Electronic Notes in Theoretical Computer Science 59 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 22 pages

As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs

Thom Frühwirth

University of Ulm
Fakultät für Informatik
Albert-Einstein-Allee 11, D-89069 Ulm, Germany

Abstract

In previous papers we showed that from a suitable termination order (called ranking) one can automatically compute the worst-case time complexity of a CHR constraint simplification rule program from its program text. We combined the worst-case derivation length of a query predicted from its ranking with a worst-case estimate of the number and cost of rule application attempts and the cost of rule applications to obtain the desired meta-theorem.

Here we generalize the approach presented in these papers and use it to analyse several non-trivial rule-based constraint solver programs. These results also hold for naive CHR implementations. We also present empirical evidence through test runs that the actual run-time of a state-of-the-art CHR implementation is much better due to optimizations like indexing.

Keywords: Program Analysis, Complexity Analysis, Cost Analysis, Rankings, Derivation Length, Termination, Constraint Solving, Constraint Handling Rules.

1 Introduction

CHR [Fru98,AAI00,SaAb00] are a committed-choice concurrent constraint logic programming language consisting of guarded rules that work on conjunctions of constraints [FrAb02]. A CHR program consists of simplification and propagation rules. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification.

Properties like rule confluence [AFM99] and program equivalence [AbFr99] have been investigated for CHR. These properties are decidable for terminat-

[1] WWW: http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/

ing programs. In a previous paper [Fru00a] we have proven termination of simplification rule programs using rankings. A *ranking* maps *lhs (left hand side)* and *rhs (right hand side)* of each simplification rule to a non-negative integer, such that the rank of the lhs is strictly larger than the rank of the rhs. Intuitively then, the rank of a given constraint problem yields an upper bound on the number of rule applications, because each rule application decreases the rank by at least one [Fru00b].

Example 1.1 Consider the constraint `even` that ensures that a natural number (written in successor notation) is even:

```
even(0) <=> true.
even(s(N)) <=> N=s(M), even(M).
```

The first rule says that `even(0)` can be simplified to `true`, a built-in constraint that is always satisfiable. In the second rule, the built-in constraint `=` stands for syntactic equality: `N=s(M)` ensures that `N` is the successor of some number `M`. The comma stands for conjunction. The rule says that if the argument of `even` is the successor of some number `N`, then the predecessor of this number, `M`, must be even.

If a constraint matches the lhs of a rule, it is replaced by the rhs of the rule. If no rule matches a constraint, the constraint delays. For example, the constraint problem (query) `even(N)` delays. When the constraint `N=0` is added, `even(N)` is woken and behaves like the query `even(0)`. It reduces to `true` with the first rule. To the query `even(s(X))` the second rule is applicable, the answer is `X=s(M),even(M)`. The query `even(s(0))` results in an inconsistency after application of the second rule, since `0=s(M)` is unsatisfiable.

An obvious ranking for the rules of `even` is

$$rank(even(N)) = size(N)$$

$$size(0) = 1$$
$$size(s(N)) = 1 + size(N)$$

The ranking not only proves termination, it also gives an upper bound on the derivation length, in case the argument of `even` is completely known: With each rule application, the rank of the argument of `even` decreases by 2.

In [Fru00b] we have shown that the derivation length is not a suitable measure for worst-case time complexity. The run-time of a CHR program not only depends on the number of rule applications, but also, more significantly, on the number of rule application *attempts* (rule tries).

In [Fru02] we combined the predicted worst-case derivation length with a worst-case estimate of the number and cost of rule tries and the cost of rule applications to obtain a meta-theorem for the worst-case time complexity of CHR constraint simplification rule programs.

Example 1.2 [Contd.] It is easy to show that the worst-case time complexity of a single `even` constraint is linear in the derivation length, i.e. the rank.

The same observation holds for a query consisting of several ground `even` constraints, if the rank is defined as the sum of the ranks of the individual constraints.

However, things change when we add the rule:

```
even(s(X)),even(X) <=> false.
```

where `false` is a built-in constraint that is always unsatisfiable. This rule may be applicable to *all pairs of* `even` constraints in a query, and again after a reduction of a single `even` constraint with one of the other two rules. Of course in most cases, the rule application attempts (rule tries) will be in vain.

Thus the number of rule tries in a single derivation step is at worst quadratic in the number of `even` constraints in the query. Since the rank of an `even` constraint is at least one, the rank of the query is a bound on the number of constraints. The number of derivation steps is also bounded by the rank of the query. Overall, this yields an algorithm that is cubic in the rank of the query.

Related Work. To the best of our knowledge, only the work of Ganzinger and McAllester [McA99,GaMc01,GaMc02] is closely related to our work in that it gives several complexity meta-theorems for a logical rule-based language. These papers investigate bottom-up logic programming as a formalism for expressing static analyses and related algorithms. [McA99] is concerned with certain propagation rules (in our terminology), while [GaMc01] extends the rule language with deletions of atomic formulae and static priorities between rules, and [GaMc02] adds dynamic priorities. Such rules correspond to CHR simplification or simpagation rules [Fru98] that are applied in textual order (dynamic priorities can be implemented as constraints).

Ganzinger and McAllester prove several complexity theorems which allow, in many cases, to determine the asymptotic running time of a bottom-up logic programm by inspection. The main difference and complementarity between their work and our paper is that they consider rules that must be applied to ground formulae at run-time, while we consider simplification rules that involve free variables at run-time and arbitrary built-in constraints. They deal with the complexity of optimally implemented programs using clever indexing and structure sharing, while our results apply also to naive implementations of CHR.

In the complexity meta-theorem of [GaMc01], the complexity is the sum of the syntactic size of the query and the worst number of potential prefix firings of the rules in the program. Prefix firings are ground sub-formulas of lhs instances of a rule that could occur in a derivation. (The paper [GaMc02] adds a logarithmic factor for rules with dynamic priorities.) Here it is - ignoring the cost of built-in constraints - the sum of the rank of the query and the number of potential rule applications. The computation of the number of prefix firings requires insights about the states in all valid computations that can be performed. The number of potential rule applications can be computed

automatically from the program text, once a ranking is known.

This paper is a companion paper to [Fru02], which was based on [Fru01]. Here we generalize the approach presented in these papers by allowing for a more general definition of ranking functions and use it to analyse several non-trivial rule-based constraint solver programs.

Overview of the Paper. We will first give syntax and semantics of CHR. In Section 3, we introduce rankings and show how they can be used to derive upper bounds for worst-case derivation lengths. In the next section we show how to use these derivation lengths to predict the worst-case complexity of CHR programs. Finally, the fifth section reviews some CHR constraint solver programs. Based on the predicted worst-case derivation lengths, the worst-case time complexity is computed according to our complexity meta-theorem. The predictions are compared with preliminary empirical run-time measurements. We conclude with a discussion of the results obtained.

2 Syntax and Semantics of CHR

In this section we give syntax and semantics for CHR, for details see [AFM99]. We assume some familiarity with (concurrent) constraint (logic) programming [MaSt98,FrAb02].

A *constraint* is a predicate (atomic formula) in first-order logic. We distinguish between *built-in (or predefined) constraints* and *CHR (or user-defined) constraints*. Built-in constraints are those handled by a given constraint solver. CHR constraints are those defined by a CHR program.

In the following definitions, upper case letters stand for conjunctions of constraints.

Definition 2.1 A CHR program is a finite set of CHR. There are two kinds of CHR. A *simplification CHR* is of the form

n @ H <=> G | B

and a *propagation CHR* is of the form

n @ H ==> G | B

where the rule has an optional name `n` followed by the symbol @. The lhs H (head) is a conjunction of CHR constraints. The optional guard G followed by the symbol | is a conjunction of built-in constraints. The rhs B (body) is a conjunction of built-in and CHR constraints.

The *operational semantics* of CHR programs is given by a state transition system. With *derivation steps (transitions, reductions)* one can proceed from one state to the next.

Definition 2.2 A *state (or: goal)* is a conjunction of built-in and CHR constraints. An *initial state (or: query)* is an arbitrary state. In a *final state*

(or: answer) either the built-in constraints are inconsistent or no derivation step is possible anymore. A *derivation* is a sequence of derivation steps $S_1 \longmapsto S_2 \longmapsto S_3 \ldots$. The *derivation length* is the number of derivation steps in a derivation.

Definition 2.3 Let P be a CHR program and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows:

Simplify
$H' \wedge C \longmapsto (H = H') \wedge G \wedge B \wedge C$
if $(H \texttt{ <=> } G \mid B)$ in P and $CT \models C \rightarrow \exists \bar{x}(H = H' \wedge G)$

Propagate
$H' \wedge C \longmapsto (H = H') \wedge G \wedge B \wedge H' \wedge C$
if $(H \texttt{ ==> } G \mid B)$ in P and $CT \models C \rightarrow \exists \bar{x}(H = H' \wedge G)$

When we use a rule from the program, we will rename its variables using new symbols, and these variables are denoted by the sequence $\bar{x}$. A rule with lhs H and guard G is *applicable* to CHR constraints H' in the context of constraints C, when the condition holds that $CT \models C \rightarrow \exists \bar{x}(H = H' \wedge G)$.

Any of the applicable rules can be applied, but it is a committed choice, it cannot be undone. If an applicable simplification rule $(H \texttt{ <=> } G \texttt{ | } B)$ is applied to the CHR constraints H', the **Simplify** transition removes H' from the state, adds the rhs B to the state and also adds the equation $H = H'$ and the guard G. If a propagation rule $(H \texttt{ ==> } G \texttt{ | } B)$ is applied to H', the **Propagate** transition adds B, $H = H'$ and G, but does not remove H'.

We finally discuss in more detail *the rule applicability condition* $CT \models C \rightarrow \exists \bar{x}(H = H' \wedge G)$. The equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. More precisely, by $(H_1 \wedge \ldots \wedge H_n) = (H'_1 \wedge \ldots \wedge H'_n)$, where conjuncts can be permuted, we mean $(H_1 = H'_1) \wedge \ldots \wedge (H_n = H'_n)$. By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $(t_1 = s_1) \wedge \ldots \wedge (t_n = s_n)$. The symbol $=$ is to be understood as built-in constraint for syntactic equality.

Operationally, the rule applicability condition can be checked as follows: Given the built-in constraints of C, try to solve the built-in constraints $(H = H' \wedge G)$ without further constraining any variable in H' and C. This means that we first check that H' matches H and then check the guard G under this matching.

As a consequence, in a CHR implementation, there are several computational phases when a rule is applied:

LHS Matching: Atomic CHR constraints in the current state have to be found that match the lhs constraints of the rule.

Guard Checking: It has to be checked if the current built-in constraints imply the guard of the rule.

RHS Handling: The built-in and CHR constraints of the rhs are added.

Before that, the CHR constraints of the lhs are removed.

In this paper we are only concerned with simplification rules. For the rest of the paper we assume that CHR programs do not contain any propagation rules.

3 Rankings and Derivation Lengths

In this section, we introduce rankings for constraint simplification rules and show how the rankings can be used to derive upper bounds for worst-case derivation lengths of CHR programs.

A ranking is an arithemtic function that maps terms and formulae to integers. It is inductively defined on the function symbols, predicate symbols and logical connectives (in our case, conjunction only). The resulting order on formulae is total. It is *well-founded* if we can prove that it is non-negative for the formulae under consideration. Of course we are looking for rankings that allow to decide the order relation.

Of particular interest are ranking functions that are linear polynomials. They are simple but seem sufficient to cover common constraint solver programs [Fru00a,Fru00b].

Definition 3.1 Let f be a function or predicate symbol of arity $n(n \geq 0)$ and let $t_i(1 \leq i \leq n)$ be terms. A *(linear polynomial) CHR ranking (function)* defines the rank of a term or constraint atom $f(t_1, \ldots, t_n)$ as follows:

$$rank(f(t_1, \ldots, t_n)) = a_0^f + a_1^f * rank(t_1) + \ldots + a_n^f * rank(t_n)$$

where the a_i^f are integers. For each built-in constraint C we impose $rank(C) = 0$. The rank of a conjunction is the sum of the ranks of its conjuncts:

$$rank((A \wedge B)) = rank(A) + rank(B)$$

For each formula B we require $rank(B) \geq 0$.

This definition generalizes the one of [Fru02] from natural numbers to integers, in particular it is not required that variables and CHR constraints have a non-zero, strictly positive rank.

The rank of any built-in constraint is 0, since we assume that their termination and time complexity is known. A built-in constraint may imply order constraints between the ranks of its arguments (interargument relations), such as $s = t \rightarrow rank(s) = rank(t)$, where s and t are terms. Note that $=$ on the lhs stands for syntactical equality between two terms s and t, and $=$ on the rhs for arithmetic equality.

These order constraints are helpful to establish termination by showing that the rank of the lhs of a rule is always strictly larger than the rank of the rhs of the rule.

Definition 3.2 Let *rank* be a CHR ranking function. The *ranking (condition) of a simplification rule* H `<=>` $G \mid B$ is the formula

$\forall\ (O \rightarrow rank(H) > rank(B))$,

where O is a conjunction of order constraints implied by the built-in constraints in the rule, $\forall\ ((G \wedge B) \rightarrow O)$.

Since termination is undecidable for CHR, a suitable ranking and suitable order constraints cannot be found completely automatically.

To prove termination, goals have to be sufficiently known.

Definition 3.3 A goal B is *bounded* if the rank of any allowed instance of B is bounded from above by a constant.

The notion of *allowed instance* allows us to ignore certain instances, for example those that we would consider as ill-typed. Of course, allowedness should be a decidable property. Bounded goals not only terminate, their ranks provide an upper bound on the number of rule applications (derivation steps), i.e. derivation lengths.

Theorem 3.4 *Let P be a CHR program containing only simplification rules.*

1. [Fru00a] If the ranking condition holds for each rule in P, then P is terminating for all bounded goals.

2. [Fru00b] If the ranking condition holds for each rule in P, then a worst-case derivation length D for a bounded goal B in P is the rank of B:

$$D = rank(B)$$

Note that the Theorems in the cited papers only apply to linear polynomial rankings over natural numbers, but can be generalized to arbitrary well-founded rankings.

4 Worst-Case Time Complexity

We first consider the worst cost of applying a single rule, which consists of the cost to try the rule on all CHR constraints in the current state and of the cost to apply the rule to some CHR constraints in the state. Then we choose the worst rule in the program and apply it in the worst possible state of the derivation. Multiplying the result with the worst-case derivation length gives us the desired upper bound on the worst-case time complexity.

In the following, we assume a naive implementation of CHR with no optimizations. The complexity of handling built-in constraints is predetermined by the built-in constraint solvers used. We assume that the time complexity of checking and adding built-in constraints is not dependent on the constraints accumulated so far in the derivation. While this is not true in general, it holds for all the constraint programs we have considered so far, because the built-in constraints that appear in CHR programs are usually simple.

Lemma 4.1 *[Fru02] Let there be a simplification rule S of the form H <=> $G \mid C \wedge B$, where H is a conjunction of n CHR constraints, G and C are built-*

in constraints and B are CHR constraints. A worst-case time complexity of applying the rule S in a state with c CHR constraints *is:*

$$O(c^n(O_H + O_G) \ + \ (O_C + O_B)),$$

where O_H is the complexity of matching the lhs H of the rule, O_G the complexity of checking the guard G, O_C the complexity of adding the rhs built-in constraints C, and O_B the complexity of removing the lhs CHR constraints and of adding the rhs CHR constraints B.

Now we are ready to give our meta-theorem about the time complexity of simplification rule programs. To compute the time complexity of a derivation, we have to find the worst-case for the application of a rule, i.e. the largest number of CHR constraints c_{max} of any state in a derivation and the most costly rule that could be tried and applied. We know that the number of derivation steps is bounded by the rank D, because each derivation step decreases the value of D by at least 1. This theorem generalizes the one of [Fru02] to the case where ranks of formulae with CHR constraints can be zero. We therefore have to redo (a part of) the proof.

Theorem 4.2 *Let P be a CHR program containing only constraint simplification rules. Given a query with worst-case derivation length D. Then the worst-case time complexity of a derivation starting with the given query is:*

$$O(D \sum_i ((c + D)^{n_i}(O_{H_i} + O_{G_i}) \ + \ (O_{C_i} + O_{B_i}))),$$

where the index i ranges over the rules in the program P.

Proof. *In the worst-case of a naive implementation, in each of the D derivation steps, all rules are tried on all combinations of the maximum possible number of constraints c_{max} and then the most costly rule is applied. Since rule application attempts are independent from each other, we can extend Lemma 4.1 to a set of rules in a straightforward way:*

$$O(\sum_i c_{max}^{n_i}(O_{H_i} + O_{G_i}) \ + \ \mathbf{Max}_i(O_{C_i} + O_{B_i})),$$

where c_{max} is the worst number of CHR constraints in a derivation from a given query and $\mathbf{Max}_i$ takes the maximum over all i. Since the functions $\mathbf{Max}$ and $+$ are equivalent in the O-notation, we can replace $\mathbf{Max}_i$ by $\sum_i$. This gives us the complexity for one derivation step.

Multiplying the resulting formula by the derivation length D yields the overall complexity:

$$O(D \sum_i (c_{max}^{n_i}(O_{H_i} + O_{G_i}) \ + \ (O_{C_i} + O_{B_i}))).$$

Now we need a bound on $c_{max}^{n_i}$ that only depends on properties of the query, namely c, the number of constraints in the query, and D, the upper bound on the derivation length. There cannot be more than $c + O(D)$ CHR constraints in any state of a derivation starting with the query, because we start from c

constraints and there are at most D derivations steps, and each of them adds at most a certain, fixed number of new constraints (it also removes old ones) which is given by the rules in the program. After replacing c_{max} by the bound $c + O(D)$, we arrive at the formula of the Theorem. □

Note that in many cases, D contains the factor c, so that $c+D$ simplifies to just D (as in the corresponding Theorem of [Fru02]). ¿From the meta-theorem it can be seen that the cost of rule tries dominates the complexity of a naive implementation of CHR.

We end this section with some general remarks on the complexities of the constituents of a simplification rule. The cost of syntactic matching O_H is determined by the syntactic size of the lhs in the given program text. Thus, its time complexity is constant. We assume that the complexity O_B of removing and adding CHR constraints (without applying any rules) is constant in a naive implementation where e.g. lists are used to store the CHR constraints.

As far as the built-in constraints are concerned, we can only make the following general remarks. The complexity of guard checking O_G is usually at most as high as the complexity of adding the respective constraints. The worst-case time complexity of adding built-in constraints O_C is often linear in their size.

5 Time Complexity of CHR Constraint Solvers

We now derive worst-case time complexities of constraint solvers for finite interval domains employing arc consistency, linear polynomials employing variable elimination and description logic [Fru98] from the CHR library of Sicstus Prolog [HoFr98,HoFr00]. As in the example of the introduction, we will use *concrete syntax* of Prolog implementations of CHR, where a conjunction is a sequence of conjuncts separated by commas.

We will contrast these results with the time complexities derived from a preliminary set of test runs with randomized data. We expect the empirical results to be better than the predicted ones, since this CHR implementation uses indexing for computing the combinations of constraints needed for lhs matching of a rule. The Sicstus Prolog and CHR source code for the test runs is available at `www.informatik.uni-muenchen.de/~fruehwir/chr/complexity.pl`. The code can be run via the WWW-interface of CHR Online [SaAb00].

For each solver, we will give a ranking that is an upper bound on the derivation length. From the ranking, we calculate the worst-case time complexity. We denote constant time complexity by the number 1 and zero time by 0 (this means that no computation is performed at all). We will summarize the empirical results of the test runs in a table, see e.g. Fig. 1. The tables have the following columns:

Goal the (abbreviated) goal that was run to produce the test data.

Worst the predicted worst-case derivation length D for the goal.

Apply the actual number of rule applications, i.e. derivation length.

Try the number of rules that have been tried, but not necessarily applied.

Time the time to run the goal with the CHR library of Sicstus Prolog, in seconds, including instrumented source code for randomization, on a recent Linux PC with medium work load.

5.1 Finite domains FD

Finite domains are one of the success stories of constraint logic programming. Many real-life combinatorial problems can be expressed in this constraint system, most prominently scheduling and planning applications. This constraint system was the result of a synthesis of logic programming and finite domain constraint networks as explored in artificial intelligence research since the late 60ties.

In this constraint system [vHSD95], variables are constrained to take their value from a given, finite set. Choosing integers for values allows for arithmetic expressions as constraints. Constraint propagation proceeds by removing values that do not participate in any solution from the sets of possible values.

Here we present an implementation of an arc consistency algorithm for integer interval constraints [vHDT92,Ben95] (a special case of finite domain constraints). Arc consistency distinguishes a special class of unary constraints of the form $X \in S$, where S is a given finite set of values.

Definition 5.1 An atomic constraint $c(X_1, \ldots, X_n)$ is *(hyper-)arc consistent* with respect to a conjunction of unary constraints $X_1 \in S_1 \ \wedge \ldots \wedge \ X_n \in S_n$, if for all $i \in \{1, \ldots, n\}$ and for all possible values for X_i taken from its domain S_i the constraint $X_1 \in S_1 \ \wedge \ldots \wedge \ X_n \in S_n \ \wedge \ c(X_1, \ldots, X_n)$ is satisfiable.

In other words, in an arc consistent atomic constraint, every value of every variable domain takes part in a solution of the atomic constraint. An atomic constraint can be made arc consistent by deleting those values from the domain of the variables that do not participate in any solution of the constraint. A conjunction of constraints is arc consistent if each atomic conjunct is arc consistent. In our case, the domains are intervals of integers, and values are deleted from domains by making intervals smaller.

In the following rules of the solver *Intv*, the unary interval constraint `X in A:B` stands for $X \in \{i \in \mathcal{Z} \mid A \leq i \wedge i \leq B\}$. `in, le, eq` and `add` are CHR constraints, the inequalities `<, =<, >, >=, <>` are built-in arithmetic constraints, and `min, max, +, -` are built-in arithmetic functions. Intervals of integers are closed under computations involving only these functions. The built-in prefix operator `not` negates its argument. The rules affect the interval constraints only, the constraints `le, eq` and `add` remain unaffected.

```
inconsistency @ X in A:B <=> A>B | false.
intersection  @ X in A:B, X in C:D <=> A=<B,C=<D |
     X in max(A,C):min(B,D).
```

The rules `inconsistency` and `intersection` remove one interval constraint each. The built-in inequalities `A=<B` and `C=<D` used in the guards of the rules ensure that these rules apply only to non-empty intervals. The remaining built-in inequalities in the guards ensure that in each rule, at least one interval gets strictly smaller. This also holds for the following rules. The next rules deal with inequalities:

```
le @ X le Y, X in A:B, Y in C:D <=> A=<B,C=<D, B>D  |
     X le Y, X in A:D, Y in C:D.
le @ X le Y, X in A:B, Y in C:D <=> A=<B,C=<D, C<A  |
     X le Y, X in A:B, Y in A:D.
eq @ X eq Y, X in A:B, Y in C:D <=> A=<B,C=<D, A<>C |
     X eq Y,X in max(A,C):B,Y in max(C,A):D.
eq @ X eq Y, X in A:B, Y in C:D <=> A=<B,C=<D, B<>D |
     X eq Y,X in A:min(B,D),Y in C:min(D,B).
```

The next rule deals with addition.

```
add @ add(X,Y,Z),
     X in A:B, Y in C:D, Z in E:F <=> A=<B,C=<D,
        not (A>=E-D,B=<F-C,C>=E-B, D=<F-A,E>=A+C,F=<B+D) |
     add(X,Y,Z),
     X in max(A,E-D):min(B,F-C),
     Y in max(C,E-B):min(D,F-A),
     Z in max(E,A+C):min(F,B+D).
```

Complexity. We rank constraints by the width (size) of their intervals:

$rank(X \ in \ A{:}B) = 2 + width(A{:}B)$
$rank(A) = 0$ otherwise

$width(A{:}B) = B - A$ if $A \leq B$
$width(A{:}B) = -1$ otherwise

For the ranking, 2 is added to the interval width such that empty and singleton intervals have positive ranks as well. ¿From the ranking we can see that any goal with given intervals is bounded. This corresponds to the intended use of the constraint solver program.

We assume that each variable in a query is associated with exactly one interval domain constraint and that in each atomic constraint, all variables are pairwise different. Let w be the the maximum rank of an interval constraint in a query and let v be the number of different variables in the query. Then the derivation length is bounded by

$$D_{Intv} = vw$$

since with each rule application, at least one interval gets smaller (or is removed).

We further assume that the arithmetic built-in constraints take constant

time to compute. All guards and all rhs take constant time. Only the number of lhs constraints differs with the rules, n ranges from 1 to 4. Hence, according to Theorem 4.2, the complexity is $O(vw((c+vw)^4(1+1)+(1+1)))$. Since by definition, $v \leq c$, we can replace v by c and give the resulting simpler complexity expression

$$O_{Intv}(c^5 w^5)$$

which also holds in case there are none or several interval constraints for a variable.

Empirical Results. In Fig. 1, the query with *tadd* takes a list of v different variables and produces the two constraints *add*($A_{2i}, A_{2i+1}, A_{2i+2}$), A_{2i} *le* A_{2i+2} where $1 \leq 2i \leq v$. Hence, for v variables, exactly $v-2$ constraints are produced. The interval domains for the variables are generated randomly, they are non-negative and the upper bound increases by 100 for every other variable to increase the probability of consistency in presence of the constraint A_{2i} le A_{2i+2}. Hence the maximum interval domain size w is $50v$.

The query *len(L,v), genless...* generates a sequence of v finally inconsistent *add* constraints involving v variables, all domains have initially width $w = 200$.

The table shows that

- The behavior of the random problem instances is quite stable.
- The actual derivation length can be much better than the predicted worst-case derivation length, but the last entries show that, depending on the problem type, the worst-case can be eventually reached as problem size increases.
- The number of rule tries is roughly linear in the number of rule applications *tadd*, but not for *genless*.
- Time is roughly linear in the number of rule tries.

For this solver, for the preliminary set of examples we investigated, the number of variables v is linear to the number of constraints c and the worst observed time complexity was just:

$$O^{obs}_{Intv}(c^2 w)$$

The empirical results are better than the predicted ones, since the investigated CHR implementation uses indexing for computing the combinations of constraints needed for lhs matching of a rule.

5.2 Linear Polynomial Equations $\Re$

For solving linear polynomial equations, a minimalistic but powerful variant of variable elimination [Imb95] is employed in the available CHR constraint solvers.

Definition 5.2 A *linear polynomial equation* is of the form $p+b=0$ where b is a constant and the polynomial p is the sum of monomials of the form $a_i * x_i$ with

Goal	Worst	Apply	Try	Time
tadd(L,100)	490196	352	1894	0.51
tadd(L,100)	490196	352	1894	0.50
tadd(L,100)	490196	340	1843	0.49
tadd(L,100)	490196	339	1831	0.50
tadd(L,100)	490196	349	1885	0.51
tadd(L,200)	1980396	718	3869	1.04
tadd(L,200)	1980396	702	3794	1.02
tadd(L,200)	1980396	706	3809	1.03
tadd(L,200)	1980396	715	3854	1.06
tadd(L,200)	1980396	714	3848	1.03
genless(U,L,Z),				
len(L,10),...	2040	884	3737	1.11
len(L,20),...	4080	1420	7243	2.12
len(L,30),...	6120	2308	13569	3.96
len(L,40),...	8160	3735	24973	7.23
len(L,50),...	10200	5482	40967	11.78
len(L,60),...	12240	7549	62251	17.87

Fig. 1. Results from Test Runs with Interval Arc Consistency Constraints

coefficient $a_i \neq 0$ and x_i is a variable. Constants and coefficients are numbers. Variables are totally ordered. In an equation $a_1 * x_1 + \ldots + a_n * x_n + b = 0$, variables appear in strictly descending order.

In constraint logic programming, constraints are added incrementally. So we cannot eliminate a variable in *all* other equations at once, but rather consider the other equations one by one. A simple normal form can exhibit inconsistency: It suffices if the left-most variable of each equation is the only left-most occurrence of this variable. Therefore the two rules below implement a solver for linear equations over both floating point numbers (to approximate real numbers) and rational numbers. In the implementation, we write `eq` for equality on polynomials.

```
eliminate @ A1*X+P1 eq 0, A2*X+P2 eq 0 <=>
   normalize(P2-P1*A2/A1,P3),
   A1*X+P1 eq 0, P3 eq 0.

empty @ B eq 0 <=> number(B) | B=0.
```

The `eliminate` rule performs variable elimination. It takes two equations that start with the same variable. The first equation is left unaffected, it is used to eliminate the occurrence of the common variable in the second equation. The auxiliary built-in constraint `normalize` simplifies a polynomial arithmetic expression into a new polynomial. The `empty` rule says that if the polynomial contains no more variables, the constant `B` must be zero.

In the solver algorithm, no variable is made explicit, i.e. no pivoting is performed. Any two equations with the same first variable can react with each other. Therefore, the solver is highly concurrent and distributed.

The solver can be extended by a few rules to create explicit variable bindings, to make implicit equalities between variables explicit, to deal with inequations using slack variables or Fourier's algorithm [SaAb00].

Complexity. Informally, each application of the rule `eliminate` removes a single occurrence of a variable form one equation, and potentially introduces new variables smaller in the ranking.

More precisely, the ranking is defined as:

$$rank(P\ eq\ 0) = arank(P) + 1,$$

where $arank(E)$ is the maximum rank of a variable occuring in the arithmetic expression E. We require that $arank(X) \geq 1$ if X is a variable and $arank(e) = 0$ if e does not contain any variables (i.e. e is ground). We rely on the following order constraints:

$$a_1 * X_1 + P\ eq\ 0 \rightarrow arank(a_1{*}X_1{+}P) > arank(P)$$
$$normalize(E, P) \rightarrow arank(E) \geq arank(P)$$

The first order constraint is a consequence of the fact that the monomials in an equation are ordered by their variables. The second order constraints holds because the built-in constraint `normalize` does not introduce new variables, but may eliminate occurrences of some.

¿From the ranking we can see that goals are bounded as long as variables are not instantiated to expressions containing other variables (because that may change the *arank* of the arithmetic expression). In other words, *allowed instances* instantiate variables by ground expressions only.

Let there be v different variables in a given query with c equations. Then the worst-case derivation length is

$$D_{\Re} = O(cv),$$

because we can choose an *arank* function that ranks variables with all the integers from 1 to v.

For the complexity, we can assume that lhs matching and `normalize` take time linear in v, so we arrive at $O_{\Re}(cv * ((c + cv)^2(v + 0)\ +\ (v + 1)))$, which yields

$$O_{\Re}(c^3v^4).$$

Empirical Results. In Fig. 2, *eqtest(N)* generates N equations with N

variables and random integer coefficients between -99 and 99. *Worst* uses the notation $c * v$ for the worst-case derivation length. In the table, we show three exemplary test cases. In the first, the set of equations is dense, i.e. each coefficient is non-zero with probability 0.99. In the second test case, the probability of a non-zero cofficient is 0.5. The number of rule tries and applications is approximatly halved. The absolute run-time increases faster now. Test runs with probability 0.25 and 0.10 showed an analogous behavior. In the third test case, in each equation, only two randomly choosen variables have non-zero coefficients.

Goal	Worst	Apply	Try	Time
N=10,eqtest(N)	10*10	44	44	0.01
N=20,eqtest(N)	20*20	190	190	0.06
...	40*40	779	779	0.48
	80*80	3160	3160	3.56
N=10,eqtest(N)	10*5	25	25	0.01
N=20,eqtest(N)	20*10	98	98	0.04
...	40*20	416	416	0.45
	80*40	1579	1579	4.82
N=10,eqtest(N)	10*2	15	15	>0.0
N=20,eqtest(N)	20*2	38	38	0.02
...	40*2	83	83	0.09
	81*2	151	151	0.40

Fig. 2. Results from Test Runs with Linear Polynomial Equation Constraints

The table shows that

- The number of rule tries is identical to the number of rule applications, due to the simple structure of the rules (no guard in main rule).
- The number of rule tries is linear in the size of the problem, cv.
- For dense problems, the run-time is of complexity cv^2, because each rule application has to consider up to v variables.
- For sparse problems, the run-time is of complexity up to c^2v^2, because each rule application needs more time to find a partner constraint.

Summarizing, for the test cases we investigated, the worst observed time complexity was:

$$O_{\Re}^{obs}(c^2v^2)$$

The empirical results are better than the predicted ones, since the investigated CHR implementation uses various optimizations.

5.3 Description Logic

Description logics [PSR99] are used to represent the terminological knowledge of a particular problem domain on an abstract logical level. To describe this kind of knowledge, one starts with atomic concepts and roles, and then defines new concepts and their relationship in terms of existing concepts and roles. Concepts can be considered as unary relations similar to types. Roles correspond to binary relations over objects. In this paper, we use a natural language like syntax to help readers not familiar with the formalism.

Definition 5.3 *Concept terms* are defined inductively: Every *concept (name)* c is a concept term. If s and t are concept terms and r is a *role (name)*, then the following expressions are also concept terms:

s `and` t (conjunction), s `or` t (disjunction), `nota` s (complement),
`every` r `is` s (value restriction), `some` r `is` s (exists-in restriction).

Objects are constants or variables. Let a, b be objects. Then $a : s$ is a *membership assertion* and $(a, b) : r$ is a *role-filler assertion*. An *A-box* is a conjunction of membership and role-filler assertions.

Definition 5.4 A *terminology (T-box)* consists of a finite set of acyclic *concept definitions*

c `isa` s,

where c is a newly introduced concept name and s is a concept term.

The CHR constraint solver *Descr* for description logics is similar to the one in [FrHa95], except that here we represent both the A-box and the T-box as constraints of the query. The solver simplifies and propagates assertions in the A-box by using the definitions in the T-box and by making information more explicit and looks for obvious contradictions such as `X : device` and `X : nota device`. This is handled by the rule:

```
I : nota S, I : S  <=> false.
```

The unfolding rules replace concept names by their definitions.

```
I : C, C isa S  <=>  I : S, C isa S.
I : nota C, C isa S  <=>  I : nota S, C isa S.
```

The conjunction rule generates two new, smaller assertions:

```
I : S and T  <=>  I : S, I : T.
```

Disjunction is handled by lazy labeling search with the connective `lazy_or`, which is not directly expressible within simplification rules without case splitting [Abd00]. Hence we have to ignore the corresponding rule for the purpose of our analysis.

```
I : S or T  <=>  (I : S lazy_or I : T).
```

An exists-in restriction generates a new variable that serves as a witness

for the restriction:

```
I : some R is S  <=>  (I,J) : R, J : S.
```

A value restriction has to be propagated to all role fillers using a propagation rule:

```
I : every R is S, (I,J) : R  ==>  J : S.
```

Since propagation rules are not covered in this paper, we ignore the rule for the purpose of our complexity analysis.

The final simplification rules push the complement operator `nota` down to the leaves of a concept term:

```
I : nota nota S  <=>  I : S.
I : nota (S or T)  <=>  I : nota S and nota T.
I : nota (S and T)  <=>  I : nota S or nota T.
I : nota (every R is S)  <=>  I : some R is nota S.
I : nota (some R is S)  <=>  I : every R is nota S.
```

Complexity. For the complexity analysis, which only applies to simplification rules, we have to ignore the treatment of disjunction and of the value restriction. In effect, this means that we analyse an incomplete constraint solver. *Incompleteness* means the solver is correct (sound), but cannot detect unsatisfiability in all cases. This property is a common phenomenon with constraint solvers.

We rank constraints by the size of their concept terms:

$rank(I : s) = size(s)$
$rank(A) = 0$ otherwise

$size(nota\ s) = 2 * size(s)$
$size(some\ r\ is\ s) = 1 + size(s)$
$size(every\ r\ is\ s) = 1 + size(s)$
$size(c) = 1 + size(s)$ if (`c isa s`) exists
$size(f(t_1, \ldots, t_n)) = 1 + size(t_1) + \ldots + size(t_n)$ otherwise.

¿From the ranking we can see that queries are bounded if the ranks of all concept terms (like s and c) are known. Since concept terms are ground and finite by definition, their ranks can always be computed.

The derivation length D_{Descr} is bounded by the sum of the sizes of the concept terms occurring in a goal. Since the size of a concept depends on its definition, the syntactic size of the goal does not properly reflect the worst-case derivation length. Let the maximum size of a concept term be bounded by a constant k. For the incomplete constraint solver program, we have that

$$D_{Descr'} = ck.$$

Actually, disjunction and value restriction both give rise to exponential worst-case time complexity.

The complexity for the incomplete solver is $O_{Descr}(ck * ((c + ck)^2(1 + 0) +$

$(1 + 1)))$, i.e.

$$O_{Descr}(c^3k^3).$$

Empirical Results. In Fig. 3, *gen_dl* randomly generates a concept term T of a given depth, here in the table 20 or 25. Each kind of concept forming operator (*nota, and, or, every, some*) has the same probability. The worst-case derivation length (*Worst*) is the size A of the concept term T. The table is divided into three parts that correspond to samples from three test sets[2]. The first test set shows that the computation usually stops quickly. This is because concept terms that are part of a disjunction or value restriction are not further evaluated. Therefore, an outermost disjunction is always replaced by a conjunction in the second test set. Consequently, more computation takes place. In the third test set, in addition value restrictions are replaced by exists-in restrictions. Even though there is more computation, the observed complexity is still at worst linear in the size of the concept term.

The table shows that

- The number of rule tries is identical to the number of rule applications, due to the simple structure of the rules in combination with indexing.
- Time is roughly linear in the number of rule tries.
- The number of rule tries is in the worst case linear in the term size.

For this solver, the worst observed time complexity was just:

$$O^{obs}_{Descr}(ck)$$

The empirical results are better than the predicted ones due to indexing.

In another test set (see the online file `complexity.pl`), we studied the effect of indexing. When the variables of the clash rule are renamed apart and explicitely checked for equality in the guard, the number of rule tries increases considerably over the number of rule applications.

6 Conclusions

Based on the worst-case derivation length, as given by a ranking, we were able to give a general complexity meta-theorem for the worst-case time complexity of CHR constraint simplification rule programs. Rankings were originally used to prove termination. They map constraints and terms to integers such that the rank of the lhs of a rule is larger than the rank of the rhs of a rule. Once a ranking has been found, our meta-theorem allows for computing the complexity automatically from the program text. Our theorem also applies to naive implementations of CHR simplification rules.

This paper is a companion paper to [Fru02]. In comparison, we have generalized the notion of ranking function from natural numbers to integers and proven the corresponding meta-theorem for this extended case. We have also

[2] More test data can be found in the online file `complexity.pl`

Goal	Worst	Apply	Try	Time
gen_dl(1,T,20,A), I::T,(I,J)::r,(I,K)::r	14274	5	5	0.01
	14432	23	23	0.02
	23608	4	4	0.02
	38306	17	17	0.05
gen_dl(1,T,20,A), I::T,(I,J)::r,(I,K)::r	14274	5	5	0.01
	14432	222	222	0.05
	23608	4	4	0.02
	38306	669	669	0.14
gen_dl(1,T,25,A), I::T,(I,J)::r,(I,K)::r	96488	504	504	0.15
	107124	8	8	0.03
	233666	176	176	0.14
	379432	1924	1924	0.67
	640114	1893	1893	0.63
gen_dl(1,T,20,A), I::T,(I,J)::r,(I,K)::r	14274	894	894	0.10
	14432	1630	1630	0.19
	23608	2662	2662	0.32
	38306	5376	5376	0.63
gen_dl(1,T,25,A), I::T,(I,J)::r,(I,K)::r	96488	7458	7458	0.92
	107124	3955	3955	0.47
	233666	6890	6890	0.84
	379432	27226	27226	3.28
	640114	16057	16057	2.03

Fig. 3. Results from Test Runs with Description Logic Constraints

introduced the notion of allowed instances for bounded goals. These extensions were necessary in order to be able to analyse three non-trivial constraint solver programs: for finite interval domains employing arc consistency, for linear polynomials employing variable elimination and for description logic.

We have found that the dominating factor in the complexity are the rule application attempts (rule tries), not the actual rule applications. The cost of rule tries depends on the number of lhs CHR constraints n, the complexity of the guard checking and the ranking D of a given query. D was bounded by the product cr, where c is the number of atomic CHR constraints in the query and r is the maximum rank of an atomic CHR constraint in the query. Built-in

constraints only contribute if they have non-constant complexity. This is the case if non-scalar datatypes like arithmetic expressions are involved. In our examples, the derived complexities were of the form $c^{n+1}r^{n+1+k}$, where k is a small constant (often zero) introduced by the built-in constraints. In one case, r corresponded to v, the number of different variables in the problem.

We started to compare the complexities predicted by our theorem with the complexities observed in preliminary empirical tests. Due to optimizations like indexing on variables in the Sicstus Prolog CHR implementation, the observed complexities were much better than the predicted ones. At this stage of the research we cannot rule out with certainty that there are cases were the implementation actually shows the predicted worst-case complexity. So far we have not managed to construct such worst-case examples. Clearly more experiments are necessary.

We could only analyse an incomplete version of the description logic constraint solver, because our approach currently does not cover disjunction and propagation rules. The difficulty is that for propagation rules, the ranking approach for derivation lengths does not apply. The approach of Ganzinger and McAllester also does not apply, since it does not deal with free variables at run-time and arbitrary built-in constraints.

Further work should also take into account the effect of indexing and other optimizations in the complexity predictions. Another open question is which aspects in finding an appropriate ranking can be automated.

Acknowledgements
The author thanks the anonymous reviewers for their helpful comments. Most of this work was performed while working at the Computer Science Department of Ludwig-Maximilians-University, Munich, while visiting the School of Computer Science and Software Engineering at Monash University, Melbourne, in March 2000, and while visiting the Department of Computer Science at the University of Pisa in October 2001.

References

[AAI00] Applied Artificial Intelligence, Special Issue on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), 2000.

[AbFr99] S. Abdennadher and T. Frühwirth, Operational Equivalence of CHR Programs and Constraints, 5th Intl Conf on Principles and Practice of Constraint Programming (CP'99), Springer LNCS 1894, 1999.

[AFM99] S. Abdennadher, T. Frühwirth and H. Meuss, Confluence and Semantics of Constraint Simplification Rules, Constraints Journal Vol 4(2), Kluwer Academic Publishers, 1999.

[Abd00] S. Abdennadher, A Language for Experimenting with Declarative Paradigms, Second Workshop on Rule-Based Constraint Reasoning and

Programming, at the 6th Intl Conf on Principles and Practice of Constraint Programming (CP'2000), Singapore, 2000.

[Ben95] F. Benhamou, Interval Constraint Logic Programming, in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), Springer LNCS 910, 1995.

[FrAb02] T. Frühwirth and S. Abdennadher, Essentials of Constraint Programming, Springer Verlag, Heidelberg, Germany, 2002.

[FrHa95] T. Frühwirth and P. Hanschke, Terminological Reasoning with Constraint Handling Rules, in Principles and Practice of Constraint Programming, (P. van Hentenryck and V.J. Saraswat, Eds.), MIT Press, Cambridge, Mass., USA, 1995.

[Fru98] T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. J. Stuckey and K. Marriot, Eds.), Journal of Logic Programming Vol 37(1-3), Elsevier, 1998.

[Fru00a] T. Frühwirth, Proving Termination of Constraint Solver Programs, in New Trends in Constraints, (K.R. Apt, A.C. Kakas, E. Monfroy and F. Rossi, Eds.), Springer LNAI 1865, 2000.

[Fru00b] T. Frühwirth, On the Number of Rule Applications in Constraint Programs, Declarative Programming - Selected Papers from AGP 2000, (A. Dovier, M. C. Meo, A. Omicini, Eds.), Electronic Notes in Theoretical Computer Science (ENTCS), Vol 48, Elsevier Science Publishers, June 2001.

[Fru01] T. Frühwirth, As Time Goes by: Complexity Analysis of Simplification Rules, Workshop on Quantitative Aspects of Programming Laguages (QAPL'01), at the Conf on Principles, Logics, and Implementations of high-level programming languages (PLI'01), Firenze, Italy, September 2001.

[Fru02] T. Frühwirth, As Time Goes By: Automatic Complexity Analysis of Simplification Rules, 8th Intl Conf on Principles of Knowledge Representation and Reasoning (KR2002), Toulouse, France, April 2002.

[GaMc01] H. Ganzinger and D. McAllester, A New Meta-Complexity Theorem for Bottum-up Logic Programs, (R. Gore, A. Leitsch and T. Nipkow, Eds.) First Intl Joint Conf on Automated Reasoning IJCAR 2001, Springer LNAI 2083, 2001.

[GaMc02]
H. Ganzinger and D. McAllester, Logical Algorithms, 18th Intl Conference on Logic Programming ICLP'02, Copenhagen, Denmark, July/August, 2002.

[HoFr98] C. Holzbaur and T. Frühwirth, Constraint Handling Rules Reference Manual for Sicstus Prolog, TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria, July 1998.

[HoFr00] C. Holzbaur and T. Frühwirth, A Prolog Constraint Handling Rules Compiler and Runtime System, Applied Artificial Intelligence, Special Issue on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), 2000.

[Imb95] J.-L. J. Imbert, Linear Constraint Solving in CLP-Languages, (A. Podelski, Ed.), Constraint Programming: Basics and Trends, LNCS 910, March 1995.

[MaFr85] A. K. Mackworth and E. C. Freuder, The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, Artificial Intelligence Vol 25, 1985.

[MaSt98] K. Marriott and P. J. Stuckey, Programming with Constraints, MIT Press, USA, 1998.

[McA99] D. McAllester, On the Complexity Analysis of Static Analyses, (A. Cortesi and G. File, Eds.), 6th Intl. Static Analysis Symposium (SAS'99), Springer LNCS 1694, 1999.

[MoHe86] R. Mohr and T.C. Henderson, Arc and Path Consistency Revisited, Artificial Intelligence 28, 1986.

[PSR99] P. Patel-Schneider and M-C. Rousset, Eds., Special Issue on Description Logics, Journal of Logic and Computation, Volume 9, Number 3, Oxford University Press, June 1999.

[SaAb00] M. Saft and S. Abdennadher, WebCHR, Ludwig-Maximilians-Universität München, `www.pms.informatik.uni-muenchen.de/~webchr`, as of October 2002.

[vHDT92] P. Van Hentenryck, Y. Deville and C.-M. Teng, A generic arc-consistency algorithm and its specializations, Artificial Intelligence, 57(2-3):291-321, October 1992.

[vHSD95] P. van Hentenryck, V. A. Saraswat, and Y. Deville, Constraint Processing in cc(FD), Chapter in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), Springer LNCS 910, 1995.