Electronic Notes in Theoretical Computer Science 64 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume64.html

JACK: A Java Constraint Kit

Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, Matthias Schmauss

Computer Science Department
University of Munich
http://www.pms.informatik.uni-muenchen.de/software/jack

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) [18] provides syntax extensions to Java to support constraint programming. DJ is especially designed to simplify the process of GUI's and Java applets.
- JSolver [4] 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) [15] 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 (JAva Constraint Kit) and consists of three parts:

- JCHR (Java Constraint Handling Rules): 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 briefly present the idea of constraint logic programming. In Section 3, we present the syntax and semantics of JCHR. In Section 4, we present the visualization tool of the JCHR computations. Section 5 introduces the Java abstract search engine JASE. Finally, we conclude with a summary and directions for future work.

2 Constraint (Logic) Programming

Constraint programming is based on the idea that many interesting and difficult problems can be expressed declaratively in terms of variables and constraints. The variables range over a (finite) set of values and typically denote alternative decisions to be taken. The constraints are expressed as relations over subsets of variables and restrict admissible value combinations for the variables. Constraints can be given explicitly, by listing all possible tuples, or implicitly, by describing a relation in some (say mathematical) form. A solution is an assignment of variables to values which satisfies all constraints. Constraint programming can be expressed over many different domains like linear terms over rational numbers, Boolean algebra, finite/infinite sets or intervals over floating point numbers. Very interesting development is possible for most of these domains or more general domain independent constraint solvers.

Constraint logic programming (CLP) is the most developed of the constraint programming paradigms [10,11]. In the last 15 years, CLP has evolved from a basic research idea to a powerful programming paradigm. CLP combines the declarativity of logic programming with the efficiency of constraint solving.

Constraint solving is the mechanism which controls the interaction of the constraints. Each constraint can deduce necessary conditions on the variable domains of its variables. The methods used for this constraint reasoning depend on the constraints, in the finite domain case they range from general but rather syntactic inference rules to complex combinations of algorithms used in the global constraints. Whenever a constraint updates a variable, the constraint propagation will wake all relevant constraints to detect further consequences.

In the beginning, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language, termed the "black-box" approach. While efficient, this approach makes it hard to modify a solver or build a solver over a new domain, let alone reason about and analyze it. As the behavior of the solver can neither be inspected by the user nor explained by the computer, debugging of constraint-based programs is hard. Also, one lesson learned from practical applications is that constraints are often heterogeneous and application specific. Several proposals have been made to allow more flexibility and customization of constraint solvers, often termed "glass-box" approaches [5,16]. The most far-reaching proposal is the "no-box" approach: Constraint Handling Rules (CHR) [7].

3 Java Constraint Handling Rules

Constraint Handling Rules (CHR) [7] 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 [8] or Sicstus Prolog [9]. In the following, we will present an implementation of CHR in Java. We call this language Java Constraint Handling Rules (JCHR).

3.1 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 5). 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.

3.1.1 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);
```

3.1.2 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 defined and 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.

3.1.3 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 {
      ...
        }
```

3.2 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 \ 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.

3.3 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`. 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`.

3.4 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. To provide JCHR for Java, we implemented 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.

4 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

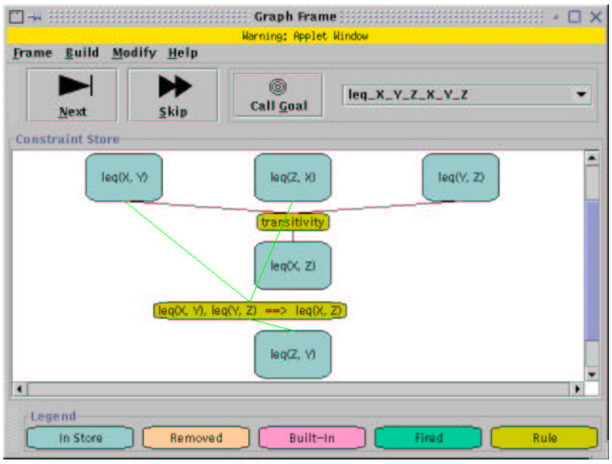

Fig. 1. Sub-Box View

propagation methods and the interaction of different constraints implemented by means of JCHR. 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].

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*.

4.1 Sub-Box View

In Figure 1, 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 1, 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

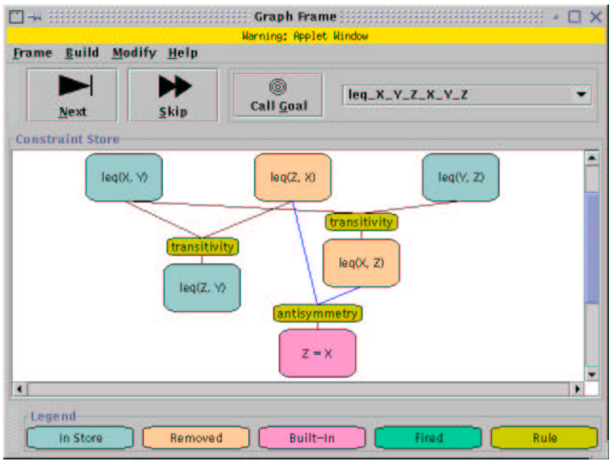

Fig. 2. Further Evaluation Step

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 2, 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 3 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 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.

4.2 Box View

In Figure 4, 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*.

The visualization tool has other functionalities, e.g. hiding nodes to ab-

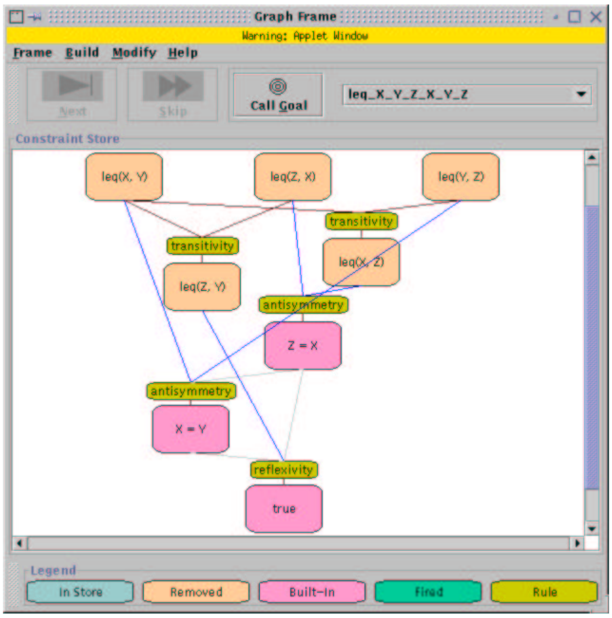

Fig. 3. Last Evaluation Step

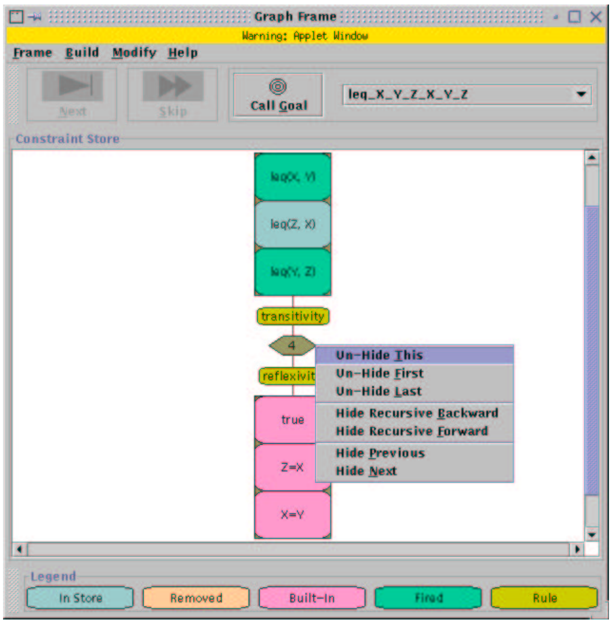

Fig. 4. Box View

stract from details that are currently irrelevant (Figure 4).

4.3 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.

5 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 [6], or support the programming of different search algorithms through special purpose language constructs, e.g. Oz [14] or CLAIRE [2]. Figaro [3] 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", 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, 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). Finite domains are represented by enumerations or by intervals. In an *enumeration domain* all values are listed ($X \in D$, 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)`. An *interval domain* specifies upper and lower bounds of a domain ($X \in [Min, Max]$). The textual representation of such a constraint is written as `fdInt(X,Min,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();
```

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

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

[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.

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());
```

6 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 global `alldifferent`-constraints using the algorithm proposed by Regin [12], Schur's Lemma taken from [17] which is part of the problem library for constraints 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`

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` [2] 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 [13]. Combining this tool with VisualCHR will be the next challenge.

References

[1] S. Abdennadher and M. Saft. A visualization tool for constraint handling rules. In *11th Workshop on Logic Programming Environments, WLPE'01*, 2001.

[2] Y. Caseau, F.-X. Josset, and F. Laburthe. Claire: Combining sets, search, and rules to better express algorithms. In *ICLP99*, 1999.

[3] T. Chew, M. Henz, and K. Ng. A toolkit for constraint-based inference engines. In *Practical Aspects of Declarative Languages*, 2000.

[4] A. Chun. Constraint programming in java with JSolver. In *Practical Application of Constraint Logic Programming*, 1999.

[5] P. Codognet and D. Diaz. Boolean constraint solving using clp(FD). In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, Vancouver, Canada, 1993. The MIT Press.

[6] 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.

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

[7] 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.

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

[9] 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.

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

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

[12] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI-94*, pages 362–367, Seattle, WA, 1994.

[13] 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.

[14] G. Smolka. The Oz programming model. *Lecture Notes in Computer Science*, 1000, 1995.

[15] 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.

[16] P. van Hentenryck. Constraint logic programming. *The Knowledge Engineering Review*, 6, 1991.

[17] T. Walsh. CSPLIB problem 15: Schur's lemma. Internet: http://www-users.cs.york.ac.uk/~tw/csplib/prob/prob015/index.html.

[18] 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 the goal $X{\in}\{2,3\} \wedge Y{\in}\{1,2\} \wedge X{\leq}Y$ (Figure A.1).

```
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)
 }

}
```

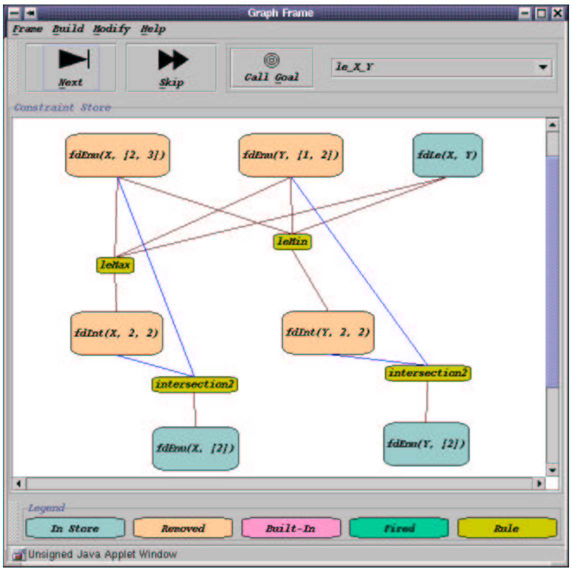

Fig. A.1. Visualization of a goal