A Generic Visualization Platform for CP

Helmut Simonis[1] and Paul Davern[1] and Jacob Feldman[1] and Deepak Mehta[1] and Luis Quesada[1] and Mats Carlsson[2]⋆

[1] Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
[2] Swedish Institute of Computer Science
SICS AB,Uppsala Science Park, SE-751 83 Uppsala, Sweden
contact: `h.simonis@4c.ucc.ie`

Abstract. In this paper we describe the design and implementation of CP-VIZ, a generic visualization platform for constraint programming. It provides multiple views to show the search tree, and the state of constraints and variables for a post-mortem analysis of a constraint program. Different to most previous visualization tools, it is system independent, using a light-weight, intermediate XML format to exchange information between solvers and the visualization tools. CP-VIZ is available under an open-source licence, and has already been interfaced to four different constraint systems.

1 Introduction

Visualization[3] is one of the best techniques for understanding the behavior of constraint programs, allowing us to directly observe the impact of changes by visual inspection instead of using tedious debugging. So far, most constraint visualization tools have been closely linked to specific solvers, making it difficult to compare alternative solvers and to reuse development effort spent on other systems. Previous attempts [4] at generic tools did not find widespread use largely due to the complexity of the specification and the level of detail captured. The new, light-weight CP-VIZ system provides a simple XML based interface for solvers, and can be easily extended for new systems and constraints. In CP-VIZ, we try to visualize the search tree and the state of variables and (global) constraints in parallel views. The search tree shows choices, assignments and failures, modeled on the tree display in the Oz Explorer [13] and later in CHIP [15]. Constraints and variables are shown in a 2D layout defined by the user, individual global constraints are shown in custom visualizations similar to [16]. A new constraint can be added to the package by simply deriving a new class with a custom drawing method. The design of the visualization tool was driven by user-requirements, coming mainly from the development of an ECLiPSe ELearning course.

---

[3] Visualization relies heavily on the use of colors, with a potential loss of information if seen in black&white only. An on-line version of the paper with colored diagrams can be downloaded from the URL `http://4c.ucc.ie/~hsimonis/cpviz.pdf`.

2 Related Work

Visualization has played a significant role in demonstrating the use of constraint programming, and helping to develop successful applications. Early systems like CHIP [6] relied on Prolog-based coroutines to visualize the assignment of variables changing throughout a search process. Visualizations were written as application specific tools which were co-developed with the constraint model. This approach restricted re-use of components and was tightly linked to a logic-programming host language. Meier [10] was the first to abstract visualization types based on collections of variables and to propose different views for them. The visualization of the search tree was pioneered in the Oz Explorer [13], its interactive use tightly linked to the generalized branching possibilities of the Oz environment. The DISCiPl project [5] produced a multitude of results for constraint debugging and visualization, the ones most relevant for this paper are the search tree tool for CHIP [15] and the idea of specialized visualizers for global constraints [16]. The French OADymPPaC project [12] considered a system independent view of visualization. But the XML-based specification for post-mortem traces was quite complex and achieved limited acceptance, and seems no longer to be actively maintained. The visualization tools for Comet [7] provide an environment for developing visualizations for constraint-based local search, which mix generic and application specific aspects of the visualization inside the modeling language.

3 Design Aims

The design of CP-VIZ was largely driven by the development of an ECLiPSe ELearning course [14], for which we wanted to be able to show and explain the solving of various models for application programmers. We did not want to restrict the use of the visualizer to ECLiPSe only, but rather tried to design a constraint system independent architecture. This led to a number of key design decisions:

- We decided to concentrate on post-mortem analysis, which minimizes the requirements of interaction between the constraint solver and the visualization environment, but still provides most of the information required for analysis.
- The output of the visualization can be studied on-screen, but can also be provided as high-quality, colored, vector based print output. Data feeds for other visualization tools are also provided.
- The tools are solver independent, written in a general purpose language (Java) and can be easily extended and specialized.
- We added invariant checking at each search node to the functionality, this allows a solver independent validation of the results, and can highlight missing propagation in individual constraints.
- The system is platform independent, and is provided as open source. The system specific XML generation requires minimal effort.

At the same time, these choices restricted some functionality that is provided in other visualization tools.

- The tool currently is not designed for interactive problem solving, controlling the search from within the visualization by making manual decisions on how the search should progress. To allow this in a system independent way seems quite difficult, and would require rather deep interaction with each solver. At the same time, there is limited evidence that such an interactive use helps application programmers in developing strategies which solve problems automatically.
- We are not considering individual propagation steps, showing in which order constraints are woken and how they make detailed domain restrictions. For most application programmers, this level of abstraction is too detailed, and it is too time consuming to follow the execution of constraint propagators through a longer search process.
- We don't collect and display the constraint graph. Many existing tools for displaying constraint graphs seem to work only for binary constraints, and heavily rely on graph layout algorithms, without finding usable displays for larger problem sizes.

Fig. 1. Sudoku: Consistency Level Comparison

Forward Checking   Bounds Consistency   Domain Consistency

Which type of design choices can be improved when using visualization? Figure 1 shows the well-known example of a Sudoku puzzle expressed in constraint programming, which compares three different consistency levels for the ALLDIFFERENT constraints in the model. The same problem is modeled using forward checking, bounds consistency and domain consistency; the pictures show the state after the initial set-up, before search is started. The variables are shown in form of a two-dimensional matrix, each cell corresponds to a variable, which shows the current values in the domain (small, in green) or the assigned value (large, in red). For this carefully selected, didactic example, different consistency levels lead to different amounts of propagation, but this is not universally true. In many applications the visualization can help to decide which consistency level to use in order to find the right compromise between speed, propagation and problem solving stability.
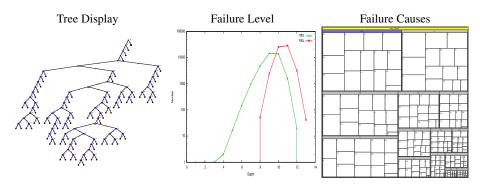
Fig. 2. Search Tree Analysis - Different Views of Search Tree Data

Tree Display   Failure Level   Failure Causes

Figure 2 shows different diagrams for visualization of the search tree. If the search space is small, the full tree can be shown (on the left). For more complex problems, this is no longer possible, and a more compact form, originally proposed in [13], which abstracts failed sub-trees, can be displayed (see Figure 7 for an example). But often this detailed analysis is not required, it suffices to have a simple quantitative analysis, as shown in the middle part of Figure 2. It plots the number of success and failure nodes with the depth of the search tree. The shape of the plot often is enough to understand how well a model is able to explore the search space. On the right we show a treemap visualization which indicates which value assignment for which variable led to an immediate failure during search. This can help to understand more clearly if the search strategy globally is making the right choices.

Finally, the diagrams in Figure 3 show an example where invariant checking was used to detect nodes in the search where the constraint propagation was not sufficient. The pictures are from a cumulative scheduling problem proposed by Robert Nieuwenhuis [11] solved in ECLiPSe. It highlights two problems with the CUMULATIVE constraint implementation of ECLiPSe, which is based on task intervals. In the left picture, some tasks in a partial assignment are restricted sufficiently so that obligatory parts (dark, in red) are generated. The sum of these obligatory parts exceeds the resource limit, which is not detected by the propagator. Invariant checking highlights the constraint and has also marked the problem in the search tree. On the right, a number of tasks have been assigned, and their resource profile reaches the resource limit, but the start times of unassigned tasks are not updated properly. This not only shows some missing propagation, but affected the search routine as well, as the heuristic for task selection picked the wrong tasks to be assigned next. The problem was resolved by developing another propagator for CUMULATIVE based on obligatory parts.

4 Architecture

Figure 4 shows the basic architecture of the CP-VIZ system. The visualization is driven by annotations in the constraint program. When run in the solver, two XML log files
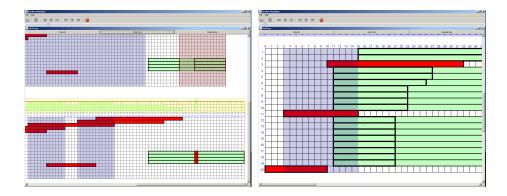
Fig. 3. Invariant Checks for Cumulative Scheduling Problem
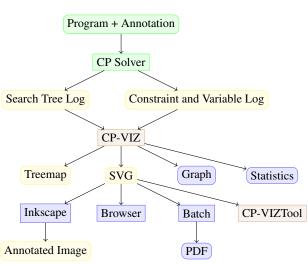

Fig. 4. CP-Vɪᴢ System Architecture

(one for the search tree, the other for the constraint and variable visualization) are produced. These files are then parsed in the main CP-VIZ application, producing graphical output as SVG, or as input for other tools (tree maps, graphs, statistics). The SVG output can be displayed interactively in the CP-VIZTOOL, or can be used in multiple ways to produce annotated or converted output for print or WEB media.

We use XML text files to link the generation of the log files to the creation of the visualization. This should allow almost any constraint programming system to be linked to the CP-VIZ visualization with minimal effort.

Search Tree Log. The log file consists of a single *tree* element, which contains a sequence of node elements which describe the search tree. There is an initial *root* node, which defines the start of the search, and *try* and *fail* nodes for successful and failed choices. In each node we have a node id, the node id of the parent node, the name of the variable currently assigned, the size of its domain, and the value assigned. A variant of these types also allows to handle arbitrary choices, not based on variable assignment. These alternatives can be useful to describe more complex branching schemes, but their analysis is slightly more restricted. A *solution* node is used to mark choice nodes which complete an assignment, i.e. to mark nodes where all constraints are satisfied. The format does not assume chronological depth first search, nodes can be added for any parent at any time.

Constraint and Variable Log. The second log file is used to describe snapshots of constraints and variables. Its top element is *visualization*, which contains a list of *visualizer* elements, describing the constraints and variables to be displayed. This is followed by a sequence of *state* elements, each containing a snapshot of the execution at a given time point. Inside each state, the *visualizer_state* elements describe the current state of a constraint or collection of variables. The syntax used roughly follows the syntax used in the global constraint catalog [3]. Constraints can be described by their named *arguments*, which may contain *collections* of basic types or *tuples*, which describe structures of disparate types. The basic types currently allowed are integers and finite domain variables, integer sets and domain variables over finite sets, plus some more specialized types.

4.1 System Dependent XML Generators

For every constraint system that wishes to use the CP-VIZ environment, we need to define an interface to generate the XML logs. Figure 5 shows such an interface for Java, based on two classes, *VisualSolver* and *VisualProblem*. The methods for the search tree log are contained in the *VisualSolver* interface, each adds or annotates a search node in the tree.

The methods for the *VisualProblem* class are split into two groups. The application programmer can use the method *register()* to register a constraint or a collection of variables with the visualization. There is also a method *snapshot()* which triggers the creation of a snapshot of all registered constraints and variables at a given program point. The snapshot is created by sending a *snapshot()* message to each registered constraint. This is then responsible for saving the current state of the constraint into the

Fig. 5. VisualSolver and VisualProblem Interface Definition

```java
public interface VisualSolver extends Visual {
        public void addRootNode(int id);
        public void addSuccessNode(int id, int parentId,
                String variableName, int size, int value);
        public void addSuccessNode(int id, int parentId,
                String variableName, int size, String choice);
        public void addFailureNode(int id, int parentId,
                String variableName, int size, int value);
        public void addFailureNode(int id, int parentId,
                String variableName, int size, String choice);
        public void labelSolutionNode(int id);
}

public interface VisualProblem extends Visual {
        public void register(Constraint constraint);
        public void register(Var var);
        public void register(Var[] varArray);
        public void register(Var[][] varMatrix);
        public void snapshot();

        // for implementors only
        public void startTagArgument(String index);
        public void startTagArgument(int index);
        public void endTagArgument();

        public void startTagCollection(String index);
        public void startTagCollection(int index);
        public void endTagCollection();

        public void startTagTuple(String index);
        public void startTagTuple(int index);
        public void endTagTuple();

        void tagVariable(Var var);
        void tagVariable(String index, Var var);
        void tagVariable(int index, Var var);

        void tagInteger(String index, int value);
        void tagInteger(int index, int value);
}
```

log. For this it might use the remaining methods of the *VisualProblem* class, which log
XML elements of different types for the constraint.

4.2 CP-VIZ

The main CP-VIZ application parses the XML log files and creates SVG output for
the user. The search tree is parsed completely before generation, while the constraint
and variable snapshots are handled one at a time. In order to see which changes have
occurred to the variables by the current search step, the tool keeps a stack of snapshots
for all parents of the current node in memory. This not only allows to see the domain
updates of the variables, but also permits to generate path based visualizations [15],
which display the evolution of a variable or some parameter through all parent nodes
from the root to the current node.

Internally, the CP-VIZ application uses an event-based SAX-2 XML parser, so that
it can minimize which part of the XML tree it needs to keep in memory. Experiments
have shown that log files of several hundred Mb do not pose any problems.

4.3 CP-VIZ tool

Figure 6 shows the CP-VIZTOOL, a Java application which displays the result of the
visualization on the screen. The application has a time-line at the top, where the user
can select a state of the execution for display. The tool will then display the state of
the search tree in the left main pane, and the corresponding snapshot of the constraint
and variable visualization in the right pane. The user can also step forward/backwards
through the execution, or display the complete solution process as a movie, progressing
automatically through the different snapshots.

5 Invariant Checking

By providing snapshots of the execution at fix points only, when all constraints have
performed their consistency checking, CP-VIZ also provides data for systematic test-
ing of execution traces. We have implemented an invariant checker, which for ev-
ery snapshot calls an *invariant()* method for each registered constraint. This method
may return *TRUE*, also the default value, or one of the values *INTERESTING*, *MISS-
ING_PROPAGATION*, *INCONSISTENT* or *FALSE*. Aggregating over all invariant checks
for a snapshot, the visualizer then marks the node in the search tree accordingly and
highlights any failed assertions in the constraint visualization. We explain the mean-
ing of the values for the example of a CUMULATIVE [1] constraint. The CUMULATIVE
constraint states that the resource consumption of a collection of n tasks with start times
s_i, fixed duration d_i and resource use r_i must stay below the resource limit l and within

Fig. 6. Interactive CP-Viz Tool for Car Sequencing Problem

the scheduling period p. A ground solution must satisfy the equations

$$\forall\, 0 \leq t < p: \qquad \sum_{\{i\ \mid\ s_i \leq t < s_i + d_i\}} r_i \leq l \tag{1}$$

$$\forall\, 1 \leq i \leq n: \quad s_i + d_i \leq p \tag{2}$$

$$\sum_{1 \leq i \leq n} d_i * r_i \leq l * p \tag{3}$$

If for a ground instance one of these equations is not satisfied, then the invariant checker will return *FALSE*.

We can rewrite constraint (3) to consider upper bounds on domain variables l and p. This produces

$$\overline{p} \geq \left\lceil \frac{\sum d_i * r_i}{\overline{l}} \right\rceil \tag{4}$$

If in any snapshot this invariant does not hold, then the snapshot is inconsistent, i.e. the constraint propagator should have failed for this node. The invariant checker returns *INCONSISTENT*. A weaker invariant checks the lower bound of p instead:

$$\underline{p} \geq \left\lceil \frac{\sum d_i * r_i}{\overline{l}} \right\rceil \tag{5}$$

If this invariant is violated, the lower bound of p has not been updated correctly, but other values in the domain of p might satisfy the condition, so the invariant checker

returns *MISSING_PROPAGATION*. In a similar way we can derive

$$\forall\, 0 \leq t < p: \quad \bar{l} < \sum_{\{i\,|\,\overline{s_i} \leq t < \underline{s_i}+d_i\}} r_i \Rightarrow INCONSISTENT \tag{6}$$

$$\forall\, 0 \leq t < p: \quad \underline{l} < \sum_{\{i\,|\,\overline{s_i} \leq t < \underline{s_i}+d_i\}} r_i \Rightarrow MISSING\_PROPAGATION \tag{7}$$

We are cumulating the resource use over all obligatory parts, i.e. time periods where we know that a task will be active.

The weakest value is *INTERESTING*, which can be used to mark snapshots where a constraint detects a special condition that the user is interested in. We will show its use in section 6.3 to mark nodes where no propagation of a global method was possible. One key advantage of an invariant checker inside a system independent tool is that the invariant code can be shared for all constraint systems that implement a given constraint. As it is written independently from any specific propagation methods, it avoids problems when reused, buggy code in the validation precludes detection of an error. Finally, the invariant checks enhance chances to detect subtle differences in the declarative meaning of a global constraint between systems.

6 Implementation

In this section we discuss the platforms which currently have been integrated with the CP-VIZ tool set, and note some details of the effort required to connect a new system to the visualizer.

6.1 ECLiPSe

We have linked the finite domain *ic* library of the Prolog based ECLiPSe system to CP-VIZ as part of the ECLiPSe ELearning course development. The main requirement was to display sufficient information of the constraint propagation to the students, without overwhelming them with unwanted detail. As custom search routines are very easy to write in ECLiPSe, we also needed an interface which could visualize such routines with minimal overhead. The current interface does not require hooks in the predefined search routines or constraint implementations, but rather uses logic programming features to express the visualization as annotations of the user programs. Visualizers for some 15 global constraints have been implemented so far, together with a series of example programs, based on the course material.

6.2 SICStus

We are currently extending the `clpfd` library module of SICStus Prolog with exported predicates producing XML files for CP-VIZ. The work is being carried out as a port of the interface code for ECLiPSe. Like for ECLiPSe, we do not use any special hooks of SICStus Prolog or its `clpfd` library module and we rely on user program annotations. The implementation replaces the normal `labeling/2` procedure, which takes a list

of domain variables, by another procedure taking a list of domain variables annotated with information for display purposes, e.g. variable name. ECLiPSe and SICStus provide different libraries of global constraints, and so the main implementation effort lies in implementing visualizers for the global constraints not provided by ECLiPSe. In particular, we plan to implement 2D and 3D visualizers for the generic multi-dimensional *geost* constraint [2]. The XML files are currently being written with standard Prolog I/O predicates. For efficiency, this is likely to be replaced later by specific XML I/O code.

6.3 Visualization of the Global Constraint SOFTPREC

As a case study for visualizing individual global constraints, consider the visualization of the SOFTPREC constraint arising in the context of the feature subscription problem for telecommunication services. A feature subscription problem is a configuration problem defined by a set of possible features, a set of hard precedence constraints, a set of soft precedence constraints, and a function that maps each feature and each soft precedence constraint to a non-zero integer weight. The objective is to maximize the value of the subscription, which is defined to be the sum of the weights of the features and soft precedences that are included. The soft global precedence constraint SOFTPREC is proposed for solving the feature subscription problem in [9] and [8]. It holds if and only if there is a strict partial order on the selected features subject to the relevant hard precedence constraints and the selected soft (user) precedence constraints, and the value of the subscription is within the provided bounds.

The algorithms for the pruning rules of SOFTPREC have been implemented in Choco (http://www.emn.fr/z-info/choco-solver), which is a Java library for constraint programming. In order to visualize the search tree and the propagation carried out at each node of the search tree, the implementation of SOFTPREC was extended in order to generate and save the trace in the CP-VIZ format. The implementation of this extension was fairly simple. We had to extend two classes of Choco and override some of their methods. While generating the required data for visualization is quite easy, deciding what to visualize and how to visualize it required several iterations.
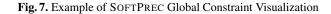
A distinct advantage of the visualizer is that it is easy to get a sense of the solutions. Visualizing solutions can give more insight than just knowing the numerical value of the solution. It can help a user in deciding whether a given solution with the optimal value is really optimal for him/her or not. The arguments of SOFTPREC that an end-user might be interested in visualizing are the states of the variables associated with optional features, soft precedences, and the value of the subscription being computed. Some of the interesting states are whether a feature (or a user precedence) is included, excluded or undecided, or whether the current state is a result of the last choice or the previous choices.

Figure 7 (left) depicts the search tree (generated by a branch and bound search algorithm) explored until the node number 38 and the states of the variables (after constraint propagation) at that node, when solving an instance of feature subscription with 20 features and 10 user precedences. In Figure 7 leaf-nodes of the search tree corresponding to feasible solutions are shown in green, dead-ends are shown in red, and the current node (node number 38) is shown in blue. Figure 7 (right) visualizes the state of the variables after reaching the fix point completing the propagation at node number 38.

The states of the features are visualized using a vector of cells (shown at the bottom and to the right). When a feature is undecided the corresponding cell is unlabeled. If a feature is included or excluded then the cell is labeled with either 1 or 0. The difference between the features that are included/excluded in the current node from those that are decided in the earlier nodes is made through the difference in the background color of the cells. The states of the variables associated with the soft precedence constraints are visualized through a matrix of cells. Each soft precedence constraint $i \prec j$ is associated with a cell in row i and column j. If a soft precedence $i \prec j$ holds then the corresponding cell is labeled with 1 and if it is violated then the corresponding cell is labeled with 0, and undecided soft precedence is labeled with 01. The bounds of the value of the subscription being computed is displayed on the left bottom of Figure 7 (right).

SOFTPREC internally maintains transitivity on the hard precedence constraints. Here, a hard precedence constraint, $i \prec j$, means that if features i and j are included then i must precede j. From a developer's point of view it is interesting to visualize the states of these variables, which is done through the background colors of the cells in the matrix. SOFTPREC also elicits and maintains incompatibilities between undecided features through the states of these variables. An incompatibility between undecided features i and j is visualized by placing a box around the cell in row i and column j. For example, in Figure 7 (right) the cell in the second row and fifth column is surrounded by a red box which denotes that feature 2 and feature 5 are incompatible. This helps in seeing patterns in the incompatibilities between pairs of features. Within SOFTPREC bounds are computed by associating a graph with a set of incompatibilities, and computing the violation cost of each component of the graph. The components are also visualized by using different colors for the incompatibilities of different components. When it comes to describing the pruning rules of SOFTPREC it is much easier to explain them through visualization. Initially it was agreed to implement a static variable ordering for SOFTPREC that chooses variables associated with soft precedences before the variables associated with features. However, after visualizing the search tree, we discovered that the intended variable ordering was not implemented in the right way. Another advantage of visualization is that it can help in understanding the impact of the strength of different pruning rules.

6.4   JSR331

The Java Specification Request (JSR) 331 (`http://jcp.org/en/jsr/detail?id=331`) is a working group in the Java Community Process trying to propose an open, standard constraint programming API for Java. As part of a reference implementation we have considered the use of CP-VIZ as an example of a visualization extension for the standard API. Figure 8 shows a code example for an annotated N-queens program in the proposed standard syntax. For most classes (Problem, Solver, individual global constraints), variants which incorporate the visualization capabilities of CP-VIZ are provided. By creating for example a new constraint from ALLDIFFERENTVISUAL instead of ALLDIFFERENT , a visualization for this constraint will be provided. Note that not all constraints and variables need to be annotated, the user can concentrate on only parts of the model, if required. Figure 9 shows an UML sequence diagram for the in-
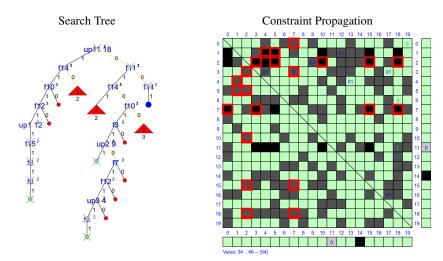
Fig. 7. Example of SOFTPREC Global Constraint Visualization

Search Tree   Constraint Propagation

Fig. 8. JSR 331 Example: Visualization of N-Queens Problem

```java
public class QueensVisual {
        public static void main(String[] args) {
                ProblemVisual problem = new ProblemVisual("Queens");
                int size = 16;
                problem.startVisualization("QueensProblem.log");
                Var[] x = problem.varArray("x",0, size-1, size);
                Var[] x1 = new Var[size];
                Var[] x2 = new Var[size];
                for (int i = 0; i < size; i++) {
                        x1[i] = x[i].add(i);
                        x2[i] = x[i].sub(i);
                }
                problem.register(x);
                new AllDifferentVisual(x).post();
                problem.snapshot();
                new AllDifferent(x1).post();
                new AllDifferent(x2).post();
                SolverVisual solver = new SolverVisual(problem);
                solver.startVisualization("QueensSolver.log");
                Solution solution = solver.findSolution();
                solver.stopVisualization();
                problem.stopVisualization();
        }
}
```

teraction of the Application, VisualProblem, VisualSolver and Constraint classes which shows how the JSR331 implementation builds on the interface of Figure 5.
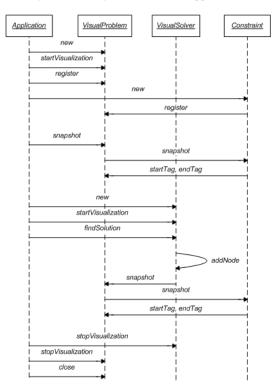
Fig. 9. UML Sequence Diagram - Message Flow between Application and Visualization Classes

7 Future Work and Conclusions

While the current CP-Viz system already provides many useful features for understanding and improving constraint programs, there are a number of features that would improve its capabilities:

- At the moment the system can tell the user which choice led to a failure, but can not provide a more detailed explanation. It would be helpful if we can integrate some explanation tools which can provide automatically derived explanations of failures.
- Much of the development time for a constraint application is taken up with comparing different possible design choices. We will study how to best compare search trees and constraint and variable visualizations from multiple runs in a single display.

– The invariant checker provides a useful paradigm for concentrating effort on interesting parts of the search effort, but at the moment the checks are compiled as part of the tool itself. It might be interesting to allow users to specify checks interactively, and display such search results inside the visualization.

By providing an open-source, system independent visualization platform, CP-VIZ can help to reduce the amount of duplicated and redundant work required by system developers, while allowing specific, new features to be added without too much effort.

References

1. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling problems. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1993.
2. N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In C. Bessiere, editor, *Proc. CP'2007*, volume 4741 of *LNCS*, pages 180–194. Springer, 2007.
3. N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global constraint catalog. Technical Report T2005:08, SICS, May 2005.
4. Pierre Deransart. Main results of the OADymPPaC project. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 456–457. Springer, 2004.
5. Pierre Deransart, Manuel V. Hermenegildo, and Jan Małuszyński, editors. *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*. Springer, 2000.
6. Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. The constraint logic programming language CHIP. In *FGCS*, pages 693–702, 1988.
7. Grégoire Dooms, Pascal Van Hentenryck, and Laurent Michel. Model-driven visualizations of constraint-based local search. *Constraints*, 14(3):294–324, 2009.
8. David Lesaint, Deepak Mehta, Barry O'Sullivan, Luis Quesada, and N. Wilson. A Soft Global Precedence Constraint. In *IJCAI-09*, Pasadena, CA, USA, 2009.
9. David Lesaint, Deepak Mehta, Barry O'Sullivan, Luis Quesada, and Nic Wilson. Consistency techniques for finding an optimal relaxation of a feature subscription. In *Proceeding of the 20th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2008)*, pages 283–290, 2008.
10. Micha Meier. Debugging constraint programs. In Ugo Montanari and Francesca Rossi, editors, *CP*, volume 976 of *Lecture Notes in Computer Science*, pages 204–221. Springer, 1995.
11. Robert Nieuwenhuis. A cumulative scheduling problem. Personal Communication, 2008.
12. The OADymPPaC project. Public deliverables, 2004.
13. Christian Schulte. Oz Explorer: A visual constraint programming tool. In *ICLP*, pages 286–300, 1997.
14. Helmut Simonis. An ECLiPSe ELearning course. `http://4c.ucc.ie/~hsimonis/ELearning/index.htm`, 2009.
15. Helmut Simonis and Abderrahmane Aggoun. Search-tree visualisation. In Deransart et al. [5], pages 191–208.
16. Helmut Simonis, Abderrahmane Aggoun, Nicolas Beldiceanu, and Eric Bourreau. Complex constraint abstraction: Global constraint visualisation. In Deransart et al. [5], pages 299–317.