Adaptive Constraint Handling with CHR in Java

Armin Wolf

Fraunhofer Gesellschaft
Institute for Computer Architecture and Software Technology (FIRST)
Kekuléstraße 7, D-12489 Berlin, Germany
Armin.Wolf@first.fraunhofer.de http://www.first.fraunhofer.de

Abstract. The most advanced implementation of adaptive constraint processing with Constraint Handling Rules (CHR) is introduced in the imperative object-oriented programming language Java. The presented Java implementation consists of a compiler and a run-time system, all implemented in Java. The run-time system implements data structures like sparse bit vectors, logical variables and terms as well as an adaptive unification and an adaptive entailment algorithm. Approved technologies like attributed variables for constraint storage and retrieval as well as code generation for each head constraint are used. Also implemented are theoretically sound algorithms for adapting of rule derivations and constraint stores after arbitrary constraint deletions. The presentation is rounded off with some novel applications of CHR in constraint processing: simulated annealing for the n queens problem and intelligent backtracking for some SAT benchmark problems.

1 Introduction

Java is a state-of-the-art, object-oriented programming language that is well-suited for interactive and/or distributed problem solving [2,5]. The development of graphical user interfaces is well supported by the JavaBeans concept and the graphical components of the Swing package (cf. [4]). There are several approaches using constraint technologies for (distributed) constraint solving that are based on Java (e.g. [3,14,15]). [14] in particular is a recent approach that integrates Constraint Handling Rules into Java. Constraint Handling Rules (CHR) are multi-headed, guarded rules used to propagate new or simplify given constraints [6,7]. However, this Java implementation of CHR only supports chronological backtracking for constraint deletions, similar to the implementations of CHR in ECLiPSe [8] and SICStus Prolog [11]. Arbitrary additions and deletions of constraints that may arise in interactive or even distributed problem solving environments are not directly supported. These restrictions have been removed by previous – mainly theoretical – work [18,19]. However, an implementation of a CHR system that allows arbitrary additions and deletions of constraints was not yet available.

This paper presents a first implementation of adaptive constraint handling with CHR (c.f. [18]). The implementation language is Java. This imperative

T. Walsh (Ed.): CP 2001, LNCS 2239, pp. 256–270, 2001.

programming language was chosen because of its properties (see above) and because it has no integrated, fixed add/delete mechanism for constraints like Prolog. This latest and advanced implementation of CHR improves the previous implementation in terms of flexibility and/or efficiency. For the user, this CHR implementation offers well-established aspects like

- no restriction of the number of heads in a rule
- compilation of rules in textual order
- constant time access to constraints
- code is compiled not interpreted

and opens up new application areas for CHR in constraint solving:

- local search
- back-jumping and dynamic backtracking
- adaptive solution of dynamic problems

There are several CHR examples in this paper. However, one example will guide us through the chapter on the system. This example is not a typical constraint handler, but it is small and still illustrates various considerations and stages during compilation and use of CHR in Java.

Example 1 (Primes). The sieve of Erathosthenes may be implemented as a kind of a "chemical abstract machine" (c.f. [11]): Assuming that for an integer $n > 2$, the constraints `prime(2)`, ..., `prime(`n`)` are generated. The CHR

```
prime(I) \ prime(J) <=> J mod I == 0 | true.
```

will filter out all non-prime "candidates". If the rule no longer applies, only the constraints `prime(`p`)`, where p is a prime number, are left. More specifically, if there is a constraint `prime(`i`)` and some other constraint `prime(`j`)` such that j `mod` $i = 0$ holds, then j is a multiple of i, i.e. j is non-prime. Thus, `prime(`i`)` is kept but `prime(`j`)` is removed. In addition, the empty body of the rule (`true`) is executed.

The paper is organized as follows. First, the syntax and operational semantics of CHR are briefly recapitulated. Then, the system's architecture, interfaces and performance are described. Specifically, the primes sieve is used as a benchmark to compare the runtime of the system with the recent implementation of CHR in SICStus Prolog. Some novel applications of CHR complete the presentation. The paper closes with some conclusions and a brief outline of future work.

2 The Syntax and Operational Semantics of CHR

Some familiarity with constraint logic programming is assumed (e.g. [13]). The presented CHR implementation supports a restricted set of built-in constraints, which are either syntactic equations or arithmetic relations over a predefined set of arithmetic terms (for details, see [18]). Arbitrary host language statements as

in the SICStus implementation of CHR (see [11]) are not (yet) supported. One reason is that for every host language statement in the body of a CHR there must be an undo-statement, which is executed whenever applications of this rule are no longer valid.

2.1 Syntax

There are three kinds of CHR:

- *Simplification*: $H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_l$.
- *Propagation*: $H_1, \ldots, H_i \Rightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k$.
- *Simpagation*: $H_1, \ldots, H_m \backslash H_{m+1}, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k$.

The *head* $H_1, \ldots, H_i$ is a non-empty, finite sequence of CHR constraints, which are logical atoms. The *guard* $G_1, \ldots, G_j$ is a possibly empty, finite sequence of built-in constraints, which are either syntactic equations or arithmetic relations. If the guard is empty, it has the meaning of `true`. The *body* $B_1, \ldots, B_k$, is a possibly empty, finite sequence of built-in or CHR constraints. If the guard is empty, it has the meaning of `true`.

2.2 Operational Semantics

The operational semantics of CHR in the actual implementation (for details, see [18,19]) is compatible with the operational semantics given in [1,7]. Owing to lack of space, a repetition of the formal definitions is omitted, though an informal description of the operational behaviour of CHR is given, adopting the ideas presented in [11]: a CHR constraint is implemented as both *code* (a Java method) and *data* (a Java object), an entry in the constraint store. Whenever a CHR constraint is added (executed) or woken (re-executed), the applicability of those CHRs is checked that contain the executed constraint in their heads. Such a constraint is called *active*; all other constraints in the constraint store are called *passive*.

Head. The head constraints of a CHR serve as constraint patterns. If the active constraint matches a head constraint of a CHR, passive partner constraints are searched that match the other head constraints of this CHR. If matching partners are found for all head constraints, the guard is executed. Otherwise, the next CHR is tried.

Guard. After successful head matching, the guard must be entailed by the built-in constraints. Entailment means that all arithmetic calculations are defined, i.e. variables are bound to numerical values, arithmetic tests succeed and syntactical equations are entailed by the current constraint store, e.g. $\exists Y\ (X = f(g(Y)))$ is entailed by the equations $X = f(Z)$ and $Z = g(1)$. If the guard is entailed the CHR applies and the body is executed. Otherwise, either other matching partners are searched or, if no matching partners are found, the next CHR is tried.

Body. If the firing CHR is a simplification, all matched constraints (including the active one) are removed from the constraint store and the body constraints are executed. In the case of a simpagation, only the constraints that match the head constraints after the '\' are removed. In the case of a propagation, the body is executed without removing any constraints. It should be noted that a propagation will not fire again with the same matching constraints (in the same order). If the active constraint has not been removed, the next CHR is tried.

Suspension and Wakeup. If all CHR have been tried and the active constraint has not been removed, it suspends until a variable that occurs in it becomes more constrained by built-in constraints, i.e. is bound. Suspension means that the constraint is inserted in the constraint store as data. Wakeup means that the constraint is re-activated and re-executed as code.

3 The System

In the beginning, only the runtime system and the compiler are given. CHR handlers and applications are the responsibility of the user. The runtime system and the compiler contain the data structures that are required to define rule-based adaptive constraint solvers and to implement Java programs that apply these solvers to dynamic constraint problems. The definition of a rule-based constraint solver is quite simple: the CHRs that define the solver for a specific domain are coded in a so-called CHR handler. A CHR handler is a Java program that uses the compiler in a specific manner. Compiling and running a CHR handler generates a Java package containing Java code that implements the defined solver and its interface: the addition or deletion of user-defined constraints or syntactical equations, a consistency test and the explanation of inconsistencies. This problem-specific solver package may be used in any Java application. Figure 1 shows the components and their interactions.

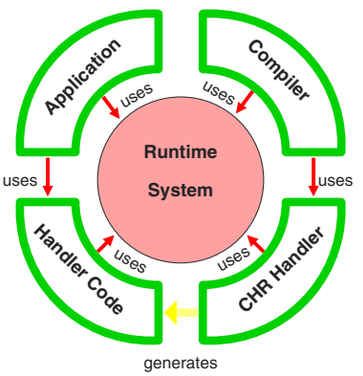


Fig. 1. The architecture of the adaptive CHR system.

3.1 The Runtime System

The core of the adaptive constraint-handling system is its runtime system. Among other things, it implements attributed logical variables (the subclass `Variable` of the class `Logical`) as presented in [10], logical terms (the subclass `Structure` of `Logical`) and data structures for CHR and built-in constraints. For dynamic constraint processing, constraints are justified by integer sets. These sets are implemented as sparse bit vectors (the class `SparseSet`, c.f. [18]). This

implementation is much more storage- and runtime-efficient than the bit-sets in the Java API.[1] Based on these sets and the other data structures, an adaptive unification algorithm [17] and an adaptive entailment algorithm [16] is implemented. The runtime system is the common basis for

- the compiler
- the CHR handlers
- the generated handler packages
- the applications using the handlers

3.2 The Compiler and Its Interface

The compiler class is also written in Java. Logical term objects that represent CHR heads, guards and bodies may be added to a compiler object. Thus, a parsing phase that transfers CHR into an internal representation is unnecessary. All CHRs are represented in a canonical form, which allows uniform treatment of simplifications, propagations and simpagations (c.f. [11]). This form consists of

- a (*remove*) array of all head constraints that are removed when the rule is applied
- a (*keep*) array of all head constraints that are kept when the rule is applied
- an array of all guard conditions that have to be entailed
- an array of all body constraints that are added when the rule is applied

At most one of the two arrays of head constraints may be empty. To define a CHR-based constraint solver, the canonical form of the rules has to be added in a CHR handler to a compiler object.

Example 2 (Primes, continued). The canonical representation of the simpagation `prime(I) \ prime(J) <=> J mod I == 0 | true.` in Java is shown in the CHR handler for the primes sieve presented in Figure 2. The head variables are defined in lines 5 and 6. In line 7, the functor of the unary constraint `prime` is defined. In lines 8 and 9, the head constraints are constructed. The guard condition is constructed in lines 10–12, where the built-in modulo operator `mod_2` and the built-in predicate `identical_2` (the equivalent of Prolog's '`==`') are used. In lines 14–17, the canonical form of the rule is added to the compiler object.

When all rules have been added, the compilation has to be activated. The compiler method `compileAll()` that activates the translation phase is called (c.f. Figure 2, line 18). The generated methods for the active constraints

- match formal parameters to actual arguments of the active (head) constraint
- find and match passive partners for the remaining head constraints
- check the guards

[1] Experiments have shown that the improvement is at least one order of magnitude for randomly generated sparse sets.

```
01  import common.*; // import the runtime system
02  import compile.DJCHR; // import the compiler class
03  public class primeHandler {
04      public static void main( String[] args ) {
05          Variable i = new Variable("I");
06          Variable j = new Variable("J");
07          Functor prime_1 = new Functor("prime", 1);
08          Structure prime_i = new Structure(prime_1, new Logical[]{ i });
09          Structure prime_j = new Structure(prime_1, new Logical[]{ j });
10          Structure cond = new Structure(DJCHR.identical_2, new Logical[] {
11              new Structure (DJCHR.mod_2, new Logical[]{ j, i }),
12              new ZZ(0) }); // j mod i == 0
13          DJCHR djchr = new DJCHR("prime", new Structure[] { prime_1 });
14          djchr.addRule(new Structure[] { prime_j },
15                        new Structure[] { prime_i },
16                        new Structure[] { cond },
17                        null);
18           djchr.compileAll();
19      }
20  }
```

Fig. 2. The CHR handler for the sieve of Eratosthenes.

```
01  public boolean prime_1_0_0 (Constraint pc0, Logical[] args, SparseSet label) {
02      pc0.lock();
03      SolutionTriple etriple = new SolutionTriple(); etriple.addToLabel(label);
04      Logical tmplogical; SparseSet tmplabel;
05      primeVariable local0 = primeVariable.newLocal("J");
06      local0.lbind(args[0], label);
07      boolean applied = false;
08      search: do {
09          primeVariableTable.Stepper st1
10              = primeVarTab.initIteration(new primeVariable[] {  }, 0);
11          while (st1.hasNext()) {
12              Constraint pc1 = st1.next();
13              if (!pc1.isUsable()) continue;
14              SparseSet plab1 = (SparseSet)pc1.getLabel();
15              SolutionTriple.Point point1 = etriple.setPoint();
16              etriple.addToLabel(plab1);
17              primeVariable local1 = primeVariable.newLocal("I");
18              local1.lbind(pc1.getArgs()[0], plab1);
19              do {
20                  SparseSet guardLabel0 = new SparseSet();
21                  Logical logical0 = local0.deref(guardLabel0);
22                  Logical logical1 = local1.deref(guardLabel0);
23                  if ( ! (logical0 instanceof ZZ && logical1 instanceof ZZ &&
24                          ((((ZZ)logical0).val % ((ZZ)logical1).val) == 0)) )
25                              continue;
26                  etriple.addToLabel(guardLabel0);
27                  etriple.add(new Conditional(
28                      new guard_0_0(new primeVariable[] {local0, local1}), guardLabel0));
29                  if (!etriple.getLabel().isEmpty())
30                      derivation.add(
31                          new RuleState_0(-1, new primeVariable[] {local0, local1},
32                          new Constraint[] {pc0, pc1}, (SolutionTriple)etriple.clone()));
33                  primeVarTab.removeConstraint(pc0);
34                  applied = true;
35                  break search;
36              } while (false);
37              etriple.backToPoint(point1);
38          } // end of iteration
39      } while (false);
40      pc0.unlock();
41      return applied;
42  }
```

Fig. 3. Code generated for `prime(J)` in `prime(I)\prime(J)<=>J mod I==0|true`.

- remove matched constraints from the constraint store if required
- execute the bodies

Furthermore, for adaptations after constraint deletions, all constraints are justified by a set of integers. These justifications are used in the generated methods to perform truth maintenance. The generated methods additionally

- unite all justifications of all constraints that are necessary for successful head matching
- unite all justifications of all constraints that are necessary for guard entailment
- justify the executed body constraints with the union of the justifications for head matching and guard entailment
- store justifications and partners of the applied rules in rule state objects

For adaptation after deletions, a rule state class is generated for each CHR. Every rule state class contains a method that retries a previously applied rule if its present justification is no longer valid. If there is no alternative justification, the previous rule application is undone: removed head-matching constraints are re-inserted in the constraint store or re-executed and the consequences of the executed body constraints are erased.

Finding Partner Constraints. Like [11], we believe the real challenge in implementations of *multi-headed* CHRs is efficient computation of joins for partner constraints. A naive solution is to compute the cross-product of all potential partner constraints. However, if there are shared variables in the head constraints, only a subset of the cross-product has to be executed. If we consider, for instance, the transitivity rule `leq(X,Y), leq(Y,Z) ==> leq(X,Z)`, which has to be tried against all active constraints `leq(`u`,` v`)`, only `leq` constraints have to be considered as potential partners that have either v in their first argument position or u in their second. In order to (partially) apply this knowledge, the idea of *variable indexing* (c.f. [11]) is also implemented in our compiler. Thus, the partner search is better focused if the arguments of the active constraints are variables, e.g. if u and v are variables. The constraints in the store are therefore distributed over all variables that occur in these constraints. The constraints are attached to their variables as attribute values (c.f. [10]). The attributes are named after the constraints. For efficient $O(1)$ access to these constraints, the compiler generates for every CHR handler a subclass of variables to which the necessary attributes are added. All constraints defined in the handler must therefore be known by the compiler. This information is passed on when a compiler object is created (e.g. in line 13 in Figure 2). The name of the variable subclass accommodates this, receiving the handler's name as a prefix (e.g. `primeVariable` for the prime handler in Figure 2).

Unlike the SICStus Prolog implementation, the attribute values are not merged when a variable binding occurs. If there is a variable binding $X = f(\ldots Y \ldots)$ or $X = Y$ in SICStus Prolog, the attribute values stored under X

are added to the attribute values in Y because all variable occurrences of X in constraints are "substituted" by $f(\ldots Y \ldots)$ or Y, respectively. In our implementation however, only a "back pointer" ($X \leftarrow Y$) from Y to X is established. The variables, together with these "back pointers", define graph structures; more precisely, rational trees[2] that are traversed to access all the attribute values, i.e. the constraints stored under an unbound variable. This design decision was made because variable bindings caused by built-in constraints might be arbitrarily deleted. In the case of a deletion of $X = f(\ldots Y \ldots)$ or $X = Y$, only the binding itself and the "back pointer" from Y to X have to be deleted. The connected attribute values of X and Y are automatically separated because the attribute values of X are no longer accessible from Y, the connecting link being removed. This approach is much simpler and more efficient than restoring the attribute values.

Example 3 (Primes, continued). The compiled method for the head constraint `prime(J)` in the CHR `prime(I) \ prime(J) <=> J mod I == 0 | true.` is presented in Figure 3. The formal parameter `J` (line 5) is matched to the actual argument `args[0]` (lines 1 and 6) of the active constraint `pc0`. To find a partner constraint matching `prime(I)`, an iteration over all stored constraints is activated until one is found that satisfies the guard condition (lines 9–38). Variable indexing is impossible because there are no common formal head parameters (the array of common `primeVariable` in line 10 is empty). The iteration continues with the next candidate if the current candidate is already being used (line 13). Otherwise, the formal parameter `I` is matched to the actual argument of the candidate `pc1.getArgs()[0]` (lines 17 and 18). Then, the guard is tested (lines 20-24) and the iteration continues with another candidate if the condition `J mod I == 0` is not satisfied (line 25). Otherwise, the rule is applicable and the rule body is normally executed. In this case, the body is empty (`true`) and so only the united justifications (lines 3, 16, 26) for head matching and guard entailment and the partners are stored for adaptation (lines 30–32) if necessary. No adaptation is necessary if the union of all justifications is empty, i.e. always true (c.f. line 29). Last, the active constraint is deleted (line 33) and a Boolean value is returned (line 41). It is true iff the rule was successfully applied and the active constraint was deactivated. This flag is used to prevent the method for the other head constraint `prime(I)` from being activated on `pc0`.

3.3 The Application Interface

During the translation phase for each head constraint of a CHR, a Java method is generated. Methods for constraints that have the same name and arity are subsumed under a method that is named after the constraints and their arities. Furthermore, for each constraint name and arity, there is a method for reading the corresponding constraints out of the constraint store. These methods form the "generic" part of the application interface of the generated constraint solver.

[2] Variable bindings like $X = f(Y)$ and $Y = g(X)$ are allowed, resulting in $X \leftrightarrows Y$.

They are complemented by "non-generic" methods to add syntactical equations to the constraint store, to delete all constraints with a specific justification, to test the consistency of the stored built-in constraints and to get an explanation (justification) for an inconsistency.

Example 4. The application interface generated for the CHR handler in Figure 2 comprises the following methods

- `public void prime_1(Logical[] args, SparseSet label)`
- `public ArrayList get_prime_1()`
- `public void equal(Logical lhs, Logical rhs, SparseSet lab)`
- `public void delete(SparseSet del)`
- `public boolean getStatus()`
- `public SparseSet getExplanation()`

of the class `prime`, the class of constraint stores that are processed by the CHR defined in the CHR handler. The variable subclass `primeVariable` of `Variable` is generated, too.[3]

The use of the interface is shown in the following program:

```
01    import common.*; // import the runtime system
02    import prime; // import the generated prime handler
03    public class primeTest {
04        public static void main( String[] args ) {
05            int n = Integer.parseInt( args[0] );
06            prime cs = new prime();
07            for (int i=2; i <= n; i++)
08                cs.prime_1(new Logical[] {new ZZ(i)}, new SparseSet(i));
09            cs.delete(new SparseSet(2));
10            cs.prime_1(new Logical[] {new ZZ(2)}, new SparseSet(2));
11        }
12    }
```

In lines 7–8, the constraints `prime(2)`, ..., `prime(`n`)` are executed, where n – a positive integer – is read from the command line (see line 5). In line 9, the constraint `prime(2)` (the only constraint justified by 2) is deleted and then re-added in line 10.

3.4 Runtime Comparisons

The sieve of Erathosthenes was used as a benchmark to compare the adaptive Java version with the recent SICStus Prolog implementation of CHR. In particular, the Java program presented in Example 4, which uses the compiled code of the handler in Figure 2, was compared to the following SICStus Prolog program

```
primetest(N) :-
    switch(Start,Phase), generate(3,N), runtime(End),
    Time is End-Start, print(Time), nl,
    Phase=delete, % causes backtracking and the deletion of prime(2)
    runtime(StartReAdd), prime(2), runtime(EndReAdd),
    ReAddTime is EndReAdd-StartReAdd, print(ReAddTime), nl.
switch(Time,process) :- runtime(Time), prime(2).
switch(Time,delete) :- runtime(Time).
runtime(Time) :- statistics(runtime, [_,Time])
generate(I,N) :- I > N, !.
generate(I,N) :- prime(I), J is I+1, generate(J,N).
```

[3] See Section 4.2 for the use of such a `Variable` subclass.

This program uses the SICStus CHR handler

```
handler prime.
constraints prime/1.
prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

runtime measurements were made on a Pentium III PC running SuSE Linux 6.2. For problem sizes $n = 1000, 2000, 4000, 8000$ and 16000, the constraints `prime(2), ..., prime(`n`)` were generated and processed. Then, the constraint `prime(2)` and its consequences were deleted and the result was adapted/re-calculated. For this purpose, in the Java implementation the interface method `delete` was used, which is based on repair algorithms presented in [18,19]. This causes a re-insertion of all constraints on even numbers `prime(`$2k$`)` $2 \leq k \leq n/2$ and a re-removal of all these constraints except `prime(4)`. In the SICStus Prolog implementation however, chronological backtracking to the top level and re-processing of the constraints `prime(3), ..., prime(`n`)` was forced, i.e. the equation `Phase=delete` causes a failure that causes backtracking to the second clause of `switch`. Then, after both kinds of adaptation, the constraint `prime(2)` was re-inserted. In both cases, this causes a removal of the previously re-inserted constraint `prime(4)`.

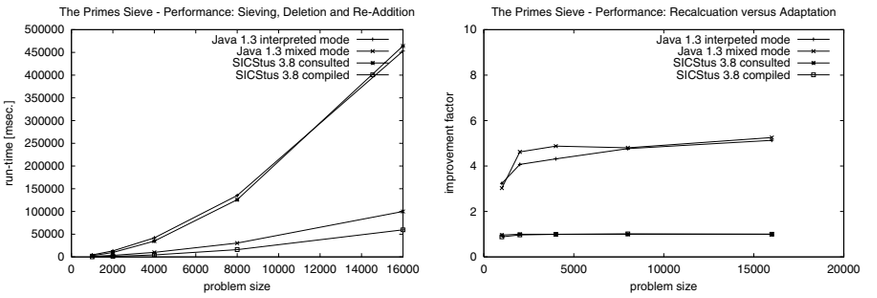
The runtimes for generation and processing show that the purely interpreted Java code is about 1.7 times slower than the consulted SICStus Prolog code and that the partially compiled Java code (Java version 1.3. in mixed mode) is about 2.9 times slower than the compiled SICStus Prolog code.

The runtimes for the deletion of `prime(2)` show the advantage of adaptation over recalculation: the purely interpreted Java code is about 2.6 times faster than the consulted SICStus Prolog code, and the partially compiled Java code is about 1.5 times faster than the compiled SICStus Prolog code.

The runtimes for re-addition of `prime(2)` show that the purely interpreted Java code is about 6 times slower than the consulted SICStus Prolog code, and that the partially compiled Java code is about 5.6 times slower than the compiled SICStus Prolog code.

Overall, the sums of the runtimes for all these operations are surprisingly comparable: Figure 4(a) shows that the performance of the interpreted/consulted code is nearly identical and that the compiled SICStus Prolog code is on the whole marginally faster than the Java code in mixed mode. However, a relative comparison of the two chosen adaptation strategies – "repair" and backtracking – with re-calculation from scratch is shown in Figure 4(b): in Java, the adaptation is 3–5 times faster than re-calculation; performance increases with problem size. Obviously, there is no performance improvement in the SICStus Prolog implementation.

A comparison of our Java implementation of CHR with the one presented in [14] was not considered further. For $n = 1000$, this implementation takes about 1 minute for the generation and processing phase. We assume that the interpretation of CHRs rather than their compilation is the reason for this runtime.



(a) Total Runtime (b) Recalculation versus Adaptation

Fig. 4. A benchmark comparison of the prime handler

4 Applications

The possibility of arbitrary constraint deletions opens up new application areas for CHR in constraint programming. One broad area is local search based on simulated annealing; another is back-jumping and dynamic backtracking. One application shows how CHR are used in a simple simulated-annealing approach to solve the well-known n queens problem. Another application compares chronological backtracking, back-jumping and dynamic backtracking in the solution of satisfiability problems.

4.1 Simulated Annealing for the n Queens Problem

The n queens problem is characterized as follows: place n queens on an $n \times n$ chessboard such that no queen is attacked by another. One simple solution of this problem is to place the n queens (one per row) randomly on the board until no queen is attacked. To detect an attack, the following CHR is sufficient:[4]

```
queen(I,J), queen(K,L) ==> I < K, (J == L ; K-I == abs(L-J))
                         |  conflict(I, 1.0), conflict(K, 1.0).
```

The constraints `conflict(`i`,1.0)` and `conflict(`k`,1.0)` are derived whenever the queens in row/column i/j and k/l are attacking each other: They are either in the same column ($j = l$) or in the same diagonal ($|k - i| = |l - j|$). To detect the queens that are "in conflict with" the maximum number of other queens, the following CHR sums up these numbers:

```
conflict(I,R), conflict(I,S) <=> T is R+S | conflict(I, T).
```

The search algorithm to solve the n queens problem is based on a simple simulated-annealing approach. An initially given temperature is cooled down

[4] The semicolon ';' represents the logical "or" ($\vee$) in the guard of the CHR.

to minimize the total number C of conflicts: $T_k = T_0 \times \rho^k$ $(0 < \rho < 1)$. The search stops if either a solution is found or the temperature is below a predefined level ($T_k < T_{\min}$). While there are conflicts, a queen that is in conflict with the maximum number of other queens is chosen and placed at another randomly selected position, i.e. the corresponding constraint `queen(`i, j`)` is deleted and a new constraint `queen(`i, j'`)` is inserted.[5] If, for the new number D of conflicts, it either holds that $D < C$ or $e^{-\frac{D-C}{T_k}} \geq \delta$, where $0 < \delta < 1$ is a random number, the search continues. Otherwise, the moved queen is placed in its original position. Using this simple simulated-annealing approach, solutions for 10, 20, 30, ..., 100 queens problems were easily found (0.5 sec. for 10 and 30 sec. for 100 queens).

The runtime performance of the implementation is rather poor; it is easily outperformed by other approaches. However, the aim of this example was to show that adaptive constraint handling with CHR can be used for rapid prototyping of local-search algorithms. These prototypes can be used for education or to examine and improve the search algorithm, e.g. the number of search steps required to find a solution.

4.2 Different Search Strategies for SAT Problems

The SICStus Prolog distribution[6] comes with several CHR handlers and example applications. One of these example applications is a SAT(isfiability) problem, called the Deussen problem ulm027r1. It is the conjunctive normal form of a propositional logic formula with 23 Boolean variables. The problem is to find a 0/1 assignment for all these variables such that the formula, a conjunction of Boolean constraints, is satisfied.

To solve such SAT problems, we coded and compiled the necessary CHRs that are part of the Boolean CHR handler in the SICStus Prolog distribution. These rules are:

```
or(0,X,Y) <=> Y=X.
or(X,0,Y) <=> Y=X.
or(X,Y,0) <=> X=0,Y=0.
or(1,X,Y) <=> Y=1.
or(X,1,Y) <=> Y=1.
or(X,X,Z) <=> X=Z.
neg(0,X) <=> X=1.
neg(X,0) <=> X=1.
neg(1,X) <=> X=0.
neg(X,1) <=> X=0.
neg(X,X) <=> fail.
```

```
or(X,Y,A) \ or(X,Y,B) <=> A=B.
or(X,Y,A) \ or(Y,X,B) <=> A=B.
neg(X,Y) \ neg(Y,Z) <=> X=Z.
neg(X,Y) \ neg(Z,Y) <=> X=Z.
neg(Y,X) \ neg(Y,Z) <=> X=Z.
neg(X,Y) \ or(X,Y,Z) <=> Z=1.
neg(Y,X) \ or(X,Y,Z) <=> Z=1.
neg(X,Z) , or(X,Y,Z) <=> X=0,Y=1,Z=1.
neg(Z,X) , or(X,Y,Z) <=> X=0,Y=1,Z=1.
neg(Y,Z) , or(X,Y,Z) <=> X=1,Y=0,Z=1.
neg(Z,Y) , or(X,Y,Z) <=> X=1,Y=0,Z=1.
```

We then implemented three different labelling algorithms to solve SAT problems. A labelling algorithm is a (systematic) search algorithm that assigns a possible value to an unassigned variable – the variable is labelled – until either

[5] From time to time the moved queen is arbitrarily chosen, avoiding starvation.
[6] See `http://www.sics.se/sicstus.html`.

all variables are assigned and the conjunction of all constraints is satisfied or some constraints are violated. If a violation occurs, a labelled variable that has an alternative value is selected. The selected variable is re-assigned an alternative value. If there is a violation but no labelled variable with an alternative value left, then the constraints are inconsistent, i.e. there is no assignment satisfying them.

The implemented labelling algorithms are based on chronological backtracking, back-jumping and dynamic backtracking. Search based on chronological backtracking and back-jumping assigns the variables systematically in a fixed order. In the case of a violation, the last labelled variable is re-assigned if it has an alternative value, otherwise the assignments of the some variables are "forgotten" (deleted). If the search is based on back-jumping, the recent variable assignment that justifies the violation and all the following assignments are forgotten; in the chronological case, e.g. if the justification is missing, only the last variable assignment is forgotten. The search "backtracks" or "back-jumps" until the violation is solved or there is no labelled variable left to backtrack or to jump to. In the latter case, the problem is unsolvable. During search with dynamic backtracking, neither the assignment nor the backtracking is in fixed order. If there is a violation and there is no alternative value for the last assigned variable, only the recent variable assignment is deleted, which justifies this "dead end" of the search process – all other assignments are untouched. A detailed, more formal description of all these algorithms is given in [9].

We implemented search procedures based on back-jumping (DJCHR BJ) and dynamic backtracking (DJCHR DBT) for SAT problems using the compiled Boolean CHR handler in Java 1.3. These implementations were compared with the search procedure, based on chronological backtracking, that comes with the Boolean CHR handler in the SICStus Prolog distribution (SICStus CBT). These three search procedures were used to solve the Deussen problem and some SAT problems that are available in the *Satisfiability Library* (SATLIB).[7] runtime measurements were made on a Pentium III PC using SICStus Prolog with consulted program code and Java 1.3 in mixed mode.

Table 1 shows the counted numbers of backtracking/back-jumping steps and the required runtime in milliseconds, used to find the (first) solution or to detect the unsatisfiability of the problem. These runtime experiments show that either back-jumping or dynamic backtracking requires less backtracking/back-jumping steps than chronological backtracking for the considered problems. Additionally, the improved search yields better absolute runtime performance of the Java implementations for nearly all the examined benchmarks.

This application impressively demonstrates the new possibilities offered by adaptive constraint handling with CHR: the existence of justifications for all derived constraints including false allows high-level implementations of sophisticated backtracking and search algorithms.

[7] The whole benchmark set is available online at `www.satlib.org`.

Table 1. Runtime comparison on SATLIB benchmark problems (except ulm027r1).

SATLIB benchmark problems	number of solutions	SICStus CBT steps	SICStus CBT msec.	DJCHR BJ steps	DJCHR BJ msec.	DJCHR DBT steps	DJCHR DBT msec.
Deussen ulm027r1	16	52	250	36	939	72	1524
The Pigeon Hole 6	0	14556	20950	3646	17837	1452121	19384972
aim-50-2_0-yes1-1	1	11110	61310	552	6062	5178	47850
aim-50-2_0-yes1-2	1	384	2000	154	2519	90	1805
aim-50-2_0-yes1-3	1	34088	168180	301	3340	978	15951
aim-50-2_0-yes1-4	1	302	2160	123	2540	167	3416
aim-50-2_0-no-1	0	906558	1706830	44492	429141	17697	184587
aim-50-2_0-no-2	0	70266	415340	944	13340	25528	418031
aim-50-2_0-no-3	0	172150	674910	46526	483830	295792	3817240
aim-50-2_0-no-4	0	53874	236130	198	4298	5689	85381

5 Conclusions and Future Work

The adaptive CHR system outlined in this paper was implemented over a six months period. The implemented system is the first system to combine recent developments in CHR implementation with dynamic constraint solving. More specifically, the number of constraints in CHR's heads is no longer limited to two, and rational trees of attributed variables are used to implement efficient access to the constraint store, especially during the partner search. Furthermore, arbitrary constraint additions and deletions are fully supported: constraint processing is automatically adapted. This opens up new areas in constraint programming for CHR. Three of these are now implemented: simulated annealing and adaptive search with back-jumping or dynamic backtracking. For the future, interactive diagrammatic reasoning with CHR is planned as well as the application of other "fancy backtracking" algorithms on harder SAT problems, e.g. all the AIM instances (c.f. [12]) will be examined and discussed.

Other future activities will concentrate on the compiler in order to produce highly optimized code. Besides general improvements like early guard evaluation and the avoidance of code generation and processing, there are improvements of the adaptation process. This will make it possible to avoid re-processing of constraints that are removed by rule applications and later re-activated by undoing these applications during adaptation. In some cases, it is correct and more efficient to put them directly back in the constraint store rather than activate them. This holds for removed constraints that would not have been re-activated by a later wake-up even if they would not have been removed.

Acknowledgement. The author wishes to thank Kathleen Steinhöfel for the crash course in simulated annealing and all the colleagues he met in Melbourne and who helped him with their valuable remarks and fruitful discussions. Special thanks go to Christian Holzbaur, Thom Frühwirth, Kim Marriott, Bernd Meyer and Peter Stuckey.

References

1. Slim Abdennadher. Operational semantics and confluence of Constraint Handling Rules. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming – CP97*, number 1330 in Lecture Notes in Computer Science. Springer Verlag, 1997.
2. Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, June 2000.
3. Andy Hon Wai C. Constraint programming in Java with JSolver. In *Proceedings of PACLP99, The Practical Application of Constraint Technologies and Logic Programming*, London, April 1999.
4. David Flanagan. *Java Foundation Classes in a Nutshell*. O'Reilly, September 1999.
5. David Flanagan. *Java in a Nutshell*. O'Reilly, 3rd edition, November 1999.
6. Thom Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science, pages 90–107. Springer Verlag, March 1995.
7. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *The Journal of Logic Programming*, 37:95–138, 1998.
8. Thom Frühwirth and Pascal Brisset. High-Level Implementations of Constraint Handling Rules. Technical report, ECRC, 1995.
9. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
10. Christian Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification*. PhD thesis, Dept. of Medical Cybernetics & AI, University of Vienna, 1990.
11. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. *Applied Artificial Intelligence*, 14(4):369–388, April 2000.
12. K. Iwama, E. Miyano, and Y. Asahiro. Random generation of test instances with controlled attributes. In *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 377–394. American Mathematical Society, 1996.
13. Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
14. Matthias Schmauss. An implementation of CHR in Java. Master's thesis, Ludwig Maximilians Universität München, Institut für Informatik, May 1999.
15. Marc Torrens, Rainer Weigel, and Baoi Faltings. Java constraint library: Bringing constraint technology on the internet using java. In *Proceedings of the CP-97 Workshop on Constraint Reasoning on the Internet*, November 1997.
16. Armin Wolf. Adaptive entailment of equations over rational trees. In *Proceedings of the 13th Workshop on Logic Programming, WLP'98*, Technical Report 1843-1998-10, pages 25–33. Vienna University of Technology, October 1998.
17. Armin Wolf. Adaptive solving of equations over rational trees. In *Proceedings of the Fourth International Conference on Principles and Practice on Constraint Programming, CP'98, Poster Session*, number 1520 in Lecture Notes in Computer Science, page 475. Springer, 1998.
18. Armin Wolf. *Adaptive Constraintverarbeitung mit Constraint-Handling-Rules – Ein allgemeiner Ansatz zur Lösung dynamischer Constraint-Probleme*, volume 219 of *Disserationen zur Künstlichen Intelligenz (DISKI)*. infix, November 1999.
19. Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On incremental adaptation of CHR derivations. *Applied Artificial Intelligence*, 14(4):389–416, April 2000.