JACK: A Java Constraint Kit

Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, Matthias Schmauss

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

Abstract. Most existing libraries providing constraint facilities are embedded in the logic programming language, Prolog, or in the object-oriented language, C++. Recently, some proposals have been made to integrate constraint handling in Java. The goal of this work is to provide a new constraint library for Java, called JACK. It consists of a high-level language for writing constraint solvers, a generic search engine and a tool to visualize the simplification and propagation of constraints.

1 Introduction

The constraint programming technology has matured to the point where it is possible to isolate some essential features and offer them as libraries or embedded cleanly in general purpose host programming languages. At the moment, most constraint systems are either extension of a programming language (often Prolog), e.g. Eclipse, or libraries which are used together with conventional programming languages (often C or C++), e.g. ILOG Solver. Due to the growing popularity of Java and the possibilities of the Internet, there is a big interest to provide constraint handling in Java to implement application servers, e.g. for planning or scheduling systems.

Recently, several proposals have been done to combine the advantages of constraint programming with the advantages of the programming language Java.

- Declarative Java (DJ) [15] provides syntax extensions to Java to support constraint programming. DJ is especially designed to simplify the process of GUI's and Java applets.
- JSolver [3] is a Java library that provides classes to built constraints and strategies to solve these constraints. Thereby it is possible to use variables of the types integer and boolean.
- The Java Constraint Library (JCL) [13] provides several algorithms to solve binary constraint satisfaction problems.

In this paper, we propose a new Java library providing constraint programming features. The library is called JACK (<u>JA</u>va <u>C</u>onstraint <u>K</u>it) and consists of three parts:

- JCHR (<u>J</u>ava <u>C</u>onstraint <u>H</u>andling <u>R</u>ules): A high-level language to write application specific constraint solvers

– VisualCHR: An interactive tool to visualize JCHR computations
– JASE (Java Abstract Search Engine): A generic search engine for JCHR to solve constraint problems

The paper is organized as follows. In Section 2, we present the syntax and semantics of JCHR. In Section 3, we present the visualization tool of the JCHR computations. Section 4 introduces the Java abstract search engine JASE. Finally, we conclude with a summary and directions for future work.

2 Java Constraint Handling Rules

Constraint Handling Rules (CHR) [5] is a high-level language especially designed for writing constraint solvers either from scratch or by modifying existing solvers. CHR allows to specify and implement both propagation and simplification for user-defined constraints using rules. With CHR one can introduce these constraints into a given host language. Most CHR libraries have been implemented in logic programming languages, e.g. Eclipse [6] or Sicstus Prolog [7]. JCHR is an implementation of CHR in Java.

2.1 Syntax of a JCHR Solver

A JCHR constraint handler (also called constraint solver) is introduced by the keyword handler followed by the name of the handler and the code of the handler written in curly brackets (blocks as known from Java):

```
handler leq {
          ...
            }
```

A JCHR constraint handler consists of three sections: declarations, rules and goals (in that order). Goals for constraints are optional, while a handler without declaring constraints and rules for them would not make much sense. There are two ways of using a constraint handler written in JCHR: Calling it from Java or running it stand-alone using goals. The former is usually the case in full-fledged applications, while the latter is helpful for testing and for small examples that do not require search (the JASE library, see Section 4). When used from Java, the goals of the constraint handler will be ignored. Variables that appear in constraints are called logical variables. Logical variables and class instances must be declared at the beginning of the rules section and at the beginning of each goal in the goals section.

JCHR Declarations In the declarations section, Java classes are imported and the signatures of the constraints are declared. The Java classes will be needed in the signatures and the code of the rules or goals. The constraints will be implemented in the rules section. As in Java, each declaration is finished by a semicolon. A class import is defined by the keyword class followed by the class name as it can be found in the class path. All classes used in the following code need to be imported, including the classes mentioned in the constraint signatures. A constraint is declared by the keyword constraint followed by the name of the constraint and its argument types (much like a Java method):

```
handler leq {
   class java.lang.Integer;
   class IntUtil;
   constraint leq(java.lang.Integer,
                  java.lang.Integer,
                  java.lang.Integer);
```

Rules In the rules section, first the variables and class instances are declared and then the rules that simplify the constraints are implemented. Variables are defined by the keyword `variable` followed by a type and variable names:

```
handler leq {
       rules {
            variable java.lang.Integer X, Y, Z;
             ...
            }
           }
```

The rules describe the propagation and simplification of constraints. As in other CHR libraries, there are three kinds of rules: A simplification rule is of the form

```
if Guard { Head } <=> { Body } Name ;
```

A propagation rule is of the form

```
if Guard { Head } ==> { Body } Name ;
```

A simpagation rule is of the form

```
if Guard { Head1 &\& Head2 } <=> { Body } Name ;
```

We distinguish between *user-defined* and *built-in* constraints. User-defined constraints are those implemented by the rules, built-in constraints are those already provided by the JCHR library. The built-in constraints are `true` and `false`, the first always holds, the second never holds. Moreover, syntactical equality `=` is provided as a built-in constraint and can be applied to constants and logical variables, regardless of their type, as long as both arguments have the same type. If a method is called on the right hand side of the equality symbol `=`, the return type needs to be equal to the type of the object on the left hand side.

A rule has an optional name, `Name`, which is a Java identifier. Besides that, a rule consists of an optional guard, a head (left hand side) and a body (right hand side). These parts are all conjunctions using the infix operator `&&`. The head `Head` is a conjunction of user-defined constraints. The guard is optional. If present, the guard is a conjunction of built-in constraints and Java methods. If the guard is not present, it has the same meaning as the guard `true`. The body `Body` is a conjunction of user-defined constraints, built-in constraints and Java methods.

Goals Typically, a goal section exists if the constraint solver has to be run stand-alone. If the handler is used from Java, the goals are ignored. The goal section consists of one or more goals. Each goal has a name and is introduced by the keyword `goal`. A goal consists of declarations for the variables and class instances followed by the goal itself. A JCHR goal is a named conjunction of constraints and Java methods (like a rule body).

```
goal g1 {
     variable java.lang.integer X, Y, Z;
     leq(X, Y) && leq(Z, X) && leq(Y, Z)
        }

goal g2 {
      ...
        }
```

2.2 Semantics

In the current implementation, two different kinds of stores are used. One store contains user-defined constraints and the other contains built-in constraints. Note that Java methods are handled as built-in constraints.

Every time a user-defined constraint is activated (posted or woken), it checks itself the applicability of rules it appears in. Such a constraint is called (currently) *active*. All the other constraints in the constraint store are called (currently) passive.

Heads. One aspect of the applicability of a rule is to find an instance of the head of the rule. Therefore the head of each rule is matched against the active constraint. If the head consists of more than one constraint, partner constraints are searched in the user-defined constraint store, to match the other heads. If matching succeeds, i.e. the active constraint and eventually a conjunction of partner constraints are an instance of the head of the rule, the guard is executed. Otherwise, the next rule is tried.

Guard. A guard is a precondition on the applicability of a rule. The guard either succeeds or fails. A guard succeeds, if the execution of the guard succeeds. The execution of an empty guard always succeeds. The execution of the guard may not have any effects on the variables used in the head or in the body. If the guard succeeds and the rule applies, we commit to it and it fires. Otherwise, it fails and the next rule is tried.

Body. If the firing rule is a simplification rule, the matched constraints are removed from the user-defined constraint store. All matching constraints of a propagation rule are kept in the store. Once a propagation has fired, it will not fire again with the same combination of user-defined constraints. A simpagation rule is a hybrid kind of rule. All constraints matching the head constraints of a simpagation rule which succeed the operator $\backslash$ are removed from the store.

The constraints matching the other head constraints are kept. In any case, the body of a firing rule is executed, i.e. the user-defined constraints of the body are stored in the user-defined store and the built-in constraints of the body are stored in the built-in constraint store. When the currently active constraint has not been removed, the next rule is tried.

(Re-)Suspension. If all rules have been tried and the active constraint has not been removed, it suspends (that means it is inserted in the user-defined constraint store) until it is reactivated. In this case, all rules are tried again.

2.3 Example

We will illustrate the syntax and semantics of JCHR by the following example (see Appendix A for an implementation of a finite domain solver in JCHR). We define a user-defined constraint for less-than-or equal, `leq/2`. It is assumed that syntactical equality, `=`, is a built-in constraint.

```
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 {
      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` (L3). 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`.

2.4 Structure of JCHR

The JCHR prototyping environment consists of several components. JCHR programs are translated into Java code by the JCHR compiler. It generates Java code which is intended to be integrated into Java applications or applets. The last step which had to be taken to provide CHR for Java was the implementation of an evaluator which is able to interpret the information built with JCHR. It is called the JCHR evaluator. A constraint solver written with JCHR is based on a common constraint system. This system receives information about the used variables, rules, and goals. It is represented in Java by an instance of the class `ConstraintSystem`. This class is also the main part of the evaluator.

3 VisualCHR

VisualCHR is a tool to support the development of constraint solvers written in JCHR. It can be used to debug and to improve the efficency of constraint solvers. VisualCHR can also be used to understand the details of constraint propagation methods and the interaction of different constraints implemented by means of JCHR. Thus, it is suitable for users at different levels of expertise.

3.1 Representation of the Constraint Store

The visualization of the constraint propagation depends on the representation of the store. A constraint store can be represented graphically by a box consisting of all its constraints. We call such representation *box view*. In Figure 1, the goal `leq(X,Y), leq(Z,X), leq(Y,Z)` from Section 2.3 is represented in a box view.

Additionally, 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*. In Figure 2, the constraints `leq(X,Y), leq(Z,X), leq(Y,Z)` are represented in a sub-box view.

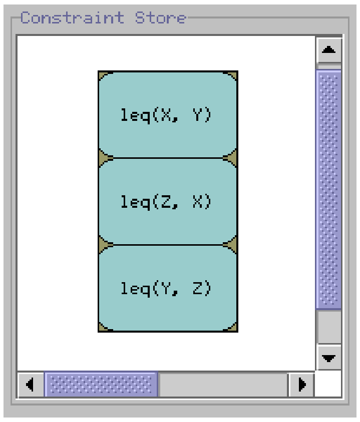

Fig. 1. Box View of a Constraint Store

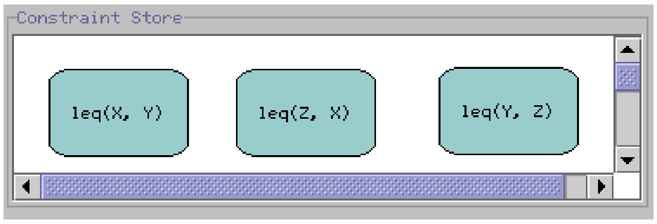

Fig. 2. Sub-Box View of a Constraint Store

3.2 Creation and Expansion of Graphs

Using a box view, the constraint propagation will be visualized by a linear tree. However, a sub-box view leads to a graph.

Initially, in a sub-box view of a constraint store the graph consists of the nodes representing the goal, i.e. each node corresponds to a constraint. VisualCHR provides an operation "next", which uses the built-in inference engine to expand the graph by applying rules applicable to nodes in the graph. For the `leq` example, the transitivity rule is applicable on `leq(X,Y)` and `leq(Y,Z)` and its body results in a node `leq(X,Z)` of the graph (Figure 3). The constraints `leq(X,Y)` and `leq(Y,Z)` remain in the constraint store since the transitivity rule is a propagation rule. To distinguish between constraints which are removed by simplification rules and constraints which remain in the constraint store, we use different colors, i.e. orange for removed constraints and blue for the constraints which are still in the constraint store.

Expansion of the graph need not proceed in a step-by-step fashion. The operation "skip" creates the whole graph. Figure 4 shows the graph resulting from "skip" activated for the root.

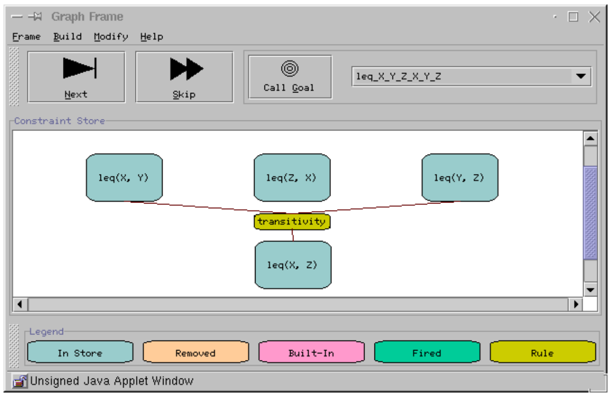

Fig. 3. Step-by-Step Expansion

The visualization tool has other functionalities, e.g. hiding constraints and rules to abstract from details that are currently irrelevant.

3.3 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.
- The user interface which provides for menus, cursor control, status bar, ...

The user interface is implemented using swing classes that eliminate Java's biggest weakness and allow to build state-of-the-art user interfaces.

4 Java Abstract Search Engine

Usually, constraint solving is not sufficient to solve combinatorial problems. Constraint solving must be combined with search, which is in general used to assign values to variables. After each assignment step constraint propagation restricts the possible values for the remaining variables, removing inconsistent values or detecting failure. If a failure is detected, the search returns to a previous decision and chooses an alternative.

Most existing libraries and languages have either only depth-first search, e.g. CHIP [4], or support the programming of different search algorithms through special purpose language constructs, e.g. Oz [12] or CLAIRE [1]. Figaro [2] was proposed to support programmable search algorithms in a C++ library by representing constraint stores as data objects. Our work is inspired by the Figaro system. We provide an abstract search engine, called JASE (to pronounce "chase",

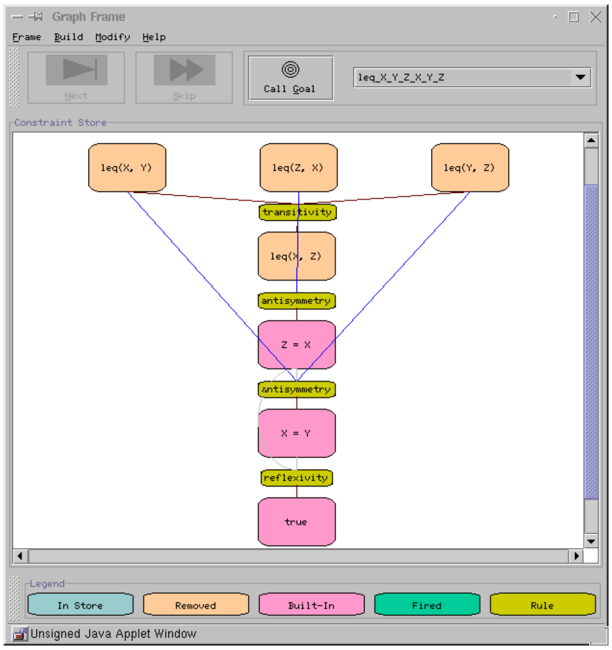

Fig. 4. Expansion

because it "chases a solution"), which has been actually designed for JCHR but can be used for any Java constraint library. JASE is called abstract because it poses no limits on the search strategies that can be implemented with it – It is a framework for a multitude of possible algorithms. In the following and due to space limits, we describe JASE by a small example.[1] For more details, we refer to `http://www.pms.informatik.uni-muenchen.de/software/jack`.

In the following, we use the finite domain solver implemented in JCHR (see Appendix A for the full code). A finite domain constraint is a constraint of the form $X \in D$, where X is a variable, and $D \subset \mathbb{N}$ is a finite subset (domain) of the natural numbers. The textual representation of such a constraint is written as `fdEnu(X,D)`[2] or `fdInt(X,Min,Max)` (which means $X \in D = [Min, Min + 1, \ldots, Max]$).

The most important Java object when using JCHR is the `ConstraintSystem`, which encapsulates the constraint solver, the constraint store, and all rules. It is the main object used by the host code.

```
ConstraintSystem cs=new ConstraintSystem();
```

[1] All the code snippets in this example are one contiguous block of source code, they are just separated to insert the explanation text between. Places, where "..." is used, are not relevant for the search, and only contain implementation details.

[2] "`fd`" is the prefix for all Finite Domain constraints; "`Enu`" stands for "enumeration", which means that `D` is simply a collection of integers.

Rules must be inserted in the the constraint system; this is done by creating a constraint handler.

```
fdHandler fd=new fdHandler();
fd.defineRules(cs);
```

Constraint variables are represented by Java objects of type `Object`. They are associated with a type and a name.

```
Object X = new Object();
cs.addVariable(X, "java.lang.Integer", "X");
```

The last step in setting up the constraint system is inserting initial constraints with the `addGoalConstraint` method of `ConstraintSystem`. Here, the constraint `fdEnu(X,[2,3,4,5,6])` is created and inserted into the constraint store:

```
cs.addGoalConstraint(new FDENUConstraint(X,createList(2,6)));
```

Now, the search engine is being set up. In this particular example, the values of the variable `X` should simply be enumerated. So, a container with the variable is created:

```
ObjectContainer vars=new ObjectContainer();
vars.add(X);
```

The next line creates an object that defines what should happen with each solution that is encountered during the search.

```
SChoice collector=new SCollectorChoice(vars,...);
```

The `collector` accumulates all solutions into a container for later use; it is applied to all successful leaves of the search tree ("solutions").

The most important part of the search are the choices made at each node:

```
SChoice rootChoice=new SFDEnuChoice(vars,collector,...);
```

`rootChoice` is the root of the search tree. It is responsible for creating more choices, and it actually modifies and runs the constraint system during the search. The `SFDEnuChoice` used in the example enumerates variables from left to right with no particular heuristic.

Now, the way to explore the search tree is defined (depth-first search).

```
SExploration exploration=
    new SDepthFirstExploration(cs,rootChoice);
```

The search is run, looking for all solutions.

```
boolean success=SSearch.all(exploration);
```

And finally, all solutions can be displayed or otherwise processed.

```
System.out.println(collector.toBeautifulString());
```

5 Conclusion and Future Work

In this paper, we have described a library, called JACK, that provides constraint programming for the host language Java. JACK consists of a high-level language JCHR to write constraint solvers, a tool VisualCHR to visualize the propagation and simplification of constraints defined by a JCHR solver, and a generic search engine JASE.

JCHR provides several classical constraint solvers, e.g. finite domains, Booleans, linear polynomials and interval arithmetics. To evaluate our search engine, we have implemented several examples that require search, e.g. n-queens problem with `noattack`-constraints, n-queens problem with the `alldifferent`-constraints using the algorithm proposed by Regin [9], Schur's Lemma taken from [14] which is part of CSPLIB, ... Note that the `alldifferent`-constraint is implemented in a Java class independent from JCHR. This example shows the ease of combination of solvers written in JCHR and solvers written in Java. These examples and more can be found at

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

JACK works quite satisfyingly. Even though, the main area for further development will be to improve the performance: Though the speed of the evaluator has been quite improved, it is still lacking when compared to constraint systems which are implemented in C++. Maybe the data structures and algorithms employed can be adapted to take more advantage of Java specifics (for example, by suppressing frequent creation of temporary objects, or by improving the `clone` operations). Maybe some key parts can even be (optionally) implemented in C++ via JNI; for example, key structures like the `ObjectContainer` or `nl`[3] utility classes

Another direction for future work will be the implementation of a visualization tool for search trees as it has been done for Oz [10]. Combining this tool with VisualCHR will be the next challenge.

References

1. Y. Caseau, F.-X. Josset, and F. Laburthe. Claire: Combining sets, search, and rules to better express algorithms. In *ICLP99*, 1999.
2. T. Chew, M. Henz, and K. Ng. A toolkit for constraint-based inference engines. In *Practical Aspects of Declarative Languages*, 2000.
3. A. Chun. Constraint programming in java with JSolver. In *Practical Application of Constraint Logic Programming*, 1999.
4. M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, and A. Herold. The CHIP system: Constraint handling in Prolog. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction*, LNCS 449. Springer-Verlag, 1990.

[3] `nl` is a class which implements a nested (linked) list.

5. 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.
6. T. Frühwirth and P. Brisset. High-level implementations of constraint handling rules. Technical report, ECRC, 1995.
7. C. Holzbaur and T. Frühwirth. A prolog constraint handling rules compiler and runtime system. *Journal of Applied Artificial Intelligence*, pages 369–388, 2000. Special Issue on Constraint Handling Rules.
8. W. K. Jackson, M. Irgens, and W. S. Havens. A re-examination of limited discrepancy search. Internet: http://fas.sfu.ca/isl/papers/jackson-lds/LDS.html.
9. J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI-94*, pages 362–367, Seattle, WA, 1994.
10. C. Schulte. Oz Explorer: A visual constraint programming tool. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. MIT Press, Cambridge, MA, USA.
11. C. Schulte. Comparing trailing and copying for constraint programming. In D. D. Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289. MIT Press, November 1999.
12. G. Smolka. The Oz programming model. *Lecture Notes in Computer Science*, 1000, 1995.
13. M. Torrens, R. Weigl, and B. Faltings. Java Constraint Library: bringing constraint technology to the internet using the java language. In *Working Notes of the Workshop on Constraints and Agents*, 1997.
14. T. Walsh. CSPLIB problem 15: Schur's lemma. Internet: http://www-users.cs.york.ac.uk/~tw/csplib/prob/prob015/index.html.
15. N. Zhou, S. Kaneko, and K. Yamauchi. DJ: A java-based constraint language and system. In *Proceedings of the Annual JSSST Conference*, 1998.

A Finite Domain Solver in JCHR

In this section, we present a cut-out of the finite domain solver written in JCHR and the visualization of a goal of the form $X \in \{2,3\} \wedge Y \in \{1,2\} \wedge X \leq Y$ (Figure 5).

```
handler fd
{
   class java.lang.Integer;
   class IntUtil;
   class nl; // linked list
   class NlIntUtil;
   class FDUtil;
   class ConstraintSystem;

   constraint fdEnu(java.lang.Integer, nl);
   constraint fdInt(java.lang.Integer,
                    java.lang.Integer,
                    java.lang.Integer);
   constraint fdLe(java.lang.Integer, java.lang.Integer);
   constraint fdLt(java.lang.Integer, java.lang.Integer);
   constraint fdNe(java.lang.Integer, java.lang.Integer);
```

```
rules {
   variable java.lang.Integer X,Y,Z;
   variable java.lang.Integer Min,Max;
   variable java.lang.Integer MinX,MinX1,MinY;
   variable java.lang.Integer MaxY,MaxY1,MaxX;
   variable nl L, L1, L2, L3, L4, L5, L6;
   variable FDAllDiff AD;

   // failure
   if (nl.isEmpty(L)) { fdEnu(X, L) } <=>
      { false } failure;

   // intersection
   { fdEnu(X, L1) && fdEnu(X, L2) } <=>
      { L = nl.intersection(L1, L2) &&
        fdEnu(X, L) } intersection;

   // interaction with intervals
   { fdEnu(X, L) && fdInt(X, Min, Max) } <=>

      { L1 = NlIntUtil.removeLower(Min, L) &&
        L2 = NlIntUtil.removeHigher(Max, L1) &&
        fdEnu(X, L2) } intersection2;

   // interaction with inequalities
   if (nl.notEmpty(L1) && MinX = NlIntUtil.minList(L1) &&
       nl.notEmpty(L2) && MinY = NlIntUtil.minList(L2) &&
       IntUtil.gt(MinX, MinY))

       { fdLe(X, Y) && fdEnu(X, L1) &&
         fdEnu(Y, L2) } ==>

          { MinX = NlIntUtil.minList(L1) &&
            MaxY = NlIntUtil.maxList(L2) &&
            fdInt(Y, MinX, MaxY) } leMin;

   if (nl.notEmpty(L1) && MaxX = NlIntUtil.maxList(L1) &&
       nl.notEmpty(L2) && MaxY = NlIntUtil.maxList(L2) &&
       IntUtil.gt(MaxX, MaxY))

       { fdLe(X, Y) && fdEnu(X, L1) &&
         fdEnu(Y, L2) } ==>

          { MinX = NlIntUtil.minList(L1) &&
            MaxY = NlIntUtil.maxList(L2) &&
            fdInt(X, MinX, MaxY) } leMax;

   if (nl.notEmpty(L1) && MinX = NlIntUtil.minList(L1) &&
       nl.notEmpty(L2) && MinY = NlIntUtil.minList(L2) &&
```

```
        MinX1 = IntUtil.inc(MinX) &&
        IntUtil.gt(MinX1, MinY))

        { fdLt(X, Y) &&
          fdEnu(X, L1) &&
          fdEnu(Y, L2) } ==>

           { MinX = NlIntUtil.minList(L1) &&
             MinX1 = IntUtil.inc(MinX) &&
             MaxY = NlIntUtil.maxList(L2) &&
             fdInt(Y, MinX1, MaxY) } ltMin;

    if (nl.notEmpty(L1) && MaxX = NlIntUtil.maxList(L1) &&
        nl.notEmpty(L2) && MaxY = NlIntUtil.maxList(L2) &&
        MaxY1 = IntUtil.dec(MaxY) &&
        IntUtil.lt(MaxY1, MaxX))

        { fdLt(X, Y) &&
          fdEnu(X, L1) &&
          fdEnu(Y, L2) } ==>

           { MinX = NlIntUtil.minList(L1) &&
             MaxY = NlIntUtil.maxList(L2) &&
             MaxY1 = IntUtil.dec(MaxY) &&
             fdInt(X, MinX, MaxY1) } ltMax;

    // interaction with fdNe
    if (NlIntUtil.member(X, L))
        { fdNe(X, Y) && fdEnu(Y, L) } <=>
           { L1 = NlIntUtil.remove(L, X) &&
             fdEnu(Y, L1) } ne1;

    if (NlIntUtil.member(X, L))
        { fdNe(Y, X) && fdEnu(Y, L) } <=>
           { L1 = NlIntUtil.remove(L, X) &&
             fdEnu(Y, L1) } ne2;

    if (NlIntUtil.notMember(X, L))
        { fdEnu(Y, L) &\& fdNe(X, Y) } <=>
           { true } ne3;

    if (NlIntUtil.notMember(X, L))
        { fdEnu(Y, L) &\& fdNe(Y, X) } <=>
           { true } ne4;

    // fdLe, fdLt trivial constraints
    { fdLe(X, Y) && fdLe(Y,X) } <=> { X = Y } leLe;
    { fdLt(X, Y) && fdLt(Y,X) } <=> { false } ltLt;
}
```

```
goal g1
{

     variable java.lang.Integer X, Y;

     fdEnu(X,new nl(2,new nl(3))) &&
     fdEnu(Y,new nl(1,new nl(2))) &&
     fdLe(X,Y)
}

}
```

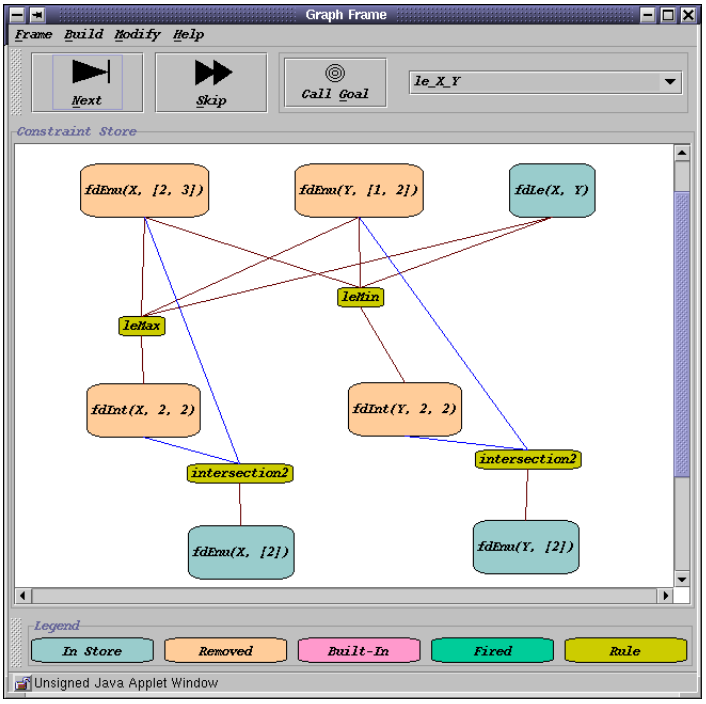

Fig. 5. Visualization of a goal