Source-to-Source Transformation for a Class of Expressive Rules

Thom Frühwirth and Christian Holzbaur

[1] Fakultät für Informatik
University of Ulm, Germany
Thom.Fruehwirth@informatik.uni-ulm.de
[2] Department of Medical Cybernetics and Artificial Intelligence
University of Vienna, Austria
christian@ai.univie.ac.at

Abstract. We argue that adding Source-to-Source transformation (STS) to Constraint Handling Rules (CHR) is easy, desirable and elegant. The central ideas are to represent CHR programs in relational form, and the utilization of CHR proper to perform the transformations, acting upon this representation. We illustrate the power and practicality of STS for CHR on three non-trivial examples. The first example shows the application of STS in a bootstrapping compiler for CHR. Then we extend the CHR language by probabilistic rule choice and Prolog-style clauses, respectively.

1 Introduction

Source-to-Source transformations (STS) [8] are an attractive component in the tool boxes that complement core implementations of computer languages. In STS, users will write STS programs to manipulate other programs during their compilation. As an example consider `CPP`, the pre-processor for `C`. Users happily apply STS to their advantage because it makes their programs more concise and reactive to conditionals during compilation.

STS is really attractive when the language used to encode the transformation is expressive, and applicable mechanically. `CPP` is certainly a useful tool, but it rather fails on the first count - expressiveness - if we compare it against macro processing facilities for (declarative) languages like Lisp, Scheme, Prolog and others, where the language for specifying the macro expansion is the same as the target language. In these cases, the user has a turing complete language at hand that he already knows.

The programming language Constraint Handling Rules (CHR) [2, 3] has accumulated credibility, yet it lacked STS. Which was unfortunate, considering the expressiveness of CHR. CHR is essentially a concurrent committed-choice language consisting of guarded rules that rewrite conjunctions of constraints into simpler ones until they are solved. CHR can define both simplification of and propagation over user-defined constraints. Simplification replaces constraints by simpler constraints. Propagation adds new constraints which may cause further

simplification. From a more general viewpoint, in the context of CHR, conjunctions of constraints can be regarded as interacting collections of concurrent agents or processes.

A preliminary short report on STS for CHR appeared in [7].

2 CHR by Example

In this section we introduce CHR by example. We define a user-defined constraint for less-than-or-equal, `=<`, that can handle variable arguments. The implementation will rely on syntactical equality, `=`, which is assumed to be a predefined (built-in) constraint.

```
reflexivity  @ X=<Y <=> X=Y | true.
antisymmetry @ X=<Y,Y=<X <=> X=Y.
transitivity @ X=<Y,Y=<Z ==> X=<Z.
```

The CHR specify how `=<` simplifies and propagates as a constraint. They implement reflexivity, antisymmetry and transitivity in a straightforward way. The rule `reflexivity` states that `X=<Y` is logically true, provided it is the case that `X=Y`. This test forms the (optional) guard of a rule, a precondition on the applicability of the rule. Hence, whenever we see the constraint `X=<X` we can simplify it to `true`. The rule `antisymmetry` means that if we find `X=<Y` as well as `Y=<X` in the current constraint, we can replace it by the logically equivalent `X=Y`. Note the different use of `X=Y` in the two rules: In the `reflexivity` rule the equality is a precondition (test) on the rule, while in the `antisymmetry` rule it is enforced when the rule fires.

The rules `reflexivity` and `antisymmetry` are *simplification CHR*. The rule `transitivity` propagates constraints. It states that the conjunction `X=<Y, Y=<Z` implies `X=<Z`. Operationally, we add logical consequences as a redundant constraint. This kind of CHR is called *propagation CHR*.

Redundancy from propagation CHR is useful, as the query `A=<B, C=<A, B=<C` shows: The first two constraints cause CHR `transitivity` to fire and add `C=<B` to the query. This new constraint together with `B=<C` matches the head of CHR `antisymmetry`, `X=<Y,Y=<X`. So the two constraints are replaced by `B=C`. In general, matching takes into account the syntactical equalities that are implied by built-in constraints. The equality is applied to the rest of the query, `A=<B,C=<A`, resulting in `A=<B,B=<A` where `B=C`. Therefore, since the built-in constraint `B=C` was added, CHR `antisymmetry` applies to the constraints `A=<B,C=<A`, resulting in `A=B`. The query contains no more inequalities, the simplification stops. The constraint solver we built has solved `A=<B,C=<A,B=<C` and produced the answer `A=B,B=C`.

3 Operational Semantics of CHR

For lack of space, we refer for detailed syntax and semantics to the paper [4]. Here we just discuss the core of the operational semantics of CHR programs, which is given by a state transition system.

Let P be a CHR program for the CHR constraints and CT be a constraint theory for the built-in constraints. We use abstract rule syntax in this section. The transition relation $\longmapsto$ for CHR is as follows (where upper case letters stand for conjunctions of constraints):

Simplify
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge D$
if $(H \Leftrightarrow G \mid B)$ in P and $CT \models \forall(D \rightarrow \exists\bar{x}(H = H' \wedge G))$

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

When we use a rule from the program, we will rename its variables using new symbols, and these variables form the sequence $\bar{x}$. A rule with lhs H and guard G is *applicable* to CHR constraints H' in the context of constraints D, when the condition holds that $CT \models D \rightarrow \exists\bar{x}(H = H' \wedge G)$. Any of the applicable rules can be applied, but it is a committed choice, it cannot be undone.

If a simplification rule $(H \Leftrightarrow G \mid B)$ 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 $(H \Rightarrow G \mid B)$ is applied to H', the **Propagate** transition adds B, $H = H'$ and G, but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints [**?**].

We now discuss in more detail the *rule applicability condition* $CT \models D \rightarrow \exists\bar{x}(H = H' \wedge G)$. The equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. Operationally, the rule applicability condition can be checked as follows: Given the built-in constraints of D, try to solve the built-in constraints $(H = H' \wedge G)$ without further constraining (touching) any variable in H' and D. This means that we first check that H' matches H and then check the guard G under this matching.

4 STS transformation for CHR

The key idea of STS for CHR is that CHR rules will be translated into relational normal form by introducing special CHR constraints for the components of a rule, which are head (lhs), guard (precondition), body (rhs) and compiler pragmas (directives). The STS component of CHR guarantees that all syntactical constituents of CHR programs can be mapped in both directions (from and to this relational form). The STS *transformer* is a constraint solver (handler) that acts on this representation. When a fixpoint is reached, the relational form is translated back into CHR rules and normal compilation continues.

The result of this approach is that STS programs are concise, compact and thus easy to inspect and analyze. Indeed, the complete STS program to implement the language extension of probabilistic CHR consists of a few rules that easily fit one page.

There is one problem to be solved: we need a means to keep the name spaces of object and transformation rules apart. Collections of CHR rules are currently already aggregated into so called (constraint) handlers. For STS, the CHR runtime system features a new builtin predicate to register handlers as transformers, their intended order of application, and options that give some additional control over the expansion, i.e. printing of intermediate results.

The constraints acted upon by transformation handlers encode CHR and associated meta information in relational form where rule identifiers connect the components of the rule that are `head/4, guard/2, body/2, pragma/2, constraint/1` (the notation specifies the name and number of arguments of each relation). For each CHR constraint symbol in the object program, there is a corresponding STS constraint `constraint`. Each of the remaining STS constraints `head, guard, body` and `pragma` starts with an identifier for the rule they come from. The second argument is the respective component of the rule. For the constraint `head`, the third argument is an identifier for the constraint matching the rule head, and the last argument indicates if the constraint is to be kept or removed. This information is necessary, because any type of CHR rule is represented in the same normalized, relational way.

5 Example: Bootstrapping the CHR compiler via STS

One major application domain of STS is the CHR compiler itself. The relational representation of CHR object programs combines very well with typical computational requirements during compilation since

1. compilation relies heavily on mappings (dictionaries)
2. CHR provide efficient mappings, including indexed lookup and iteration
3. operations on mappings are brief and to the point in CHR

Note that CHR compilation is non-local in the sense that we need to know in which rules the constraints occur, and what constraints appear together forming the left hand sides of the rules [6]. The task of computing these occurrences is expressed precisely by the three rules taken from the bootstrapped compiler:

```
crossref_each_head @
  head(R,Head1,Id1,T1) ==>
    functor(Head1,F1,A1),
    occ(F1/A1,head(R,Head1,Id1,T1),[]).

crossref_multi_headed_rule @
  head(R,Head1,Id1,T1),
    head(R,Head2,Id2,T2) ==>
    functor(Head1,F1,A1),
    occ(F1/A1,head(R,Head1,Id1,T1),[(Head2,Id2,T2)]).

crossref_combine @
```

```
occ(FA,Head,Ps1),
  occ(FA,Head,Ps2) <=>
    merge(Ps1,Ps2,Ps3),
    occ(FA,Head,Ps3).
```

Due to the STS rule `crossref_each_head`, every source code rule head gives rise to an occurrence constraint `occ/3`, that acts like an entry in the cross reference. Such an entry is identified by: functor name and arity of the head `F/A`, the head constraint itself and a list of partner constraints in a given source code rule `R`.

The second STS rule `crossref_multi_headed_rule` only applies to source code rules with more than one head constraint. It takes a pair of head constraints from the same rule and generates a occurrence constraint for one of them. The rule will also apply to the same pair of constraints in reversed order, so that the occurrence constraint will also be generated for the other head constraint.

With the third STS rule, the lists of partner constraints of two occurrences `occ/3` are merged if they occurrences refer to the same head constraint `Head`.

Note that all dictionary lookups in the computation of this cross-reference are implicit, as are the nested iterations over these dictionaries required to compute the crossproducts. One can think of the above program fragment as reactive system, concurrently and incrementally computing parts of the cross-reference crossproduct as `head/5` constraints arrive in the constraint store.

6 Example: Probabilistic CHR

In this section we extend the CHR language with randomness in rule applications using STS. In probabilistic CHR (PCHR) [5] randomness is expressed by *probabilistic rule choice*. Among the rules that are applicable, the committed choice of the rule is performed randomly by taking into account the relative probability associated with each rule.

The following PCHR program implements tossing a coin. We use concrete Prolog-style CHR syntax in the program examples. Syntactically, the probabilities (weights) are the argument of the `pragma` annotation that is used in normal CHR to give hints to the compiler. Here it will initiate source to source transformation.

```
toss(Coin) <=> Coin=head pragma 0.5.
toss(Coin) <=> Coin=tail pragma 0.5.
```

Each side of the coin has the same probability. This behavior is modelled by two rules that have the same probability to apply to a query `toss(Coin)`, either resulting in `Coin=head` or `Coin=tail`.

The example below shows how PCHR can be used to generate an n bit random number. The random number is represented as a list of `N` bits that are generated recursively and randomly one by one.

```
r1 @ rand(N,L) <=> N=:=0 | L=[].
r2 @ rand(N,L) <=> N>0   | L=[0|L1],
       rand(N-1,L1) pragma 0.5.
r3 @ rand(N,L) <=> N>0   | L=[1|L1],
       rand(N-1,L1) pragma 0.5.
```

As long as there are bits to generate, the next bit will either get value 0 or 1, both with same probability. When the remaining list length `N` is zero, a non-probabilistic simplification rule closes the list.

The three rules above will be represented as the following conjunction of constraints to which the STS program will be applied:

```
constraint(rand/2),

head(r1,rand(N,L),id1,remove),
guard(r1,N=:=0),
body(r1,L=[]),

head(r2,rand(N,L),id2,remove),
guard(r2,N>0),
body(r2,(L=[0|L1],rand(N-1,L1))),
pragma(r2,0.5),

head(r3,rand(N,L),id3,remove),
guard(r3,N>0),
body(r3,(L=[1|L1],rand(N-1,L1))),
pragma(r3,0.5).
```

Now we consider the STS program for PCHR which will be applied to the above example code in relational form. It simply states how the components of the rules should be translated in case the rule is probabilistic. The STS basically transforms the rules such that they generate a conflict set. Finally, we have to extend the run-time system with some rules for conflict resolution.

Conflict Set Generation Transformation

The *conflict set* is the set of all rules that are applicable at a particular computation step. While in normal CHR, any rule can be chosen and it is a committed choice, in probabilistic CHR we have to collect the unnormalized probabilities (weights) from all candidates in the conflict set and then randomly choose one rule according to their probabilities.

The two rules below define a generic standard transformation that makes the conflict set of the object rules explicit.

```
make_propagation @
pragma(R,N),
  head(R,H,I,remove),
```

```
    body(R,B) <=>
      pragma(R,N),
      head(R,H,I,keep),
      body(R,(remove(I),B)).

wrap_body @
pragma(R,N),
  body(R,B) <=>
    body(R,cand(N,B)).
```

The transformation rule `make_propagation` maps all rules into propagation rules (replacing `head(R,H,I,remove)` by `head(R,H,I,keep)`) that explicitly remove the head constraint(s) in the body of the rule using the standard CHR built-in `remove` (cf. `body(R,(remove(I),B))`). (The same effect could also be achieved using an auxiliary variable and without this standard CHR built-in, but it would be less efficient.) The operational behaviour of the transformed rule at this stage is the same, however the removal of head constraints has been made explicit.

The second transformation rule wraps the body of a rule with the run-time CHR constraint `cand`, whose first argument is the information from the pragma. This transformation changes the behavior of the rules, because their bodies will not be executed, but only collected at run-time. The collection of `cand` constraints forms the current conflict set of the computation.

Note that it is essential that the transformation rules are always applied to exhaustion and in textual order (in order of appearance).

Last but not least there is a final, third rule that adds a last object rule for each defined CHR constraint `C`:

```
ensure_collection @
constraint(C) ==>
  head(rx,C,I,keep),
  guard(rx,true),
  body(rx,collect(0,R)).
```

The resulting propagation rule just calls the CHR constraint `collect(0,R)` which triggers the conflict resolution. It has to be made sure that this rule is added at the end of the object program (see discussion below). Note that names for (generated) rules need not be unique. Here there will be a rule named `rx` for every type of head constraint.

For our example of random n-bit numbers, the application of the STS rules and the final translation back into rule syntax results in the following code (variable names have been generated automatically):

```
r1 @ rand(A,B)#C <=> A=:=0 | B=[].

r2 @ rand(A,B)#C ==> A>0 |
  cand(0.5,(remove(C),B=[0|D],rand(A-1,D))).
r3 @ rand(A,B)#C ==> A>0 |
```

```
  cand(0.5,(remove(C),B=[1|D],rand(A-1,D))).

rx @ rand(A,B)#C ==> collect(0,D).
```

The `#C` added to the rule heads is CHR syntax for accessing the identifier of the constraint that matched the head. Note that the first rule is left untranslated since it was not probabilistic.

Conflict Resolution

Conflict resolution chooses one rule (body) to apply from the conflict set of applicable rules. In our case, the probability normalisation and evaluation of the conflict set is achieved by the following rules that are defined in the STS program for PCHR and that are added to the transformed object program (where `cand/2` is replaced by `cand/4`):

```
collect(M,R),
  cand(N,B) <=>
    cand(R,M,M+N,B),
    collect(M+N,R).

collect(M,R) <=> random(0,M,R).

cand(R,M,MN,B) <=> R < M | true.
cand(R,M,MN,B) <=> R >= MN | true.
cand(R,M,MN,B) <=> M =< R, R < MN | call(B).
```

The constraint `collect(M,R)` takes a candidate rule body `cand(N,B)` and replaces it by `cand(R,M,M+N,B)` before continuing with `collect(M+N,R)`. The effect of this rule is that each candidate constraint is extended by the common variable `R` and by the interval `M` to `M+N`, where `N` is its unnormalized probability measure (weight).

Instead of explicitly normalizing the probabilities (weights), `collect` adds them up and finally calls `random(0,M,R)` to produce a random number in the interval from `0` to `M`. Note that this random number will be bound to the variable `R`.

The conjunction of extended candidate rule bodies acts as a concurrent collection of agents. As soon as they receive the random number through the variable (channel) `R`, they can proceed. If the value of `R` is outside of their range of weights `M` to `MN`, the candidate agent simply goes away. Otherwise, it is the randomly chosen candidate and it will call its original rule body `B`.

In this way, from the set of applicable rules, one of the rules is randomly chosen and applied. The probability distribution is according to the weights of the individual rules.

Discussion: Ordering generated rules

STS transformation in CHR is concurrent and incremental, i.e. transformation rules are applied while the components of the original rules arrive. In most cases, the order of the generated rules reflects the order of the original rules. While the order of rules does not matter in most CHR programs that solve constraints (they are confluent), the order still has an impact on efficiency. Moreover, CHR for e.g. the bootstrapping compiler or the probabilistic language extension are order sensitive in some parts.

In particular, in the example discussed here, `collect` has to be executed *after* all `cand/4` constraints have been generated. This can be achieved by relying on textual execution order of rules. But then the rules named `rx` must be the last ones in the generated code. We currently have not found a completly convincing means to elegantly ensure this order. There are many possibilities to put rules at the end of the generated code,

- sort generated rules according to rule name (ad hoc solution),
- test for the absence of the `pragma` constraints (ad hoc solution),
- introduce a dummy constraint `finally` at the end of the relational rule constraints (not elegant),
- put the last transformation rule into a separate transformer (clean but a bit tedious),
- use rule applications strategies (an overkill),

but more experiments are necessary to come up with the right way to do it. Hence this issue is a topic for future work.

7 Example: Clauses for CHR

In this example, we introduce Prolog-style clauses as new type of rules into CHR. A logical clause $H \leftarrow B$ is represented by the pragma-annotated rule `H <=> B pragma clause`. Operationally, the application of a clause rule is only speculative (don't know indeterminism), while normal CHR rules are committed-choice (don't care indeterminism). This means that the head `H` is unified with the current constraints, not matched, and that if the body B fails, the execution is simply undone by chronological backtracking. In contrast to Prolog, however, the head `H` may be a conjunction.

Consider the following program that computes paths in a graph *bottom-up*, which is not directly possible in Prolog due to the lack of multiple head atoms:

```
r1 @
path(X,Y,[X,Y]), edge(X,Y) <=> true pragma clause.
r2 @
path(X,Y,[X|P]), edge(X,Z) <=> path(Z,Y,P) pragma clause.
r3 @
path(X,Y,T) <=> fail.
```

The last rule encodes Prolog's closed world assumption. Note that it is a normal simplification CHR. Also note that rules are applied in textual order. Note that the computation will always terminate, since each rule application consumes one `edge` constraint or results in failure.

To the transformer (defined below) these rules are represented as a set of constraints:

```
head(r1,path(X,Y,[X,Y]),Id1,remove),   constraint(edge/2),
head(r1,edge(X,Y),      Id2,remove),   constraint(path/3),
head(r2,path(X,Y,[X|P]),Id3,remove),
head(r2,edge(X,Z),      Id4,remove),
head(r3,path(X,Y,T),    Id5,remove),

guard(r1,true),  body(r1,true),          pragma(r1,clause),
guard(r2,true),  body(r2,path(Z,Y,P)),   pragma(r2,clause),
guard(r3,true),  body(r3,fail)
```

The transformation maps all clause-annotated rules into propagation rules that *either* explicitly remove the head constraints and execute their body *or* just execute `true`, i.e. go away unnoticed. In the latter case, another rule will be tried.

For the implementation we have to rely on *disjunction*, as it is implemented by the operator `';'` in Prolog. The disjuncts are tried left-to-right by chronological backtracking. This construct is available in an extension of CHR, called CHR$^\vee$ [1], that is available in all Prolog-based CHR implementations.

The following rules concisely implement the complete transformation in an ultra-compact way:

```
make_heads @
pragma(R,clause), head(R,H,I,remove), body(R,B) <=>
       pragma(R,clause),
       same_functor(H,H1),
       head(R,H1,I,keep),
       body(R,(remove(I),H=H1,B)).

make_guard_body @
pragma(R,clause), guard(R,C), body(R,B) <=>
       guard(R,true),
       body(R,(C,B;true)).
```

The rule `make_heads` replaces each rule head `H` that was to be removed by a sceleton `H1` that is kept but explicitly removed by the CHR built-in `remove` in the body of the rule. The *sceleton* `H1` has the same function name and arity as `H`, but its arguments are fresh, pairwise different variables. The head and its sceleton are explicitly unified in the body of the rule. The effect of this change is to replace matching in the head by unification in the body. Note the similarity with the rule `make_propagation` that was used in probabilistic CHR.

After all heads are processed in this way, the second rule `make_guard_body` moves the guard into the body and introduces a *disjunction* with `true` in the body. The effect of this change is to replace guard checking by trying to assert the guard constraints or backtrack and do nothing.

The result of applying the transformation to our example is as follows:

```
r1 @ path(A,B,C)#D, edge(E,F)#G ==> (remove(D), remove(G),
                                     A=E, B=F, C=[A,B]
                                     ; true).
r2 @ path(A,B,C)#D, edge(E,F)#G ==> (remove(D), remove(G),
                                     A=E, C=[A|H],
                                     path(F,B,H)
                                     ; true).
r3 @ path(A,B,C)#D <=> fail.
```

In the code, the equalties between the head and its sceleton have been automatically simplified by the CHR compiler.

With the generated rules, the query `edge(a,b), edge(b,c), edge(c,d), edge(d,a), path(X,X,T)` will produce all cyclic paths in the given graph:

```
T = [d,a,b,c,d], X = d
T = [c,d,a,b,c], X = c
T = [b,c,d,a,b], X = b
T = [a,b,c,d,a], X = a
```

8 Conclusions

We motivated the incorporation of STS into CHR and proposed a way to do it. Based on the experience gathered, the mechanism seems adequate, concise and elegant. So far, we have implemented as language extensions a fair version of CHR, probabilistic CHR [5], linear logic CHR, and (multi-headed) clauses for CHR. The other major STS application is the bootrapped CHR compiler itself, where the transition from preprocessing (e.g. syntactic de-sugaring) to substantial compilation tasks is smooth and natural.

In summary, STS for CHR consists of the following steps

- CHR rules are translated into relational normal form.
- There are special CHR constraints for the components of a rule (head, guard, body, compiler pragmas).
- The STS program is just a regular CHR program/solver.
- The relational form resulting from applying the transformation is translated back into CHR rules.

All future (performance) improvements to the CHR system are immediately reflected in the transformation process. Apart from actual transformations, our STS also naturally provides basic support for program analysis like proving certain algebraic properties of CHR programs like symmetries, set semantics,

etc., which are important during the actual compilation [6]. We also expect that STS in CHR will have synergies with the Literate CHR system that is currently developed [9].

In the future, we have to investigate means to influence the order and scheduling of rules and constraints in the code generated from STS as discussed at the end of section 6. We plan a new release of CHR that features the bootstrapping compiler together with the STS capabilities described in this paper.

Another important issue is the correctness of the STS. Note that all the transformations we discussed turn a non-operational CHR program into an operational one. In this case, correctness means to check the result against a specification of the behavior of the desired language. We also plan to address correctness preserving STS.

Acknowledgements Thanks to Walter Guttmann and Marc Meister for critical comments on earlier versions of this paper. Part of this work was performed by the authors while at the Institut für Informatik at the Ludwig-Maximilians-University Munich, Germany.

References

1. S. Abdennadher and H. Schütz, CHR$^\vee$: A Flexible Query Language, International conference on Flexible Query Answering Systems, FQAS'98, Springer LNCS, Roskilde, Denmark, May 1998.
2. Constraint Handling Rules, Special Issue Journal of Applied Artificial Intelligence (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), April 2000.
3. Documents Mentioning Constraint Handling Rules, `www.google.com/search?q="constraint+handling+rules"+filetype:ps+OR+filetype:pdf`, 2003.
4. T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming, Vol 37(1-3), pp 95-138, October 1998.
5. T. Frühwirth, A. Di Pierro, and H. Wiklicky, Probabilistic Constraint Handling Rules, 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002), Selected Papers, Guest Editors: Marco Comini and Moreno Falaschi, Vol. 76 of Electronic Notes in Theoretical Computer Science (ENTCS), 2002.
6. C. Holzbaur, M.G. de la Banda, D. Jeffrey, P. J. Stuckey, Optimizing Compilation of Constraint Handling Rules, in Proceedings of the International Joint Conference on Logic Programming (ICLP'01), 2001.
7. C. Holzbaur, Source-to-Source Transformation for Constraint Handling Rules, Workshop on Functional and (Constraint) Logic Programming (WFLP 2002), University of Udine, Italy, June 2002.
8. D.B. Loveman, Program improvement by source-to-source transformation, Journal of the ACM, 24(1):121–145, 1977.
9. S. E. Torres, A Literate Programming System for Logic Programs with Constraints, Workshop on Functional and (Constraint) Logic Programming (WFLP 2002), University of Udine, Italy, June 2002.