Analysing the CHR Implementation of Union-Find

Tom Schrijvers* and Thom Frühwirth

[1] Department of Computer Science, K.U.Leuven, Belgium
www.cs.kuleuven.ac.be/~toms/
[2] Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. CHR (Constraint Handling Rules) is a committed-choice rule-based language that was originally intended for writing constraint solvers. Over time, CHR is used more and more as a general-purpose programming language. In companion paper [12] we show that it is possible to write the classic union-find algorithm and variants in CHR with best-known time complexity, which is believed impossible in Prolog. In this paper, using CHR analysis techniques, we study logical correctness and confluence of these programs. We observe the essential destructive update of the algorithm which makes it non-logical.

1 Introduction

When a new programming language is introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. For example, one often hears the argument that in Prolog some graph algorithms cannot be implemented with best known complexity because Prolog lacks destructive assignment that is needed for efficient update of the graph data structures. In particular, it is not clear if the union-find algorithm can be implemented with best-known complexity in pure (i.e. side-effect-free) Prolog [10].

We give a positive answer for the Constraint Handling Rule (CHR) programming language. There is an CHR implementation with the optimal worst case and amortized time complexity known for the classical union-find algorithm with path compression for find and union-by-rank. This is particularly remarkable, since originally, CHR was intended for implementing constraint solvers only.

CHR is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) into simpler ones until they are solved. In CHR, one distinguishes two main kinds of rules: Simplification rules replace constraints by simpler constraints while preserving logical equivalence, e.g. $\texttt{X}\geq\texttt{Y} \wedge \texttt{Y}\geq\texttt{X} \Leftrightarrow \texttt{X}=\texttt{Y}$. Propagation rules add new constraints, which are logically redundant, but may cause

* Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen). Part of this work was performed while visiting the University of Ulm in November 2004.

further simplification, e.g. $\mathtt{X} \geq \mathtt{Y} \wedge \mathtt{Y} \geq \mathtt{Z} \Rightarrow \mathtt{X} \geq \mathtt{Z}$. The combination of propagation and multi-set transformation of logical formulae in a rule-based language that is concurrent, guarded and constraint-based make CHR a rather unique and powerful declarative programming language.

Closest to our work is the presentation of a logical algorithm for the union-find problem in [10]. In a hypothetical bottom-up inference rule programming system with permanent deletions and rule priorities, a set of rules for union-find is given. The direct efficient implementation of these inference rule system seems not feasible. It is also not clear if the rules given in [10] describe the standard union-find algorithm as can be found in text books such as [4]. The authors remark that giving a rule set with optimal amortized complexity is complicated.

In contrast, we give an executable and efficient implementation that directly follows the pseudo-code presentations found in text books and that has also optimal amortized time complexity. Moreover, we do not need to rely on rule priorities. Here we analyse confluence and logical reading as well as logical correctness of our union-find program.

This paper is an revised extract of our technical report [13]. A programming pearl describing the implementation and giving a proof for the optimal time complexity is under submission [12]. This paper is structured as follows. In the next Section, we review the classical union-find algorithms. Constraint Handling Rules (CHR) are briefly introduced in Section 3. Then, in Section 4 we present the first basic implementation of the classical union-find algorithm in CHR. Relying on established analysis techniques for CHR, we investiagte the logical meaning of the program. The logical reading shows that there is an inherent destructive update in the union-find algorithm. In Section 6, the detailed confluence analysis helps to understand under which conditions the algorithm works as expected. It also shows in which way the results of the algorithm depend on the order of its operations. An improved version of the implementation, featuring path compression and union-by-rank, is presented and analysed next in Section 7. Finally, Section 8 concludes.

2 The Union-Find Algorithm

The classical union-find (also: disjoint set union) algorithm was introduced by Tarjan in the seventies [14]. A classic survey on the topic is [9]. The algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations on the sets:

- `make(X)`: create a new set with the single element `X`.
- `find(X)`: return the representative of the set in which `X` is contained.
- `union(X,Y)`: join the two sets that contain `X` and `Y`, respectively (possibly destroying the old sets and changing the representative).

In the naive algorithm, these three operations are implemented as follows.

- `make(X)`: generate a new tree with the only node `X`, i.e. `X` is the root.
- `find(X)`: follow the path from the node `X` to the root of the tree by repeatedly going to the parent node of the current node until the root is reached. Return the root as representative.
- `union(X,Y)`: find the representatives of `X` and `Y`, respectively. To join the two trees, it suffices to `link` them by making one root point to the other root.

The naive algorithm requires $\mathcal{O}(N)$ time per find (and union) in the worst case, where N is the number of elements (make operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve quasi-constant (i.e. almost constant) *amortized* running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root after a find. After `find(X)` returned the root of the tree, we make every node on the path from `X` to the root point directly to the root. The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth. If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one.

For each optimization alone and for using both of them together, the worst case time complexity for a single find or union operation is $\mathcal{O}(log(N))$. For a sequence of M operations on N elements, the worst complexity is $\mathcal{O}(M + Nlog(N))$. When both optimizations are used, the amortized complexity is quasi-linear, $\mathcal{O}(M + N\alpha(N))$, where $\alpha(N)$ is an inverse of the Ackermann function and is less than 5 for all practical N (see e.g. [4]).

The union-find algorithm has applications in graph theory (e.g. efficient computation of spanning trees). We can also view the sets as equivalence classes with the union operation as equivalence. When the union-find algorithm is extended to deal with nested terms to perform congruence closure, the algorithm can be used for term unification in theorem provers and in Prolog. The WAM [3], Prolog's traditional abstract machine, uses the basic version of union-find for variable aliasing. While *variable shunting*, a limited form of path compression, is used in some Prolog implementations [11], we do not know of any implementation of the optimized union-find that keeps track of ranks or other weights.

3 Constraint Handling Rules (CHR)

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) [6, 8, 5] and about termination and confluence analysis.

3.1 Syntax of CHR

We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols which are solved by a given

constraint solver, and CHR (user-defined) constraint symbols which are defined by the rules in a CHR program. There are three kinds of rules:

$$\begin{array}{ll} \textit{Simplification rule:} & \textit{Name} \ @ \ H \Leftrightarrow C \mid B, \\ \textit{Propagation rule:} & \textit{Name} \ @ \ H \Rightarrow C \mid B, \\ \textit{Simpagation rule:} & \textit{Name} \ @ \ H \setminus H' \Leftrightarrow C \mid B, \end{array}$$

where *Name* is an optional, unique identifier of a rule, the *head* H, H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal (query)* is a conjunction of built-in and CHR constraints. A trivial guard expression "`true |`" can be omitted from a rule. Simpagation rules abbreviate a simplification rules of the form $\textit{Name} \ @ \ H, H' \Leftrightarrow C \mid H, B$.

3.2 Operational Semantics of CHR

Given a query, the rules of the program are applied to exhaustion. A rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice (in contrast to Prolog). When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule, when a propagation rule is applied, the body of the rule is added to the goal without removing any constraints. When a simpagation rule is applied, all constraints to the right of the backslash are replaced by the body of the rule.

This high-level description of the operational semantics of CHR leaves two main sources of non-determinism: the order in which constraints of a query are processed and the order in which rules are applied. As in Prolog, almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program [3]. This behavior has been formalized in the so-called refined semantics that was also proven to be a concretization of the standard operational semantics [5].

In this refined semantics of actual implementations, a CHR constraint in a query can be understood as a procedure that goes efficiently through the rules of the program in the order they are written, and when it matches a head constraint of a rule, it will look for the other, *partner constraints* of the head in the *constraint store* and check the guard until an applicable rule is found. We consider such a constraint to be *active*. If the active constraint has not been removed after trying all rules, it will be put into the constraint store. Constraints from the store will be reconsidered (woken) if newly added built-in constraints constrain variables of the constraint, because then rules may become applicable since their guards are now implied. Obviously, ground constraints need never to be considered for waking.

[3] Nondeterminism due to wake-up order of delayed constraints and multiple matches for a rule are not relevant for our union-find programs [12].

3.3 Well-Behavedness: Termination and Confluence

For many existing CHR programs simple well-founded orderings are sufficient to prove termination [7]. Problems arise with non-trivial interactions between simplification and propagation rules.

Confluence of a CHR program guarantees that the result of a terminating computation for a given query is independent from the order in which rules are applied. This also implies that the order of constraints in a goal does not matter. The papers [1, 2] give a decidable, sufficient and necessary condition for confluence for terminating CHR programs. (It is also shown that confluent CHR programs have a consistent logical reading.) The condition can be readily implemented by an algorithm that is described informally in the following.

For checking confluence, one takes copies (with fresh variables) of two rules (not necessarily different) from a terminating CHR program. The heads of the rules are *overlapped* by equating at least one head constraint from each rule. For each overlap, we consider the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If the critical pair is not joinable, we have found a non-confluence. In any consistent state that contains the overlap of a non-joinable critical pair, the application of the two rules to the overlap will usually lead to different results.

4 Implementing Union-Find in CHR

The following CHR program in concrete ASCII syntax implements the operations and data structures of the basic union-find algorithm without optimizations.

ufd_basic

```
make      @ make(A) <=> root(A).

union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot  @ root(A) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
link      @ link(A,B), root(A), root(B) <=> B ~> A, root(A).
```

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the operations. `link/2` is an auxiliary relation for performing union. The constraints `root/2` and `~>/2` represent the tree data structure.

Remark. The use of the built-in constraint `=` in the rule `findRoot` is restricted to returning the element `A` in the parameter `X`, in particular no full unification is ever performed (that could rely on union-find itself).

Remark. The rule `link` can be interpreted as performing *abduction*. If the the nodes `A` and `B` are not equivalent, introduce the minimal assumption `B ~> A` so that they are equivalent (i.e. performing union afterwards leads to application of rule `linkEq`).

As usual in union-find, we will allow the following queries:

- An *allowed query* consists of `make/1`, `union/2` and `find/2` constraints only. We call these the *external* operations (constraints). The other constraints (including those for the data structure) are generated and used internally by the program only.
- The elements we use are constants. A new constant must be introduced exactly once with `make/1` before being subject to `union/2` and `find/2`.
- The arguments of all constraints are constants, with exception of the second argument of `find/2` that must be a variable that will be bound to a constant, and the second argument of `root/2`, that must be an integer.

5 Logical Properties

The logical reading of our `ufd_basic` union-find CHR program is as follows:

make	$make(A) \Leftrightarrow root(A)$
union	$union(A,B) \Leftrightarrow \exists XY(find(A,X) \wedge find(B,Y) \wedge link(X,Y))$
findNode	$find(A,X) \wedge A{\rightarrow}B \Leftrightarrow find(B,X) \wedge A{\rightarrow}B$
findRoot	$root(A) \wedge find(A,X) \Leftrightarrow root(A) \wedge X{=}A$
linkEq	$link(A,A) \Leftrightarrow true$
link	$link(A,B) \wedge root(A) \wedge root(B) \Leftrightarrow B{\rightarrow}A \wedge root(A)$

From the logical reading of the rule `link` it follows that $B{\rightarrow}A \wedge root(A) \Rightarrow root(B)$, i.e. *root* holds for every node in the tree, not only for root nodes. Indeed, we cannot adequately model the update from a root node to a non-root node in first order logic, since first order logic is monotonic, formulas that hold cannot cease to hold. In other words, the link step is where the union-find algorithm is *non-logical* since it requires an update which is destructive in order to make the algorithm efficient.

In the union-find algorithm, by definition of set operations, a union operator working on representatives of sets is an equivalence relation observing the usual axioms:

reflexivity	$union(A,A) \Leftrightarrow true$
symmetry	$union(A,B) \Leftrightarrow union(B,A)$
transitivity	$union(A,B) \wedge union(B,C) \Rightarrow union(A,C)$

To show that these axioms hold for the logical reading of the program, we can use the following obervations: Since the unary constraints `make` and `root`

must hold for any node in the logical reading, we can drop them. By the rule `findRoot`, the constraint `find` must be an equivalence. Hence its occurrences can be replaced by `=`. Now `union` is defined in terms of `link`, which is reflexive by rule `linkEq` and logically equivalent to `~>` by rule `link`. But `~>` must be syntactic equivalence like `find` because of rule `findNode`. Hence all binary constraints define syntactic equivalence. After renaming the constraints accordingly, we arrive at the following theory:

union	$A{=}B \Leftrightarrow \exists XY(A{=}X \wedge B{=}Y \wedge X{=}Y)$
findNode	$A{=}X \wedge A{=}B \Leftrightarrow B{=}X \wedge A{=}B$
findRoot	$A{=}X \Leftrightarrow X{=}A$
linkEq	$A{=}A \Leftrightarrow \mathit{true}$
link	$A{=}B \Leftrightarrow B{=}A$

It is easy to see that these formulas are logically equivalent to the axioms for equality, hence the program is logically correct.

6 Confluence

We have analysed confluence of the union-find implementation with a small confluence checker written in Prolog and CHR. For the union-find implementation `ufd_basic`, we have found 8 non-joinable critical pairs. Two non-joinable critical pairs stem from overlapping the rules for `find`. Four non-joinable critical pairs stem from overlapping the rules for `link`. The remaining two critical pairs are overlaps between `find` and `link`.

We found one non-joinable critical pair that is unavoidable (and inherent in the union-find algorithm), three critical pairs that feature incompatible tree constraints (that cannot occur when computing allowed queries), and four critical pairs that feature pending link constraints (that cannot occur for allowed queries in the standard left-to-right execution order). In the technical report [13] associated with this paper, we also add rules by completion and by hand to make the critical pairs joinable.

The Unavoidable Non-Joinable Critical Pair The non-joinable critical pair between the rule `findRoot` and `link` exhibits that the relative order of `find` and `link` operations matters.

Overlap	`find(B,A),root(B),root(C),link(C,B)`
`findRoot`	`root(C),B~>C,A=B`
`link`	`root(C),B~>C,A=C`

It is not surprising that a `find` after a `link` operation has a different outcome if linking updated the root. As remarked in Section 5, this update is unavoidable and inherent in the union-find algorithm.

Incompatible Tree Constraints Cannot Occur The two non-joinable critical pairs for `find` correspond to queries where a `find` operation is confronted with two tree constraints to which it could apply. Also the non-joinable critical pair involving the rule `linkEq` features incompatible tree constraints.

Overlap	A~>B,A~>D,find(A,C)
findNode	A~>B,A~>D,find(B,C)
findNode	A~>B,A~>D,find(D,C)
Overlap	root(A),A~>B,find(A,C)
findNode	root(A),A~>B,find(B,C)
findRoot	root(A),A~>B,A=C
Overlap	root(A),root(A),link(A,A)
linkEq	root(A),root(A)
link	root(A),A~>A

The conjunctions (`A~>B, A~>D`), (`root(A), A~>B`), (`root(A), A~>A`) and (`root(A), root(A)`) that can be found in the overlaps (and non-joinable critical pairs) correspond to the cases that violate the definition of a tree: a node with two parents, a root with a parent, a root node that is its own parent, and a tree with two identical roots, respectively. Clearly, these four conjunctions should never occur during a run of the program.

We show now that the four dangerous conjunctions indeed cannot occur as the result of running the program for an allowed query. We observe that the rule `make` is the only one that produces a `root`, and the rule `link` is the only one that produces a `~>`. The rule `link` needs `root(A)` and `root(B)` to produce `A ~> B`, and it will absorb `root(A)`.

In order to produce one of the first three dangerous conjunctions, the link operation(s) need duplicate `root` constraints (as in the fourth conjunction) to start from. But only a query containing identical copies of `make` (e.g. `make(A),make(A)`) can produce the fourth dangerous conjunction. Since duplicate make operations are not an allowed query, we cannot produce any of the dangerous conjunctions (and non-joinable critical pairs) for allowed queries.

Pending Links Cannot Occur The remaining four non-joinable critical pairs stem from overlapping the rule for `link` with itself. They correspond to queries where two `link` operations have at least one node in common such that when one link is performed, at least one node in the other link operation is not a root anymore. When we analyse these non-joinable critical pairs we see that the two conjunctions (`A~>C,link(A,B)`) and (`A~>C,link(B,A)`) are dangerous.

Overlap	root(A),root(B),link(B,A),link(A,B)
link	root(B),A~>B,link(A,B)
link	root(A),link(B,A),B~>A
Overlap	root(A),root(B),root(C),link(B,A),link(C,B)
link	root(C),A~>B,B~>C
link	root(A),root(C),link(B,A),B~>C
Overlap	root(A),root(B),root(C),link(B,A),link(A,C)
link	root(B),root(C),A~>B,link(A,C)
link	root(B),C~>A,A~>B
Overlap	root(A),root(B),root(C),link(B,A),link(C,A)
link	root(B),root(C),A~>B,link(C,A)
link	root(B),root(C),link(B,A),A~>C

Once again, we argue now that the critical pairs can never arise in practice in an allowed query. `link` is an internal operation, it can only be the result of a `union`, which is an external operation. In the union, the link constraint gets its arguments from `find`. In the standard left-to-right execution order of most sequential CHR implementations [5], first the two find constraints will be executed and when they have finished, the link constraint will be processed. In addition, no other operations will be performed inbetween these operations. Then the results from the find constraints will still be roots when the link constraint receives them. Note that such an execution order is always possible, provided `make` has been performed for the nodes that are subject to union (as is requiered for allowed queries).

7 Optimized Union-Find

The following CHR program implements the optimized classical Union-Find Algorithm, derived from the basic version by adding path compression for find and union-by-rank [14].

ufd_rank

```
make      @ make(A) <=> root(A,0).

union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B, find(A,X) <=> find(B,X), A ~> X.
findRoot  @ root(A,_) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
linkLeft  @ link(A,B), root(A,N), root(B,M) <=> N>=M |
              B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight @ link(B,A), root(A,N), root(B,M) <=> N>=M |
              B ~> A, N1 is max(N,M+1), root(A,N1).
```

When compared to the basic version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. The rule `findNode` has been extended for path compression already during the first pass along the path to the root of the tree. This is achieved by the help of the logical variable `X` that serves as a place holder for the result of the find operation. The `link` rule has been split into two rules `linkLeft` and `linkRight` to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is chosen (both rules are applicable) and the rank is incremented by one.

Remark. Path compression (cf. rule `findNode`) can be interpreted as *memoization* or *tabling* of all the (intermediate) results of the recursive find operation, where the memoized `find(A,X)` is stored as `A ~> X`.

The results for logical reading and logical correctness of the optimized union-find are analogous to the ones for `ufd_basic`.

Confluence Revisited The non-joinable critical pairs (CPs) are in principle analogous to the ones discussed for `ufd_basic` in Section 6, but their numbers significantly increases due to the optimizations of path compression and union-by-rank that complicate the rules for the find and link operations.

Our confluence checker found 73 non-joinable critical pairs. The number of critical pairs is dominated by those 68 of the link rules. Not surprisingly, each critical pair involving `linkLeft` has a corresponding analogous critical pair involving `linkRight`.

The CPs between `findRoot` and a link rule are the unavoidable critical pairs as in `ufd_basic`. These show the expected behavior that the result of `find` will differ if its executed before or after a link operation, for example:

Overlap	`find(B,A),root(B,C),link(E,B),root(E,D),D>=C`
`findRoot`	`A=B,D>=C,N is max(D,C+1),root(E,N),B~>E`
`linkLeft`	`A=E,D>=C,N is max(D,C+1),root(E,N),B~>E`

Two `findNode` rule applications on the same node will interact, because one will compress, and then the other cannot proceed until the first find operation has finished:

Overlap	`find(B,A),B~>C,find(B,D)`
`findNode`	`find(A,D),find(C,A),B~>D`
`findNode`	`find(D,A),find(C,D),B~>A`

We see that `A` and `D` are interchanged in the states of the critical pair. In the first state, since the result of `find(C,A)` is `A`, the `find(A,D)` can eventually only reduce to `A=D`. Analogously for the second state. But under `A=D` the two states of the critical pair are identical. The other two critical pairs involving a `findNode` rule correspond to impossible queries `B~>C,B~>D` and `root(B,N),B~>C` as discussed for the confluence of `ufd_basic`.

All critical pairs between link rules only, except those for `linkEq`, consist of pairs of states that have the same constraints and variables, but that differ in

the tree that is represented. Just as in the case of `ufd_basic` the problem of pending links occurs without a left-to-right execution order. For more details see [13].

8 Conclusion

We have analysed in this paper basic and optimal implementations of classical union-find algorithms. We have used and adapted established reasoning techniques for CHR to investigate the logical properties and confluence (rule application order independence). The logical reading and the confluence check showed the essential destructive update of the algorithm when trees are linked. Non-confluence can be caused by incompatible tree constraints (that cannot occur when computing with allowed queries), and due to competing link operations (that cannot occur with allowed queries in the standard left-to-right execution order).

Clearly, inspecting dozens of critical pairs is cumbersome and error-prone, so a refined notion of confluence should be developed that takes into account allowed queries and syntactical variations in the resulting answer.

At `http://www.cs.kuleuven.ac.be/~toms/Research/CHR/UnionFind/` all presented programs as well as related material are available for download. The programs can be run with the proper time complexity in the latest release of SWI-Prolog.

In future work we intend to investigate implementations for other variants of the union-find algorithm. For a parallel version of the union-find algorithm parallel operational semantics of CHR have to be investigated (confluence may be helpful here). A dynamic version of the algorithm, e.g. where unions can be undone, would presumably benefit from dynamic CHR constraints as defined in [15].

Acknowledgements. We would like to thank the participants of the first workshop on CHR for raising our interest in the subject. Marc Meister and the students of the constraint programming course at the University of Ulm in 2004 helped by implementing and discussing their versions of the union-find algorithm.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer, 1997.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2), 1999.
3. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

5. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, 2004.
6. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, October 1998.
7. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.
8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
9. Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
10. H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *International Joint Conference on Automated Reasoning*, LNCS 2083, pages 514–528. Springer, 2001.
11. D. Sahlin and M. Carlsson. Variable Shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.
12. T. Schrijvers and T. Fruehwirth. Optimal union-find in constraint handling rules. Technical report, November 2004.
13. T. Schrijvers and T. Fruehwirth. Union-find in chr. Technical Report CW389, Department of Computer Science, K.U.Leuven, Belgium, July 2004.
14. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.
15. A. Wolf. Adaptive constraint handling with chr in java. In *7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, LNCS 2239. Springer, 2001.