INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

An Implementation of CHR in Java

Matthias Schmauß

Diplomarbeit

Beginn der Arbeit:	13. Mai 1999
Abgabe der Arbeit:	26. November 1999
Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Dr. Slim Abdennadher

Contents

Chapter 1

Introduction

1.1 Objectives

Constraint programming is the topic of both scientific and commercial investigation and it is used all over the world for solving combinational problems, for generating new information out of incomplete or imprecise information and for decision support. Great efforts concerning constraint solving have been made in constraint logic programming (CLP) [?, ?, ?, ?]. In the beginning of CLP, 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 [?].

The most far-reaching proposal is the "no-box" approach. Constraint Handling Rules (CHR) [?]. CHR is a high-level language for writing constraint solvers either from scratch or by modifying existing solvers. CHR comes as a library that enhance a host language with constraint solving capabilities. CHR is essentially a language consisting of guarded rules with multiple heads that replace (conjunctions of) constraints by simpler ones until they are solved. Most implementations of CHR are done in logic programming languages like ECL^{i}PS^e Prolog [?] and SICStus Prolog [?].

These languages provide useful mechanisms like backtracking, which support efficient search for solutions of constraint problems. But there are missing some aspects, which are provided by the programming language Java. It combines the advantages of an object oriented programming language with the advantages of platform independence and the possibility of distributing applications in the Internet.

This work describes an approach of the implementation of a Java class library, which provides CHR in Java. The first step, which had to be taken, was the definition of a CHR syntax, similar to Java. With this language, called Java-CHR (JCHR), it is possible to define the guarded rules and goals for testing purposes. 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. This Java code is called JCHRCore, as long the definition of rules and constraints is concerned. The last step, which had to be taken to provide CHR for Java, was the implementation of an evaluator, which is able to interprete the information, built with JCHR. It is called the JCHRCore-Evaluator. Finally this work describes an approach of the integration of a constraint handler written in JCHR into a

Java applet. It tries to describe the interface between Java and JCHR for a special case and for other applications similar to the given example. This interface will be interesting for a project called JACK [?], which intends to integrate JCHR into the programming language JavaLog. This language is an implementation of Prolog in Java. The JACK project aims at defining an integrated Prolog-CHR system, platform independent oriented to industrial applications.

1.2 Related Work

There has been done some other work, which intends to combine the advantages of constraint programming with the advantages of the programming language Java.
For instance, the project JSolver [?] is a java library, which 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 efficiency of finding a solution for a constraint satisfaction problem, depends on the strategies, how these variables are selected for instantiation and on the order of their instantiation. Therefore JSolver provides predefined algorithms. However, it is possible to influence the solving algorithm by using user-defined variable-selection and variable-instantiation algorithms. The following steps have to be taken, when trying to solve a constraint problem with JSolver:

1. define variables and constraints
2. define heuristics for variable-selection and variable-instantiation
3. call the constraint handler

It is possible to implement new constraint handlers, but this has to be done on a low level (in this case with Java). As a consequence it is also difficult to combine existing constraint handlers.
The Java Constraint Library (JCL) [?] is an approach on solving constraint satisfaction problems[1] in Java. Similar to JSolver these constraint satisfaction problems are "hardwired". The description of the problems resembles a graph, where variables are represented with nodes and the constraints, which describe relations between them, are represented as edges. This representation allows to apply algorithms of the graph theory on constraint satisfaction problems. In order to increase flexibility and efficiency, the JCL provides several backtracking and preprocessing strategies. Preprocessing strategies are used to reduce the complexity of the constraint satisfaction problems, represented as graphs. However the main aspect of the JCL is the distribution of constraint satisfaction problems in networks, like the Internet, with client-server techniques and the reduction of computing time, caused by this distribution. Thereby the following methodology is propagated: The user sends a request from a client (e.g. from a WWW page) to a server, where the JCL is installed and working. The server generates a constraint satisfaction problem, using JCL and several databases, which are required for solving the problem. The constraint satisfaction problem and the data is sent back to the client, which autonomously solves the problem using the JCL by itself. This minimizes response time and server workload.
However, both approaches differ in some aspects from the Java library, introduced in this work. JSolver provides some constraint handlers and appropriate solving algorithms. They are ready to solve constraints problems, which need these constraint handlers, but it

[1] A *constraint satisfaction problem* consists of a finite set of variables, each associated with a finite domain, a finite set of constraints. A *solution* of a CSP maps each variable to a value of its domain such that all the constraints are satisfied.

is difficult to implement and modify new or to merge existing constraint handlers. With JCL it is possible to write own constraint handlers and to merge it with Java applications or applets, but this has to be done on a low level, too. Against it, JCHR provides a syntax to create, modify and merge constraint handlers very easily. Solving algorithms (like backtracking) or additional functionalities (like arithmetic operations, etc.) need to be implemented in Java.

Chapter 2

Preliminaries

Solving a problem with constraint techniques often requires problem oriented constraint handlers, which may not be available or which are very difficult to create out of existing constraints. CHR has been developed, in order to simplify the implementation of new constraint handlers and the extension, the specialization and combination of existing implementations. CHR extends a language. So the capabilities of CHR can be combined with the specialties of the current host language (e.g. backtracking in Prolog, the usage of the Internet in Java). Furthermore, auxiliary computations can be done by the host language. There are CHR implementations for the languages ECL^{i}PS^e Prolog [**?**], Common LISP [**?**] , OZ[1] [**?**] and SICStus Prolog [**?**]. The main principle of CHR is to apply a set of guarded rules on a set of constraints. Depending on the rule, one or more constraints are added or removed to the set of constraints. The syntax of the rules and their effects on constraints is described in the following two sections. An Example of CHR will be described in Section **??**. Some familiarity with (constraint) logic programming and the implementation of logic programming languages (e.g. logical variables and unification) is assumed.

2.1 Syntax of CHR

The following description of the CHR syntax is kept very common. There are syntactical differences between the implementations of CHR (e.g. position of rule names and guards), which are not mentioned in this section [**?**].
A CHR program consists of rules, which define the properties of user-defined constraints. Beside these user-defined constraints, two other disjunct sets of predicates are used in CHR. The first kind of constraints are the built-in constraints, the second kind are the predicates of the host language. A rule consists of a head, an optional guard and a body (Figure **??**).
A head is a conjunction of user-defined constraints. A guard is a conjunction of built-in constraints and expressions of the host language. Built-in constraints and host language expressions are not allowed in the head. Vice versa, user-defined constraints are not allowed in the guard. The body of a rule is a conjunction of all kinds of constraints and host language expressions. All constraints and all host language expressions may use logical variables as arguments. There are three kinds of rules. The *simplification rule*, the *propagation rule* and the *simpagation rule*. A *simpagation rule* is a combination of the other two kinds.

[1] a logical object-oriented language

Simplification rule

$$\text{Head} \Leftrightarrow [\ \text{Guard}\ |\ ]\ \text{Body}$$

Propagation rule

$$\text{Head} \Rightarrow [\ \text{Guard}\ |\ ]\ \text{Body}$$

Simpagation rule

$$\text{Head}_1 \setminus \text{Head}_2 \Leftrightarrow [\ \text{Guard}\ |\ ]\ \text{Body}$$

Figure 2.1: The syntax of CHR

2.2 Semantics of CHR

This section describes an operational semantic of CHR [**?**]. A CHR program consists of a finite set of rules. Every time a user-defined constraint is activated, this set is searched for applicable rules concerning this (currently) active constraint. All the other constraints in the constraint store are called (currently) passive. Two different kinds of stores are used. One store contains user-defined constraints and the other contains built-in constraints.
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 constraint 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 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 a 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 the rule applies, one commits 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 combination of the two other types. 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.

2.3 Example

This section shows a standard example in CHR: the 'less or equal'-constraint (leq). It uses the syntactical equality ($\doteq$), *true* and *false* as built-in constraints.

reflexivity

$$\text{X leq Y} \Leftrightarrow \text{X} \doteq \text{Y} \mid \mathit{true}$$

antisymmetry

$$\text{X leq Y} \wedge \text{Y leq X} \Leftrightarrow \text{X} \doteq \text{Y}$$

idempotence

$$\text{X leq Y} \setminus \text{X leq Y} \Leftrightarrow \text{true}$$

transitivity

$$\text{X leq Y} \wedge \text{Y leq Z} \Rightarrow \text{X leq Z}$$

Figure 2.2: The rules for the 'less or equal'-constraint.

The rules for the user-defined constraint 'leq' are specified in Figure **??**. The *reflexivity rule* simplifies the constraint $X\ leq\ Y$ to *true*, when X $\doteq$ Y holds, with $\doteq$ as the syntactic equality. The *antisymmetry rule* simplifies the constraints $X\ leq\ Y$ and $Y\ leq\ X$ to $X \doteq Y$. The *idempotence rule* removes redundant constraints. These rules reduce the set of user-defined constraints in contrast to the *transitivity rule*, which adds a body constraint without removing the head constraints. This makes sense in combination with the two other rules as the following example demonstrates. Let $A\ leq\ B$, $C\ leq\ A$, $B\ leq\ C$ a goal. The initial state is $(A\ leq\ B \wedge C\ leq\ A \wedge B\ leq\ C,\ true)$, whereas a state consists of a store for user-defined constraints and a store for built-in constraints.

$$
\begin{array}{ll}
 & (\underline{A\ leq\ B} \wedge \underline{C\ leq\ A} \wedge B\ leq\ C,\ true) \\
\rightarrow_{propagate} & (A\ leq\ B \wedge C\ leq\ A \wedge \underline{B\ leq\ C} \wedge \underline{C\ leq\ B},\ true) \\
\rightarrow_{simplify} & (\underline{A\ leq\ B} \wedge \underline{C\ leq\ A},\ B \doteq C) \\
\rightarrow_{simplify} & (\top,\ A \doteq B \wedge B \doteq C)
\end{array}
$$

The goal causes the *transitivity rule* to fire, that means, that the instantiated body of the *transitivity rule* is added to the constraint stores (user-defined and built-in constraints are separated). In the example the constraint $C\ leq\ B$ is added to the store. Then the constraints $C\ leq\ B$ and $B\ leq\ C$ are simplified to $B \doteq C$ by the *antisymmetry rule*. In the following each occurrence of the variable C is regarded as the variable B. So the constraints $C\ leq\ A$ and $A\ leq\ B$ are simplified to $A \doteq B$ by the *antisymmetry rule*. Since there are no more constraints left in the constraint store, the transition is finished.

Chapter 3

Implementation

In this chapter, we go into details of the implementation of the CHR in Java. The implementation process consisted of three steps. At first a language for describing the rules of CHR in a syntax, similar to Java, was developed. This language is called JCHR. An introduction to JCHR is given in Section **??**.
The next step was to implement CHR in Java directly. Therefore it was necessary to develop a Java library, which provides classes to define constraints and rules in Java. This library is called JCHRCore (Section **??**). The implementation of CHR in Java then was completed by another Java library, which provides the evaluation of the rules and goals defined in JCHRCore. Section **??** describes some details of the implementation of this library called JCHRCore-Evaluator. Section **??** illustrates implementation details of the JCHR-Compiler, which translates JCHR to JCHRCore. The basic interaction of JCHR-Compiler and JCHRCore-Evaluator with the Java Runtime Environment is illustrated in Figure **??**. Thereby, rules which describe the properties of a constraint, are specified in the language JCHR. This specification is translated into the JCHRCore class library by the JCHR-Compiler. The result is integrated into a Java application or a Java Applet. The Java Runtime Environment and the JCHRCore-Evaluator interprete this Java-Constraint program. Besides defining rules in JCHR, it is possible to define goals, in order to test the constraint handler. After compiling the JCHR program with the JCHR-Compiler, it can executed without merging it with Java code, in order to evaluate the goals, defined in JCHR previously. In any case, the advantages of Java and CHR are combined with this package.

3.1 Introduction to Java Constraint Handling Rules (JCHR)

As we have seen in Chapter **??** there are several advantages of Java and CHR, so that a combination of this two languages will offer a large spectrum of new possibilities. This combination is intended to be very similar to Java, in order to provide an intuitive programming. This combination of Java and CHR, called JCHR, will be introduced in the following section.

3.1.1 Syntax

The most obvious modification of JCHR (concerning CHR in a logic programming language) are the syntactic elements of Java identifying the beginning and the end of a logic block, the brackets "{" and "}". They signalize the beginning and the end of a head and a body. Furthermore, the guard is not positioned after the rule operator anymore. In order to make a rule more 'Java-like', the if-statement is written in front of the heads. The most

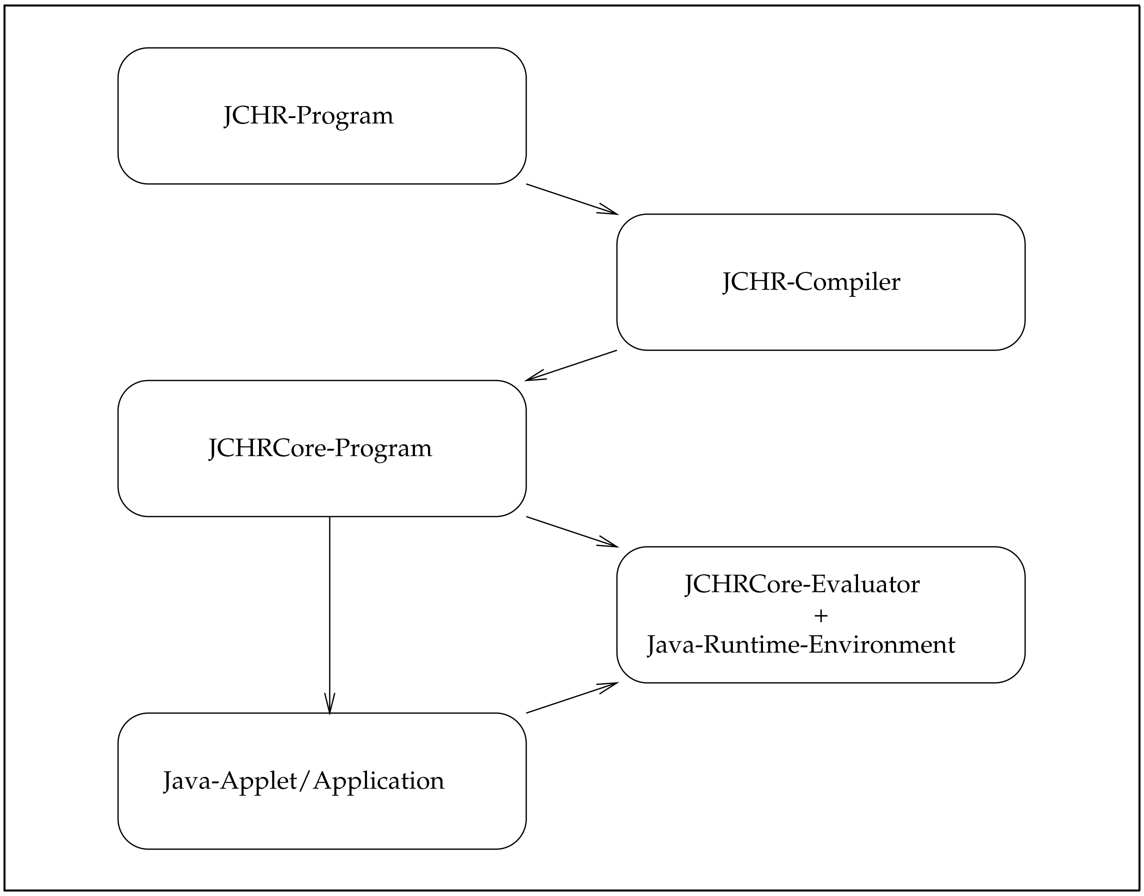

Figure 3.1: The parts of JCHRCore.

important change comes with the fact, that Java is a typed language. This is necessary in order to ensure the security of software, hardware and data while using Java in the Internet [?]. Therefore JCHR has to be typed, too. That means that every expression in JCHR needs a type, even logical variables. So a constraint handler and the parts of such a handler, written in JCHR, are divided in declaration and implementation sections. The declaration sections provide information about the used classes, the variables, their types and about the instances of objects. The constraints and their argument classes are declared in these sections, too. In the implementation sections, the rules and (optional) goals are specified. JCHR is formalized in EBNF in Figure **??** (NOTE: It is possible to insert single line ('//') oder multi line ('/* */') comments everywhere in the constraint handler. An IDENTIFIER may also contain dots, like it is needed in Java class names (e.g. java.lang.Integer)).
All classes, used in the handler, need to be imported in the declaration section of the constraint handler. This includes both, the classes used in declarations and the classes used for static method calls. The following line indicates, that the class `java.lang.Math` is going to be used in the current handler.

```
class java.lang.Math;
```

A user-defined constraint has to be declared in the declaration section of the handler, too, e.g.

```
constraint leq(java.lang.Integer, java.lang.integer);
```

```
ConstraintHandler              ::=  ‘handler’ HandlerName ‘{’ HandlerDeclaration
                                    HandlerImplementation ‘}’.
HandlerDeclaration             ::=  { ClassImport | ConstraintDeclaration }.
HandlerImplementation          ::=  RuleImplementation { GoalImplementation }.
ClassImport                    ::=  ‘class’ JavaClassName ‘;’.
ConstraintDeclaration          ::=  ‘constraint’ ConstraintName ‘(’ [
                                    JavaClassNameList ] ‘)’ ‘;’.
RuleImplementation             ::=  ‘rules’ ‘{’ StandardDeclaration { Rule }.
GoalImplementation             ::=  ‘goal’ ‘{’ StandardDeclaration Goal ‘}’.
JavaClassNameList              ::=  JavaClassName { ‘,’ JavaClassNameList }.
StandardDeclaration            ::=  { VariableDeclaration | InstanceDeclaration }
                                    .
VariableDeclaration            ::=  ‘variable’ JavaClassName VariableNameList
                                    ‘;’.
VariableNameList               ::=  VariableName { ‘,’ VariableNameList }.
InstanceDeclaration            ::=  JavaClassName InstanceName ‘=’
                                    InstanceDeclarationRightside ‘;’.
InstanceDeclarationRightside   ::=  ‘new’ JavaClassName ‘(’ ArgumentList ‘)’.
Rule                           ::=  SimplificationRule | PropagationRule |
                                    SimpagationRule .
SimplificationRule             ::=  [ Guard ] ‘{’ Heads ‘}’ ‘<=>’ ‘{’ Body ‘}’
                                    [ RuleName ] ‘;’.
PropagationRule                ::=  [ Guard ] ‘{’ Heads ‘}’ ‘==>’‘{’ Body ‘}’ [
                                    RuleName ] ‘;’.
SimpagationRule                ::=  [ Guard ] ‘{’ SimpagationHead ‘}’ ‘<=>’‘{’
                                    Body ‘}’ [ RuleName ] ‘;’.
Guard                          ::=  ‘if’ ‘(’ BuiltInConstraintList ‘)’.
SimpagationHead                ::=  Heads ‘&\&’ Heads.
Heads                          ::=  UserDefinedConstraint { ‘&&’ Heads } .
Body                           ::=  Constraint { ‘&&’ Body }.
Goal                           ::=  ‘goal’ GoalName ‘{’ { StandardDeclaration } {
                                    ConstraintList } ‘}’.
Constraint                     ::=  BuiltInConstraint | UserDefinedConstraint.
ConstraintList                 ::=  Constraint { ‘&&’ ConstraintList }.
UserDefinedConstraint          ::=  ConstraintName ‘(’ [ ArgumentList ] ‘)’.
BuiltInConstraintList          ::=  BuiltInConstraint { BuiltInConstraintList }.
BuiltInConstraint              ::=  ‘true’ | ‘false’ | Equality |
                                    MethodInvokation.
Equality                       ::=  Argument ‘=’ Argument.
MethodInvokation               ::=  ( JavaClassName | JavaInstanceName ) ‘.’
                                    JavaMethodName ‘(’ [ ArgumentList ] ‘)’.
ArgumentList                   ::=  Argument { ‘,’ ArgumentList }.
Argument                       ::=  Constraint | InstanceDeclarationRightside |
                                    IDENTIFIER | STRING | NUMBER.
InstanceName                   ::=  IDENTIFIER.
RuleName                       ::=  IDENTIFIER.
GoalName                       ::=  IDENTIFIER.
HandlerName                    ::=  IDENTIFIER.
ConstraintName                 ::=  IDENTIFIER.
JavaClassName                  ::=  IDENTIFIER.
ConstraintIdentifier           ::=  IDENTIFIER.
```

Figure 3.2: The EBNF of JCHR (part 2)

means, that a new constraint with the identifier 'leq' and two arguments of the type java.lang.integer is declared. The ClassImports and the ConstraintDeclarations are valid for the RuleImplementation and GoalImplementation sections. Instead of this, logical variables and instance declarations are only valid in the current section (rules definition or goal definition). The instance declarations are performed as in Java. The instructions

```
IntUtil iu = new IntUtil();
```

create a new instance of the class `IntUtil` called `iu`. It is not possible to apply a method call of a certain class to a logical variable of this type. So the programmer is forced to use methods, which return the result of their computation. It is not helpful, when the methods change the value of the object, they are belonging to. The following examples tries to make this circumstance more clear. Let `X` be a logical variable of the type `java.lang.Integer`. The variable is bound and the value is needed to be increased by `1`. The class `IntUtil` provides a static method called `inc`, which increases a given value by `1` and returns the result. This is implemented in JCHR as follows:

```
X = IntUtil.inc(X);
```

It is a question of the design, whether to provide this functionality or not. JCHRCore does not implement this property of Java variables in JCHR variables, in order to stay closer to the properties of logic variables.

3.1.2 Examples

The following examples for JCHR are standard examples for the constraint `leq` (less-or-equal) and `min` (minimum). The implementation of `min` uses a class called `IntUtil` (see Appendix **??**), which provides static methods for comparing integer objects (e.g. the method `le` tests whether an integer is less or equal than another integer).

Example: less or equal

The constraint handler `leq`, introduced by the keyword `handler` can be divided into tree main sections. In the handler declaration section, the constraint `leq` is defined by the keyword `constraint`.
The constraint `leq` expects two arguments of the type `java.lang.Integer`.
The second section is the rule implementation section, initiated by the keyword `rules`. In the beginning of this section, three variables (`X`, `Y` and `Z`) of the type `java.lang.integer` are declared. They are used only by the rules in the rule definition section. The four rules of the `leq`-handler are the same rules as described in the CHR example in Section **??**. So we omit a detailed description.
The difference between the both examples is the position of the guard in the `reflexivity` rule and the operator, that divides the head of the `idempotence` rule (\ vs. `&\&`).
Furthermore, the rule identifiers are placed behind the rules. The third main section is the goal implementation section, introduced by the keyword `goal`. The identifier `g1` is assigned to this goal, which also equals to the example in Section **??**.

```
handler leq {

    constraint leq(java.lang.Integer, java.lang.Integer);

    rules {
```

```
        variable java.lang.Integer X, Y, Z;

        // reflexivity
        if (X = Y) { leq(X, Y) } <=> { true } reflexivity;

        // antisymmetry
        { leq(X, Y) && leq(Y, X) } <=> { X = Y } antisymmetry;

        // idempotence
        { leq(X, Y) &\& leq (X, Y) } <=> { leq(X, Y) } idempotence;

        // transitivity
        { leq(X, Y) && leq(Y, Z) } ==> { leq(X, Z) } transitivity;

    }

    goal g1 {

        variable java.lang.Integer A, B, C;

        leq(A, B) && leq(C, A) && leq(B, C)

    }

}
```

Example: minimum

The JCHR `min` handler has the same structure as the leq handler. There is only one difference concerning the handler declaration section, where the Java class `IntUtil` is declared. This class provides some static methods, for the comparison of integer objects. The rules `r1` and `r2` handle the cases, when the variables `X` and `Y` are bound to integer objects. Then the variable `Z` is bound to the integer object which fulfills the minimum constraint. The rules `r3` and `r4` handle the reverse cases, where the variables `Z` and `X` are bound to integer objects and `Y` is unbound, or vice versa. Then `Y` is bound according the minimum constraint.

```
handler min {

    class IntUtil;

    constraint min(java.lang.Integer, java.lang.Integer, java.lang.Integer);

    rules {

        variable java.lang.Integer X, Y, Z;

        // min rules
        if (IntUtil.le(X, Y)) { min(X, Y, Z) } <=> { X = Z } r1;
        if (IntUtil.le(Y, X)) { min(X, Y, Z) } <=> { Y = Z } r2;
```

```
        if (IntUtil.lt(Z, Y)) { min(X, Y, Z) } <=> { Y = Z } r3;
        if (IntUtil.le(Z, X)) { min(X, Y, Z) } <=> { X = Z } r4;

    }

    goal g1 {

        variable java.lang.Integer A;
        min(1, 2, A)

    }

}
```

Goal `g1` of the `min` handler causes rule `r1` to fire. The body of rule `r1` is executed. This delivers the result `A = 1`.

3.2 The JCHRCore-Evaluator

In this section, the implementation of the operational semantic of CHR (see Section **??**) is described. Section **??** illustrates the basic package structure of the evaluator. Section **??** introduces the library JCHRCore, which provides classes for rules and goal definition in Java. Implementation details of the JCHRCore-Evaluator are mentioned in Section **??**.

3.2.1 The Package Structure

The basic package structure of the evaluator is shown in Figure **??**. The main part is the class `ConstraintSystem`. It provides data structures to store rules and goal constraints and the functionality to evaluate the goal constraints with the rules. In order to manage this, the class has to handle logical variables. An instance of `ConstraintSystem` consists of a variable table (class `VariableTable`), a memory to store the rules (class `RuleMemory`), and two memories to store constraints (the classes `UserDefinedConstraintMemory` and `BuiltinConstraintMemory`).
Every rule, stored in `RuleMemory` owns a pointer to `VariableTable`. The same holds for all constraints used in the constraint handler object. This is necessary, to keep the argument values of all constraints consistent. The following example clarifies this circumstance. Regarding the rule

```
{ p(X) && q(Y) } <=> { r(X, Y) };
```

with the constraints `p`, `q`, `r` and the variables `X` and `Y`. If the variables in the head constraints are instantiated, it is indispensable, that they are instantiated with the same values in the body constraint `r`, too. This is achieved by a centralized variable handling. Another advantage of a variable table, known by each constraint, is that the physical size of a variable object is kept small. In object oriented languages, the programmer has the possibility to accumulate data and functionality in objects. This may cause problems, when such an object needs to be cloned very often, because a large amount of data has to be moved. In this special case, variable objects are simply Java standard objects. All additional information about a variable (name, type, etc.) is stored in the variable table. There are two types of constraints. User defined constraints (class `UDConstraint`) and built-in constraints (class `BuiltinConstraint`). User-defined constraints are stored in the data

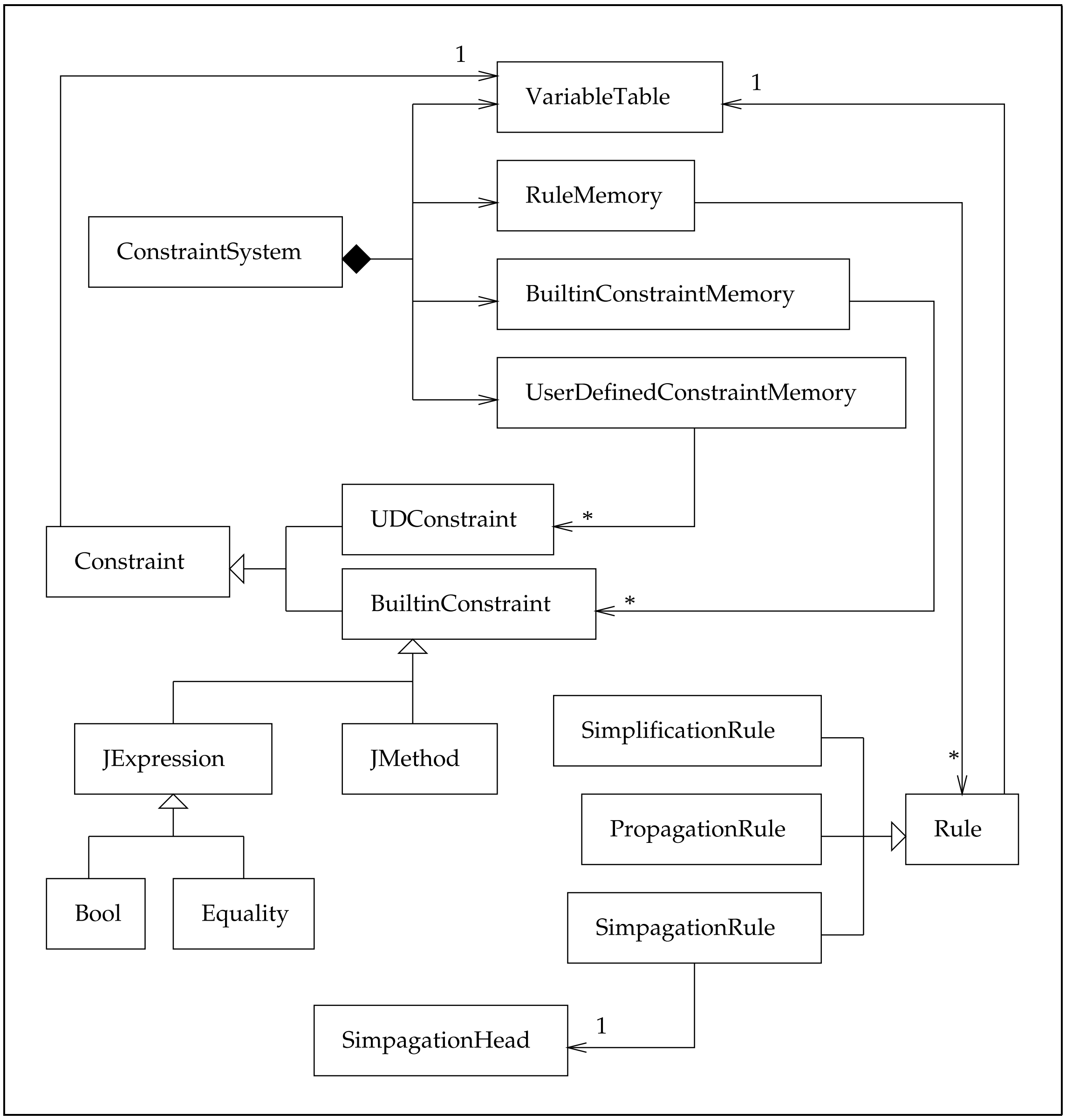

Figure 3.3: The basic class structure of the evaluator.

structure represented by an instance of the class `UserDefinedConstraintMemory`, and built-in constraints are stored in `BuiltinConstraintMemory`. Both, java expressions (like boolean values and the comparison and equality of objects, respectively) and java methods are handled as built-in constraints.

The rules are stored in the rule memory. Thereby there are three kinds of rules represented by the three classes `SimplificationRule`, `PropagationRule` and `SimpagationRule`. As described in Section **??**, the simpagation rule has a special head, indicating which constraints are removed from the constraint memory and which remain in the memory. So the class `SimpagationRule` is associated with the class `SimpagationHead`, in order to provide this functionality.

Furthermore, several Java exceptions are thrown by the classes. Figure **??** gives an overview.

While a built-in constraint is executed, all arguments need to have a value. This definition is crucial concerning logical variables, which may not have been bound at the time

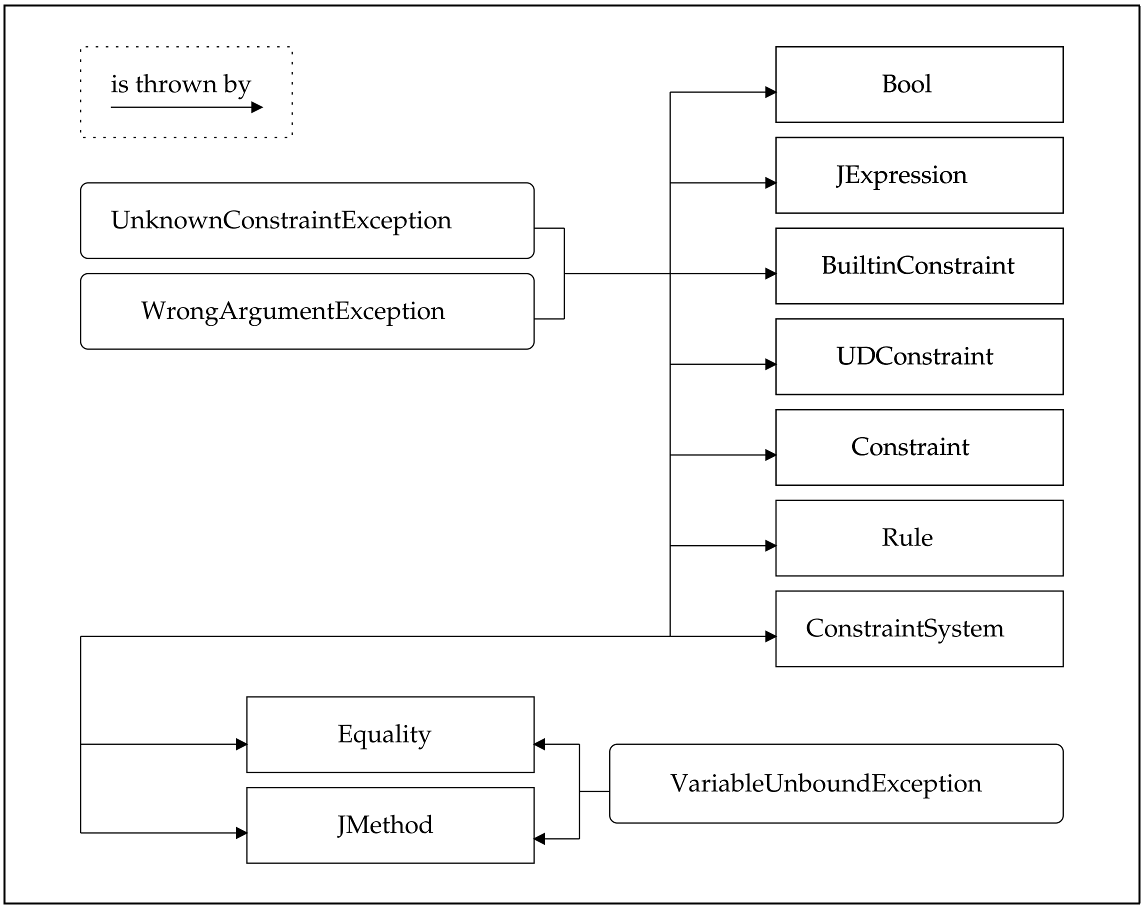

Figure 3.4: Exceptions

of execution. In this case the `VariableUnboundException` is thrown by the built-in constraints `Equality` and `JMethod`. This exception is not thrown by the classes `JExpression` and `BuiltinConstraint` in order to keep this definition flexible. An alternative would be the delayed execution of built-in constraints with unbound variables as arguments (see Chapter **??**).

When a rule is added to the constraint system, the elements of head, guard and body are tested, whether they are constraints. When an user-defined constraint is added to the goal, the constraint system tests whether the constraint is declared (the argument classes and a pointer to the variable table need to be specified). If the tests fail, the exception `UnknownConstraintException` is thrown. This exception is actually not needed for the currently implemented built-in constraints. But the possibility to throw it for a built-in constraint, keeps the system flexible for future work.

The exception `WrongArgumentException` is thrown, when the argument classes do not fit to the declaration of a constraint. Last but not least, the class `ConstraintSystem` provides an interface to store and reload the current state of the constraint handler. This is required for applications, which use backtracking algorithms (or the like; more information about this in Section **??**). Furthermore, this functionality can be used for debugging purposes, when the current states of the constraint system are printed out during a stepwise evaluation.

3.2.2 Introduction to JCHRCore

In this section, the class library JCHRCore is introduced. With this language it is possible to build constraint handler directly in Java. Aspects concerning the implementation of the evaluator are explained in Section **??**. Basic knowledge in Java is required.

The Constraint System

As we have seen in Section **??**, a constraint handler, written with JCHRCore 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`. In the following section the interfaces of this class are described. Let `cs` be an instance of `ConstraintSystem`.

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

Furthermore, the class `ConstraintSystem` provides some methods for variable and constraint access and for the evaluation itself. The following three methods provide the functionality for binding variables (`setVarValue`), reading the bindings (`getVarValue`) and testing whether a variable is bound (`hasVarValue`).

```
public void setVarValue(Object var, Object val)
public Object getVarValue(Object var)
public boolean hasVarValue(Object var)
```

Sometimes it is necessary to access not only the variables, but a complete user-defined constraint, which is not exactly known. Maybe the user only has information about the identifier and the arity of the constraint. Therefore the method `getInstances` was implemented. It expects a user-defined constraint as the argument and returns all possible instances. Now the programmer only has to perform a search over this reduced search area.

```
public ObjectContainer getInstances(UDConstraint c)
```

Before the constraint system is activated, it needs to be initialized with a set of rules and a goal. A goal is a conjunction of constraints. These constraints are added to the constraint system with the method `addGoalConstraint`. The rules are added to the constraint system with the method `addRule`.

```
public void addRule(Rule r)
public void addGoalConstraint(Constraint c)
```

Finally, after all initialization procedures have been done, the constraint system is activated by the method `callGoal`. If the evaluation has come to a successful finish, the method return `true`, otherwise it returns `false`.

```
public boolean callGoal()
```

Most constraint satisfaction problems are solved by backtracking algorithms. Therefore it is necessary to restore the state of the constraint system (consisting of constraint memories and variable table) at each node of the search tree. The methods `getEnvironment` and `setEnvironment` provide this functionality. The method `setEnvironment` causes a reinitialization of the rule memory. That means, after the state of the constraint system has been restored by this method, the search in the rule memory restarts with the first rule.

```
public Environment getEnvironment()
public void setEnvironment(Environment env)
```

All other functions of the class `ConstraintSystem` are documented in the Java code [**?**].

Logical Variables

Logical Variables are represented by objects of the Java standard class `Object`. The assignment of the type of a new variable is done by the method `addVariable` provided by the class `ConstraintSystem`:

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

It is optionally possible to assign an identifier to the variable for debugging purposes:

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

After this declaration, the variable can be used in the current java block in each constraint.

Expressions of the host language Java

It is only possible to use methods in guards and bodies of rules. The arguments of methods an arbitrary Java objects. A method invocation in is similar to a method invocation in Java. First an instance of the class, where the method is defined, has to be built. Then a new instance of the class `JMethod`, which knows the instance, the method name and the method arguments needs to be created. If the method is used in the guard or in the body of a rule, the return value has to be a boolean value. Only, when the method invocation is used as an argument of a constraint (this includes another method invocation), the return value can be any Java object. In the following example a method invocation is demonstrated:

```
// AClass contains the method 'foo',
// which needs an integer object as an argument
AClass o = new AClass();
JMethod foo = new JMethod("foo", o);
ObjectContainer[1] args = new ObjectContainer();
args.add(new Integer(3));
foo.setArguments(args);
```

In order to make a method invocation less complicated, it is recommended to create a new class, which is a direct heir of the class JMethod. This heir, needs a constructor with the same arity as the method which is desired to call. This constructor handles the adding of the method arguments. This make the code more readable. Example for the method "foo":

```
public class FOOMethod extends JMethod {

   ObjectContainer args;

   public FOOMethod(AClass object, Object arg0) {
```

[1]see Appendix **??**

```
        super("foo", object);
        args = new ObjectContainer();
        args.add(arg0);
        args.add(arg1);
        setArguments(args);

    }

}
```

The arguments of the constructor have the type "Object", in order to enable the usage of logical variables. After the arguments have been added to an instance of `ObjectContainer`, they are added to the current instance of `JMethod` with the method `setArguments`. The keyword `super` calls the constructor of the superclass JMethod. The class `FOOMethod` enables the following use of the method invocation of the method "foo":

```
// AClass contains the method 'foo',
// which needs an integer object as an argument
AClass o = new AClass();
FOOMethod fm = new FOOMethod(o, new Integer(3));
```

If the method is a static class method, the class name has to be given to the constructor instead of the object:

```
// AClass contains the static method 'boo',
// which needs an integer object as an argument
JMethod foo = new JMethod("foo", "AClass");
ObjectContainer args = new ObjectContainer();
args.add(new Integer(3));
foo.setArguments(args);
```

Constraints

Built-in Constraints A built-in constraint are inserted into the guard or the body of a rule. A built-in constraint, inserted into the goal, will be executed. If the execution fails, the evaluation of the goal fails. The implementation of the evaluator provides two kinds of built-in constraints: the syntactical equality ($\doteq$) and boolean constraints (`true` and `false`). A new boolean constraint is built with the constructor `Bool`:

```
Bool b1 = new Bool(``true'');
Bool b2 = new Bool(``false'');
```

The execution of `b1` will succeed and the execution of `b2` will fail. A new equality constraint is built with the constructor `Equality`. It has got two arguments, which can be any java objects (including constraints).

```
Object v = new Object(); cs.addVariable(v, "java.lang.Integer");
Equality e = new Equality(v, new Integer(1));
```

The execution of the equality constraint `v` would assign the value `1` to the variable `v`, if the variable is unbound. The return value of the execution would be "true". If `v` is bound to a value, the execution compares this value to `1`. If the value is equals to `1`, the execution succeeds, otherwise it fails. This equality can be compared to the unification in Prolog. If two unbound variables are compared to each other, the execution would have the effect, that all occurrences of the variable on the left hand side will be replaced by the variable on the right hand side. The execution of such an 'assignment' will have the return value "true". The return value of a comparison depends on the arguments of this constraint.

User-Defined Constraints The definition and usage of user-defined constraints represents a similar problem as the method invocation. There is a lot of additional information, which has to be given to a user-defined constraint. So it is recommended to create an extra heir of the class `UDConstraint` for each new user-defined constraint. A heir of `UDConstraint` information about the arity of the new user-defined constraint, about the argument classes and it needs a constructor with an adequate number of arguments. The following example represents a class with a constructor, to create a new constraint with the identifier "c", with arity two and with two float values as arguments. Additionally two methods for the argument access (`getArg1` and `getArg2`) and for cloning the constraint (`clone`) are added:

```
public class CConstraint extends UDConstraint {

   ObjectContainer args;
   ObjectContainer classes;

   public CConstraint(Object arg0, Object arg1) {

      super("c", 2);

      classes = new ObjectContainer();
      classes.add("java.lang.Float");
      classes.add("java.lang.Float");

      try {

         declareConstraint(classes);

         } catch(java.lang.ClassNotFoundException e) {

            System.err.println(""+e);

         }

         args = new ObjectContainer();
         args.add(arg0);
         args.add(arg1);
         setArguments(args);

      }
```

```
      public Object getArg1() {

         return args.get(0);

      }

      public Object getArg2() {

         return args.get(1);

      }

      public Object clone() {

         CConstraint tmp = new CConstraint(args.get(0), args.get(1));
         cloneInwards(tmp);

         return tmp;

      }

}
```

Comparing the constructors of `FOOMethod` and `CConstraint`, there is only one difference. A java method is declared by the signature of its definition. Therefore the heir of a class `JMethod` does not need any information about the classes of its arguments. A user-defined constraint is defined in the moment of its usage, so the information about the argument classes has to be added in the constructor. This is done by the method `declareConstraint`, which expects an instance of `Objectcontainer` as an argument. This `ObjectContainer` contains the names of the argument classes.
The method `cloneInwards`. handles the cloning of the arguments and some other flags. The heirs of `UDConstraint` and `JMethod` are automatically created by the compiler (see Section **??**).

Adding a goal to the constraint system We are now regarding a user-defined constraint with the identifier `leq` (the arity is two). This constraint is represented by a heir of `UDConstraint` called `LEQConstraint`. The heir is implemented analogously to `CConstraint`. A goal constraint is added to the constraint system by the method `addGoalConstraint` provided by the class `Constraint System`. So the goal (in JCHR)

```
variable java.lang.Integer A, B, C;
leq(A, B) && leq(Y, Z) && leq(B, C)
```

will be defined by the following JCHRCore instructions:

```
Object A = new Object; cs.addVariable(A, "java.lang.Integer","A");
Object B = new Object; cs.addVariable(B, "java.lang.Integer","B");
Object C = new Object; cs.addVariable(C, "java.lang.Integer","C");

cs.addGoalConstraint{new LEQConstraint(A, B));
```

```
cs.addGoalConstraint{new LEQConstraint(C, A));
cs.addGoalConstraint{new LEQConstraint(B, C));
```

Rules

As we have seen in Section **??** there are three kinds of rules. The rules are represented in by the classes `SimplificationRule`, `SimpagationRule` and `PropagationRule`. Basically the constructors of these classes have the same structure. They expect two or three arguments, depending on the existence of a guard. The first argument is an `ObjectContainer` object containing the head constraints. The second argument is either the guard or the body. If a guard is existing, the third argument is the body. Body and guard are `Objectcontainer` objects, too. There is only one exception. The constructor of the class `SimpagationRule` expects an object of the type `SimpagationHead` as first argument, in order to specify the head constraints, which are to be removed from the constraint memory. The following example shows the definition of the rules for the `leq` constraint in Section **??**.

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

The upper three lines declare three variable objects, which are required for the rule definitions. A variable declaration consists of the creation of a new variable object (`Object X = new Object;`) and of the registration of this variable in the constraint system object (`cs.addVariable(X, "java.lang.Integer","X");`). The registration of the variable is done with the method `addVariable`. It requires three arguments: the object, which represents the variable `X`, the type of the variable (`"java.lang.Integer"`) and optionally the name of the variable (`"X"`).
The definition of a rule always runs according to the same pattern. The constructor of a rule usually expects three `ObjectContainer` objects as arguments. Each `ObjectContainer` object contains the constraints or java method calls, which define the rule. The constraints or method calls are added to the `ObjectContainer` object with the method `add`. E.g. `refHead.add(new LEQConstraint(X, Y));` means, that the constraint `leq(X, Y)` is added to the `ObjectContainer` object, which represents the head of the the reflexivity rule of the `leq` handler.

```
// reflexivity
ObjectContainer refHead = new ObjectContainer();
ObjectContainer refGuard = new ObjectContainer();
ObjectContainer refBody = new ObjectContainer();

refHead.add(new LEQConstraint(X, Y));
refGuard.add(new Equality(X, Y));
refBody.add(new Bool("true"));

SimplificationRule reflexivity = new
       SimplificationRule(refHead, refGuard, refBody);
cs.addRule(reflexivity);
```

If a rule has an empty guard, the JCHRCore library provides also rule constructors, which expect only `ObjectContainer` objects for the head and the body (see the definition of the *antisymmetry* rule below). A rule object is added to the rule memory of the constraint system object by the method `add` provided by the class `ConstraintSystem`.

```
// antisymmetry
ObjectContainer antiHead = new ObjectContainer();
ObjectContainer antiBody = new ObjectContainer();

antiHead.add(new LEQConstraint(X, Y));
antiHead.add(new LEQConstraint(Y, X));
antiBody.add(new Equality(X, Y));

SimplificationRule antisymmetry =
    new SimplificationRule(antiHead, antiBody);
cs.addRule(antisymmetry);
```

The *idempotence* rule is a simpagation rule. The constructor of a simpagation rule expects an object of the class `SimpagationHead` instead of an `ObjectContainer` object for the head. This class is a wrapper for two `ObjectContainer` objects, representing the heads of a simpagation rule `new SimpagationHead(idemPlusHead, idemMinusHead)`: the first object `idemPlusHead` contains the constraints which are kept in the constraint store, the second object `idemMinusHead` contains the constraints, which are removed from the constraint store.

```
// idempotence
ObjectContainer idemMinusHead = new ObjectContainer();
ObjectContainer idemPlusHead = new ObjectContainer();
ObjectContainer idemBody = new ObjectContainer();

idemMinusHead.add(new LEQConstraint(X, Y));
idemPlusHead.add(new LEQConstraint(X, Y));
idemBody.add(new LEQConstraint(X, Y));

SimpagationRule idempotence =
    new SimpagationRule(new SimpagationHead(idemPlusHead, idemMinusHead),
                        idemBody);
cs.addRule(idempotence);
```

Last but not least, we need a constructor for propagation rules. For this constructor, it is possible to add a guard, too, but this is not needed for the *transitivity* rule. Everything else, concerning the construction of a rule, is equal to the other rule constructors.

```
// transitivity
ObjectContainer transHead = new ObjectContainer();
ObjectContainer transBody = new ObjectContainer();

transHead.add(new LEQConstraint(X, Y));
transHead.add(new LEQConstraint(Y, Z));
transBody.add(new LEQConstraint(X, Z));

PropagationRule transitivity = new PropagationRule(transHead, transBody));
cs.addRule(transitivity);
```

3.2.3 Implementation Aspects

Up to now, structural and syntactical aspects of this project have been discussed. This section intends to give an overview about the technical side of the JCHR evaluator.

The Constraint System

The class `ConstraintSystem` represents the main part of this package. So it has to provide an interface for the rule and goal definition, for the access to the evaluation results and the evaluation process itself. Furthermore, there are methods to store the current state of the system (see Section **??**) for debugging purposes. The interface for the rule and goal definition is described in Section **??**. In the following the evaluation algorithm will be explained.

When the constraint system is initialized with rules and a goal, the evaluation process is started with the method `callGoal`. The constraints, stored in the constraint system, will be evaluated, until the execution of a built-in constraint fails (`false` will be inserted into the built-in constraint memory) or until no rule can be applied to the user-defined constraint memory. This is the case, when all user-defined constraints have been removed from the memory (a), or when all rules have been tried to be applied on the user-defined constraint memory, the memory state did not change during this period and there is no untested (active) constraint left (b).

If a rule fires for an active constraint, the memories (user defined constraint memory, built-in constraint memory) are updated, according to the rule properties (details on firing and applying rules see Section **??**). Furthermore, this rule is added to a rule history for debugging purposes. If the currently active constraint has been removed from the user-defined constraint memory (in case of a simplification or a simpagation rule), the next user-defined constraint, which is waiting in a queue, has to be activated. If there is no untested constraint left, the algorithm finishes. Otherwise, when the currently active constraint has not been removed, the rule loop is reinitialized and the search for firing rules begins again. But now, rules, which have already fired with this active constraint, may not fire again, at least not with the same combination of partner constraints. When no rule can be found to fire with the currently active constraint, it is set passive and the next constraint in the queue is taken to be active. An activity diagram of this algorithm can be seen in Figure **??**.

Handling of Variables

As we have seen above, the handling of logical variables in the evaluator is centralized in a variable table, in order to keep the occurrences of each variable consistent. Furthermore, the additional information for each variable can be stored in the table. This keeps the variable objects small. A central variable handling presupposes, that every constraint, used in a rule or as a goal constraint, needs to know the variable table. As we have seen in Figure **??**, the variable table (class `VariableTable`) is part of the main class `ConstraintSystem`. So this problem can be solved by adding the variable table to each constraint added to the system as a goal constraint (with the method `addGoalConstraint`) and to each rule added to the system as a rule (with the method `addRule`). Each type of constraint inherits a method called `setVTable`. When a constraint is added to the system, this method is called with the variable table object as argument. The same is done for each rule. The method `setVTable` of the class `Rule` adds the variable table to each head, guard and body constraint. This technique makes changes or addings, done to the variable table, known to each constraint and rule.

JCHR like Java is typed. So, beside the binding of a variable some other information has to be stored in the variable table. The most important one, are the java classes associated to a certain variable. For debugging purposes, the evaluator needs a variable name. This name can be given to the variable table optionally. Otherwise a name, which is automatically generated by the variable table is associated to the variable object. With all

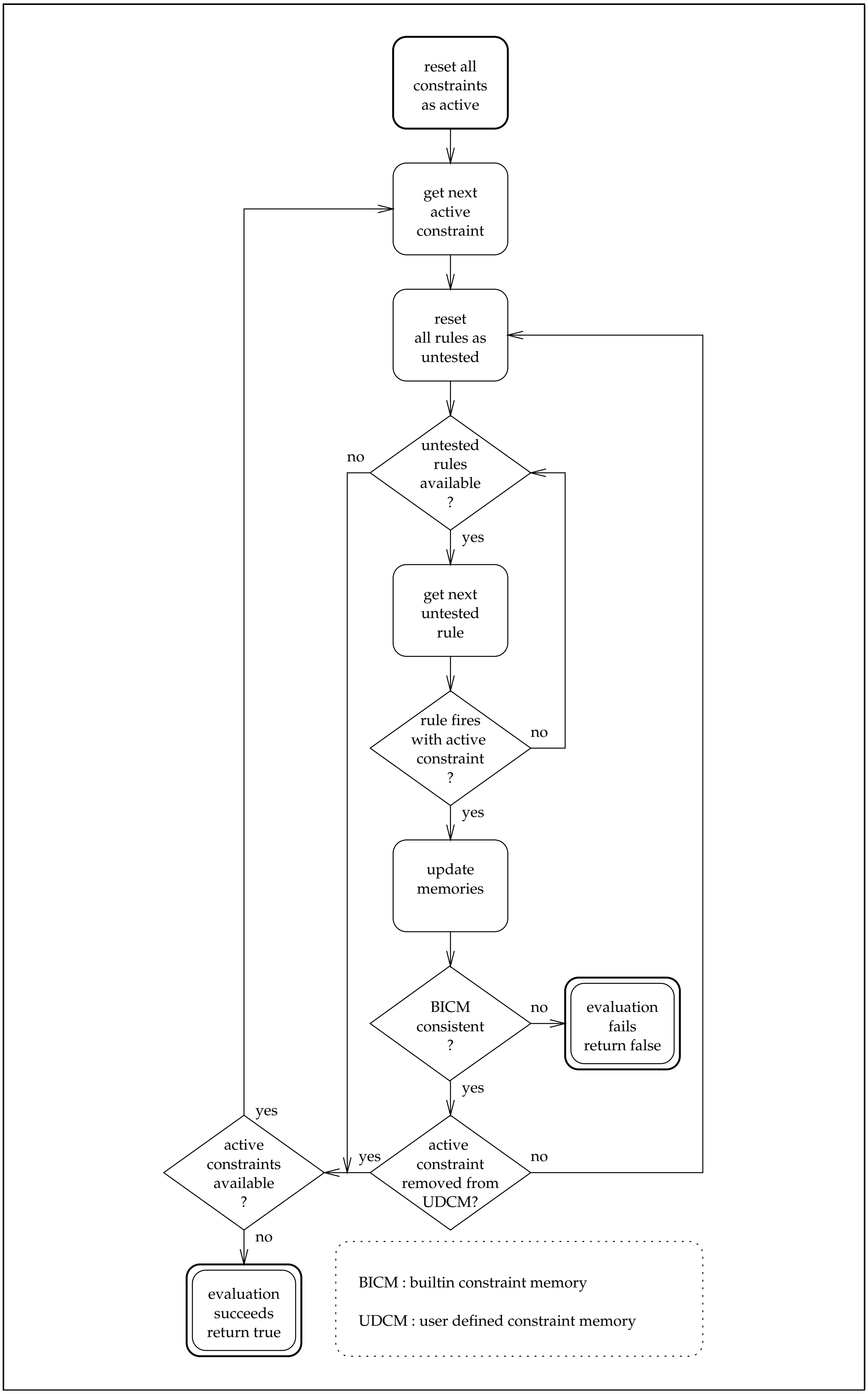

Figure 3.5: The evaluation algorithm.

this, the class `VariableTable` consists of three objects of the type `java.util.Hashtable`, to associate each variable object with the respective information. Of course there are also methods for the access to each hash table.

Constraints

JCHR distinguishes between two kinds of constraints: user-defined constraints and built-in constraints.

User-Defined Constraints The main task user-defined constraints have to perform, is the matching of user-defined constraints. The decision, whether a rule fires or not is made, when there are instances of the head constraints of the rule (and when the execution of the guard succeeds) in the user-defined constraint memory. An instance of a constraint is a less common variant of this constraint. The common constraint matches the instance. The following examples try to clarify this circumstance. Let `p/2`, `p/3` and `r/1` be constraint symbols. `X`, `Y` and `Z` are variables and `a` and `b` are constants.

```
p(a, b) is an instance of p(a, X)
p(Z, a) is an instance of p(X, a)
p(X, Y) is not an instance of p(a, Z)
p(a, b, c) is not an instance of p(X, Y)
r(X) is not an instance of p(a, b)
```

Formally matching can be introduced as follows: T_1 and T_2 are Term, which may have variables as arguments. T_1 is an instance of T_2, if

$$\exists\sigma\ T_1 \doteq T_2\sigma.$$

Thereby, σ is a substitution, that is a set of bindings for the variables in T_2. The symbol $\doteq$ describes the syntactical equality of terms.

	bound variable	unbound variable	constant
bound variable	Matching succeeds if the bindings are equal, or the binding of the possible instance appears in the chain, possibly started by the generalization	Matching succeeds.	Matching fails.
unbound variable	Matching succeeds only, if the generalization starts a variable chain, which includes the possible instance.	Matching succeeds.	Matching fails.
constant	Matching succeeds, if binding equals constant, otherwise it fails	Matching succeeds.	Matching succeeds, if constants are equal, otherwise it fails

Figure 3.6: The cases of the matching algorithm.

The implementation of the matching of two constraints starts with the comparison of the identifiers and the arities of both constraints. After that, the argument lists are compared. In any case the arguments of the possible instance have to be less general, as the arguments of the possible generalization, or the arguments are equal. Figure **??** illustrates all cases of matching the arguments of terms. Thereby, the first row describes the cases, an argument of T_2, the generalization, can take. The first column describes the cases, an argument of T_1, the possible instance, can take. Actually, there is no difference in the matching of two bound variables and two constants. But the implementation differs this cases to achieve a better structure. When matching an unbound variable against a bound variable or a constant, it is necessary, that the types of the unbound variable and the the types of the constant and the binding of the bound variable are equal, because JCHR (and of course JCHRCore as a Java library) is typed.

Built-In Constraints Built-in constraints are used in goals, in guards and bodies of rules. Inserting a built-in constraint into a goal causes its execution. The built-in constraints in the guard of a rule have to be executed, too. For this case there exists the restriction, that this execution may not cause variable bindings, which have effects on the body of the rule. So the interesting aspect, concerning built-in constraints, is the execution.
The built-in constraint equality, in this case, is not only a syntactic test, but also an assignment. The second functionality is used, when one of the both arguments of an equality constraint is an unbound variable. There is only one exception. In the guard of a rule an equality constraint is only a test, because the execution in a guard may not have any effects on the body of the rule.
Figure **??** shows all cases of the execution of a equality constraint, depending on the arguments. The first row and the first column represent the left and the right argument of an equality constraint, respectively. Each case has an identifier, positioned in front of the text (e.g. **AA**), in order to simplify the referencing.
If a bound variable is involved in an equality constraint, a test similar to an occur check has to be performed, in order to avoid unnecessary assignments in variable chains.
For Instance, the variable `A` initiates a variables chain (so `B`, `C`, `D`, `E`, `F` are variables, too):

$$\texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{C} \rightarrow \texttt{D} \rightarrow \texttt{E}$$

The execution of the equality constraint `A = X` is true and the variable `X` is added at the end of the chain. The execution of the equality constraint `A = C` is true, too. But `C` is not appended, because it already occurs in the chain. So it has to be tested, whether a variable occurs in an existing chain, before each assignment. If this test is not performed, the execution of `A = C` would cause the following circle:

$$\texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{C} \rightarrow \texttt{D} \rightarrow \texttt{E} \rightarrow \texttt{C} \rightarrow \texttt{D} \rightarrow \texttt{E} \rightarrow \texttt{C} \ldots$$

This is crucial for matching and for the execution of equality constraints which have one or more of the upper variables as arguments. Of course it is necessary, to compare the class of an unbound variable and the object, to which the variable is intended to be bound, in order to keep the system typed.

	bound variable	unbound variable	constant
bound variable	(**AA**) Each binding is evaluated (un-chaining a variable chain, method invocation, etc.). Then, both results are compared. The execution succeeds, if both objects are equal, otherwise it fails. If one of the bindings is a unbound variable, the execution can be reduced to the case (**BB**). If the results are two unbound variables, the execution is reduced to case (**BA**).	(**AB**) The binding of the bound variable is evaluated. If the result is a unbound variable, the execution is reduced to case (**BA**), otherwise the unbound variable is bound to the result (see case (**BB**)).	(**AC**) The binding of the variable is evaluated . Then the binding and the constant are compared. The execution succeeds, if binding and constant are equal, otherwise it fails.
unbound variable	see case (**AB**)	(**BA**) The variable on the left hand side is bound to the variable on the right hand side. Each further appearance of the left variable will be replaced by the right variable. The execution fails, when the variables are unequal, otherwise it succeeds.	(**BB**) The constant is assigned to the unbound variable. The execution succeeds.
constant	see case (**AC**)	see case (**BB**)	(**CA**) Both constants (objects) are compared. The execution succeeds, if both objects are equal, otherwise it fails.

Figure 3.7: The cases of the equality execution.

Executing a boolean built-in constraint is trivial, because the value of the respective constraint represents the return value of the execution.

Even, if method is not a built-in constraint, the presuppositions of an execution of a method are very similar to those of the execution of a built-in constraint. The execution of methods is handed over to the mechanisms supplied by Java (implemented in `java.lang.reflect`). For a correct execution of a java method the name of the method and the classes of its arguments have to be handed over to the method `getMethod` provided by the class `java.lang.Class`. The Method returns a `java.reflect.Method` object, which provides a method called `invoke`. This method expects the instance, where the method, which is intended to be executed, belongs to and the arguments of the method. The call of the method `invoke` executes the method. All this is implemented in the class

`JMethod` by the methods `execute` and `executeMethod`.

Rules

The main functionality of the rules is implemented in the superclass `Rule`. The subclasses `SimplificationRule`, `SimpagationRule` and `PropagationRule` only provide comfortable constructors and `toString` methods for nice output. The class `Rule` provides methods for executing the guard and for applying the rule. The applicability of a rule includes the following steps:

- find instances for the head constraints
- execute the guard
- insert body constraints into the appropriate constraint memory

These steps are performed by the method `apply`:

```
public apply(UserDefinedConstraintMemory udcm,
             BuiltInConstraintMemory bcm,
             UDConstraint constraint,
             boolean simpRule)
```

The arguments `udcm` and `bcm` are memories of the current constraint system. The argument of the type `UDConstraint` is the currently active constraint and the `boolean` argument indicates whether the current rule is a rule where head constraints are to be removed from the user-defined constraint memory. This argument is true, when the current rule is a simplification rule or a simpagation rule. It is necessary to indicate this aspect to the method `apply`, because the main evaluation algortihm tests the applicability of a rule not for the special instance (`SimpagationRule`, `SimplificationRule` or `PropagationRule`), but for the more common super class `Rule`, in order to avoid expensive type casting.
The search for instances of the head constraints of the rule is done in the following way. For each head constraint, except the one, which matches with the current active constraint, an instance is searched in the user-defined constraint memory. In case of success the guard of the rule is executed. If the guard succeeds, the body constraints are inserted into the memories. Thereby, the built-in constraints are executed immediately, in order to recognize a failure earlier. The user-defined constraints of the body are inserted into the user-defined constraint memory. After all this, the method `apply` returns `true`. It returns false in all other cases, described above.
There is a particularity, when the rule is a propagation rule. A propagation rule does not remove the active constraint from the user defined constraint memory. So it has to be avoided, that the currently active constraint causes the rule to fire with the same combination of user-defined constraints again. This is achieved by storing the combinations of firing head instances in an extra data structure (the `ObjectContainer firingHeadInstance` in `Rule.java`) provided by the class `Rule`, and searching for occurances of other firing combinations in this data structure.

User-Defined Constraint Memory

In this section, the user-defined memory is described, because it is the more complicated, compared to the rule memory and the built-in constraint memory, which are only simple containers for java objects (see Appendix **??**), with some additional methods for access. The user-defined constraint memory provides two main functionalities.

- handling of active constraints
- storing user-defined constraints with a path index [?]

The active constraints are stored in a queue, in order to ensure fairness. Before newly added constraints become active, the old stored constraints need to be worked off. Otherwise a certain combination of goals and rules may cause evaluation loops. Let $p_{/1}$ and $q_{/1}$ be constraints, `X` and `Y` variables, `a` and `b` constant and `p(a), q(b)` a goal. With the rules

```
p(X), q(Y) ==> q(Y)
p(X) <=> true
```

and `p(a)` as active constraint, this circumstance can be clarified: The goal causes the first rule to fire, so that the constraint `q(b)` is inserted into the constraint memory. Since the active constraint has not been removed, it is crucial to set the newly inserted constraint `q(b)` as the active constraint. This would cause the propagation rule to fire again and a loop is initiated. Using a queue for the active constraints guarantees, that both rules are tried with the current active constraint `p(a)`. The simplification rule removes the active constraint from the memory. The evaluation of the goal terminates.

The user-defined constraints are stored in a data structure called path index. Terms (as constraints) can be described by paths. For example the user-defined constraint `coo(7, X)` can be identified by the following paths:

```
[0, coo]
[0, coo, 1, 7]
[0, coo, 2, *]
```

The first path says, that the constraint has the identifier `coo`. It delivers no more information. The second path enlarges the information about the constraint. It says, that the first argument of the constraint is an integer with the value 7. The third path indicates, that the constraint has a variable as the second argument (represented by *). So each constraint can be described by a set of paths. This can be used, when instances of a constraint are searched. Searching all instances for the constraint `coo(7, X)` (out of a set of constraints) requires to create a set of pattern paths, which describe the instances. For each pattern path, a set of constraints have to be searched. The result is a set of sets of constraints. The intersection of all sets of constraints is a set of instance candidates for `coo(7, X)`. The set of pattern paths for this constraint consists only of one path:

```
[0, coo, 1, 7]
```

An instance of `coo(7, X)` only needs the integer 7 as the first argument. The second argument is arbitrary. Now all constraints, which are stored under this path have to be searched in the path index. The results are instance candidates for `coo(7, X)`. The constraint `coo(7, 3, X)` is stored under the path `[0, coo, 1, 7]`, too. It differs in the arity from the constraint `coo` mentioned above, but if will be added to the set of candidate constraints, because it is store in the same data structure like the constraint `coo(7, 3)`, concerning the path, which describes the first argument. So the results still need to be tested, whether they are instances of `coo(7, X)` or not. But this technique decreases the number of candidates [?].

3.3 The JCHR-Compiler

With the JCHR-Compiler, the rules and (testing) goals, specified in JCHR, are translated to Java and JCHRCore, respectively. Since JCHRCore is a Java library, which is easy to understand, it is possible, to create rules directly in Java. But the compiler is a useful tool, if CHR is already known from Prolog (or a similar CHR based language). Nevertheless it is necessary to acquire some rudimentary knowledge in JCHRCore, in order to formulate own goals, when merging Java and JCHR (more information about this in Section **??** and Chapter **??**). This section describes the package structure of the compiler, goes into detail concerning the implementation, gives a helping hand, with the usage of the compiler and explains the result of a compilation process.

3.3.1 The Package Structure

The basic package structure of the compiler is shown in Figure **??**. There are three main classes is this package: `JCHRLexer`, `JCHRParser` and `JCHRGenerator`. The class `JCHRLexer` scans a JCHR file and generates a token stream. The class `JCHRParser` generates an abstract syntax tree out of this token stream. The class `JCHRGenerator` transforms this abstract syntax tree into Java and JCHRCore, respectively.

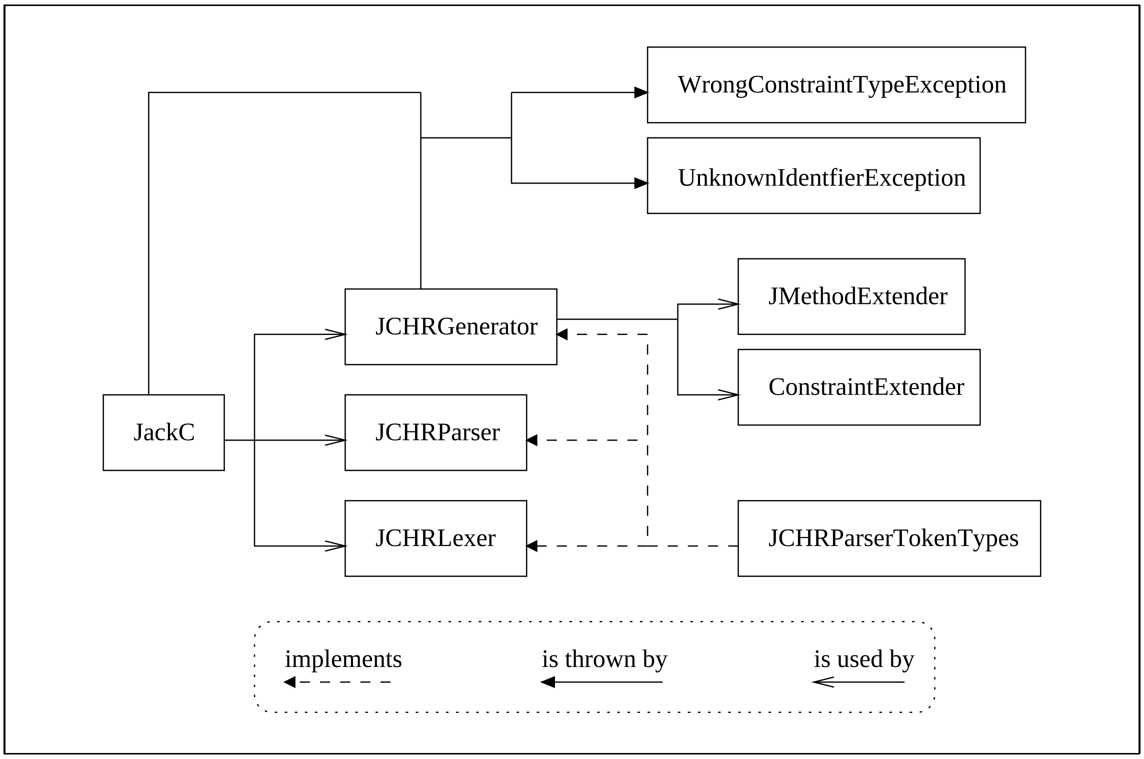

Figure 3.8: The basic class structure of the compiler.

As we have seen in Section **??**, it is useful for the readability of the generated JCHRCore code, to built heirs of `UDConstraint` and `JMethod` for each new user-defined constraint and for each method call. This is done bye the classes `JMethodExtender` and `ConstraintExtender`, which are called by the class `JCHRGenerator`. The interface `JCHRParserTokenTypes` provides constants for a symbolic usage of the tokens generated by `JCHRLexer`.
The exception `WrongConstraintTypeException` is thrown by the class `JCHRGenerator`, when a constraint, which is not declared as a user-defined constraint, is tried to be added to the head of a rule. It is also thrown, when a user-defined constraint is intended to

be added to a guard of a rule. The exception `UnknownIdentifierException` is thrown, wherever an identifier, which is not declared in one of the declaration sections, is used at a position, which needs a declared identifier.
The classes `JCHRLexer`, `JCHRParser` and `JCHTParserTokenTypes` are automatically generated by the compiler tool ANTLR [?]. The grammar of JCHR is specified in a language provided by ANTLR in a file called `jchr.g`.

3.3.2 Implementation Aspects

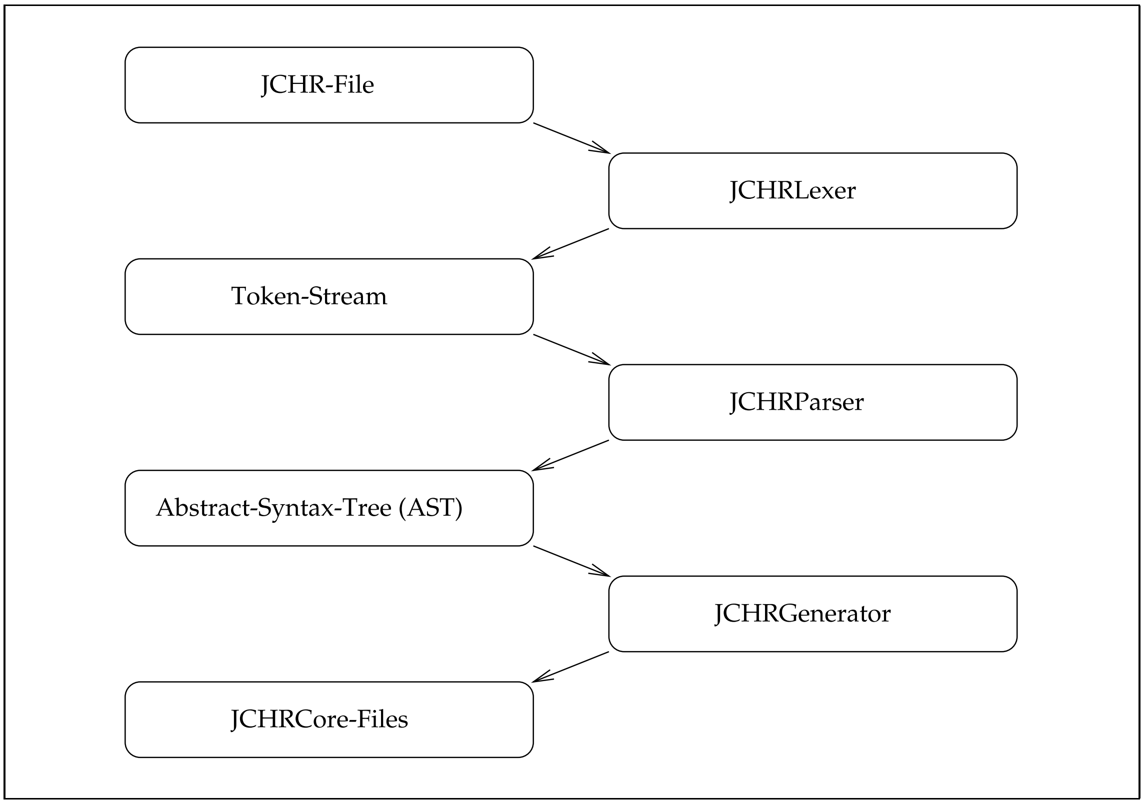

Figure 3.9: The detailed compilation process.

As mentioned above, a compilation task can be divided into three parts: the generation of a token stream, the generation of an abstract syntax tree and the generation of JCHRCore code. The detailed steps are shown in Figure **??**. This segmentation is advantageous concerning modularity. When a changement in JCHR is necessary, it is done in the JCHR grammar file (`jchr.g`). Compiling this grammar file with the ANTLR compiler, the classes `JCHRLexer`, `JCHRParser` and `JCHRParserTokenTypes` are generated [?]. Even the abstract syntax tree (AST) is generated automatically. After this only class `JCHRGenerator`, which produces the JCHRCore classes out of the AST, needs to be adapted to the changes. However, splitting the compilation process, causes a loss of information. For instance it is not possible to assign the line number to a detected error, when the error occurs after the parsing process. This is a hint to ANTLR, to provide the line number information in the automatically generated AST.
The main class of this package is `jchrc`. An object of this class is created with the following constructor:

```
public jchrc(Reader read,
```

```
            Writer out,
            Writer err,
            String outdir,
            boolean quiet)
```

The argument **read** is an object of the class `java.io.Reader`, from where the compiler can read the JCHR file (e.g. an object of the class `java.io.FileReader`). In order to redirect output and error messages the new `jchrc` object expects two Writers (**out** and **err**). Sometimes it is useful to write the result files to another directory as the current directory. This can be specified by the argument **outdir**. The boolean value **quiet** causes (if `true`) the compiler to suppress the standard output messages. The class `jchrc` provides a method called `compile`, which may throw a `java.io.IOException`. The result of this method is a boolean value. If it returns `true`, the compilation process has been successful and all necessary files have been generated. If it returns `false`, an error message is written to the member variable `error`. The main translation process is done by the class `JCHRGenerator`, which reads the AST from the parser. Then the AST is walked through by the method `writeFile` provided by `JCHRGenerator`. More details about the implementation can be obtained by viewing the commented code [**?**].

3.3.3 Using the JCHR-Compiler

This section is intended to be a quick start manual to use the JCHR-Compiler and its results. So we provide a description of the command line call of the JCHR-Compiler and its parameters and a description of the results of the compilation process.

The JCHR-Compiler is a Java application which is called as follows:

```
java jchrc [options] "filename".jchr
```

The input file needs to have an "jchr" extension. With the option **-o** an output directory for the compilation result can be specified. If no output directory is given, all results will be written to the current directory. The option **-quiet** causes the compiler not to generate messages.
Assuming, the handler described in the JCHR file is called "`leq`". Then the compiler, in any case, produces a file called `leqHandler.java`. Depending on the number of user-defined constraints and method calls, several additional output files are generated. Let the constraint `leq` be defined by the following line:

```
constraint leq(java.lang.Integer, java.lang.Integer);
```

The compiler generates a file with the name `LEQConstraint.java`. It contains java code similar to the code shown in Section **??**. As explained in this section, this class serves to increase the readability of the JCHRCore code in `leqHandler.java`. The first call of a method induces to compiler to create an additional file, too.

```
AClass aclass = new AClass();
... && aclass.foo(1, 2) && ...
```

Section **??** describes the circumstances of a method call like `aclass.foo(1, 2)`. So a file `ACLASSFOO2Method.java` is generated. The names of all additional generated files are printed to the standard output, unless the option **quiet** is not used. Furthermore, all those files are documented in a comment in the main file ("handlername"Handler.java). For instance, the compilation of the handler `min` in Section **??** generates the following output:

```
jchrc (JCHR to JCHRCore Compiler) Version 0.0.1
./MINConstraint.java was created!
./INTUTILLT2Method.java was created!
./INTUTILLE2Method.java was created!
./minHandler.java was created!
./minHandler.java compiled! Class file generated!
```

The integration of these results into an own java application or applet is described in next section.

Chapter 4

Finite Domains in JCHR

This chapter introduces the constraint system Finite Domains (*FD*). With *FD* combinational problems, where variables are only bound to values of a certain domain, can be solved. The following sections describe the implementation of the constraint system *FD* in JCHR.
Finite domains are represented by enumerations or by intervals. In an enumeration representation all values of the domain are listed. An interval finite domain only specifies upper and lower bounds of a domain. To each finite domain, a variable is assigned. This variable can be bound to each value of the domain. An implementation of the finite domains in Prolog is given in Appendix **??**, to support a better understanding for prolog programmers. The constraint handler is initiated by the keyword `handler` and has the name `fd`.

```
handler fd {

...

}
```

Between the brackets, used classes, constraints, rules and goals are specified. First, we explain the common declarations of the used classes and the constraints.

```
class java.lang.Integer;
class IntUtil;
class nl;
```

The handler uses three classes. `java.lang.Integer` is a java standard class and provided by the java development kit. This is the type for the finite domains. The class `IntUtil` provides methods for handling `java.lang.Integer` objects and comparing them. Since we use the enumeration representation of the finite domains, a class is required, which represents lists in java. The class `nl` (i.e. nested list) provides such a functionality. Beyond this, several constraints handling the finite domains are needed.

```
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);
constraint fdAdd(java.lang.Integer, java.lang.Integer, java.lang.Integer);
constraint fdMult(java.lang.Integer, java.lang.Integer, java.lang.Integer);
```

`fdEnu` and `fdInt` are the JCHR representations for enumeration and interval domains. The first argument of each constraint, is a variable, associated with the domain. The second argument of `fdEnu` is a nested list, which contains all values of the domain. The last two arguments of `fdInt` represent the borders of the domain. In the following, only the rules for the enumeration representation are implemented, because the implementation of the interval representation is analogous. The meaning of both kinds of domains, can be described by the following definitions:

$$\texttt{fdEnu}(X,\texttt{new nl}(k_1,\texttt{new nl}(k_2,\ldots\texttt{new nl}(\texttt{k}_\texttt{n}))\ldots)) \leftrightarrow X = k_1 \vee \ldots \vee X = k_n$$
$$\texttt{fdInt}(X,n,m) \leftrightarrow n \leq X \wedge X \leq m$$

`fdLe`, `fdLt` and `fdNe` are comparing constraints, expecting variables as arguments, which are associated with a finite domain. `fdAdd` and `fdMult` generate new domains out of two given domains, by adding and multiplicating the values of the given domains by pairs.

The declaration of the rules is included by the keyword rules and the usual brackets.

```
rules {

...

}
```

At first we need to declare some variables used by the rules.

```
variable java.lang.Integer X, X1, X2, X3, Y, Z, N;
variabel java.lang.Integer Min, Max, MinX, MaxX, MinY, MaxY, MaxY1, MinX1;
variable nl D, D1, D2, D3, L, L1, L2, L3;
```

The first rule (called `empty`) replaces a domain constraint with an empty list by false. So empty enumeration domains cause the constraint handler to fail.

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

The rules `intersection1` and `intersection2` implement the intersection of two domains for one variable. Each rule replaces both domains with one domain. The first rule intersects two enumeration domains by calling the method `intersection` of the class `nl`. The second rule adapts the enumeration list to the borders of the interval domain. Therefore the method `removeLower` and `removeHigher`, implemented by the class `IntUtil` are needed. Both expect `java.lang.Integer` objects and nested lists as arguments. From this lists, all integer values, which are lower or higher than the values in the first argument position, are removed from the given lists.

```
{ fdEnu(X, L1) && fdEnu(X, L2) } <=>
  { L = nl.intersection(L1, L2) && fdEnu(X, L) } intersection1;

{ fdEnu(X, L) && fdInt(X, Min, Max) } <=>
  { L1 = IntUtil.removeLower(Min, L) &&
    L2 = IntUtil.removeHigher(Max, L1) &&
    fdEnu(X, L2) } intersection2;
```

Comparisons between domains are handled by rules `leMax` and `leMin`. Comparing finite domains means, that every value from the one domain has to fulfill the named relation concerning every value from the other domain. For instance, let a goal[1] be

```
fdEnu(A, new nl(1, nl(2, new nl(3)))) &&
fdEnu(B, new nl(2, new nl(3, new nl(4)))) &&
fdLe(A, B)
```

This goal causes no change of the constraint memory, because every value of the domain associated with the variable `A` is smaller or equal to a corresponding value in the domain associated with the variable `B`. Against it, the goal

```
fdEnu(A, new nl(1, nl(2, new nl(3)))) &&
fdEnu(B, new nl(2, new nl(3, new nl(4)))) &&
fdLe(B, A)}
```

would cause the rules `leMin` and `leMax` to fire. They would add the constraints `fdInt(B, 2, 3) fdInt(A, 2, 3)` to the constraint memory. Then the rule `intersection2` would generate new representations for the domains, represented by the variables `A` and `B`. Thereby the following replacements are performed:

```
fdEnu(A, new nl(1, nl(2, new nl(3)))) && fdInt(A, 2, 3) <=>
fdEnu(A, nl(2, new nl(3)))

fdEnu(B, new nl(2, new nl(3, new nl(4)))) && fdInt(B, 2, 3) <=>
fdEnu(B, new nl(2, new nl(3)))
```

Now all values from the domain associated with the variable `B` are less or equal to the values of the domain associated with the variable `A`.

```
if (MinX = IntUtil.minList(L1) &&
    MinY = IntUtil.minList(L2) &&
    IntUtil.gt(MinX, MinY))
   { fdLe(X, Y) && fdEnu(X, L1) && fdEnu(Y, L2) } ==>
     { MinX = IntUtil.minList(L1) &&
       MaxY = IntUtil.maxList(L2) &&
       fdInt(Y, MinX, MaxY) } leMin;

if (MaxX = IntUtil.maxList(L1) &&
    MaxY = IntUtil.maxList(L2) &&
    IntUtil.gt(MaxX, MaxY))
   { fdLe(X, Y) && fdEnu(X, L1) && fdEnu(Y, L2) } ==>
     { MinX = IntUtil.minList(L1) &&
       MaxY = IntUtil.maxList(L2) &&
       fdInt(X, MinX, MaxY) } leMax;
```

The rules `ne1` and `ne2` handle equalities between domains and integer values, too. They determine, when two domains are unequal. Both rules are simpagation rules. `ne1` adapts a given domain `D`, if the value, bound to the variable `X`, is a member of the domain. The

[1] `new nl(1, nl(2, new nl(3)))` corresponds with the creation of a new list `[1,2,3]` in Prolog style and `A` and `B` are variables of the type `java.lang.Integer`

rule inserts a new domain, which does not contain the value and removes the old domain. Than the next can be applied. It removes the `fdNe` constraint from the memory, if the given domain does not contain the value bound to the variable `X`.

```
if (IntUtil.member(X, D))
   { fdNe(X, Y) &\& fdEnu(Y, D) } <=>
     { D1 = IntUtil.remove(D, X) && fdEnu(Y, D1) } ne1;

if (IntUtil.notMember(X, D))
   { fdEnu(Y, D) &\& fdNe(X, Y) } <=>
     { true } ne2;
```

The last two rules `addition` and `multiplication` are adding and multiplicating, respectively, all values of two domains with each other and unify the results in one list. For example the goal `fdAdd(X, Y, Z) && fdEnu(X, new nl(1, new nl(2))) && fdEnu(Y, new nl(3, new nl(4)))` would cause the rule `addition` to fire. Then the constraint `fdEnu(Z, new nl(4, new nl(5, new nl(6))))` would be inserted into the memory. The rule `multiplication` reacts analogous.

```
{ fdAdd(X, Y, Z) && fdEnu(X, L1) && fdEnu(Y, L2) } ==>
  { L3 = IntUtil.allAddition(L1, L2) && fdEnu(Z, L3) } addition;

{ fdMult(X, Y, Z) && fdEnu(X, L1) && fdEnu(Y, L2) } ==>
  { L3 = IntUtil.allMultiplication(L1, L2) && fdEnu(Z, L3) } multiplication;
```

Additional rules for the cases, when only `X` or `Y` are unknown, need to be implemented. The complete implementation of the FD constraint handler can be seen in Appendix **??**

Chapter 5

Interaction between Java and JCHR (finite domains)

This chapter describes the interface between Java and JCHR and shows an approach of the implementation of a constraint problem, in order to explain how to write constraint based applications with the class library, introduced in this work. Therefore the finite domains constraint handler is used. Furthermore, it is intended to clarify, which parts of such an application need to be changed, when another constraint problem, similar to the existing and based in the finite domains, is going to be implemented. The implementation of constraint problems with Java and JCHR is always done according to the same pattern:

1. modeling of the problem
2. labeling of the variables.

Thereby the first step is problem dependent. It consists of building a theoretical model for the problem, implementing the constraints which need to be fulfilled and the association of the problem variables with domains. This means to declare the variables for the constraint system (as it is done in Section ??) and to insert constraints, which correlate these variables with finite domains (known from Chapter ??). Labeling the variables means to instantiate them systematically. Beyond it the labeling algorithm actives the constraint system in this implementaton, in order to test, whether the instantiation fulfills the problem dependent constraints. This is done by a method called `labeling` provided by the class `fdUtil`. The class `fdUtil` is a utility class for handling constraint satisfaction problems, which use the finite domains constraint handler. The method `labeling` expects a `ConstraintSystem` object and an `ObjectContainer` object as arguments. In the constraint system object all rules, which are needed to solve the given problem, are implemented. The `ObjectContainer` object contains the variables which are to be labeled. The method `labeling` calls the method `treeSolve`, which implements the actual backtracking algorithm. The method `treeSolve` is also responsible for the instantiation of the variables, for the activation of the constraint system and for the result retrieval. It is called with three arguments. The first argument `n` indicates the number of queens, which are to be placed on the board. The second argument `var` is a counter variable and the third argument `variables` is an `ObjectContainer` object, containing the variables.
As long as the counter variable `var` is lower than `n` the method is calling itself recursively. If the counter variable is higher than `n` the algorithm has reached a leave. This means that a solution has been found and that all variables in the `ObjectContainer` object `variables` are bound to an integer value. Then, those values are collected and stored into a set of

solutions (in line 35)[1] by the method `getVarValue`. If no more solutions can be found, this set is returned to the calling methods `labeling` and `solve`.
A recursive call means, that the algorithm is walking down to the next level of the search tree. A node of the search tree is represented by an instantiation of a variable, which in turn represents a column of the board. So a node of the search tree gives information about the position of the current queen. The sons of a node are represented by the instantiation possibilities of the next column, i.e. the values in the enumeration of the domain, associated to the variable, which represents the next column. Therefore, before a branch of the search tree is able to be followed, the enumeration of the domain, associated to the current variable, has to be retrieved from the constraint system. This is done in the lines 10 and 11 by the method `getVariableEnumeration`, provided by the class `fdUtil`, too. The current variable is instantiated with each value in the enumeration. For each instantiation, the constraint system is activated (by the method `callGoal` in line 18). This evaluates the enumeration domains for the next variable (representing the next column). If the evaluation is successful, save positions for the next queen in the next column have been found. In this case, the method `treeSolve` is called recursively, with an increased counter variable. The algorithm ends, when there has been evaluated a save instantiation for all variables.
In the lines 14 and 26 an important functionality of the class `ConstraintSystem` is used. In order to provide the same states for each sons of a node in the search tree, the memories of the constraint system have to be rescued, before the algorithm is walking down the current branch. This is done by the method `getEnvironment` in line 14. The class `Environment` is a container for the user-defined memory, for the built-in constraint memory, for a rule history and for the variable table. The method `getEnvironment` creates a copy of these memories and tables. After the algorithm has returned to the same level of the tree, memories states are rewritten to the constraint system by the method `setEnvironment`.

```
1:  public static ObjectContainer labeling(ConstraintSystem cs,
2:                                          ObjectContainer solution) {
3:
4:      return treeSolve(cs, solution.size(), 0, solution);
5:
6:  }
```

```
1:  public ObjectContainer treeSolve(ConstraintSystem cs, int n, int var,
2:                                   ObjectContainer variables) {
3:
4:      ObjectContainer solutions = new ObjectContainer();
5:      ObjectContainer result = new ObjectContainer();
6:      Environment env;
7:
8:      if (var < n) {
9:
10:         ObjectContainer enumeration =
11:                     getVariableEnumeration(cs, variables.get(var));
12:         for (int i=0; i<enumeration.size(); i++) {
13:
14:             env = cs.getEnvironment();
15:
```

[1]Line numbers refer to the code segments below the respective text segment.

```
            cs.setVarValue(variables.get(var),
                           enumeration.get(i));
            if (cs.callGoal()) {

                result = treeSolve(cs, n, var+1, variables);
                for (int k=0; k<result.size(); k++)
                    solutions.add(result.get(k));

            }

            cs.setEnvironment(env);

        }

        return solutions;

   } else {

      for (int j=0; j<variables.size(); j++)
          result.add(cs.getVarValue(variables.get(j)));

      if (result.size() > 0)
         solutions.add(result);

      return solutions;

   }

}
```

The method `getVariableEnumeration`, provided by the class `fdUtil` is responsible for the retrieval of the enumeration of the domain, which is associated to the given logical variable (argument `var`). Thereby all enumeration-domain-constraints are retrieved from the constraint memory with the method `getInstances` provided by the class `ConstraintSystem` (line 7 and 8). This method uses the path index, which organizes the user-defined constraints, to find instances of the given constraint. Then the constraint with the variable `var` as the first argument is searched by comparing all first arguments of the enumeration domain constraints with the given variable. If the constraint is found, the second argument, which is a nested list, is transformed to an `ObjectContainer` object and is returned (line 16). It is obvious, that further work has to be done, to make the result retrieval more efficient.

```
1:  public ObjectContainer getVariableEnumeration(ConstraintSystem cs,
2:                                                Object var) {
3:      Object dummy1 = new Object();
4:      Object dummy2 = new Object();
5:      cs.addVariable(dummy1, "java.lang.Integer");
6:      cs.addVariable(dummy2, "java.lang.Integer");
7:      ObjectContainer candidates =
8:                      cs.getInstances(new FDENUConstraint(dummy1, dummy2));
9:      UDConstraint tmp;
```

```

    for (int i=0; i<candidates.size(); i++) {

        tmp = (UDConstraint)candidates.get(i);

        if (tmp.getArgAt(0).equals(var))
            return ((nl)tmp.getArgAt(1)).toObjectContainer();

    }

    return new ObjectContainer();

}
```

Example: N-Queens

This section describes the problem dependent part of the implementation of a constraint satisfaction problem by means of the n-queens example. the problem is to place n queens (known from chess) on a board, which is partitioned into *N*N* squares, so that they will not threaten each other. The queens are able to move as they do in chess. Two queens threaten each other, when they are positioned in the same row, in the same column or in the same diagonal.

One way of modeling this problem is to recognize, that in any solution there is exactly one queen in every column. In this case, each of the *N* columns has *N* rows. We associate each column with a logical variable. In each column there are *N* possibilities to place a queen. So, each logical variable is associated with a domain, which contains the integer values *1* to *N*. A solution is specified as a permutation of the list of the numbers *1* to *N*. The first element of the list represents the row number, where a queen is placed in the first column, etc. In order to solve the n-queens problem with JCHR and Java, we first need to implement a constraint solver, which determines, whether a board situation, described by the above given model, is safe or not. The rules, which support the solving of the n-queens problem are defined in the class `nqHandler`. This class contains the results of the compilation of the file `nq.jchr`, where the rules for the n queens problem are implemented in JCHR. The constraint handler for the n-queens problem uses finite domains, too. But not all rules, which implement the functionality of the finite domains are needed for this problem. Here we can see the advantages of JCHR, when an existing constraint handler needs to be extended or changed. Rules, which are not needed any more are removed from the constraint system, while rules, which are needed to solve the new problem are simply added. nearly no programming efforts in Java have to be taken. In the following, the rules for the n-queens problem are described.

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

if (IntUtil.ground(X))
   { noattack(X, Y, N) && fdEnu(Y, D) } <=>
     { X1 = X &&
       D1 = IntUtil.remove(D, X1) &&
       X2 = IntUtil.add(X1, N) &&
       X3 = IntUtil.sub(X1, N) &&
```

```
        D2 = IntUtil.remove(D1, X2) &&
        D3 = IntUtil.remove(D2, X3) &&
        fdEnu(Y, D3) } noattack1;

if (IntUtil.ground(X))
   { noattack(Y, X, N) && fdEnu(Y, D) } <=>
     { X1 = X &&
        D1 = IntUtil.remove(D, X1) &&
        X2 = IntUtil.add(X1, N) &&
        X3 = IntUtil.sub(X1, N) &&
        D2 = IntUtil.remove(D1, X2) &&
        D3 = IntUtil.remove(D2, X3) &&
        fdEnu(Y, D3) } noattack2;
```

The rules `noattack1` and `noattack2` handle symmetric cases, so only the functionality of `noattack1` will be explained. This rule is applicable, when the variable `X` is bound to a value. This is ensured in the guard, with the call of the method `ground`. The `noattack` constraint in the body, describes the distance `N` between the two columns of the board, which are represented by variables `X` and `Y`. The constraint `noattack(1, B, 1)` means, that the distance between the column, represented by the variable `B` and the preceeding column is `1` (column). Thereby a queen is placed in the preceeding column in the first row (third argument of `noattack`. Adding this `noattack` constraint and the constraint `fdEnu(B, [1, 2, 3, 4])`[2] to the goal, would cause the rule `noattack1` to fire. Declaratively, this rule application computes all safe positions for a queen in the row, represented by the variable B, concerning the position of the queen in the preceeding row. This is done by the execution of the body. As a result the constraint `fdEnu(B, [3, 4])` is inserted into the constraint memory. This indicates, that, with the given board situation, there are only two save positions for a queen in the column represented by the variable B: in row `3` and in row `4`. When the 4 queens problem has to be solved, the following `noattack` constraints need to be added to the constraint system, in order to give a correct description of the chess board:

```
noattack(A, B, 1) &&
noattack(A, C, 2) &&
noattack(A, D, 3) &&
noattack(B, C, 1) &&
noattack(B, D, 2) &&
noattack(C, D, 1)
```

Thereby the variables `A`, `B`, `C` and `D` represent the four columns of the board. The constraint `noattack(A, D, 3)` indicates, that between the columns, represented by the variables `A` and `D`, three columns are placed (inclusive the column represented by the variable `D`). This information about the board is generated once, in the beginning of the solving algorithm. When an enumeration of a variable is reduced to the empty enumeration, no safe position could have been found for the column, which is represented by the variable. The evaluation of the goal fails (see rule `empty`).

The connection between these JCHR rules and Java is implemented in the class `NQueens`. It contains only one member variable:

[2]The more simple Prolog notation is used for lists in this example.

```
private ConstraintSystem cs;
```

The variable `cs` is an instance of the class `ConstraintSystem` which provides the main functionality for solving constraint satisfaction problems in Java (see Section **??**). It is instantiated in the the constructor `NQueens`. Thereby the rules, which help to solve the n-queens problem, are added to the constraint system by the method `defineRules` provided by the interface `Handler` and implemented by the class `nqHandler` (see line 5 in in the following code example).

```
1:  public NQueens() {
2:
3:     cs = new ConstraintSystem();
4:     nqHandler nq = new nqHandler();
5:     nq.defineRules(cs);
6:
7:  }
```

The class `NQueens` provides a method `solve`, which initializes the constraint satisfaction problem and then calls the backtracking algorithm, which generates the solutions. The method `solve` at first generates a description of the n-queens problem. Therefore a representation of the board, where the queens are going to be placed, is built. The columns of a board (consisting of n rows and n columns) are represented by n variables. These variables are generated by the method `generateVariableList`. They are going to be bound to integer values between 1 and n, which indicate the rows 1 to n. This enumeration is generated by the method `generateEnumeration` (see lines 3 and 4). Thence, a solution is an object of the class `ObjectContainer`, containing n variables, which are bound to row numbers. In line 5 the variables are associated with enumeration domains, which consist of values between 1 and n. Before the backtracking algorithm, which searches the solution tree for save board situations is started, a description of the board has to be generated. This is done by the method `generateNoattacks` in line 6. It inserts the `noattack` constraints into the constraint system as it is described above. The class fdUtil provides the method `labeling`, which implements a simple backtracking algorithm, as described above. With this algorithm all problems of this kind can be solved. A different problem simply needs a different initialization, as it is done in the methods `makeDomains` and `generateNoattacks`.

```
1:  public ObjectContainer solve(int n) {
2:
3:     ObjectContainer solution = generateVariableList(n);
4:     ObjectContainer list = generateEnumeration(n);
5:     makeDomains(list, solution);
6:     generateNoattacks(solution);
7:     return fdUtil.labeling(cs, solution);
8:
9: }
```

The Prolog implementation of the n-queens problem can be seen in Appendix **??**. All methods, which are not completely listed here can be seen in Appendix **??**.

Chapter 6

Conclusions and Future Work

This work described a Java class library, which provides the language Constraint Handling Rules (CHR) for the host language Java. It introduced the language JCHR, which is a Java style CHR and it gave an overview about the compiler, translating JCHR to Java and the evaluator, which is needed to evaluate constraints in Java. Furthermore, some examples (e.g. finite domains) illustrated the flexibility of this library, when it comes to extend or to combine existing constraint handlers. In addition an approach to illustrate the interface between Java and JCHR was appended.

On the one hand, an intention of this approach was to provide a Java class library, in order to write platform independent, constraint based Java applications, due to the growing popularity of Java and the possibilities of the Internet. Thereby this library may be interesting for supporting intelligent agents or it may be used for electronic commerce.
On the other hand, this class library, is intended to be embedded into the programming language JavaLog [**?**], which is an implementation of Prolog in Java. This project aims at defining an integrated Prolog-CHR system, platform independent and oriented to industrial applications.

Unfortunately, the JCHRCore-Evaluator still has some efficiency problems, when it comes to working with large amount of data. Further work has to be done at the data management and the evaluation algorithms, in order to improve evaluation speed. An improved constraint retrieval, would increase efficiency, too.

Beyond it, JCHR provides not the complete scope of CHR. In JCHR *ask/tell* guards and *pragmas* are missing. Another interesting aspect, which should be added in the future, is the delayed execution of built-in constraints and Java methods. In the current implementations, it is not allowed to call built-in constraints and Java methods with unbound variables as arguments. An alternative would be, to wait with the execution of the respective expression, until all arguments have a valid value. This would require a modification of the classes, which implement methods and constraints, and a modification of the evaluation algorithm.
Another future project will investigate aspects of debugging CHR and JCHR, respectively. Therefore this library is predestinated, too, due to the graphical abilities, provided by the Java standard class libraries.

We hope, that this work supports the propagation of constraint technologies in the Internet. A handbook for will be added soon on http://www.pms.informatik.uni-muenchen.de/jack/

Appendix A

The Class ObjectContainer

The class `ObjectContainer` provides a data structure to store any objects dynamically. It extends the class `java.util.Vector` by some additional methods. Furthermore, the class `ObjectContainer` is a wrapper for future data structures of Java. If another collection is intended to be used, it has only to be replaced in this class and not in the hole project.

The class `ObjectContainer` provides following additional functionality:
The methods `first` and `rest` simulate the functionality of a nested list. `first` returns the first object in the data structure and rest returns an `ObjectContainer` object, which contains all objects of the current `ObjectContainer` object except the first one.

```
public Object first()
public ObjectContainer rest()
```

The methods `intersection`, `union` and `concatenate` create new `ObjectContainer` objects out of the current and the given `ObjectContainer` object. `concatenate` appends the given `ObjectContainer` object to the current. The methods `intersection` and `union` are self explaining.

```
public ObjectContainer intersection(ObjectContainer oc)
public ObjectContainer union(ObjectContainer oc)
public void concatenate(ObjectContainer oc)
```

Beyond this, the class `ObjectContainer` provides a cloning method `generalClone`, which is able to clone each object of an unknown class, which implements the java standard method `clone`. Furthermore, it is able to clone all wrapper objects for Java simple types (e.g. `int` or `float`) like `java.lang.Integer` or `java.lang.Float`. Calling the method `clone` for an object of an unknown class would invoke the `clone` method of the class `java.lang.Object`. This would mean a loss of information, because the method `clone` of `java.lang.Object` only copies data, concerning the object of the class `java.lang.Object`.

```
public static Object generalClone(Object o)
```

All other methods of the class `ObjectContainer` are similar to the methods of the class `java.util.Vector`.

Appendix B

Java Classes

The classes `nl` and `IntUtil` are used by the JCHR rules for the implementation of the n-queens problem in Java (see Chapter **??**)

B.1 The N-Queens Problem in Java

```
public class NQueens {

    private ConstraintSystem cs;

    public NQueens() {

        cs = new ConstraintSystem();
        nqHandler nq = new nqHandler();
        nq.defineRules(cs);

    }

    private ObjectContainer generateEnumeration(int count) {

        ObjectContainer oc = new ObjectContainer();

        for (int i = 0; i < count; i++)
            oc.add(new Integer(i+1));

        return oc;

    }

    private ObjectContainer generateVariableList(int count) {

        ObjectContainer oc = new ObjectContainer();

        for (int i = 0; i < count; i++) {

            Object variable = new Object();
            cs.addVariable(variable, "java.lang.Integer");
            oc.add(variable);

        }

        return oc;

    }
```

```
    private void makeDomains(ObjectContainer enumeration,
                             ObjectContainer varlist) {

        for (int i = 0; i < varlist.size(); i++)
            cs.addGoalConstraint(new
               FDENUConstraint(varlist.get(i), nl.tonl(enumeration)));

    }

    private void generateNoattacks(ObjectContainer solution) {

        if (solution.size() > 0) {

            safe(solution.first(), solution.rest(), 1);
            generateNoattacks(solution.rest());

        }

    }

    private void safe(Object x, ObjectContainer list, int i) {

        if (list.size() > 0) {

            cs.addGoalConstraint(new
               NOATTACKConstraint(x, list.first(), new Integer(i)));
            safe(x, list.rest(), i+1);

        }

    }

    public ObjectContainer solve(int n) {

        ObjectContainer list = generateEnumeration(n);
        ObjectContainer solution = generateVariableList(n);
        makeDomains(list, solution);
        generateNoattacks(solution);
        return fdUtil.labeling(cs, solution);

    }

    public static void main(String args[]) {

        NQueens nl = new NQueens();
        int n = Integer.parseInt(args[0]);

        ObjectContainer oc = nl.solve(n);
        if (oc.size() > 0) {

            System.out.println("solutions for the "+n+
                               " ladies problem: ");
            for (int i=0; i<oc.size(); i++)
                System.out.println(""+oc.get(i));

        } else
            System.out.println("No solutions found for the "+n+
                               " ladies problem!");

    }

}
```

B.2 Nested Lists

```
public class nl {

    //////////////////////////////////////
    // fields

    // a nested list consists of an object called first,
    // and a rest list
    public Object first;
    public nl rest;

    //////////////////////////////////////
    // constructors

    // constructs a list with an object and a rest list
    public nl(Object o, nl l) {

        first = o;
        rest = l;

    }

    // constructs a list with onlu one object and an empty
    // rest list
    public nl(Object o) {

        this(o, new nl());

    }

    // constructs an empty list
    public nl() {

        this(null, null);

    }

    // transforms an ObjectContainer to a nested list
    public static nl tonl(ObjectContainer oc) {

        if (oc.size() > 0) {

            ObjectContainer tmp = (ObjectContainer)oc.clone();
            Object o = tmp.get(0); tmp.remove(o);
            if (tmp.size() > 0)
                return new nl(o, nl.tonl(tmp));
            else
                return new nl(o);

        } else
            return new nl();

    }

    // tests, whether the given nested list is empty
    public static boolean isEmpty(nl l) {

        return ((l.first == null) && (l.rest == null));

    }
```

```
    // returns an intersection of the two given lists
    public static nl intersection(nl l1, nl l2) {

        ObjectContainer oc1 = l1.toObjectContainer();
        ObjectContainer oc2 = l2.toObjectContainer();
        ObjectContainer oc3 = oc1.intersection(oc2);
        return nl.tonl(oc3);

    }

    // returns the number of objects stored in the given list.
    public static int size(nl l) {

        return l.toObjectContainer().size();

    }

    // tests, whether the current list is empty
    public boolean isEmpty() {

        return ((first == null) && (rest == null));

    }

    // transforms the current list into an ObjectContainer
    public ObjectContainer toObjectContainer() {

        ObjectContainer oc = new ObjectContainer();

        if (!isEmpty()) {

            oc.add(first);
            if (!rest.isEmpty())
                oc.concatenate(rest.toObjectContainer());

        }

        return oc;

    }

    // transforms the current list into a String
    public String toString() {

        return "["+toObjectContainer()+"]";

    }

    // creates a copy of the current list
    public Object clone() {

        ObjectContainer oc = (ObjectContainer)
                             (toObjectContainer().clone());
        return nl.tonl(oc);

    }

}
```

B.3 Integer Utilities

```
// this method provides methods for comparing and computing
// java.lang.Integer objects. Furthermore, special methods for
// handling with nested lists (see class nl) are provided.
// In Java it is not possible work with java.lang.Integer objects
// directly; this is provided by this class.
public class IntUtil {

    // only methods, whoose arguments are bound, are executed in.
    // The execution of this method delivers true as a
    // result. Thence an integer variable in JCHR is ground (i.e.
    // bound to a value, if the execution of this method returns
    // true.
    public static boolean ground(java.lang.Integer i) {

        return true;

    }

    // returns true, if i1 is less than i2, otherwise false.
    public static boolean lt(java.lang.Integer i1, java.lang.Integer i2) {

        return i1.intValue() < i2.intValue();

    }

    // returns true, if i1 is greater than i2, otherwise false.
    public static boolean gt(java.lang.Integer i1, java.lang.Integer i2) {

        return i1.intValue() > i2.intValue();

    }

    // return true, if i1 is less or equal than i2, otherwise false.
    public static boolean le(java.lang.Integer i1, java.lang.Integer i2) {

        return i1.intValue() <= i2.intValue();

    }

    // returns true, if i1 is greater or equal than i2, otherwise false.
    public static boolean ge(java.lang.Integer i1, java.lang.Integer i2) {

        return i1.intValue() >= i2.intValue();

    }

    // return true, if i1 is not equal to i2, otherwise false.
    public static boolean ne(java.lang.Integer i1,
                             java.lang.Integer i2) {

        return i1.intValue() != i2.intValue();

    }

    // returns the object with the larger integer value.
    public static java.lang.Integer max(java.lang.Integer i1,
                                        java.lang.Integer i2) {

        if (i1.intValue() > i2.intValue())
            return i1;
```

```
        return i2;

    }

    // returns the object with the smaller integer value.
    public static java.lang.Integer min(java.lang.Integer i1,
                                        java.lang.Integer i2) {

        if (i1.intValue() > i2.intValue())
            return i2;
        return i1;

    }

    /////////////////////////////////////////////////////
    // arithmetics

    // increases the value of the given integer object by
    // one and returns it.
    public static java.lang.Integer inc(java.lang.Integer i) {

        return new Integer(i.intValue()+1);

    }

    // decreases the value of the given integer object
    //by one and returns it.
    public static java.lang.Integer dec(java.lang.Integer i) {

        return new Integer(i.intValue()-1);

    }

    // adds the values of the given integer objects and
    // returns the result.
    public static java.lang.Integer add(java.lang.Integer i1,
                                        java.lang.Integer i2) {

        return new java.lang.Integer(i1.intValue() + i2.intValue());

    }

    // substrates i2 from i1 and returns the result.
    public static java.lang.Integer sub(java.lang.Integer i1,
                                        java.lang.Integer i2) {

        return new java.lang.Integer(i1.intValue() - i2.intValue());

    }

    // multiplicates the values of the given integer objects
    // and returns the result.
    public static java.lang.Integer mult(java.lang.Integer i1,
                                         java.lang.Integer i2) {

        return new java.lang.Integer(i1.intValue() * i2.intValue());

    }

    ///////////////////////////////////////////
    // list operations
    // all nested lists are supposed to contain
```

```
// java.lang.Integer objects

// returns the first object of the given list.
public static java.lang.Integer first(nl l) {

    return (java.lang.Integer)l.first;

}

// returns true, if the given integer object is a member
// of the given nested list, otherwise false.
public static boolean member(java.lang.Integer i, nl l) {

    nl tmp = l;

    while (!tmp.isEmpty()) {

        if (((java.lang.Integer)tmp.first).equals(i))
            return true;

        tmp = tmp.rest;

    }

    return false;

}


// returns true, if the given integer object is not
// a member of the given nested list, otherwise false.
public static boolean notMember(java.lang.Integer i, nl l) {

    return !member(i, l);

}


// removes the given integer object from the given list;
// the result is a nested list without the given object.
public static nl remove(nl l, java.lang.Integer i) {

    ObjectContainer oc = l.toObjectContainer();
    oc.remove(i);

    return nl.tonl(oc);

}

// removes all integer objects from the given list, whoose
// values are lower than the value of the given integer object,
// the result is a nested list without those objects.
public static nl removeLower(java.lang.Integer min, nl l) {

    nl l1 = (nl)l.clone();

    while (!nl.isEmpty(l1)) {

        if (((java.lang.Integer)l1.first).intValue() <
             ((java.lang.Integer)min).intValue())
            l1 = l1.rest;
        else
```

```
                return l1;

        }

        return l1;

    }

    // removes all integer objects from the given list, whose
    // values are higher than the value of the given integer object,
    // the result is a nested list without those objects.
    public static nl removeHigher(java.lang.Integer max, nl l) {

        ObjectContainer oc1 = l.toObjectContainer();
        ObjectContainer oc2 = new ObjectContainer();

        for (int i=0; i< oc1.size(); i++) {

            if (IntUtil.le((java.lang.Integer)oc1.get(i), max))
                oc2.add(oc1.get(i));

        }

        return nl.tonl(oc2);

    }

    // returns the integer object with the lowest value.
    public static java.lang.Integer minList(nl l) {

        java.lang.Integer min = (Integer)l.first;
        nl tmp = l.rest;

        while (!tmp.isEmpty()) {

            if (IntUtil.lt((Integer)tmp.first, min))
                min = (Integer)tmp.first;

            tmp = tmp.rest;

        }

        return min;

    }

    // returns the integer object with the highest value
    public static java.lang.Integer maxList(nl l) {

        java.lang.Integer max = (Integer)l.first;
        nl tmp = l.rest;

        while (!tmp.isEmpty()) {

            if (IntUtil.gt((Integer)tmp.first, max))
                max = (Integer)tmp.first;

            tmp = tmp.rest;

        }

        return max;
```

```
    }

    // all integer values of the given nested lists are added to each
    //  other, the result lists are unified and returned as a result.
    public static nl allAddition(nl l1, nl l2) {

        ObjectContainer oc1 = l1.toObjectContainer();
        ObjectContainer oc2 = l2.toObjectContainer();
        ObjectContainer oc3 = new ObjectContainer();
        ObjectContainer oc4 = new ObjectContainer();

        for (int i=0; i<oc1.size(); i++) {

            for (int j=0; j<oc2.size(); j++)
                oc3.add(add((java.lang.Integer)oc1.get(i),
                            (java.lang.Integer)oc2.get(j)));

            oc4 = oc3.union(oc4);
            oc3.removeAll();

        }

        return nl.tonl(oc4);

    }

    // all integer values of the given nested lists are multiplicated
    // with each other,the result lists are unified and returned as
    // a result.
    public static nl allMultiplication(nl l1, nl l2) {

        ObjectContainer oc1 = l1.toObjectContainer();
        ObjectContainer oc2 = l2.toObjectContainer();
        ObjectContainer oc3 = new ObjectContainer();
        ObjectContainer oc4 = new ObjectContainer();

        for (int i=0; i<oc1.size(); i++) {

            for (int j=0; j<oc2.size(); j++)
                oc3.add(mult((java.lang.Integer)oc1.get(i),
                             (java.lang.Integer)oc2.get(j)));

            oc4 = oc3.union(oc4);
            oc3.removeAll();

        }

        return nl.tonl(oc4);

    }

}
```

Appendix C

Constraint Handlers, implemented in JCHR

C.1 Primes

```
handler primes {

        class IntUtil;
        class java.lang.Integer;

        constraint prime(java.lang.Integer);
        constraint primes(java.lang.Integer);

        rules {

                variable java.lang.Integer N, M, I, J;

                { primes(1) } <=> { true } simple;

                if (IntUtil.lt(1, N)) {

                        primes(N)

                } <=> {

                        M = IntUtil.dec(N) && prime(N) && primes(M)

                } recursive;

                if (IntUtil.modNull(J, I)) {

                        prime(I) &\& prime(J)

                } <=> {

                        true

                } absorbJ;

        }

        goal g1 { primes(1000) }

}
```

C.2 Finite Domains

```
handler fd {

        class java.lang.Integer;
        class IntUtil;
        class nl; // nested list

        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);
        constraint fdAdd(java.lang.Integer,
                         java.lang.Integer, java.lang.Integer);
        constraint fdMult(java.lang.Integer,
                          java.lang.Integer, java.lang.Integer);

        rules {

                variable java.lang.Integer X, X1, X2, X3, Y, Z, N;
                variable Min, Max, MinX, MaxX, MinY, MaxY, MaxY1, MinX1;
                variable nl D, D1, D2, D3, L, L1, L2, L3;

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

                //if (1 = nl.size(L)) { fdEnu(X, L) } <=>
                                      { X = IntUtil.first(L) } oneElt;

                //if (IntUtil.ground(X)) { fdEnu(X, L) } ==>
                                         { IntUtil.member(X, L) } isInEnu;1

                ///////////////////////////////////////////////
                // intersection of domains for the same variable
                {

                        fdEnu(X, L1) && fdEnu(X, L2)

                } <=> {

                        L = nl.intersection(L1, L2) && fdEnu(X, L)

                } intersection1;

                // ------------------------------------
                {

                        fdEnu(X, L) && fdInt(X, Min, Max)

                } <=> {

                        L1 = IntUtil.removeLower(Min, L) &&
                        L2 = IntUtil.removeHigher(Max, L1) &&
                        fdEnu(X, L2)

                } intersection2;
```

```
/////////////////////////////////////////////
// interaction with inequalities
if (MinX = IntUtil.minList(L1) &&
    MinY = IntUtil.minList(L2) &&
    IntUtil.gt(MinX, MinY)) {

        fdLe(X, Y) &&
        fdEnu(X, L1) &&
        fdEnu(Y, L2)

} ==> {

        MinX = IntUtil.minList(L1) &&
        MaxY = IntUtil.maxList(L2) &&
        fdInt(Y, MinX, MaxY)

} leMin;

// ------------------------------------
if (MaxX = IntUtil.maxList(L1) &&
    MaxY = IntUtil.maxList(L2) &&
    IntUtil.gt(MaxX, MaxY)) {

        fdLe(X, Y) &&
        fdEnu(X, L1) &&
        fdEnu(Y, L2)

} ==> {

        MinX = IntUtil.minList(L1) &&
        MaxY = IntUtil.maxList(L2) &&
        fdInt(X, MinX, MaxY)

} leMax;

// ------------------------------------
if (MinX = IntUtil.minList(L1) &&
    MinY = IntUtil.minList(L2) &&
    MinX1 = IntUtil.inc(MinX) &&
    IntUtil.gt(MinX1, MinY)) {

        fdLt(X, Y) &&
        fdEnu(X, L1) &&
        fdEnu(Y, L2)

} ==> {

        MinX = IntUtil.minList(L1) &&
        MinX1 = IntUtil.inc(MinX) &&
        MaxY = IntUtil.maxList(L2) &&
        fdInt(Y, MinX1, MaxY)

} ltMin;

// ------------------------------------
if (MaxX = IntUtil.maxList(L1) &&
    MaxY = IntUtil.maxList(L2) &&
    MaxY1 = IntUtil.dec(MaxY) &&
    IntUtil.lt(MaxY1, MaxX)) {

        fdLt(X, Y) &&
        fdEnu(X, L1) &&
```

```
        fdEnu(Y, L2)

} ==> {

        MinX = IntUtil.minList(L1) &&
        MaxY = IntUtil.maxList(L2) &&
        MaxY1 = IntUtil.dec(MaxY) &&
        fdInt(X, MinX, MaxY1)

} ltMax;

// ------------------------------------
if (IntUtil.member(X, D)) {

        fdNe(X, Y) &\& fdEnu(Y, D)

} <=> {

        D1 = IntUtil.remove(D, X) &&
        fdEnu(Y, D1)

} ne1;


// ------------------------------------
if (IntUtil.member(X, D)) {

        fdNe(Y, X) &\& fdEnu(Y, D)

} <=> {

        D1 = IntUtil.remove(D, X) &&
        fdEnu(Y, D1)

} ne11;

// ------------------------------------
if (IntUtil.notMember(X, D)) {

        fdEnu(Y, D) &\& fdNe(X, Y)

} <=> {

        true

} ne2;


// ------------------------------------
if (IntUtil.notMember(X, D)) {

        fdEnu(Y, D) &\& fdNe(Y, X)

} <=> {

        true

} ne22;

////////////////////////////////////////////////
// interaction with addition
{
```

```
                fdAdd(X, Y, Z) &&
                fdEnu(X, L1) &&
                fdEnu(Y, L2)

        } ==> {

                L3 = IntUtil.allAddition(L1, L2) &&
                fdEnu(Z, L3)

        } addition;


        ////////////////////////////////////////////////
        // interaction with multiplication
        {

                fdMult(X, Y, Z) &&
                fdEnu(X, L1) &&
                fdEnu(Y, L2)

        } ==> {

                L3 = IntUtil.allMultiplication(L1, L2) &&
                fdEnu(Z, L3)

        } multiplication;


    }

}
```

Appendix D

Prolog implementations

The next table correlates the constraints names of the JCHR (see Chapter **??**) and the Prolog implementations.

JCHR	Prolog	description
fdEnu/2	(::)/2	enumeration domain
fdInt/3	(::)/2	interval domain
fdLe/2	le/2	domain comparision and adaption (less or equal)
fdLt/2	lt/2	domain comparision and adaption (less than)
fdNe/2	ne/2	domain comparision (not equal)
fdAdd/3	add/3	union of an addition by pairs
fdMult/3	mult/3	union of a multiplication by pairs

D.1 The Finite Domains in Prolog

```
:- use_module( library(chr)).

:- use_module( library(lists),
              [member/2,memberchk/2,select/3,
               last/2,is_list/1,min_list/2, max_list/2,
               remove_duplicates/2]).

handler listdom.

option(debug_compile,on).
option(already_in_heads, on).
option(check_guard_bindings, off).

% for domain constraints
operator( 700,xfx,'::').
operator( 600,xfx,'..').

% for inequality constraints
operator( 700,xfx,lt).
operator( 700,xfx,le).
operator( 700,xfx,ne).

constraints (::)/2, le/2, lt/2, ne/2, add/3, mult/3.
% X::Dom - X must be element of the finite list domain Dom

% special cases
X::[] <=> fail.
```

```
% intersection of domains for the same variable
X::L1, X::L2 <=> is_list(L1), is_list(L2) |
                         intersection(L1,L2,L) , X::L.

X::L, X::Min..Max <=> is_list(L) |
        remove_lower(Min,L,L1), remove_higher(Max,L1,L2),
        X::L2.


% interaction with inequalities
X le Y, X::L1,  Y::L2 ==> is_list(L1),is_list(L2),
                     min_list(L1,MinX), min_list(L2,MinY), MinX > MinY |
                     max_list(L2,MaxY), Y::MinX..MaxY.
X le Y, X::L1,  Y::L2 ==> is_list(L1),is_list(L2),
                     max_list(L1,MaxX), max_list(L2,MaxY), MaxX > MaxY  |
                     min_list(L1,MinX), X::MinX..MaxY.

X lt Y, X::L1,  Y::L2 ==> is_list(L1), is_list(L2),
                     max_list(L1,MaxX), max_list(L2,MaxY),
                     MaxY1 is MaxY - 1, MaxY1 < MaxX |
                     min_list(L1,MinX), X::MinX..MaxY1.
X lt Y, X::L1,  Y::L2 ==> is_list(L1), is_list(L2),
                     min_list(L1,MinX), min_list(L2,MinY),
                     MinX1 is MinX + 1, MinX1 > MinY |
                     max_list(L2,MaxY), Y :: MinX1..MaxY.

X ne Y \  Y::D <=> ground(X), is_list(D), member(X,D) | select(X,D,D1), Y::D1.
Y ne X \  Y::D <=> ground(X), is_list(D), member(X,D) | select(X,D,D1), Y::D1.
Y::D \ X ne Y <=>  ground(X), is_list(D), \+ member(X,D) | true.
Y::D \ Y ne X <=>  ground(X), is_list(D), \+ member(X,D) | true.


% interaction with addition
% no backpropagation yet!
add(X,Y,Z), X::L1, Y::L2 ==> is_list(L1), is_list(L2) |
       all_addition(L1,L2,L3), Z::L3.

% interaction with multiplication
% no backpropagation yet!
mult(X,Y,Z), X::L1, Y::L2 ==> is_list(L1), is_list(L2) |
       all_multiplication(L1,L2,L3), Z::L3.


% auxiliary predicates =============================================
remove_lower(_,[],L1):- !, L1=[].
remove_lower(Min,[X|L],L1):-
        X@<Min,
        !,
        remove_lower(Min,L,L1).
remove_lower(Min,[X|L],[X|L1]):-
        remove_lower(Min,L,L1).

remove_higher(_,[],L1):- !, L1=[].
remove_higher(Max,[X|L],L1):-
        X@>Max,
        !,
        remove_higher(Max,L,L1).
remove_higher(Max,[X|L],[X|L1]):-
        remove_higher(Max,L,L1).

intersection([], _, []).
intersection([Head|L1tail], L2, L3) :-
```

```
        memberchk(Head, L2),
        !,
        L3 = [Head|L3tail],
        intersection(L1tail, L2, L3tail).
intersection([_|L1tail], L2, L3) :-
        intersection(L1tail, L2, L3).

all_addition(L1,L2,L3) :-
        setof(Z, X^Y^(member(X,L1), member(Y,L2), Z is X + Y), L3).

all_multiplication(L1,L2,L3) :-
        setof(Z, X^Y^(member(X,L1), member(Y,L2), Z is X * Y), L3).
```

D.2 The N-Queens Problem in Prolog

```
:- use_module( library(chr)).

:- use_module( library(lists),
             [member/2, memberchk/2,select/3,length/2]).

:- use_module( library(lists),
             [member/2,last/2,is_list/1,
              min_list/2, max_list/2, remove_duplicates/2]).

handler domain.


option(debug_compile,on).
option( already_in_heads, on).
option( check_guard_bindings, off).

% for domain constraints
operator( 700,xfx,'::').
operator( 600,xfx,':').

% for inequality constraints
operator( 700,xfx,lt).
operator( 700,xfx,le).
operator( 700,xfx,ne).

% X::Dom - X must be element of the finite  domain Dom
constraints (::)/2, le/2, lt/2, ne/2, add/3, mult/3,
            alldistinct/1, labeling/1, noattack/3.

% special cases
X::[] <=> fail.

noattack(X,Y,N),  Y::D <=> ground(X) |
                           X1 is X, remove(X1,D,D1),
                           X2 is X1 + N, X3 is X1 - N,
                           remove(X2,D1,D2), remove(X3,D2,D3),
                           Y::D3.

noattack(Y,X,N),  Y::D <=> ground(X) |
                           X1 is X, remove(X1,D,D1),
                           X2 is X1 + N, X3 is X1 - N,
                           remove(X2,D1,D2), remove(X3,D2,D3),
                           Y::D3.

labeling([]) <=> true.
labeling([X|Xs]),X::L  <=> member(X,L), labeling(Xs).
```

```
solve(N,Solution):- N > 0,
    gen(1,N,List),
    length(Solution,N),
    make_domains(List,Solution),
    queens(Solution),
    labeling(Solution).

gen(N,N,[N]).
gen(N1,N2,[N1|Liste]) :-
        N1 < N2,
        M is N1 + 1,
        gen(M,N2,Liste).

queens([]).
queens([X|Xs]):-
    safe(X,Xs,1),
    queens(Xs).

safe(X,[],N).
safe(X,[Y|T],N) :-
        noattack(X,Y,N),
        N1 is N + 1,
        safe(X,T,N1).

make_domains(_,[]).
make_domains(L,[X|R]) :-
        X :: L,
        make_domains(L,R).

remove(_,[],[]).
remove(X,[X|L],L).
remove(X,[Y|L],[Y|L1]) :- X \== Y,
                          remove(X,L,L1).
```

Appendix E

The Package Files

The following sections list all files, which are needed for each package, with a brief introduction.

E.1 JCHR-Compiler

jchr.g	grammar file to create scanner and parser (see [?])
jchrc.java	the main compiler class; calls scanner, parser, and code generator
JCHRGenerator.java	generates code out of the abstract syntax tree
JCHRParser.java	generates an abstract syntax tree out of a token stream
JCHRLexer.java	generates a token stream out of a JCHR file
JCHRParserTokenTypes.java	applies constant identifiers to the tokens
ConstraintExtender.java	extends the class `UDConstraint` for simplified use
JMethodExtender.java	extends the class `JMethod` for simplified use
ObjectContainer.java	data structure for java objects (see Section **??**)
Handler.java	super class for the result of a compilation
UnknownIdentifierException.java	see Section **??**
WrongConstraintTypeException.java	see Section **??**

E.2 JCHR-Evaluator

Bool.java	built-in constraint for boolean values
BuiltinConstraint.java	abstract class; further built-in constraint are heirs of this
BuiltinConstraintMemory.java	data structure for built-in constraints
Constraint.java	abstract super class for all constraints
ConstraintSystem.java	main class of the package
Environment.java	contains information about the memories of a ConstraintSystem object
Equality.java	built-in constraint for equality
Handler.java	super class for the result of a compilation

JExpression.java	super class for java style built-in constraints
JMethod.java	super class for method invocation built-in constraints
ObjectContainer.java	data structure for java objects (see Section **??**)
Path.java	data structure to for term description (used by PathNode)
PathNode.java	data structure for efficient term retrieval
PropagationRule.java	rule wrapper
Rule.java	super class for all rules
UDCMState.java	contains information about the user-defined constraint memory
RuleMemory.java	data structure for rules
RuleMemoryState.java	contains information about the rule memory
SimpagationHead.java	container for simpagation rule heads
SimpagationRule.java	rule wrapper
SimplificationRule.java	rule wrapper
UDConstraint.java	user-defined constraint
UserDefinedConstraintMemory.java	data structure for user-defined constraints
VTableState.java	contains information about the variable table
VariableTable.java	data structure for centralized variable handling
VariableUnboundException.java	see Section **??**
WrongArgumentTypeException.java	see Section **??**
UnknownConstraintException.java	see Section **??**

E.3 JCHR-Example: NQueens

ObjectContainer.java	data structure for java objects (see Section **??**)
fdHandler.java	constraint handler for finite domains; implemented in JCHRCore
nl.java	simulation of a nested list
BoardCanvas.java	Java AWT element; draws the board for the n ladies problem
IntUtil.java	provides comparison methods for integer objects
NQueens.java	application for computing the n ladies problem
NQueensApplet.java	applet for visualization of the n ladies problem

The following files are subclasses of `UDConstraint` (see Section **??**)

NOATTACKConstraint.java	user-defined constraint `noattack`
FDADDConstraint.java	user-defined constraint `fdAdd`
FDENUConstraint.java	user-defined constraint `fdEnu`
FDINTConstraint.java	user-defined constraint `fdInt`
FDLEConstraint.java	user-defined constraint `fdLe`
FDLTConstraint.java	user-defined constraint `dLt`
FDMULTConstraint.java	user-defined constraint `fdMult`
FDNEConstraint.java	user-defined constraint `fdNe`

The following files are subclasses of `JMethod` (see Section **??**). The methods are implemented in the classes `nl` and `IntUtil` (see Appendix ??)

INTUTILADD2Method.java	java method `add`
INTUTILALLADDITION2Method.java	java method `allAddition`
INTUTILALLMULTIPLICATION2Method.java	java method `allMultiplication`
INTUTILDEC1Method.java	java method `dec`
INTUTILFIRST1Method.java	java method `first`
INTUTILGROUND1Method.java	java method `ground`
INTUTILGT2Method.java	java method `gt`
INTUTILINC1Method.java	java method `inc`
INTUTILLT2Method.java	java method `lt`
INTUTILMAXLIST1Method.java	java method `maxList`
INTUTILMEMBER2Method.java	java method `member`
INTUTILMINLIST1Method.java	java method `minList`
INTUTILNOTMEMBER2Method.java	java method `notMember`
INTUTILREMOVE2Method.java	java method `remove`
INTUTILREMOVEHIGHER2Method.java	java method `removeHigher`
INTUTILREMOVELOWER2Method.java	java method `removeLower`
INTUTILSUB2Method.java	java method `sub`
NLINTERSECTION2Method.java	java method `intersection`
NLISEMPTY1Method.java	java method `isEmpty`
NLSIZE1Method.java	java method `size`

Bibliography

[AC98] Andy Hon Wai Chun. Constraint Programming in Java with JSolver. City University of Hong Kong. 1998.

[ANT99] The ANTLR Translator generator. Internet: http://www.antlr.org.

[CS99] Christian Schulte. Comparing Trailing and Copying for Constraint Programming. 999. Internet: http://ps.uni-sb.de/~schulte/papers/copying.ps.gz

[FA97] Thom Frühwirth, Slim Abdennadher. Constraint-Programmierung: Grundlagen und Anwendungen. Springer. 1997.

[FB95] Thom Frühwirth, Pascal Brisset. High-Level Implementations of Constraint Handling Rules. ECRC Technical Report EXRC-TR-95-20. 1995. Internet: http://www.pst.informatik.uni-muenchen.de/~fruehwir/Papers/ecrc91-95/ECRC-95-20.ps.Z

[FB94] T. Frühwirth and P. Brisset, The Constraint Handling Rules Library, Chapter in ECLiPSe 3.4 Extensions User Manual, ECRC Munich, Germany, 1994.

[FRU98] Thom Frühwirth. Constraint Handling Rules. 1998. Internet: http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/drafts/jlp-chr1.ps.Z.

[HER93] B. Herbig, Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, 1993.

[HF99a] 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 (C. Holzbaur and T. Frhwirth, Eds.), Taylor & Francis, to appear 2000. Internet: http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/Papers/aai99-final.ps.gz

[HF99b] Christian Holzbaur and Thom Frühwirth, Compiling Constraint Handling Rules into Prolog with Attributed Variables, International Conference on Principles and Practice of Declarative Programming (PPDP'99), Paris, France, September/October 1999. Internet: http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/Papers/ppdp99.ps.gz.

[HF98] C. Holzbaur and T. Frhwirth, Constraint Handling Rules Reference Manual for Sicstus Prolog, Technical Report TR-98-01, sterreichisches Forschungsinstitut fr Artificial Intelligence, Vienna, Austria, 1998, Internet: http://www.sics.se/isl/sicstus/sicstus_34.html

[JAC98] JACK: Java Constraint Kit. Project Description. Internet: http://www.fast.de/~mandel/jack/jack.html.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. Journal of Logic Programming, 20:503–581, 1994.

[MC92] W. McCune. Experiments with discrimination-tree indexing and path-indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147-167, 1992.

[MS98] K. Marriott and P. Stuckey. Programming with Constraints: An Introduction. The MIT Press, 1998.

[MT97] Marc Torrens. An application using the Java Constraint Library: The Air Travel Planning system. Diploma Thesis. Swiss Federal Institute of Technology. 1997.

[POS98] Joachim Posegga. Die Sicherheitsaspekte von Java. Informatik-Spektrum. 21:16-22. 1998.

[SRC99] Java sources of the diploma thesis. Internet: http://www.informatik.uni-muenchen.de/_~schmauss/work/diploma_thesis/

[ST94] G. Smolka and R. Treinen (Eds.), DFKI Oz Documentation Series, DFKI, Saarbrücken, Germany, 1994.

[TWF97] Marc Torrens, Rainer Weigl, Boi Faltings. Java Constraint Library: bringing constraint technology to the Internet using the Java language. Working Notes of the Workshop on Constraints and Agents, Technical Report WS-97-05, AAAI-97, Providence, Rhode Island, USA, July 1997. Internet: http://liawww.epfl.ch/torrens-cgi-bin/form.cgi?AAAI97.ps.gz.

[VH91] P. Van Hentenryck. Constraint logic programming. The Knowledge Engineering Review, 6:151–194, 1991.