution | failure | remove |
          restore | reduce | suspend | solved | reject | awake |
          schedule | annotation |  new-stage | start-stage |
          suspend-stage | resume-stage | stop-stage )* , state?) >

<!ELEMENT complement ( state | new-variable | new-constraint | post |
    choice-point | back-to | solution | failure | remove |
    restore | reduce | suspend | solved | reject | awake | schedule |
    annotation | new-stage | start-stage | suspend-stage |
    resume-stage | stop-stage ) >

<!ENTITY % integer "CDATA" >

<!ENTITY % number "CDATA" >

<!ENTITY % eventAttributes
  "chrono %integer; #REQUIRED
   depth %integer; #IMPLIED
   time  %integer; #IMPLIED
   context CDATA #IMPLIED
   line %integer; #IMPLIED
   file CDATA #IMPLIED" >
```

```
<!ENTITY % constraintAttributes
  "cident CDATA #REQUIRED
   cinternal CDATA #IMPLIED
   cname CDATA #IMPLIED
   cexternal CDATA #IMPLIED" >

<!ENTITY % variableAttributes
  "vident CDATA #REQUIRED
   vinternal CDATA #IMPLIED
   vname CDATA #IMPLIED
   vexternal CDATA #IMPLIED" >

<!ELEMENT state ((constraint)*, (variable)*, (update)*, misc?) >

<!ATTLIST state
   chrono %integer; #IMPLIED
   depth %integer; #IMPLIED
   time  %integer; #IMPLIED
   context CDATA #IMPLIED
   line CDATA #IMPLIED
   file CDATA #IMPLIED
   current-node CDATA #IMPLIED
   nname CDATA #IMPLIED
   status CDATA #IMPLIED
   choice-constraint CDATA #IMPLIED
   next-node CDATA #IMPLIED >

<!ELEMENT constraint ( variables? , (update)* ) >
<!ATTLIST constraint
  %constraintAttributes;
  orig CDATA #IMPLIED
  status CDATA #IMPLIED >

<!ELEMENT variables (#PCDATA) >

<!ELEMENT variable  ( vardomain? ) >
<!ATTLIST variable
  %variableAttributes;
  type CDATA #IMPLIED >

<!ELEMENT update EMPTY >
<!ATTLIST update
  vident CDATA #REQUIRED
  types CDATA #IMPLIED
  status CDATA #IMPLIED >

<!ELEMENT misc (#PCDATA) >

<!ELEMENT new-variable ( vardomain? , state?) >
<!ATTLIST new-variable
  %eventAttributes;
  %variableAttributes;
  type CDATA #IMPLIED >

<!ENTITY % valueList "( values | range )*" >

<!ELEMENT vardomain %valueList; >
<!ATTLIST vardomain
```

```
  min %integer; #IMPLIED
  max %integer; #IMPLIED
  size %integer; #IMPLIED >

<!ELEMENT values (#PCDATA) >

<!ELEMENT range EMPTY>
<!ATTLIST range
  from CDATA #REQUIRED
  to CDATA #REQUIRED >

<!ELEMENT new-constraint ( variables?, (update)*, state?) >
<!ATTLIST new-constraint
  %eventAttributes;
  %constraintAttributes;
  orig CDATA #IMPLIED>

<!ELEMENT post (state?) >
<!ATTLIST post
  %eventAttributes;
  cident CDATA #REQUIRED>

<!ELEMENT choice-point ( (choice-constraint)* , state? ) >
<!ATTLIST choice-point
   %eventAttributes;
   nident CDATA #IMPLIED
   nname CDATA #IMPLIED >

<!ELEMENT choice-constraint EMPTY >
<!ATTLIST choice-constraint
    vident CDATA #IMPLIED
    value %integer; #IMPLIED
    constraints CDATA #IMPLIED >

<!ELEMENT back-to ( (delta)*, (removed-values)*, state? ) >
<!ATTLIST back-to
  %eventAttributes;
  node CDATA #IMPLIED
  node-before CDATA #IMPLIED>

<!ELEMENT removed-values %valueList;  >
<!ATTLIST removed-values
  vident CDATA #IMPLIED >

<!ELEMENT solution ( (choice-constraint)* , state?) >
<!ATTLIST solution
  %eventAttributes;
  nident CDATA #IMPLIED
  nname CDATA #IMPLIED
  val %number; #IMPLIED>

<!ELEMENT failure ( (choice-constraint)* , state?) >
<!ATTLIST failure
  %eventAttributes;
  nident CDATA #IMPLIED
  nname CDATA #IMPLIED >

<!ELEMENT remove (state?) >
```

```
<!ATTLIST remove
  %eventAttributes;
  cident CDATA #REQUIRED>

<!ELEMENT restore ( delta?, vardomain?, update?, state? ) >
<!ATTLIST restore
  %eventAttributes;
  vident CDATA #IMPLIED >

<!ELEMENT reduce ( delta?, vardomain?, update?, explanation*, state? ) >
<!ATTLIST reduce
  %eventAttributes;
  cident CDATA #IMPLIED
  vident CDATA #IMPLIED
  algo CDATA #IMPLIED >

<!ELEMENT delta %valueList;  >
<!ATTLIST delta
  vident CDATA #IMPLIED >

<!ELEMENT explanation ( %valueList; , (cause)*, constraints?) >

<!ELEMENT cause %valueList; >
<!ATTLIST cause
  vident CDATA #REQUIRED
  ctype CDATA #IMPLIED>

<!ELEMENT constraints EMPTY>
<!ATTLIST constraints
  cidents CDATA #REQUIRED >

<!ELEMENT suspend (state?) >
<!ATTLIST suspend
  %eventAttributes;
  cident CDATA #REQUIRED>

<!ELEMENT solved (state?) >
<!ATTLIST solved
  %eventAttributes;
  cident CDATA #REQUIRED>

<!ELEMENT reject ( state? ) >
<!ATTLIST reject
  %eventAttributes;
  cident CDATA #REQUIRED>

<!ELEMENT awake (update?, state?) >
<!ATTLIST awake
  %eventAttributes;
  cident CDATA #REQUIRED>

<!ELEMENT schedule ( update?, state?) >
<!ATTLIST schedule
  %eventAttributes;
  cident CDATA #IMPLIED
  actions CDATA #IMPLIED>

<!ELEMENT annotation ( acmd?, state?) >
```

```
<!ATTLIST annotation
  %eventAttributes;
  aident CDATA #REQUIRED
  type CDATA #IMPLIED
  aname CDATA #IMPLIED
  refs CDATA #IMPLIED >

<!ELEMENT acmd (#PCDATA) >

<!ENTITY % stageAttributes
  "sident CDATA #REQUIRED
   sname CDATA #IMPLIED
   refs CDATA #IMPLIED
   detail CDATA #IMPLIED" >

<!ELEMENT new-stage (scomm?, state?) >
<!ATTLIST new-stage
  %eventAttributes;
  %stageAttributes;>

<!ELEMENT scomm (#PCDATA) >

<!ELEMENT start-stage ( state? ) >
<!ATTLIST start-stage
  %eventAttributes;
  sident CDATA #REQUIRED >

<!ELEMENT suspend-stage ( state? ) >
<!ATTLIST suspend-stage
  %eventAttributes;
  sident CDATA #REQUIRED >

<!ELEMENT resume-stage ( state? ) >
<!ATTLIST resume-stage
  %eventAttributes;
  sident CDATA #REQUIRED >

<!ELEMENT stop-stage ( state? ) >
<!ATTLIST stop-stage
  %eventAttributes;
  sident CDATA #REQUIRED >
```

Appendix B

Examples of Tracer and Trace Specification

Here are the specifications of some tracers, using the <provide> element in the prologue. It gives all the XML elements and attributes the tracer is able to provide. Knowing what several tracers may provide in their traces is the right information to use to develop more portable debugging tools.

The second section give examples of trace specification given with a <provide> element.

B.1 Specification of the Codeine GNU-Prolog Tracer

This the specification of the trace produced by Codeine, the tracer for GNU-Prolog. There is no port remove, nor restore. The time attribute is associated with the search-tree construction and stages events only to avoid slowing down the tracer.

```
<provide>

 <new-variable chrono="" depth="" context="" vident=""  vinternal=""
              vname="" type="">
   <vardomain min="" max="" size=""> <values/> <range from="" to="" />
   </vardomain>
  </new-variable>
<!-- context is the last called predication (Prolog tracer needed) -->
<!-- vident is the character 'v' followed by a unique positive number -->
<!-- vinternal is the internal pointer of the variable ('_' followed by a
positive offset -->
<!-- vname is the name of the variable (in the query or given
        by annotation -->
<!-- the type is 'int' for all finite domain variables -->

 <new-constraint chrono="" depth="" context="" cident="" cname=""
                cinternal="" cexternal="" orig="" />
<!-- cident is the character c followed by a number -->
<!-- vident is the character v followed by a number (as defined in
new-variable) -->
<!-- the orig is system or user when the constraint occurs in the source
program -->

 <post chrono="" depth="" cident="" />

 <choice-point chrono="" depth="" time="" nident="" />
<!-- nident is an integer -->
```

```
 <back-to chrono="" depth="" time="" node="" node-before="" />
<!-- no update in the back-to, not to slow down the tracer -->

 <solution chrono="" depth="" time="" nident="" val="" />

<!-- val is set only in the case of optimisation problem -->

 <failure chrono="" depth="" time="" nident="" />

 <reduce chrono="" depth="" cident="" vident="" algo="">
    <delta> <values/> <range from="" to="" />
    </delta>
    <vardomain min="" max="" size=""> <values/> <range from="" to="" />
    </vardomain>
    <update  vident="" types="" />
 </reduce>
<!-- no explanation at the moment-->

 <suspend chrono="" depth="" cident="" />

 <solved chrono="" depth="" cident="" />

 <reject chrono="" depth="" cident="" />

 <awake chrono="" depth="" cident="">
    <update vident="" types="" />
 </awake>
<!-- types is limited to one type -->

 <schedule chrono="" depth="" actions="">
  <update vident="" types="" />
 </schedule>
<!-- always the same action in GNU-Prolog, only one type
     (the one that is dequeued)  -->

 <annotation chrono="" depth="" time="" context="" aident="" type=""
            aname="" refs="" >
    <acmd/>
 </annotation>

 <new-stage chrono="" depth="" context="" sident="" sname="" refs=""
           detail="">
    <scomm/>
 </new-stage>

 <start-stage chrono="" depth="" time="" sident="" />

 <suspend-stage chrono="" depth="" time="" sident="" />

 <resume-stage chrono="" depth="" time="" sident="" />

 <stop-stage chrono="" depth="" time="" sident="" />

 <state chrono="" depth="" time="" current-node="" nname="" status=""
       choice-constraint="" next-node="">
    <constraint cident="" cinternal="" orig="" status="" >
       <variables/>
```

```
    </constraint>
    <variable vident="" type="" vinternal="" vname="">
       <vardomain min="" max="" size="">
         <values/> <range from="" to=""/>
       </vardomain>
    </variable>
    <update vident="" types="" status="" />
    <misc/>
 </state>
<!-- status: active, suspended, solved or rejected -->
<!-- the updates in the states do not have variable domains -->

</provide>
```

B.2 Specification of the JPaLM Tracer

This is the specification of the trace produced by the tracer for JPaLM, a java based version of PaLM [12].

```
<provide>

 <new-variable chrono="" depth="" time ="" line="" file="" vident=""
               vname="" type="">
      <vardomain min="" max="" size=""> <values/> <range from=" to=""/>
      </vardomain>
 </new-variable>

 <new-constraint chrono="" depth="" time="" line="" file="" cident=""
                 cexternal="" orig="" />
     <update vident="" types=""/>
 </new-constraint>

 <choice-point chrono="" depth="" time="" line="" file="">
    <choice-constraint vident="" value="" constraints=""/>
 <choice-point/>

 <solution chrono="" depth="" line="" file="" time="" val="" >
    <choice-constraint vident="" value="" constraints=""/>
 </solution>

 <failure chrono="" depth="" time="" line="" file="">
   <choice-constraint vident="" value="" constraints=""/>
 <failure/>

 <suspend chrono="" depth="" time="" line="" file="" cident="" />

 <reject chrono="" depth="" time="" line="" file="" cident="" />

 <awake chrono="" depth="" time="" line="" file="" cident="">
   <update vident="" types=""></update>
 </awake>

 <annotation chrono="" depth="" time="" context="" aident="" type=""
             aname="" refs="">
   <acmd/>
 </annotation>

 <new-stage chrono="" depth="" time="" line="" file=""  refs=""
```

```
           detail="" sident="" />

<start-stage chrono="" depth="" time="" line="" file="" sident="" />

<suspend-stage chrono="" depth="" time="" line="" file="" sident="" />

<resume-stage chrono="" depth="" time="" line="" file="" sident=""/>

<stop-stage chrono="" depth="" time="" line="" file="" sident="" />

<!-- PaLM specific part of the provide wrt choco -->

<reduce chrono="" depth="" time="" line="" file="" cident=""
        vident="" algo="">
  <update vident="" types=""/>
  <delta vident=""> <values/> <range from="" to=""/>
  </delta>
  <explanation>
     <range from="" to="" />
     <cause vident=""> <values/> </cause>
     <constraints cidents=""/>
  </explanation>
</reduce>

<restore chrono="" depth="" time="" line="" file="">
   <delta vident=""> <values/> <range from="" to=""/>
   </delta>
   <update vident="" types=""/>
</restore>
<!-- Restore can indicate how the variable is updated: min/max bound,
or value restoration for instance -->

<post chrono="" depth="" time="" line="" file="" cident="" context="" />
   <!-- "context" is used to specify indirect constraints. Indirect
   constraints are constraints whose activation depends on the
   activation of other constraints. In PaLM, this concept is crucial
   to ensure completeness of search. The context here is a set of
   constraints (like cidents in an explanation) that represent the
   conditions of activation of the constraints. For debugging
   purposes, it allows to explain why constraints - other than asked
   for by the user - are removed (these are indirect constraints
   needed to be removed because of the removal of a "father"
   constraint) -->

<remove chrono="" depth="" time="" line="" file="" cident="" context="" />
    <!-- Removing a constraint implies removing all indirect constraints which
    depend on this removed one. Thus a constraint can be removed either by the
    search algorithm (this is a choice point) or because it is not valid
    anymore. In the last case, remove events can provide a context attribute
    specifying which constraint has been directly removed and is responsible
    for this indirect remove -->

<state chrono="" depth="" time="" >
    <constraint cident="" cinternal="" orig="" status="">
      <variables />
    <constraint/>
    <variable vident="" vname="" type="">
       <vardomain  min="" max="" size=""> <values/><range from="" to=""/>
```

```
      </vardomain>
    </variable>
  <update vident="" types="" />
 </state>

</provide>
```

B.3 Specification of the JChoco Tracer

This is the specification of the trace produced by the tracer for Choco.

```
<provide>

   <new-variable chrono="" depth="" time ="" line="" file="" vident=""
                 vname="" type="">
      <vardomain> <values/> <range from="" to="" />
      </vardomain>
   </new-variable>

   <new-constraint chrono="" depth="" time="" line="" file="" cident=""
                   cexternal="" orig="">
      <update vident="" types=""/>
   </new-constraint>

   <post chrono="" depth="" time="" line="" file="" cident="" />

   <choice-point chrono="" depth="" time="" line="" file="">
      <choice-constraint vident="" value="" constraints=""/>
   </choice-point>

   <back-to chrono="" depth="" time="" line="" file="" />

   <solution chrono="" depth="" line="" file="" time="" val="" />

   <failure chrono="" depth="" time="" line="" file="" />

   <reduce chrono="" depth="" time="" line="" file="" cident=""
           vident="" algo="">
      <delta vident=""> <values/> <range from="" to=""/>
      </delta>
      <update vident="" types=""/>
   </reduce>

   <suspend chrono="" depth="" time="" line="" file="" cident="" />

   <reject chrono="" depth="" time="" line="" file="" cident="" />

   <awake chrono="" depth="" time="" line="" file="" cident="">
      <update vident="" types=""/>
   </awake>

   <new-stage chrono="" depth="" time="" line="" file=""  refs="" detail=""
              sident="" />

   <start-stage chrono="" depth="" time="" line="" file="" sident="" />

   <suspend-stage chrono="" depth="" time="" line="" file="" sident="" />
```

```
   <resume-stage chrono="" depth="" time="" line="" file="" sident=""/>

   <stop-stage chrono="" depth="" time="" line="" file="" sident="" />

    <state chrono="" depth="" time="" >
       <constraint cident="" cinternal="" orig="" status="" />
       <variable vident="" vname="" type="">
          <vardomain> <values/><range from="" to=""/>
          </vardomain>
       </variable>
       <update vident="" types="" />
    </state>

</provide>
```

B.4 Specification of Traces for visualization Tools

This section gives some minimal trace requirements for some visualization tools. Knowing what several debugging tools are requesting a tracer to provide in order to observe accurately constraint resolution in the right information to know to develop tracers able to use more debugging tools.

B.4.1 Specification of Traces for Search-tree Tracing Tool

A sufficient trace to draw search-tree and display some information at the nodes: constraints and status, variables and domains. The tool should use the infomations provided in the header (`<provide>` element to decide which ports it uses to build nodes in the search-tree.

```
<provide>
   <new-variable chrono="" depth="" vident="" vname="" vexternal="">
      <values/><range from="" to=""/>  </new-variable>
   <new-constraint chrono="" depth="" cident="" cname="" cexternal="">
      <state>
          <constraint cident="" >
             <variables/>
          </constraint>
       </state>
   </new-constraint>
   <choice-point chrono="" depth="" nident="" />
   <back-to chrono="" depth="" node="" node-before="" />
   <solution chrono="" depth="" nident="" val="" >
      <state>
         <variable vident="" >
            <vardomain> <values/><range from="" to=""/>
            </vardomain>
         </variable>
      </state>
   <failure chrono="" depth="" nident="" />
</provide>
```

B.4.2 Specification of Traces for Variables Observation

Specification for a propagation tree: search tree as previously and reduce elements.

```
<reduce chrono="" depth="" cident="" vident="" />
```

B.4.3 Specification of Traces for the INFOVIS Tool

Specification of the trace for visualization of adjacency matrices (var x constraints) and (constraints x constraints) by the INFIVIS tool (general approach described in [11]). This allows to build dynamic weighted graphs (new arrows may be added by observation of the reductions and the weights depend on the importance of the variable domain reduction).

Maximal specification (in fact maximally sufficient since no other information may be taken into account).

```
<provide>
 <new-variable chrono="" vident=""  vname=""  >
    <vardomain size="" />
 </new-variable>

 <new-constraint chrono="" cident="" cname=""  >
    <update vident="" />
    <state>
       <constraint cident="" >
          <variables />
       </constraint>
    </state>
 </new-constraint>

 <awake chrono="" cident="">                 <!-- intermediate precision (-->
    <update vident="" />
 </awake>

 <reduce chrono="" cident="" vident="" >
    <delta vident="" > <values/> <range from="" to="" /> <!-- vident optional  -->
    </delta>                                             <!--delta or vardomain-->
    <vardomain size="" > <values/> <range from="" to="" /> <!-- size or domain -->
    </vardomain>
    <update  vident="" /> <!-- intermediate precision -->
    <explanation>                  <!-- maximal precision (no awake needed) -->
      <constraints cidents=""/>
    </explanation>
 </reduce>
</provide>
```

Minimal specification: sufficient to build a static weighted graph (static in the sense that the determination of the arrows is based on the variable and constraints declaration).

```
<provide>
 <new-variable chrono="" vident=""  />

 <new-constraint chrono=""  cident="" />

 <reduce chrono="" cident="" vident="" />
</provide>
```

Appendix C

Examples of Trace

Here are some trace examples. All are trace of the same program: the comparison of two variables `X` and `Y` ranging over domain `[1..3]`, with one constraint `X #> Y`. There are three solutions `(1,2)`,`(1,3)` and `(2,3)`.

C.1 A Trace by the Codeine Tracer (GNU-Prolog)

The trace corresponds to the resolution of the sequence of constraints given in the <parameter> element. It corresponds to the follwing program:

```
fd_domain([X,Y],1,3), X #> Y, fd_labeling([X,Y]).
```

The trace of the resolution is as follows.

```
<!DOCTYPE gentra4cp SYSTEM
"http://contraintes.inria.fr/OADymPPaC/Public/Trace/gentra4cp.2.0.2.dtd">
<gentra4cp>
<header>
<date>2004-04-28 20:19:28</date>
   <source>sorted-gnu</source>
   <creator>deransar</creator>
   <contributor></contributor>
   <description></description>
   <rights>This trace is public domain.</rights>
   <solver>Codeine, Version 0.9.2 (2004-04-21).
    COnstraint DEbugging INteractive EnvironmentBy Ludovic Langevine
    and Tristan  Denmat</solver>
   <parameters>
      multsorted(2,_15)
   </parameters>
</header>

<provide>
   <new-variable chrono="" depth=""  vident="" vname="">
   <vardomain min="" max="" size="">
      <values/> <range from="" to=""/>
   </vardomain>
   </new-variable>
   <new-constraint chrono="" depth=""  cident="" cinternal="">
   </new-constraint>
   <post chrono="" depth=""  cident="" />
   <choice-point chrono="" depth="" nident="" />
   <back-to chrono="" depth="" before="" node-before="" />
```

```
   &lt;solution chrono="" depth="" nident="" &gt;
      &lt;state&gt;
         &lt;variable vident="" type="" vinternal="" vname=""&gt;
            &lt;vardomain min="" max="" size="" &gt; &lt;/vardomain&gt;
         &lt;/variable&gt;
      &lt;/state&gt;
   &lt;/solution&gt;
   &lt;failure chrono="" depth="" nident="" /&gt;
   &lt;reduce chrono="" depth=""  algo="" cident="" vident="" &gt;
      &lt;delta&gt; &lt;values/&gt; &lt;range from="" to=""/&gt; &lt;/delta&gt;
      &lt;update vident="" types=""/&gt;
   &lt;/reduce&gt;
&lt;/provide&gt;

&lt;choice-point chrono="1" depth="0" nident="0" nname="root" /&gt;
&lt;new-variable chrono="2" depth="1" vident="v1" &gt;
   &lt;vardomain min="1" max="3" size="3"&gt;&lt;range from="1" to="3" /&gt;
   &lt;/vardomain&gt;
&lt;/new-variable&gt;
&lt;new-constraint chrono="3" depth="1" cident="c1" cinternal="fd_domain(v1,1,3)"/&gt;
&lt;post chrono="4" depth="1" cident="c1" /&gt;
&lt;solved chrono="5" depth="1" cident="c1" /&gt;
&lt;new-variable chrono="6" depth="1" vident="v2" &gt;
   &lt;vardomain min="1" max="3" size="3"&gt;&lt;range from="1" to="3" /&gt;
   &lt;/vardomain&gt;
&lt;/new-variable&gt;
&lt;new-constraint chrono="7" depth="1" cident="c2" cinternal="fd_domain(v2,1,3)"/&gt;
&lt;post chrono="8" depth="1" cident="c2" /&gt;
&lt;solved chrono="9" depth="1" cident="c2" /&gt;
&lt;choice-point chrono="10" depth="1" nident="1" /&gt;
&lt;new-constraint chrono="11" depth="2" cident="c3" cinternal="x_lt_y(v1,v2)" /&gt;
&lt;post chrono="12" depth="2" cident="c3" /&gt;
&lt;reduce chrono="13" depth="2" algo="initial[2]" cident="c3" vident="v1" &gt;
   &lt;delta&gt;&lt;range from="3" to="3" /&gt;&lt;/delta&gt;
   &lt;update vident="v1" types="max minmax dom" /&gt;
&lt;/reduce&gt;
&lt;reduce chrono="14" depth="2" algo="initial[2]" cident="c3" vident="v2" &gt;
   &lt;delta&gt;&lt;range from="1" to="1" /&gt;&lt;/delta&gt;
   &lt;update vident="v2" types="min minmax dom" /&gt;
&lt;/reduce&gt;
&lt;suspend chrono="15" depth="2" cident="c3" /&gt;
&lt;choice-point chrono="16" depth="2" nident="2" /&gt;
&lt;new-constraint chrono="17" depth="3" cident="c6" cinternal="assign(v1,1)"/&gt;
&lt;post chrono="18" depth="3" cident="c6" /&gt;
&lt;reduce chrono="19" depth="3" algo="initial[1]" cident="c6" vident="v1" &gt;
   &lt;delta&gt;&lt;range from="2" to="2" /&gt;&lt;/delta&gt;
   &lt;update vident="v1" types="max minmax dom val" /&gt;
&lt;/reduce&gt;
&lt;solved chrono="20" depth="3" cident="c6" /&gt;
&lt;choice-point chrono="21" depth="3" nident="3" /&gt;
&lt;new-constraint chrono="22" depth="4" cident="c7" cinternal="assign(v2,2)" /&gt;
&lt;post chrono="23" depth="4" cident="c7" /&gt;
&lt;reduce chrono="24" depth="4" algo="initial[1]" cident="c7" vident="v2" &gt;
   &lt;delta&gt;&lt;range from="3" to="3" /&gt;&lt;/delta&gt;
   &lt;update vident="v2" types="max minmax dom val" /&gt;
&lt;/reduce&gt;
&lt;solved chrono="25" depth="4" cident="c7" /&gt;
&lt;schedule chrono="26" depth="4" actions="dequeue" &gt;
```

```
   <update vident="v2" types="max" />
</schedule>
<solution chrono="27" depth="4" nident="4" >
   <state>
   <variable vident="v1" type="int" vinternal="_#3" vname="var2">
      <vardomain min="1" max="1" size="1" />
   </variable>
   <variable vident="v2" type="int" vinternal="_#25" vname="var1">
      <vardomain min="2" max="2" size="1" />
   </variable>
   </state>
</solution>
<back-to chrono="28" depth="3" node="3" node-before="4" />
<new-constraint chrono="29" depth="4" cident="c8" cinternal="assign(v2,3)"/>
<post chrono="30" depth="4" cident="c8" />
<reduce chrono="31" depth="4" algo="initial[1]" cident="c8" vident="v2" >
   <delta><range from="2" to="2" /></delta>
   <update vident="v2" types="min minmax dom val" />
</reduce>
<solved chrono="32" depth="4" cident="c8" />
<solution chrono="33" depth="4" nident="5" >
   <state>
   <variable vident="v1" type="int" vinternal="_#3" vname="var2">
      <vardomain min="1" max="1" size="1" />
   </variable>
   <variable vident="v2" type="int" vinternal="_#25" vname="var1">
      <vardomain min="3" max="3" size="1" />
   </variable>
   </state>
</solution>
<back-to chrono="34" depth="2" node="2" node-before="5" />
<new-constraint chrono="35" depth="3" cident="c9" cinternal="assign(v1,2)" />
<post chrono="36" depth="3" cident="c9" />
<reduce chrono="37" depth="3" algo="initial[1]" cident="c9" vident="v1" >
   <delta><range from="1" to="1" /></delta>
   <update vident="v1" types="min minmax dom val" />
</reduce>
<solved chrono="38" depth="3" cident="c9" />
<schedule chrono="39" depth="3" actions="dequeue" >
   <update vident="v1" types="min" />
</schedule>
<awake chrono="40" depth="3" cident="c3" >
   <update vident="v1" types="min" />
</awake>
<reduce chrono="41" depth="3" algo="f_5[1]" cident="c3" vident="v2" >
   <delta><range from="2" to="2" /></delta>
   <update vident="v2" types="min minmax dom val" />
</reduce>
<suspend chrono="42" depth="3" cident="c3" />
<solution chrono="43" depth="3" nident="6" >
   <state>
   <variable vident="v1" type="int" vinternal="_#3" vname="var2">
      <vardomain min="2" max="2" size="1" />
   </variable>
   <variable vident="v2" type="int" vinternal="_#25" vname="var1">
      <vardomain min="3" max="3" size="1" />
   </variable>
   </state>
```

```
</solution>
<back-to chrono="44" depth="2" node="2" node-before="6" />
<failure chrono="45" depth="3" nident="7" />
<back-to chrono="46" depth="0" node="0" node-before="7" />
</gentra4cp>
```

C.2 A Trace by the JPaLM Tracer

The trace corresponds to the following program:

```
Problem pb = new PalmProblem();
IntVar var0 = pb.makeBoundIntVar("Var0", 1, 3);
IntVar var1 = pb.makeBoundIntVar("Var1", 1, 3);
pb.post(pb.gt(var1, var0));
pb.solve(true);
```


```
<!DOCTYPE gentra4cp SYSTEM
"http://contraintes.inria.fr/OADymPPaC/Public/Trace/gentra4cp.2.0.2.dtd">
<gentra4cp>
 <header>
   <date>2004-05-02 16:28:47</date>
   <source>NSort.java</source>
   <contributor>Ecole des Mines de Nantes/LINA</contributor>
   <solver>Palm traced by Oadymppac trace aspect</solver>
 </header>
 <provide>
   <new-variable chrono=""  vident="" vname="">
      <vardomain min="" max="" size="">
         <values/> <range from="" to=""/>
      </vardomain>
   </new-variable>
   <new-constraint chrono=""  cident="" cexternal="" orig="">
   </new-constraint>
   <choice-point chrono="" />
   <solution chrono="" >
      <state>
         <variable vident=""><vardomain><values/></vardomain></variable>
      </state>
   </solution>
   <failure chrono="" />
   <reduce chrono=""  cident="">
      <delta> <values/> <range from="" to=""/> </delta>
      <update vident="" types=""/>
      <explanation>
         <range from="" to=""/>
         <constraints cidents=""/>
      </explanation>
   </reduce>
   <restore chrono="" >
      <delta> <values/> <range from="" to=""/> </delta>
   </restore>
   <post chrono=""  cident="" context=""/>
   <remove chrono=""  cident="" context=""/>
 </provide>

 <new-variable chrono="0" vident="v0" vname="Var0">
  <vardomain min="1" max="3" size="3">
   <range from="1" to="3"/>
```

```
 </vardomain>
</new-variable>
<new-variable chrono="1" vident="v1" vname="Var1">
 <vardomain min="1" max="3" size="3">
  <range from="1" to="3"/>
 </vardomain>
</new-variable>
<new-constraint chrono="2"  cident="c0" cexternal="Var1 >= Var0 + 1"
                orig="user">
</new-constraint>
<post chrono="3"  cident="c0">
</post>
<reduce chrono="4"  cident="c0">
 <delta>
  <range from="1" to="1"/>
 </delta>
 <update  vident="v1" types="min"/>
 <explanation>
  <range from="1" to="1"/>
  <constraints cidents="c0 "/>
 </explanation>
</reduce>
<reduce chrono="5"  cident="c0">
 <delta>
  <range from="3" to="3"/>
 </delta>
 <update  vident="v0" types="max"/>
 <explanation>
  <range from="3" to="3"/>
  <constraints cidents="c0 "/>
 </explanation>
</reduce>
<choice-point chrono="6" />
<new-constraint chrono="7"  cident="c1" cexternal="Var0 == 1"
                orig="system">
</new-constraint>
<post chrono="8"  cident="c1">
</post>
<reduce chrono="9"  cident="c1">
 <delta>
  <range from="2" to="2"/>
 </delta>
 <update  vident="v0" types="max"/>
 <explanation>
  <range from="2" to="2"/>
  <constraints cidents="c0 c1 "/>
 </explanation>
</reduce>
<choice-point chrono="10" />
<new-constraint chrono="11"  cident="c2" cexternal="Var1 == 2"
                orig="system">
</new-constraint>
<post chrono="12"  cident="c2">
</post>
<reduce chrono="13"  cident="c2">
 <delta>
  <range from="3" to="3"/>
 </delta>
```

```
 &lt;update  vident="v1" types="max"/&gt;
 &lt;explanation&gt;
  &lt;range from="3" to="3"/&gt;
  &lt;constraints cidents="c2 "/&gt;
 &lt;/explanation&gt;
&lt;/reduce&gt;
&lt;solution chrono="14" &gt;
 &lt;state&gt;
  &lt;variable vident="v0"&gt;&lt;vardomain&gt;&lt;values&gt;1&lt;/values&gt;&lt;/vardomain&gt;
  &lt;/variable&gt;
  &lt;variable vident="v1"&gt;&lt;vardomain&gt;&lt;values&gt;2&lt;/values&gt;&lt;/vardomain&gt;
  &lt;/variable&gt;
 &lt;/state&gt;
&lt;/solution&gt;
&lt;reduce chrono="15" &gt;
 &lt;delta&gt;
  &lt;range from="0" to="1"/&gt;
 &lt;/delta&gt;
 &lt;update vident="v-1" types="empty"/&gt;
 &lt;explanation&gt;
  &lt;range from="0" to="1"/&gt;
  &lt;constraints cidents="c1 c2 "/&gt;
 &lt;/explanation&gt;
&lt;/reduce&gt;
&lt;failure chrono="16" /&gt;
&lt;choice-point chrono="17" /&gt;
&lt;remove chrono="18"  cident="c2"/&gt;
&lt;restore chrono="19" &gt;
 &lt;delta vident="v1"&gt;
  &lt;range from="3" to="3"/&gt;
 &lt;/delta&gt;
&lt;/restore&gt;
&lt;restore chrono="20" &gt;
 &lt;delta vident="v-1"&gt;
  &lt;range from="0" to="1"/&gt;
 &lt;/delta&gt;
&lt;/restore&gt;
<new-constraint chrono="21"  cident="c3" cexternal="Var1 !== 2"
                orig="system">
&lt;/new-constraint&gt;
&lt;post chrono="22"  cident="c3" context="c1 "/&gt;
&lt;reduce chrono="23"  cident="c3"&gt;
 &lt;delta&gt;
  &lt;range from="2" to="2"/&gt;
 &lt;/delta&gt;
 &lt;update  vident="v1" types="min"/&gt;
 &lt;explanation&gt;
  &lt;range from="2" to="2"/&gt;
  &lt;constraints cidents="c0 c1 "/&gt;
 &lt;/explanation&gt;
&lt;/reduce&gt;
&lt;solution chrono="24" &gt;
 &lt;state&gt;
  &lt;variable vident="v0"&gt;&lt;vardomain&gt;&lt;values&gt;1&lt;/values&gt;&lt;/vardomain&gt;
  &lt;/variable&gt;
  &lt;variable vident="v1"&gt;&lt;vardomain&gt;&lt;values&gt;3&lt;/values&gt;&lt;/vardomain&gt;
  &lt;/variable&gt;
 &lt;/state&gt;
```

```
</solution>
<reduce chrono="25" >
 <delta>
  <range from="0" to="1"/>
 </delta>
 <update vident="v-1" types="empty"/>
 <explanation>
  <range from="0" to="1"/>
  <constraints cidents="c1 "/>
 </explanation>
</reduce>
<failure chrono="26" />
<choice-point chrono="27" />
<remove chrono="28"  cident="c1"/>
<remove chrono="29"  cident="c3" context="c1"/>
<restore chrono="30" >
 <delta vident="v0">
  <range from="2" to="2"/>
 </delta>
</restore>
<restore chrono="31" >
 <delta vident="v-1">
  <range from="0" to="1"/>
 </delta>
</restore>
<restore chrono="32" >
 <delta vident="v1">
  <range from="2" to="2"/>
 </delta>
</restore>
<new-constraint chrono="33"  cident="c4" cexternal="Var0 !== 1"
                orig="system">
</new-constraint>
<post chrono="34"  cident="c4" context=""/>
<reduce chrono="35"  cident="c4">
 <delta>
  <range from="1" to="1"/>
 </delta>
 <update  vident="v0" types="min"/>
 <explanation>
  <range from="1" to="1"/>
  <constraints cidents=""/>
 </explanation>
</reduce>
<reduce chrono="36"  cident="c0">
 <delta>
  <range from="2" to="2"/>
 </delta>
 <update  vident="v1" types="min"/>
 <explanation>
  <range from="2" to="2"/>
  <constraints cidents="c0 "/>
 </explanation>
</reduce>
<solution chrono="37" >
 <state>
  <variable vident="v0"><vardomain><values>2</values></vardomain>
  </variable>
```

```
   <variable vident="v1"><vardomain><values>3</values></vardomain>
   </variable>
  </state>
 </solution>
 <reduce chrono="38" >
  <delta>
   <range from="0" to="1"/>
  </delta>
  <update vident="v-1" types="empty"/>
  <explanation>
   <range from="0" to="1"/>
   <constraints cidents=""/>
  </explanation>
 </reduce>
 <failure chrono="39" />
</gentra4cp>
```

C.3 A Trace by the JChoco Tracer

The trace corresponds to the resolution of the same program.

```
<!DOCTYPE gentra4cp SYSTEM
"http://contraintes.inria.fr/OADymPPaC/Public/Trace/gentra4cp.2.0.2.dtd">
<gentra4cp>
<header>
  <date>2004-05-02 16:28:04</date>
  <source>NSort.java</source>
  <contributor>Ecole des Mines de Nantes/LINA</contributor>
  <solver>Choco traced by Oadymppac trace aspect</solver>
</header>

<provide>
   <new-variable chrono="" depth=""  vident="" vname="">
   <vardomain min="" max="" size="">
      <values/> <range from="" to=""/>
   </vardomain>
   </new-variable>
   <new-constraint chrono="" depth=""  cident="" orig="">
   </new-constraint>
   <post chrono="" depth=""  cident=""/>
   <choice-point chrono="" depth="" />
   <back-to chrono="" depth="" />
   <solution chrono="" depth="" >
      <state>
         <variable vident=""><vardomain><values/></vardomain>
         </variable>
      </state>
   </solution>
   <failure chrono="" depth="" />
   <reduce chrono="" depth=""  cident="">
      <delta> <values/> <range from="" to=""/> </delta>
      <update vident="" types=""/>
   </reduce>
 </provide>

 <new-variable chrono="0" depth="0"  vident="v0" vname="Var0">
  <vardomain min="1" max="3" size="3">
```

```
  <range from="1" to="3"/>
 </vardomain>
</new-variable>
<new-variable chrono="1" depth="0"  vident="v1" vname="Var1">
 <vardomain min="1" max="3" size="3">
  <range from="1" to="3"/>
 </vardomain>
</new-variable>
<new-constraint chrono="2" depth="0"  cident="c0" orig="user">
</new-constraint>
<post chrono="3" depth="0"  cident="c0">
</post>
<reduce chrono="4" depth="0"  cident="c0">
 <delta>
  <range from="1" to="1"/>
 </delta>
 <update  vident="v1" types="min"/>
</reduce>
<reduce chrono="5" depth="0"  cident="c0">
 <delta>
  <range from="3" to="3"/>
 </delta>
 <update  vident="v0" types="max"/>
</reduce>
<choice-point chrono="6" depth="0" />
<choice-point chrono="7" depth="1" />
<reduce chrono="8" depth="2" >
 <delta>
  <range from="2" to="2"/>
 </delta>
 <update  vident="v0" types="ground"/>
</reduce>
<choice-point chrono="9" depth="2" />
<reduce chrono="10" depth="3" >
 <delta>
  <range from="3" to="3"/>
 </delta>
 <update  vident="v1" types="ground"/>
</reduce>
<solution chrono="11" depth="3" >
 <state>
  <variable vident="v0"><vardomain><values>1</values></vardomain>
  </variable>
  <variable vident="v1"><vardomain><values>2</values></vardomain>
  </variable>
 </state>
</solution>
<back-to chrono="12" depth="2" />
<reduce chrono="13" depth="2" >
 <delta>
  <range from="2" to="2"/>
 </delta>
 <update  vident="v1" types="ground"/>
</reduce>
<choice-point chrono="14" depth="2" />
<solution chrono="15" depth="3" >
 <state>
  <variable vident="v0"><vardomain><values>1</values></vardomain>
```

```
    &lt;/variable&gt;
    &lt;variable vident="v1"&gt;&lt;vardomain&gt;&lt;values&gt;3&lt;/values&gt;&lt;/vardomain&gt;
    &lt;/variable&gt;
   &lt;/state&gt;
  &lt;/solution&gt;
  &lt;back-to chrono="16" depth="2" /&gt;
  &lt;reduce chrono="17" depth="2" &gt;
   &lt;delta&gt;
    &lt;range from="3" to="3"/&gt;
   &lt;/delta&gt;
   &lt;update vident="v1" types="empty"/&gt;
  &lt;/reduce&gt;
  &lt;failure chrono="18" depth="2" /&gt;
  &lt;back-to chrono="19" depth="1" /&gt;
  &lt;reduce chrono="20" depth="1" &gt;
   &lt;delta&gt;
    &lt;range from="1" to="1"/&gt;
   &lt;/delta&gt;
   &lt;update  vident="v0" types="ground"/&gt;
  &lt;/reduce&gt;
  &lt;reduce chrono="21" depth="1"  cident="c0"&gt;
   &lt;delta&gt;
    &lt;range from="2" to="2"/&gt;
   &lt;/delta&gt;
   &lt;update  vident="v1" types="ground"/&gt;
  &lt;/reduce&gt;
  &lt;choice-point chrono="22" depth="1" /&gt;
  &lt;solution chrono="23" depth="2" &gt;
   &lt;state&gt;
    &lt;variable vident="v0"&gt;&lt;vardomain&gt;&lt;values&gt;2&lt;/values&gt;&lt;/vardomain&gt;
    &lt;/variable&gt;
    &lt;variable vident="v1"&gt;&lt;vardomain&gt;&lt;values&gt;3&lt;/values&gt;&lt;/vardomain&gt;
    &lt;/variable&gt;
   &lt;/state&gt;
  &lt;/solution&gt;
  &lt;back-to chrono="24" depth="1" /&gt;
  &lt;reduce chrono="25" depth="1" &gt;
   &lt;delta&gt;
    &lt;range from="2" to="2"/&gt;
   &lt;/delta&gt;
   &lt;update vident="v0" types="empty"/&gt;
  &lt;/reduce&gt;
  &lt;failure chrono="26" depth="1" /&gt;
  &lt;back-to chrono="27" depth="0" /&gt;
  &lt;reduce chrono="28" depth="0" &gt;
   &lt;delta&gt;
    &lt;range from="1" to="1"/&gt;
   &lt;/delta&gt;
   &lt;update  vident="v0" types="ground"/&gt;
  &lt;/reduce&gt;
  &lt;reduce chrono="29" depth="0" &gt;
   &lt;delta&gt;
    &lt;range from="2" to="2"/&gt;
   &lt;/delta&gt;
   &lt;update  vident="v1" types="ground"/&gt;
  &lt;/reduce&gt;
 &lt;/gentra4cp&gt;
```

C.4 A Trace by the CHIP Tracer (Cosytec)

The trace corresponds to the same program as for GNU-Prolog.


```
<!DOCTYPE gentra4cp SYSTEM
"http://contraintes.inria.fr/OADymPPaC/Public/Trace/gentra4cp.2.0.2.dtd">
 <gentra4cp>
   <header>
     <date>2004-05-03</date>
     <source>mult sorted in CHIP</source>
     <creator>COSYTEC SA</creator>
     <contributor>COSYTEC SA</contributor>
     <solver>CHIPC++ 5.5.0.9</solver>
     <parameters>multsorted(2)</parameters>
   </header>

<provide>
   <new-variable chrono="" vident="">
      <vardomain min="" max="" size="">
         <values/> <range from="" to=""/>
      </vardomain>
   </new-variable>
   <new-constraint chrono="" cident="" cname="">
   </new-constraint>
   <post chrono="" cident=""/>
   <choice-point chrono="" depth="" nname="">
      <choice-constraint value="" vident="" />
   </choice-point>
   <solution />
   <reduce chrono="" depth=""  vident="">
      <delta> <values/> <range from="" to=""/>
      </delta>
      <vardomain min="" max="" size="">
         <values/> <range from="" to=""/>
      </vardomain>
      <update vident="" types=""/>
   </reduce>
   <awake chrono="" cident="" vident="" />
 </provide>

  <new-variable chrono="1" vident="1">
     <vardomain max="3" min="1" size="3">
       <range to="3" from="1" />
     </vardomain>
  </new-variable>
  <new-variable chrono="2" vident="2">
     <vardomain max="3" min="1" size="3">
        <range to="3" from="1" />
     </vardomain>
  </new-variable>
  <new-constraint cname=">=_1" chrono="3" cident="1" />
  <post chrono="4" cident="1" />
  <reduce chrono="5" cident="1" vident="2">
     <delta vident="2">
        <values>1</values>
     </delta>
     <vardomain max="3" min="2" size="2">
        <range to="3" from="2" />
     </vardomain>
```

```
   <update types="min" vident="2" />
</reduce>
<reduce chrono="6" cident="1" vident="1">
   <delta vident="1">
      <values>3</values>
   </delta>
   <vardomain max="2" min="1" size="2">
      <range to="2" from="1" />
   </vardomain>
   <update types="max" vident="1" />
</reduce>
<suspend chrono="7" cident="1" />
<choice-point depth="0" nname="root" chrono="8" />
<choice-point depth="1" chrono="9">
   <choice-constraint value="2" vident="2" />
</choice-point>
<reduce chrono="10" cident="choice-ctr" vident="2">
   <delta vident="2">
      <values>3</values>
   </delta>
   <vardomain max="2" min="2" size="1">
      <values>2</values>
   </vardomain>
   <update types="ground" vident="2" />
</reduce>
<awake chrono="11" cident="1" vident="2" />
<reduce chrono="12" cident="1" vident="1">
   <delta vident="1">
      <values>2</values>
   </delta>
   <vardomain max="1" min="1" size="1">
      <values>1</values>
   </vardomain>
   <update types="ground" vident="1" />
</reduce>
<suspend chrono="13" cident="1" />
<choice-point depth="2" chrono="14">
   <choice-constraint value="1" vident="1" />
</choice-point>
<solution />
<choice-point depth="1" chrono="15">
   <choice-constraint value="3" vident="2" />
</choice-point>
<reduce chrono="16" cident="choice-ctr" vident="2">
   <delta vident="2">
      <values>2</values>
   </delta>
   <vardomain max="3" min="3" size="1">
      <values>3</values>
   </vardomain>
   <update types="ground" vident="2" />
</reduce>
<awake chrono="17" cident="1" vident="2" />
<suspend chrono="18" cident="1" />
<choice-point depth="2" chrono="19">
   <choice-constraint value="1" vident="1" />
</choice-point>
<reduce chrono="20" cident="choice-ctr" vident="1">
```

```
   <delta vident="1">
      <values>2</values>
   </delta>
   <vardomain max="1" min="1" size="1">
      <values>1</values>
   </vardomain>
   <update types="ground" vident="1" />
</reduce>
<solution />
<choice-point depth="2" chrono="21">
   <choice-constraint value="2" vident="1" />
</choice-point>
<reduce chrono="22" cident="choice-ctr" vident="1">
   <delta vident="1">
      <values>1</values>
   </delta>
   <vardomain max="2" min="2" size="1">
      <values>2</values>
   </vardomain>
   <update types="ground" vident="1" />
</reduce>
<solution />
</gentra4cp>
```

Glossary

Attributes There are *Trace event attributes* and *XML element attributes*.

Each trace event type or port (see *event* bellow) has a set of *attributes* which corresponds to the description of the modified elements of the solver abstract state. The main attributes are described in Chap. 3 (Semantics), but many other attributes, whose meaning is clear, are described in the syntactic description of the event (Chap.4, 5, 6 and 7).

XML elements have contents and attributes. The distinction is just syntactic. The decision to describe a trace event attribute by an XML element attribute or a content is just a design decision, not related with the semantics.

Control In this document "control" is used in different places with different meanings: control relative to the way of exploring the search space, and control related to the tracer-tool interactions.

The *control relative to the way of exploring the search space* is described in Chap. 3. If the search space is described by a search-tree then the control correspond to the way to visit the search-tree.

The *control related to the tracer-tool interactions* must be defined by a protocol of communication between the tracer and the debugging tool. A general approach is discussed in Chap 8.

Debugging Tool A debugging tool is a software used to debug a program. There is no particular limitation on the kind of debugging tool considered in this document except that a debugging tool should be able to handle the generic trace as input data and find all the information it need in this trace.

Debugging tools are called *analyser* in [15].

Defined Many features in this document are qualified *tracer defined*. For example the form of the variable or constraint identifiers, the content of a `<misc>` element, or the attributes of the `<choice-constraint>`. It means that they should be completely and exactly described in the documentation of the tracer or that their form in the trace must be documented.

Dependent Many features in this document are qualified *tracer dependent*. For example if the variable or constraint identifiers may be ordered, this order is tracer dependent, i.e. it has no particular meaning in this semantics and no tool should rely on this order. Therefore a tool should not rely on a tracer dependent feature.

Some features are *application dependent*. It is the case of the content of the `<annotation>` element. It means that the content of the `<acmd>` element depends entirely from the application but not from the solver nor the tracer. A tool should not rely on the form of its content.

Some features are *solver dependent*. It is the case for example of the awakening conditions, the scheduling actions, the solver events generated by a `<reduce>` or the `vinternal` (resp. `cinternal`) representation of the variable (resp. constraint). A tool should not rely on the form or the order of these features. They should be taken "as is".

Event This work inherits from two areas, constraint solving and debugging, which both use the word "event" in correlated but different meanings: there are solver events and trace events respectively.

Solver events are defined in Sec. 3.1.3. They are produced by the solver and have to be collected to be used for constraints awakening. They characterize the dynamic modifications of the constraint variable domains (e.g. the update of the domain bounds of a variable);

A trace is composed of *Trace events*. It is a sequence of events reflecting the processor activity and allowing the observation of its behaviour. Each *trace event* corresponds to an execution step which is worth reporting about. If a trace may have unlimited number of trace events, all are instance of a finite set of event types (also called *port*. The description of a tracer consists of the description of the finite set of all event types.

Port Trace event types (see the above Event definitions) are also called *ports*. In practice the *port* is the name of the trace event type and it is the first attribute of a trace event.

Recomended Some features are partially described in the text. It is the case of many recomended attribute values. The recommendations frequently concern context sensitive conditions or semantics are are mandatory. See 2.2.5 for more details.

Index