

# Result-directed CHR Execution

Jon Sneyers

Dept. of Computer Science, K.U.Leuven, Belgium  
jon.sneyers@cs.kuleuven.be

**Abstract.** The traditional execution mode of CHR is bottom-up, that is, given a goal, the result is computed by exhaustively applying rules. This paper proposes a result-directed execution mode for CHR, to be used when both the goal and the result are known, and the task is to find all corresponding derivations. Result-directed execution is needed in the context of CHRiSM, a probabilistic extension of CHR in which goals typically have a large number of possible results. The performance of result-directed execution is greatly improved by adding early-fail rules.

## 1 Introduction

Traditionally, execution of Constraint Handling Rules [1, 2] works as follows. Starting from an initial state  $\sigma_i$  (also called the goal or query), execution consists of applying the rules of the program exhaustively according to some execution strategy. If the program terminates, some final state  $\sigma_f$  is reached in which no more rules are applicable (called the result or answer). The traditional execution mode is, in a sense, a blind bottom-up computation. Hence it is often desirable that programs are (observably) confluent [3], i.e., every goal has a *unique* result. More precisely, if a program is non-confluent w.r.t. the very nondeterministic theoretical operational semantics  $\omega_t$ , then it should at least be confluent w.r.t. the strategy class [4] for which it was written, e.g. the less nondeterministic refined strategy class [5].

CHRiSM [6] is a rule-based probabilistic programming language based on an implementation of CHR(PRISM); PRISM is a probabilistic extension of Prolog [7]. In particular, the current CHRiSM implementation is based on the K.U.Leuven CHR system in B-Prolog. In the context of CHRiSM, most programs are non-confluent because the result of a goal depends on probabilistic choices made during the computation. The traditional execution mode corresponds to sampling, that is, given a goal, one of many possible results is returned, with a probability that depends on the probabilities of the choices that were made.

Traditional execution works well for sampling CHRiSM programs. However, there are other probabilistic inference tasks. Given a goal  $G$  and a result  $R$ , one may want to know the probability that  $R$  is returned given  $G$ . Or one may want to find the most likely (Viterbi) explanation, that is, the most likely sequence of probabilistic choices that lead to  $R$ . Or one may want to perform a learning algorithm to estimate the probability distributions of the choices in order to maximize the likelihood of given observations. For all these tasks, the basic challenge is to find all explanations of a result given a goal, that is, all computation paths from  $G$  to  $R$ .

PRISM is designed to find explanations for Prolog goals. It does this essentially by making the probabilistic choices backtrackable (during sampling they are committed-choice) and using a failure driven loop to find all explanations. CHRiSM computations can be wrapped in Prolog as follows:

```
computation(Goal,Result) :-  
    metacall(Goal),  
    get_chr_store(Store),  
    compare_multiset(Result,Store).
```

This mechanism works, but it is essentially a generate-and-test approach, which uses traditional execution to produce *all possible* results for `Goal`. This is horribly inefficient. We need a *result-directed* execution mode.

## 2 Result-directed Execution

When executing CHR in a result-directed way, the aim is not to compute the result  $R$  for a given goal  $G$ , but rather to find derivations from  $G$  to  $R$ , given both  $G$  and  $R$ . Using the notation and terminology of CHRiSM, we are looking for an *explanation* for the (full) *observation*  $G \Leftarrow R$ . Sometimes only a *partial* observation  $G \Rightarrow P$  is given, which means that the result is only partially known, that is,  $P$  is a (multiset) subset of the actual result.

Result-directed execution is needed in CHRiSM to find all explanations of an observation, which is a necessary step for probability computation and for parameter learning. However, it is also a useful execution mode for CHR in general, especially in variants of CHR that involve search, like  $\text{CHR}^\vee$  [8].

**Approach.** The approach is as follows. We start from the naive generate-and-test approach described in the introduction: compute all results for  $G$ , and after each computation, compare the final CHR store with the desired result  $R$ , failing if there is a difference. In order to make this more efficient, we now try to do the failing as soon as possible, pruning away redundant computations. We use a source-to-source transformation of the CHR program to achieve this pruning. Three new constraints are introduced: `result/1`, `observation/1`, and `cleanup/0`. We adapt the wrapper predicate from the introduction as follows:

```
computation(Goal,Result,S) :-
    metacall((observation(S),result(Result),Goal,cleanup)),
    get_chr_store(Store),
    compare_multiset(Result,Store).
```

The argument `S` indicates the status of the observation: `full` or `partial`. If the observation is `full`, then `Result` encodes the entire final store; if the observation is `partial`, then `Result` is some subset of the final store.

We add the following rules to (the bottom of) the original program:

```
result((A,B)) <=> result(A), result(B).
cleanup \ observation(_) <=> true.
cleanup \ result(_) <=> true.
cleanup <=> true.
```

The first rule recursively splits the result conjunction in its conjuncts; the `cleanup` rules make sure that none of these new constraints are visible in the actual result. Note that we use the refined operational semantics [5].

Now we add a number of *early-fail rules* to the top of the original program.

## 3 Early-fail Rules

Early-fail rules detect situations in which it has become impossible to reach the desired result. The redundant further computation is pruned by failing immediately.

### 3.1 Never-removed Constraints

Some programs have *never-removed* constraints. These constraints do not occur in the removed part of rule heads. As a consequence, once a never-removed constraint is added, it will remain in the store until the final result is reached.

If we know that a constraint  $c$  is never-removed (and ground), we can add an early-fail rule of the following form:

```
observation(full), c( $\bar{X}$ ) ==> check(c( $\bar{X}$ )).
result(c( $\bar{X}$ )) \ check(c( $\bar{X}$ )) <=> true.
check(c( $\bar{X}$ )) <=> fail.
```

We introduce a new constraint, `check/1`, which searches for a matching `result/1` constraint and fails if no match is found. We use the refined semantics for this, in a similar way as in  $\text{CHR}^1$  [9].

For efficiency reasons we do some flattening, as described in [10]:

```
result(c( $\bar{X}$ )) <=> result_c( $\bar{X}$ ).
cleanup \ result_c( $\square$ ) <=> true.
observation(full), c( $\bar{X}$ ) ==> check_c( $\bar{X}$ ).
result_c( $\bar{X}$ ) \ check_c( $\bar{X}$ ) <=> true.
check_c( $\bar{X}$ ) <=> fail.
```

Detecting never-removed constraints is straightforward; however, note that the above early-fail rules are only sound if the full result is known.

### 3.2 Partial Observations

If we only know part of the result, the above approach cannot be used. However, if we know that a never-removed constraint  $c$  has a *functional dependency* [11] between its arguments, we can add another kind of early-fail rules. We say a constraint  $c/n$  exhibits a functional dependency  $K \rightsquigarrow V$  (where  $K$  and  $V$  partition the set of  $n$  argument positions) if the *key* arguments  $K$  uniquely determine the arguments  $V$ . E.g., if a program starts with the rule

$$c(\bar{X}, \bar{Y}, \square), c(\bar{X}, \bar{Z}, \square) ==> \bar{Y} = \bar{Z}.$$

then the constraint  $c(\bar{X}, \bar{Y}, \square)$  has a functional dependency  $\bar{X} \rightsquigarrow \bar{Y}$ . In [11], a method is described to detect functional dependencies.

Now if we have a never-removed constraint  $c(\bar{X}, \bar{Y}, \square)$  with a functional dependency  $\bar{X} \rightsquigarrow \bar{Y}$ , we can add early-fail rules of the following form:

$$c(\bar{X}, \bar{Y}, \square), \text{result\_c}(\bar{X}, \bar{Z}, \square) ==> \bar{Y} \neq \bar{Z} \mid \text{fail}.$$

If a new  $c/n$  constraint is added, and the (partial) result contains a constraint with the same key, then we compare the functionally dependent arguments from both constraints, failing if they are different.

### 3.3 Surviving Constraints

Some constraints are not quite never-removed, but still behave in a sufficiently monotonous way in order to be used in an early-fail rule. Consider for example:

$$\text{min}(Y) \setminus \text{min}(X) <=> X \geq Y \mid \text{true}.$$

In this rule, `min(X)` is removed, but only if there is a `min(Y)` with a smaller argument. So if a `min(X)` constraint is added, it should still be in the result, or else there has to be some constraint `min(Y)` in the result, with  $X \geq Y$ . Hence we can add the following early-fail rule:

```

observation(full), min(X) ==> check(min(X)).
result(min(Y)) \ check(min(X)) <=> X >= Y | true.
check(min(_)) <=> fail.

```

Since `min/1` has a functional dependency from  $\emptyset$  to its only argument (i.e. it is a singleton constraint), we can also write an early-fail rule for partial observations:

```

result(min(Y)), min(X) ==> X >= Y.

```

In general we say `c/n` is a *surviving* constraint if all of its removed occurrences satisfy the following property: either the rule also has a kept occurrence of `c/n`, or the rule body unconditionally (but possibly indirectly) re-inserts `c/n`.

Another example is the rule “`sum(X,A), sum(X,B) <=> C is A+B, sum(X,C)`”. In this case, we can check that once we add a `sum/2` constraint, the result must also contain a `sum/2` constraint with the same first argument:

```

observation(full), sum(X,A) ==> check(sum(X,A)).
result(sum(X,_)) \ check(sum(X,_)) <=> true.
check(sum(_,_)) <=> fail.

```

If we also know that the type of the second argument of `sum/2` is restricted to non-negative numbers, then we can infer that  $C \geq A$  and  $C \geq B$  and thus we can add this as a guard: “`result(sum(X,C)) \ check(sum(X,A)) <=> C >= A | true`”.

### 3.4 User-defined Early-fail Rules

The above kinds of early-fail rules can be added automatically by static program analysis. For a specific program, the programmer can also write additional user-defined early-fail rules. Those rules can take particular invariants into account that are hard to detect automatically, or they can be specific to the intended use of the program (possibly unsound if the program is used in unintended ways).

Normal (non-result-directed) program execution is not affected if the early-fail rules only apply when `observation/1` and `result/1` constraints are given. It is the responsibility of the programmer to write sound early-fail rules, i.e. no computations are pruned that could actually lead to the given result.

## 4 Experimental Evaluation

In order to evaluate the effect of adding early-fail rules, we consider the music generation program from the APOPCALEAPS system [12]. The benchmark consists of computing the probability of a simple piece of music of  $n$  measures of chords and drums. Still, the number of possible pieces is exponential in  $n$ .

Most of the output constraints of the APOPCALEAPS program are never-removed, so early-fail rules like the ones in Section 3.1 can be added. However, the constraint `beat/5`, which encodes rhythm, is not never-removed. The following probabilistic rule<sup>1</sup> removes `beat/5` constraints:

```

split_beat(V) ??
  meter(_,OD), phase(split_beats(M)), shortest_duration(V,SD),
  \ beat(V,M,N,X,D), next_beat(V,M,N,X,NM,NN,NX) <=> D<SD |
    D2 is D*2, X2 is X+1/(D2/OD),
    next_beat(V,M,N,X,M,N,X2), next_beat(V,M,N,X2,NM,NN,NX),
    beat(V,M,N,X,D2), beat(V,M,N,X2,D2).

```

<sup>1</sup> In case you are not familiar with CHRiSM syntax: a probabilistic rule is just a regular CHR rule that is optionally *not* applied; in this case, the probability of rule application depends on the stochastic experiment `split_beat(V)`.

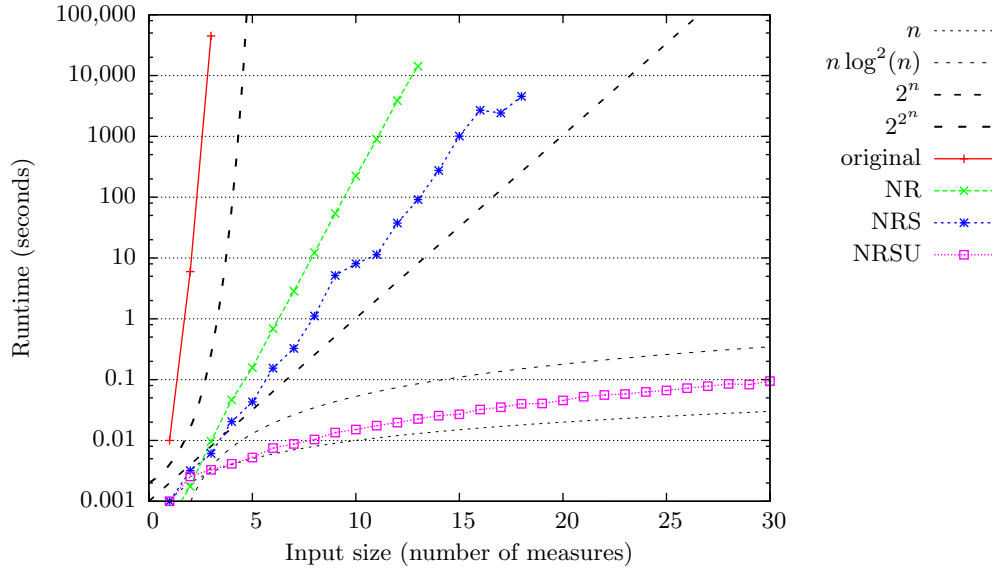


Fig. 1. Benchmark results.

This rule immediately reinserts a new `beat/5` constraint (even two), so `beat/5` is surviving (cf. Section 3.3). We can add the following early-fail rule: (the guard  $Y \geq X$  can be added because the last argument of `beat/5` always increases)

```
observation(full), beat(A,B,C,D,N) ==> check(A,B,C,D,N).
result_beat(A,B,C,D,Y) \ check(A,B,C,D,X) <=> Y >= X | true.
check(_,_,_,_,_) <=> fail.
```

We can do better if we know more about the way the APOPCALEAPS program works. In particular, the `beat-splitting` rule splits beats of a given measure `M` when the program is in a phase denoted by `phase(split_beats(M))`. Once the next phase is entered, beats from previous measures become never-removed. This justifies the following user-defined early-fail rule:

```
observation(full), phase(split_beats(Ms)), beat(V,M,N,X,D)
==> M < Ms | check_NR_beat(V,M,N,X,D).
result_beat(V,M,N,X,D) \ check_NR_beat(V,M,N,X,D) <=> true.
check_NR_beat(V,M,N,X,D) <=> fail.
```

Figure 1 shows the benchmark results for several variants of the program, with increasingly sophisticated early-fail rules. All runtimes are averages over at least 5 random problem instances. The following variants are considered:

**original:** The original program, without any early-fail rules.

**NR:** The program with early-fail rules for all never-removed constraints

**NRS:** NR + an early-fail rule for the surviving constraint `beat/5`

**NRSU:** NRS + a user-defined early-fail rule

In this benchmark, the time complexity is reduced from exponential to almost linear. This is not surprising, since in this case there is only one explanation for every result, and if all never-fail rules are added, only a linear number of failing branches is investigated. Without never-fail rules, the entire tree is investigated, and it has an exponential number of leaves.

## 5 Conclusion

We have introduced an alternative execution mode for CHR, in which the aim is not to compute results for a goal, but instead to find derivations given both the goal and (part of) the result. The notion of surviving constraints (of which never-removed constraints are a special case) can be used to add early-fail rules which significantly improve the performance during result-directed execution.

**Related Work.** Pruning a search space by early detection of failure is obviously not a novel idea, and it has been applied in many contexts. A similar general idea underlies for example A\* search [13], NOGOOD assertions in expert systems [14], search control for planning [15], and clause learning in SAT solvers [16]. Many of these existing approaches can undoubtedly be an inspiration in the context of result-directed CHR execution. However, there are important differences as well. For instance, clause learning is a dynamic approach and the result (a satisfying assignment) is not known in advance, while we derive early-fail rules statically and the result is given in advance.

Result-directed execution should not be confused with backward chaining as in Prolog [17] (and  $\text{CHR}^\vee$  [8]) or the goal-directed reasoning of ECLIPS [18]. In ECLIPS, goals are a control mechanism; they can be described in CHR as syntactic sugar for phase constraints implemented using the refined operational semantics.

In PRISM, efficient explanation search is greatly helped by tabling the probabilistic predicates [19]. In the context of CHR and CHRiSM, tabling is problematic since rules implicitly take the entire execution state as input.

**Future Work.** An implementation that automatically detects never-removed and surviving constraints and adds the corresponding early-fail rules has still to be made.

The notions of never-removed and surviving constraints can be extended as in Section 4, to a conditional form: a constraint can be never-removed or surviving *under certain circumstances*. These circumstances can be properties of the current computation (as in Section 4) or properties of the goal and result; in any case, it is straightforward to come up with corresponding early-fail rules.

More complicated kinds of early-fail rules could be defined. In fact, every execution state property that behaves monotonically (e.g. a level mapping derived in a termination proof as in [20], even if they have no lower bound) could be converted to an early-fail rule. Also, a different approach for result-directed execution can be imagined, in which rules are applied in reverse, starting from the result and working towards the goal (perhaps guided by reverse early-fail rules).

For partial observations, the notion of early-*succeed* rules makes sense. That is, if we are in a state  $\sigma$  in which the partial result has been obtained, and we can somehow show that all further execution paths will succeed and not remove the partial result (e.g. if the partial result consists only of never-removed constraints and the program has no failing derivations), then we do not need to investigate the subtree rooted at  $\sigma$ .

The methods described here could be adapted to solve the related problem of determining if a derivation from  $G$  to  $R$  exists under a given (large) strategy class. For example, we may be interested in knowing whether there is a computation  $G \mapsto^* R$  under the theoretical semantics  $\omega_t$ . If the program is confluent, it suffices to execute it with goal  $G$  under any semantics that instantiates  $\omega_t$ , and compare the result with  $R$ . However, if the program is non-confluent, that does not work. Instead one could make a general implementation of the theoretical semantics  $\omega_t$ , which at each step computes all applicable transitions and chooses (in a backtrackable way) one transition to perform. Such a general implementation could generate all

possible computations starting from  $G$ , but if the program is highly non-confluent, the majority of those computations do not lead to  $R$  and can be pruned using early-fail rules (which should get priority over all original rules).

## References

1. Frühwirth, T.: *Constraint Handling Rules*. Cambridge University Press (2009)
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming* **10**(1) (January 2010)
3. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In Dahl, V., Niemelä, I., eds.: *ICLP '07*. Volume 4670 of LNCS, Porto, Portugal, Springer (September 2007) 224–239
4. Sneyers, J., Frühwirth, T.: Generalized CHR machines. [21] 143–157
5. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: *ICLP '04*. LNCS, vol. 3132, Saint-Malo, France, Springer (2004) 90–104
6. Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., Sato, T.: CHR(PRISM)-based probabilistic logic learning. In Hermenegildo, M., Niemelä, I., Schaub, T., eds.: *26th International Conference on Logic Programming*, Edinburgh, UK (July 2010)
7. Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems* **31** (2008)
8. Abdennadher, S., Schütz, H.:  $\text{CHR}^\vee$ , a flexible query language. In Andreassen, T., Christiansen, H., Larsen, H., eds.: *FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems*. Volume 1495 of LNAI, Roskilde, Denmark, Springer (1998) 1–14
9. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with negation as absence. In Schrijvers, T., Frühwirth, T., eds.: *CHR '06*. K.U.Leuven, Dept. Comp. Sc., Technical report CW 452, Venice, Italy (July 2006) 125–140
10. Sarna-Starosta, B., Schrijvers, T.: Transformation-based indexing techniques for Constraint Handling Rules. [21] 3–18
11. Duck, G.J., Schrijvers, T.: Accurate functional dependency analysis for Constraint Handling Rules. In Schrijvers, T., Frühwirth, T., eds.: *CHR '05*. K.U.Leuven, Dept. Comp. Sc., Technical report CW 421, Sitges, Spain (2005) 109–124
12. Sneyers, J., De Schreye, D.: APOPCALEAPS: Automatic music generation with CHRiSM. In Downie, J., Veltkamp, R., eds.: *11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, Utrecht, The Netherlands (August 2010) Submitted.
13. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* **4**(2) (1968) 100–107
14. Stallman, R.M., Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* **9**(2) (1977) 135–196
15. Bacchus, F., Kabanza, F.: Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* **116**(1-2) (2000) 123 – 191
16. Zhang, H.: SATO: An efficient propositional prover. In: *14th International Conference on Automated Deduction*. Volume 1249 of LNCS, Springer (1997) 272–275
17. Clocksin, W.F., Mellish, C.S.: *Programming in Prolog*. Springer-Verlag (1984)
18. Homeier, P.V., Le, T.C.: ECLIPS: An extended CLIPS for backward chaining and goal-directed reasoning. In: *Proc. 2nd CLIPS Users Group Conference*. Volume 2 of NASA Conference Publication 10085, Houston, Texas, USA (1991) 213–225
19. Zhou, N.F., Sato, T., Shen, Y.D.: Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming* **8**(1) (2008) 81–109
20. Pillozzi, P., Schreye, D.D.: Automating termination proofs for CHR. In Hill, P.M., Warren, D.S., eds.: *25th International Conference on Logic Programming*. Volume 5649 of *Lecture Notes in Computer Science*, Pasadena, CA, USA, Springer (2009) 504–508
21. Schrijvers, T., Frühwirth, T., Raiser, F., eds.: *CHR '08: Proc. 5th Workshop on Constraint Handling Rules*, Hagenberg, Austria (July 2008)