A Visualization Tool for Constraint Handling Rules

Slim Abdennadher and Matthias Saft
Computer Science Department
University of Munich
Slim.Abdennadher@informatik.uni-muenchen.de

Abstract

This paper presents a visualization tool, called VisualCHR, that supports the development of constraint solvers written in the high-level language Constraint Handling Rules (CHR). VisualCHR can be used to debug and to improve the efficiency of constraint solvers. It can also be used to understand the details of constraint propagation methods and the interaction of different constraints implemented by means of CHR. Thus, it is suitable for users at different levels of expertise.

VisualCHR offers a rich functionality to display, inspect, rearrange, and manipulate a graph interactively, including means to influence the granularity with which it is displayed and to compactify what is not in the user's current focus of interest.

1 Introduction

Constraint Logic Programming (CLP) [7, 8] combines the declarativity of logic programming with the efficiency of constraint solving. Recently, CLP has become a promising approach for solving combinatorial problems. It presents in many cases advantages over imperative programming or other declarative paradigms. Nevertheless, CLP is not as widely used as it should in industry. One of the factors that can probably make the use of CLP more propagated in industry is the availability of advanced programming environments which facilitate the development, debugging, and visualization of systems based on this paradigm.

In CLP, a problem can be expressed declaratively in terms of variables and constraints. The variables range over a set of values and typically denote alternative decisions to be taken. The constraints are expressed as relations over subsets over variables and restrict feasible value combinations for the variables. Constraints are handled by a constraint solver that has the task to collect, combine, and simplify the constraints, and detect their inconsistency. For many interesting constraint systems, constraint solving is incomplete, i.e. it can not detect inconsistency at all times. Constraint solving must be combined with search, which is used to assign values to variables.

Current constraint visualization tools focus on representing and analyzing the search tree of a constraint program [11, 9, 12]. There is a lack of intuitive interactive tools for visualizing the constraint solving part, i.e. the constraint simplification and propagation of constraints.

In this paper, we present a visualization tool, called VisualCHR, that supports the development of constraint solvers written in the high-level language Constraint Handling Rules (CHR) [4]. VisualCHR can be used to debug and to improve the efficiency of constraint

solvers. It also can be used to understand the details of constraint propagation methods and the interaction of different constraints implemented by means of CHR. Thus, it is suitable for users at different levels of expertise.

VisualCHR is a part of the Java library JACK that additionally consists of an implementation of CHR in Java and a generic search engine [1]. Alternatively, VisualCHR can import a graph that reflects an external inference engine's process, provided that this engine can produce a trace, a protocol of its computation steps in a defined format. Regardless whether a graph is internally produced or imported, VisualCHR offers a rich functionality to display, inspect, rearrange, and manipulate a graph interactively, including means to influence the "granularity" with which it is displayed and to compactify what is not in the user's current focus of interest.

The paper is organized as follows: In Section 2, we introduce CHR by example. Section 3 presents some functionalities of VisualCHR. In Section 4, we present some implementation issues of VisualCHR. Finally, we conclude with a summary and directions for future work.

2 Constraint Handling Rules

Constraint Handling Rules (CHR) [4] is a declarative high-level language extension especially designed for writing constraint solvers. With CHR, one can introduce *user-defined* constraints into a given host language. Most CHR libraries have been implemented in logic programming languages, e.g. Eclipse [5] or Sicstus Prolog [6]. Recently, we have provided a new constraint library for Java, called JACK (JAva Constraint Kit) [1]. This library consists of the high-level language JCHR (an implementation of CHR in Java), a generic search engine for JCHR to solve constraint problems, and the VisualCHR tool which we will present in this paper.

CHR consists of guarded rules with multiple heads. There are three kinds of rules: A simplification rule replaces constraints by simpler constraints while preserving logical equivalence. A propagation rule adds new constraints, which are logically redundant but may cause further simplification. A simpagation rule is a hybrid kind of rule.

Due to space limitations, we cannot give a formal account of syntax and semantics of CHR in this paper. An overview on CHR and JCHR can be found in [4] and [1], respectively. We introduce JCHR by the following example that will guide us through the paper. We define a user-defined constraint for less-than-or-equal, `leq`, that can handle variable arguments. It uses the syntactical equality, `=`, and `true` as built-in constraints[1].

```
handler leq {
  class IntUtil;
  constraint leq(java.lang.Integer, java.lang.Integer);
  rules {
      variable java.lang.Integer X, Y, Z;
      { leq(X,X) } <=> { true }                  reflexivity;
      { leq(X,Y) && leq(Y,X) } <=> { X = Y }     antisymmetry;
      { leq(X,Y) && leq(Y,Z) } ==> { leq(X,Z) } transitivity;
      { leq(X, Y) &\& leq (X, Y) } <=> { true } idempotence;
      if (IntUtil.ground(X) && IntUtil.ground(Y))
           { leq(X, Y) } <=> {IntUtil.le(X, Y)} ground;
        }
  goal g1 {
```

[1]Built-in constraints are those handled by an already existing, predefined constraint solver.

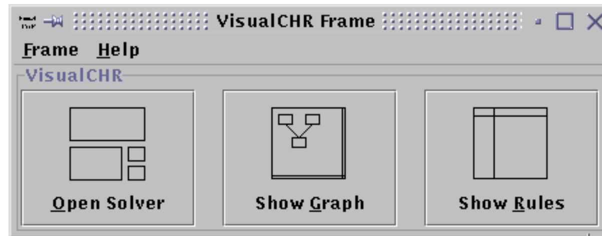

Figure 1: VisualCHR Start Screen

```
variable java.lang.Integer X, Y, Z;
leq(X, Y) && leq(Z, X) && leq(Y, Z)
      }
        }
```

The first line states that this is the definition of the solver `leq`. In the declaration section, the constraint `leq` is defined by the keyword `constraint`. The constraint `leq` expects two arguments of the type `java.lang.Integer`. In the rule section, three variables `X`, `Y` and `Z` of the type `java.lang.Integer` are declared. They are only used by the rules defined in the rule section. The rule section implements reflexivity, antisymmetry, transitivity, idempotence, and a ground rule. The reflexivity rule states that `leq(X,X)` is logically true. Hence, whenever we see the constraint `leq(X,X)` we can simplify it to `true`. The antisymmetry rule means that if we find `leq(X,Y)` as well as `leq(Y,X)` in the current store, we can replace them by the logically equivalent `X=Y`. The transitivity rule propagates constraints. It states that the conjunction `leq(X,Y), leq(Y,Z)` implies `leq(X,Z)`. Operationally, we add the logical consequence `leq(X,Z)` as a redundant constraint. The idempotence rule absorbs multiple occurrences of the same constraint. It can be expressed by a simpagation rule. The `ground` rule states that if the values of `X` and `Y` are known then the constraint `leq(X,Y)` can be replaced by the Java method `IntUtil.le(X,Y)` which is provided by a class `IntUtil`. In the goal section, the goal `leq(X,Y),leq(Z,X),leq(Y,Z)` is stated. The first two constraints cause the transitivity rule to fire and add `leq(Z,Y)`. This new constraint together with `leq(Y,Z)` matches the head of the antisymmetry rule. So the two constraints are replaced by `Y=Z`. The built-in equality is applied to the rest of the goal, `leq(X,Y),leq(Z,X)`, resulting in `leq(X,Y),leq(Y,X)`. The antisymmetry rule applies resulting in `X=Y`. The goal contains no more inequalities, the process stops and the result of the goal is `X=Y,Y=Z`.

3 VisualCHR

3.1 How to run VisualCHR?

VisualCHR can be run as an applet or application. If VisualCHR was started successfully, the start screen shown in Figure 1 appears.

With the applet, only the given constraint solvers can be debugged. For debugging one's own constraint solvers, VisualCHR has to be executed as application. To load a solver one clicks on *Open Solver*. If the applet was used, the dialog box *Solver Selection Frame* appears as in Figure 2.

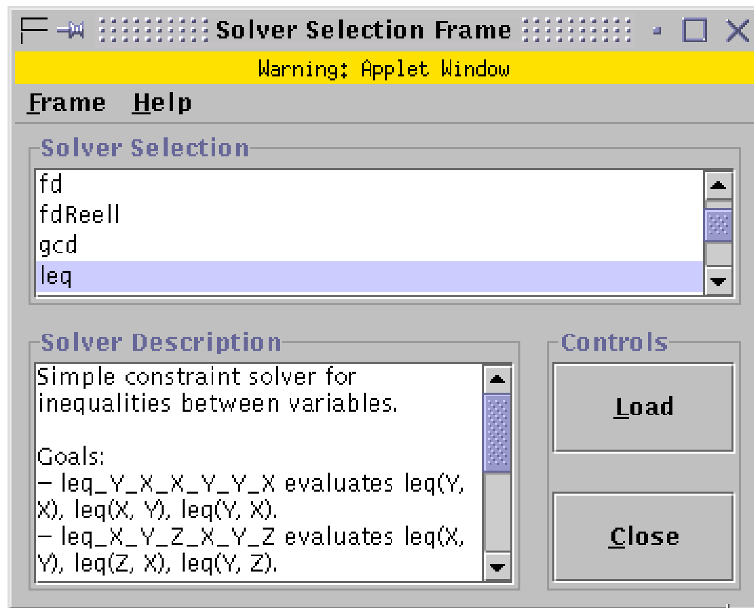

Figure 2: Loading a Solver in the Applet

From the shortlist *Solver Selection* one can select an example solver by clicking on it. In the field *Solver Description* a short description of the selected solver will appear. The selected solver can be loaded by clicking on *Load* or a double click in the shortlist. The *Load* button changes to *Show Source*. A click on this button or onto the selected solver in the shortlist will show the source code of the solver in readonly mode (cf. Figure 3).

By clicking *Show Rules* in the main window (Figure 1) a window called *Rules Frame* is opened (Figure 4). All rules belonging to the current JCHR constraint solver will be displayed together with their names. Note that the display does not show the actual source code, but rather a normalized version of it as produced by the JCHR compiler.

When clicking the button *Show Graph* in the main window (Figure 1), the *Graph Frame*

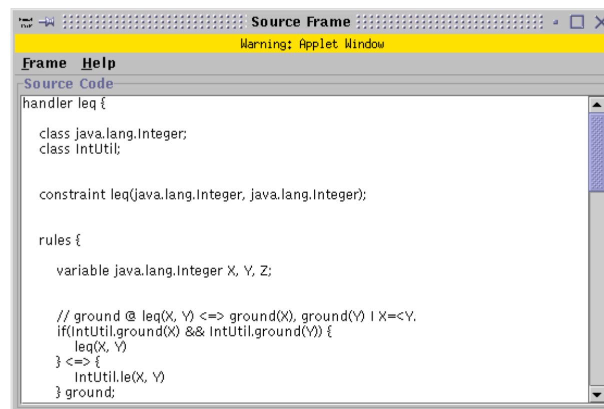

Figure 3: Display of the Source Code of a JCHR Solver

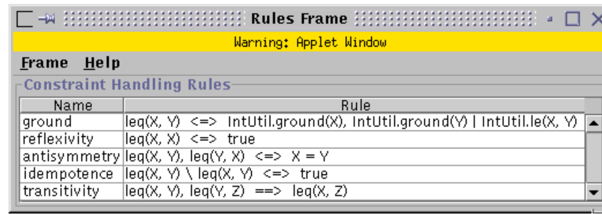

Figure 4: Display of the Rules

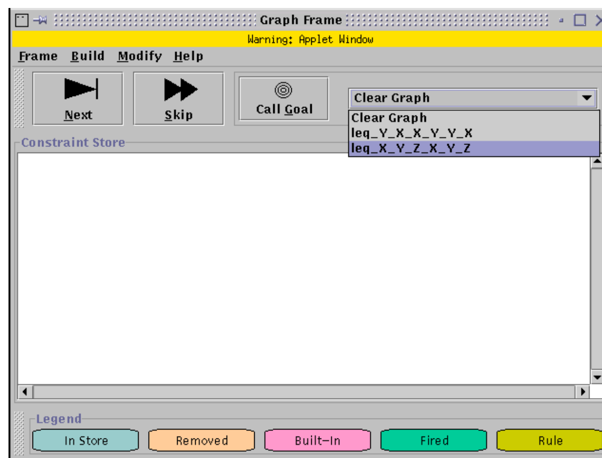

Figure 5: Graph Frame

(Figure 5) appears. The control and the legend bar can be pulled out of the windows to allow for more space for the graph.

The control bar contains a selection menu for goals. In the applet version, this list is predefined. In the application version, a goal can be entered manually. After starting the goal, the constraints are inserted into the constraint store, and thus into the graph, too. *Next* carries out the next evaluation step if evaluation is not yet completed. *Skip* skips over the step by step evaluation and completely executes the selected goal. After clicking the *Skip* button, this is transformed into *Pause*. If it is clicked, the automatic evaluation stops and the button is transformed again into *Skip*. In order to restart the selected goal one has to click on *Call Goal*.

3.2 Visualization of the Constraint Propagation

In constraint programming, the constraint store stores information about variables expressed by constraints and the constraint solver tries to simplify the store by constraint propagation and simplification. The visualization of the constraint propagation depends on the representation of the store. On one hand, a constraint store can be represented by a set of sub-boxes, where each sub-box consists of only one constraint. We call such representation *sub-box view*. On the other hand, a constraint store can be represented graphically by a box consisting of all its constraints. We call such representation *box view*.

To distinguish between constraints which are removed by simplification rules and con-

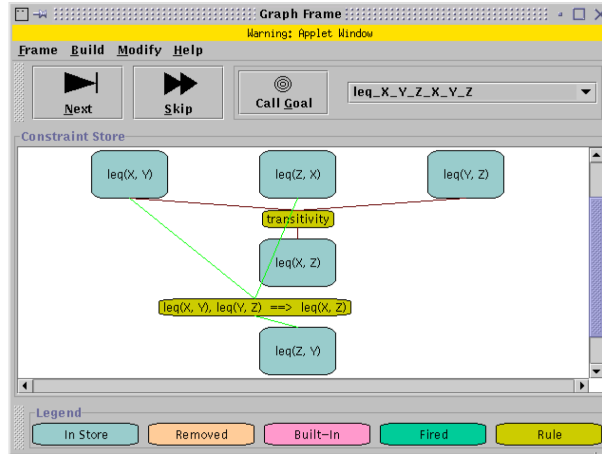

Figure 6: Sub-Box View

straints remaining in the constraint store, we use different colors, i.e. orange for removed constraints and blue for constraints which remain in the constraint store.

3.2.1 Sub-Box View

In Figure 6, the goal `leq(X,Y), leq(Z,X), leq(Y,Z)` is represented in a sub-box view.

Every large node in the graph stands for an individual constraint. These nodes are called *constraint nodes*. The small nodes represent the rules and include their names. They are called *rule nodes*. A mouse click toggles between the display of the rule name and the display of the actual code of the rule.

A rule node connects the constraint nodes which are involved in the application of a CHR rule: The constraints to which the rule is applied lead to the rule, and from the rule there are edges to the constraints that are added by the rule. If a constraint was removed by the rule, the connecting edge is blue. If built-in constraints were applied for firing a rule, the edge is gray.

In Figure 6, the first row shows the goal constraints `leq(X,Y), leq(Z,X)` and `leq(Y,Z)` inserted into the constraint store. In the first step, the transitivity rule was applied to the constraints `leq(X,Y)` and `leq(Y,Z)` and therefore the new constraint `leq(X,Z)` has been generated. The two constraints `leq(X,Y)` and `leq(Y,Z)` remain in the constraint store. In the second step the transitivity rule was applied, this time to the constraints `leq(X,Y)` and `leq(Z,X)`, which remain in the store, too. The new constraint `leq(Z,Y)` is added.

In Figure 7, the third step shows the application of the antisymmetry rule applied to the constraints `leq(Z,X)` and `leq(X,Z)`. These two user-defined constraints are removed by the rule application from the constraint store. The new built-in constraint `Z=X` is added.

In the next evaluation step, the antisymmetry rule is applied to the two constraints `leq(X,Y)` and `leq(Y,Z)`. These two constraints are removed by the rule application and the built-in constraint `X=Y` is inserted. Figure 8 shows the state at the last evaluation step. The reflexivity rule is applied to the constraint `leq(Z,Y)`. This constraint is removed and the built-in constraint `true` is inserted. The application of this rule is possible here since `Z=Y` holds due to `Z=X` and `X=Y`. All user-defined constraints are now marked as removed and only

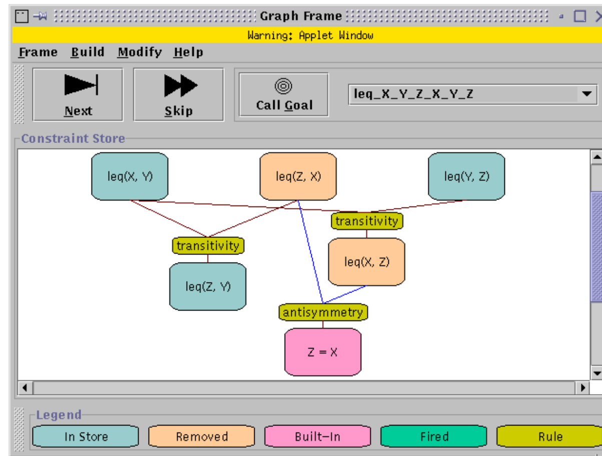

Figure 7: Further Evaluation Step

the built-in constraints `Z=X`, `X=Y` and `true` remain. No more rule is now applicable and the evaluation terminates. The solution is `Z=X`, `X=Y`.

3.2.2 Box View

In Figure 9, the goal `leq(X,Y), leq(Z,X),leq(Y,Z)` is represented in a box view. The entire contents of the constraint store after each evaluation step is represented as an individual node, called *store node*. Since all constraints present at one time are shown in own large node, the graph is just a chain of constraint stores and rule applications. Application of rules induces a dependency relationship between the constraints in the constraint store. This relationship can be displayed by marking one or more constraints which cause a rule to fire with a different color. Constraints to which a rule was applied are shown in color *Fired*.

3.3 Hiding Nodes

Typical graphs and trees are too complex to be handled conveniently by the techniques described so far. They require means to change (temporarily) their structure, such that the user sees a compactified version abstracting from details that are currently irrelevant. VisualCHR offers the concept of hiding nodes representing either the constraints or the rules.

3.3.1 Hiding Store Nodes

If one clicks with the right mouse button onto a store node, a popup menu appears. If *Hide This Store* is selected, the clicked node is collapsed to a small hexagon. *Hide Recursive Backward* or *Hide Recursive Forward* also collapses all previous or subsequent store nodes, respectively.

If one clicks with the right mouse button onto a collapsed node, another popup menu appears (see Figure 9). *UnHide This* will unhide all the collapsed store nodes. *UnHide First* or *Un Hide Last* only unhide the first or last hidden node, respectively. *Hide Recursive Backward* and *Hide Recursive Forward* work as before. With *Hide Previous* or *Hide Next* the previous or next node of the clicked node can be hidden.

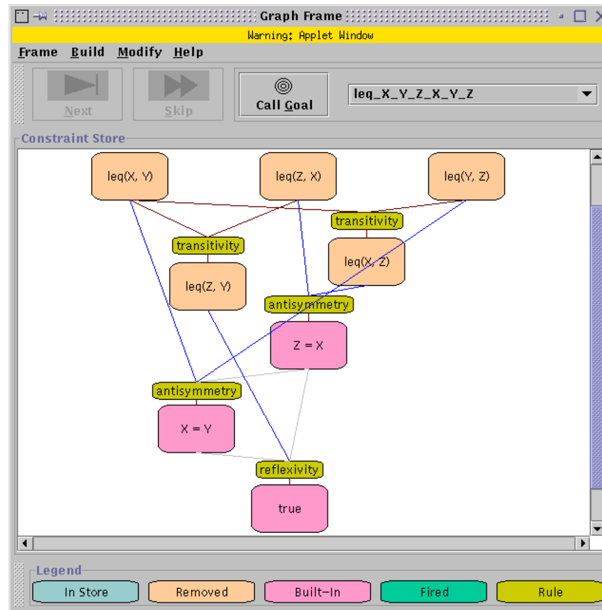

Figure 8: Last Evaluation Step

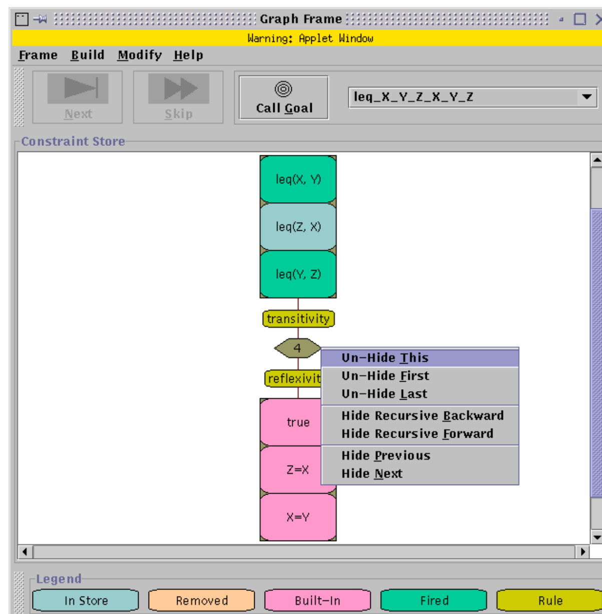

Figure 9: Box View

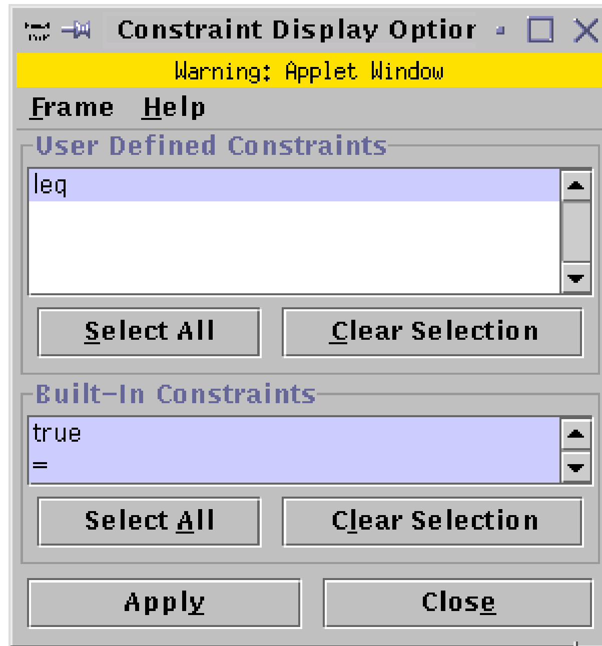

Figure 10: Hiding Constraint Nodes

3.3.2 Hiding Constraint Nodes

By selecting *Constraint Display Options* for hiding constraint nodes in the menu *Modify*, a dialog box as in Figure 10 appears. It consists of two lists, one for user-defined constraint symbols, the other for the built-in constraint symbols.

One can also click with the right mouse button directly onto constraint nodes in the graph to hide them. A popup menu will appear with the possibilities to *Hide This Constraint* or to *Hide All Constraints* with the same symbol.

3.3.3 Hiding Rule Nodes

Analogous to hiding constraint nodes, rule nodes can be hidden as well. Rules, like constraint nodes, can be hidden in both the sub-box view and the box view. When a rule is hidden, also all the constraints it produced are hidden.

The menu *Modify* contains the item *Rule Display Options*. When it is selected, a dialog box as in Figure 11 will appear. Only rules will be shown, whose names have been selected in the list *Rule Names*.

If one clicks with the right mouse button onto a rule node in the graph, a popup menu appears. Options are Hide This Rule and Hide All Rules.

4 Implementation Issues

VisualCHR is implemented in Java. The implementation is divided into two parts:

- Laying out and drawing the graph. That includes support for scaling the graph, as well as support for hiding and unhiding of nodes.

Figure 11: Hiding Rule Nodes

- The user interface which provides for menus, cursor control, status bar, ...

The user interface is implemented using Swing [3].

The method for computing the layout is still primitive. However, the user has the possibility to interact and to change the layout manually. We first considered to use existing tools for drawing layouts for graphs, e.g. the graph visualization system *daVinci* [10]. Unfortunately, it is hard to design a powerful user interface since the tools have a user interface on their own which can be customized in a limited fashion only. Nevertheless, the method for method computing and drawing the graphs has to be improved by using more sophisticated methods, e.g. [3].

5 Conclusion

We have presented an interactive tool, called VisualCHR, to visualize the propagation and simplification of constraints. VisualCHR is used to debug and to improve the efficiency of constraint solvers written in the high-level language Constraint Handling Rules. It can also be used to understand the details of constraint propagation methods and the interaction of different constraints.

VisualCHR has been tested with several classical constraint solvers, e.g. finite domains, Booleans, linear polynomials and interval arithmetics. It has helped to understand and to debug these solvers.

VisualCHR comes with a built-in inference engine, but it can also be hooked to external inference engines. It is accessible through the World Wide Web:

`http://www.pms.informatik.uni-muenchen.de/software/jack/`

Currently, we are trying to provide a plug-in mechanism for changing the representation of the constraints. For example, cumulative constraints [2] in a scheduling application could be represented as Gantt-charts, reflecting their role in concrete application. VisualCHR is a part of a constraint library called JACK (Java Constraint Kit). This library consists of the

high-level language JCHR, a generic search engine for JCHR to solve constraint problems and VisualCHR. A direction for future work will be the design of an interaction between VisualCHR and a visualization tool for search trees.

References

[1] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. Jack: A java constraint kit. In *International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, 2001.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[3] R. Eckstein, M. Loy, and D.Wood. *Java Swing.* O'Reilly, 1998.

[4] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

[5] T. Frühwirth and P. Brisset. High-level implementations of constraint handling rules. Technical report, ECRC, 1995.

[6] C. Holzbaur and T. Frühwirth. A prolog constraint handling rules compiler and runtime system. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 2001.

[7] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20, 1994.

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

[9] M. Meier. Debugging constraint programs. *Lecture Notes in Computer Science*, 976:204–221, 1995.

[10] Group of Prof. Dr. Bernd Krieg-Brückner. The graph visualization system davinci. `www.informatik.uni-bremen.de/daVinci/`.

[11] C. Schulte. Oz Explorer: A visual constraint programming tool. In *Proceedings of the Fourteenth International Conference on Logic Programming*, 1997.

[12] H. Simonis and A. Aggoun. Search-tree visualisation. In *Analysis and Visualization Tools for Constraint Programming.* LNCS 1870, Springer Verlag, 2000.