Angelic CHR

Thierry Martinez

EPI Contraintes, INRIA Paris-Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France.
thierry.martinez@inria.fr

Abstract. Implementations of CHR follow a committed-choice forward-chaining execution model: the non-determinism of the abstract semantics is partly refined with extra-logical syntactic convention on the program order and possibly notations for weighted semantics (with priorities or probabilities), and partly left unspecified in the underlying compiler. This paper proposes an alternative execution model which explores all the possible choices, by opposition to the committed-choice strategy. This execution model is angelic in the sense that if there exists a successful execution strategy (with respect to a given observable), then this strategy will be found. Formally, the set of computed goals is complete with respect to the set of the logical consequences of the interpretation of the initial goal in linear logic. In practice, this paper introduces a new data representation for sets of goals, the derivation nets. Sharing strategies between computation paths can be defined for derivation nets to make execution algorithmically tracktable in some cases where a naive exploration would be exponential. Control for refined execution is recovered with the introduction of user constraints to encode sequencing, fully captured in the linear-logic interpretation. As a consequence of angelic execution, CHR rules become decomposable while preserving accessibility properties. This decomposability makes natural the definition in angelic CHR of meta-interpreters to change the execution strategy. More generally, arbitrary computation can be interleaved during head matching, for custom user constraint indexation and deep guard definition.

1 Introduction

Since the introduction of Prolog, logic programming has been living with the dichotomy: “programs = logic + control”. As every logic-based language, the declarativity of Constraint Handling Rules [1,2] relies on the logical interpretation of the programs. However, this interpretation hides syntactic conventions, like the order of the rules, distinguished abbreviations such as propagation rules, and annotations which control the effective execution of the program. These control features are formally described in a hierarchy of semantics, from the abstract semantics ω_{va} [1] to more fine-grained semantics, describing the handling of propagations (the theoretical semantics ω_{t}), of the rule ordering (the refined semantics ω_{r} [3]), or of annotations like priorities [4] or probabilities [5].

CHR enjoys two logical interpretation: the first to have been introduced historically interprets rules and goals as first-order classical logic formulae; more recently [6], an interpretation as first-order linear logic [7] formulae has been given. The latter provides a finer reading of the dynamics of the rules and will be the logical interpretation considered in this paper.

All these semantics are correct with respect to the linear-logic interpretation: if a configuration is reachable through any of these operational semantics, then this configuration is indeed a linear-logic consequence of the initial goal. However, only the abstract semantics enjoys completeness: the purpose of all other semantics is to provide syntactic construction to force the execution to choose some particular branches. The downside is that these scheduling choices escape the declarative framework provided by logic. The programmer should ensure that the scheduler can only make the good choice, either by writing a confluent program or by relying on extra-logical traits (order of the rules, priorities, etc.) to drive the scheduler.

Focusing on completeness entails the exploration of all the logical consequences of the interpretation of the initial goal in linear logic. For this purpose, we propose angelic scheduling as an alternative execution model for CHR. Observationaly, the scheduler always makes the good choice: more precisely, if a successful (i.e., non-blocking) choice exists, it will be explored. Accessible configurations exactly match the set of logical consequences of the linear interpretation of the initial goal: the operational behavior is fully described by the linear-logic interpretation, including the control. More formally, linear logic is the most faithful logic for CHR [6], since it captures the non-monotonous evolution of configurations. Control structures like sequencing and branching have natural encoding in this logic and their usage for CHR have already been showed through the log-linear encoding of RAM machines [8].

In angelic settings, the atomicity of head consumption is not essential, in opposition to the committed-choice case. Since absence of user constraints cannot be observed, partial head consumptions just lead to silent unsuccessful computation branches. This property allows the interleaving of arbitrary computations between multiple head consumptions. Meta-interpreters for CHR rules can therefore be written by sequencing the consumption of the successive parts of the head. Specific representations can be chosen for some heads to enable user-defined indexation strategy. To reduce the combinatorial explosion among computation branches, the formalism of derivation nets is introduced: this formalism provides a graphical representation for sets of computation paths. Non-determinism during the execution of a CHR program can be intrinsic to the rule dynamics, and all choices should be explored, but the abstract operational semantics suffers from a large part of scheduling non-determinism between independent paths of the computation that should be quotiented for a tractable execution. The derivation nets are a convenient representation to define sharing strategies between computation paths to eliminate scheduling non-determinism. Two decidable sharing strategies are explored in this paper. The first strategy shows that optimal sharing is decidable but is computationaly expensive. The second one is polynomial in the worst case and induces essentially a constant overhead in practice while being optimal relatively to a conservative interpretation of user constraint identity.

In the following section, angelic semantics is formally defined through derivation nets, and sharing strategies are presented. In Section 3, the specificity of angelic programming is formally explored through the decomposition property of head consumption and control mechanisms. In Section 4, concrete usage of angelism are given for meta-interpreter implementation, custom indexing definition and deep guards in CHR.

Note that this application of angelic semantics to CHR is only a preliminary work. A prototype implementation of angelic semantics along the lines presented here has been developped for the LCC language[1] (Linear-logic Concurrent Constraint), and despite showing good asymptotical behavior, large implementation work still has to be done to optimize execution time and memory usage. Despite divergences in the syntax, we have shown that LCC and CHR are equivalent for the abstract semantics considered here [9]. Therefore, the implemented prototype can already be used to execute the examples given in this article, modulo their trivial encoding in LCC.

Angelic semantics have been identified as the natural semantics for Concurrent Constraint (CC) programming languages [10] since the very beginning of the introduction of this language family: in this forward-chaining framework, the set of accessible computations is more natural to link with a logical interpretation than a particular computation path. However, the CC language and its angelic semantics is considered in [10] as an abstract language to reason about concurrency-related questions that can be captured in this formalism: there is no consideration about implementation. Moreover whenever CC languages have only a monotonous interpretation in classical logic, LCC and CHR handle non-monotonous traits with consumptions.

2 Angelic Semantics

In this section, derivation nets are first introduced for Constraint Simplification Rules, the fragment of CHR without propagation. Sharing strategies are introduced to algorithmically build derivation nets that reduce scheduling non-determinism. Derivation nets are then generalized to the full CHR language through the separation of CHR store between linear and persistent constraints, in the sense of the $\omega_!$ semantics [11].

An (oriented) *multigraph* is a pair $(V; \mathbf{i})$ where V is a set of *vertices* and $\mathbf{i} : V \times V \to \mathbb{N}$ is an *incidence function* giving the weight of the *edge* between each pair of vertices (with the convention that identifies the absence of edge with an edge of weight 0). Equivalenty, $\mathbf{i}$ is a multiset of binary edges in $V \times V$. For each vertex v, the multiset of *prevertices* ${}^\bullet v$, vertices that lead to v, is defined by the characteristic function $u \mapsto \mathbf{i}(u, v)$ and the multiset of *postvertices* $v^\bullet$, vertices that come from v, is defined by the characteristic function $u \mapsto \mathbf{i}(v, u)$.

[1] `http://contraintes.inria.fr/~tmartine/silcc`

A multigraph is *bipartite* if V is the disjoint union of two sets $V_1 \uplus V_2$ such that $\mathbf{i}(v, v') = 0$ for all $v, v' \in V_i$ for $i = 1$ or 2. An (oriented) *multihypergrap*h is a tuple (V, E, i) such that $(V \uplus E, i)$ is a bipartite multigraph: V is the set of the *vertices of the multihypergraph* and E are the *hyperarcs*. For each hyperarc $e \in E$, ${}^\bullet e$ is the set of *input vertices* of e and $e^\bullet$ is the set of *output vertices* of e. A *labeled multihypergraph* is a tuple (V, E, i, ℓ) such that (V, E, i) is a multihypergraph and $\ell : V \uplus E \to A$ is a mapping from vertices and hyperarcs to an alphabet of *labels* A.

2.1 Derivation nets for Constraint Simplification Rules (CSR)

Given a language for built-in constraints $\mathcal{L}_b$ equipped with a constraint theory $\mathcal{T}$ and a language for user-defined constraints $\mathcal{L}_u$, a CSR program is a set of constraint simplification rules.

Definition 1. *A* constraint simplification rule *has the form*

$$n@H \Leftrightarrow G|B_b, B_u$$

where n is the name *of the rule, the* head *H is a multi-set of user-defined constraints, the* guard *G is a built-in constraint, and the* body *is a conjunction of a built-in constraint B_b and a multi-set of user-defined constraints B_u.*

Consider the following rules describing the calculus of a two-dimensional scalar product with a concurrent product.

Example 1 (Concurrent two-dimensional scalar product).

```
init @ scalar(X1, Y1, X2, Y2, P) ⇔
  product(X1, X2, X), product(Y1, Y2, Y), sum(X, Y, P) .
product @ product(A, B, C) ⇔
  V is A * B, value(C, V).
sum @ sum(A, B, C), value(A, VA), value(B, VB) ⇔
  V is VA + VB, value(C, V) .
```

There are essentially two possible derivations in the abstract semantics from a query `scalar(X1, Y1, X2, Y2, P)`, revealing scheduling non-determinism, depending upon which of the two products is evaluated first: `product(X1, X2, X)` or `product(Y1, Y2, Y)`.

We introduce derivation nets to describe sharing strategies which quotient these scheduling choices.

Definition 2. *A derivation net for a CSR program P is a labeled multi-hypergraph $(V, E, \mathbf{i}, \ell)$, where the vertices V are labeled with built-in or user-defined constraints and the hyperarcs E are labeled with rule names ($\ell : V \uplus E \to \mathcal{L}_b \uplus \mathcal{L}_u \uplus \mathcal{N}$), such that for each hyperarc $e \in E$, there exists a rule $\langle n@H \Leftrightarrow G|B_b, B_u\rangle \in P$ and a renaming ρ for fresh variables occurring in the rule with*

– $\ell(e) = n$,
– $\ell(^\bullet e) = H\rho \uplus G'$,
– $\ell(e^\bullet) = B_b\rho \uplus B_u\rho \uplus G'$.

with G' a logical consequence of G under the hypotheses of the theory $\mathcal{T}$.

The following derivation net shows the quotiented derivation path for the scalar product.

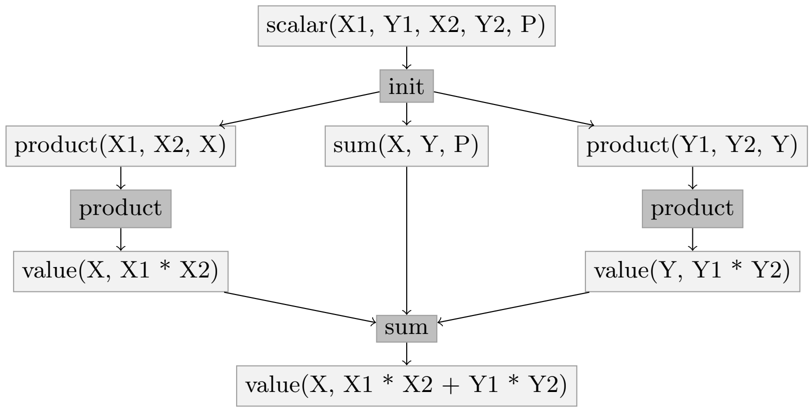

Fig. 1. Derivation net for the query `scalar(X1, Y1, X2, Y2, P)` with the scalar product program (Example 1)

Derivation nets can be equipped with a Petri-net semantics: vertices can be viewed as places marked with tokens that give their number of occurrences in the constraint store. Hyperarcs give the transitions between the places. Compared to the interpretation of CHR programs in Petri-nets [12] that interprets programs themselves as (a colored extension of) Petri-nets independently from the execution, the nets considered here give interpretations for partial executions of programs and grow as long as the execution continues.

Definition 3. *A* marking *is a multiset of vertices.* Derivations *are given by a binary relation $\rightarrow_\mathrm{d}$ between markings such that $m \rightarrow_\mathrm{d} m'$ if there exists a hyperarc e such that $^\bullet e \subseteq m$ (*i.e.*, for all v, $^\bullet e(v) \leqslant m(v)$) and for all v, $m'(v) = m(v) - {}^\bullet e(v) + e^\bullet(v)$.*

Markings are multisets of user constraints and built-in constraints: as such, they can be identified to CHR configurations.

Theorem 1 (Correction). *If there exists a derivation $m \rightarrow_\mathrm{d} m'$, then there exists a transition in the abstract semantics from the configuration m to the configuration m'.*

Moreover, a derivation net can always be extended with a new hyperarc for any possible transition, leading to new vertices according to the goal of the associated rule. By iterating this construction, it is possible to define a potentially infinite derivation net representing all the possible transitions.

Theorem 2 (Completeness). *For any initial configuration m, there exists a (possibly infinite) derivation net such that if m' is a configuration accessible from m in the abstract semantics, then m' is a marking accessible from m by derivation.*

Angelic execution consists therefore in the iterative construction of such a complete derivation net, keeping only the hyperarcs which are involved in a reachable marking from the initial configuration. Since the exploration is potentially infinite, the exploration should be done in breadth first to give an equal chance of execution to every computation path.

2.2 Sharing strategies

Derivation nets do not structurally force any sharing to reduce scheduling non-determinism. This is typically the case for *simpagation* rules: a simpagation rule is of the form $n@H_1\backslash H_2 \Leftrightarrow B$ where H_1 is a *persistent head*. Such a rule has the same logical interpretation as $n@H_1, H_2 \Leftrightarrow H_1, B$. Two firings of simpagation rules with a common persistent head can lead to two computation paths whether which rule is fired before the other.

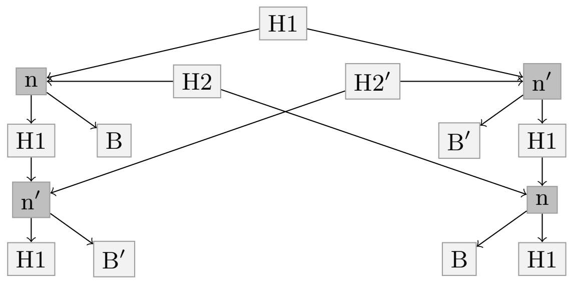

Fig. 2. Derivation net without sharing for the query `H1, H2, H2'` with two simpagation rules `n @ H_1 H_2` $\Leftrightarrow$ `B` and `n' @ H_1 H_2'` $\Leftrightarrow$ `B'`

This non-determinism can be reduced considering the derivation net where each simpagation rule is a hyperarc such that the vertex of the persistent head is the same both for input and output.

The iterative construction of such a derivation net can be done by sharing all equal user constraints to the same vertex: interpreting the derivation net as a Petri net, testing the reachability of a hyperarc reduces to testing the reachability in a Petri net, which is decidable [13] but computationaly expensive [14].

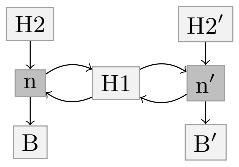

Fig. 3. Derivation net with sharing for the query `H1, H2, H2'` with two simpagation rules `n @ H_1 H_2` $\Leftrightarrow$ `B` and `n' @ H_1 H_2'` $\Leftrightarrow$ `B'`

Proposition 1. *The complete derivation net where all hyperarcs are reachable and all equal user constraints are shared to the same vertex can be iteratively constructed by solving an EXPSPACE-complete problem for each new hyperarc.*

However, in the context of derivation nets where all cycles are trivial, that is to say that every cycle consist in only one hyperarc having some vertices both as input and output, then there exists a log-linear algorithm to decide the reachability. The case of trivial cycles is particularily important as those cycles appear naturally when considering simpagation rules.

- Preparation: at each new vertex creation v0, computes a table $t(v_0)$ which associates each ancestor vertice v (outside trivial cycles) to its potential immediate successor hyperarc e (there is at most one!): $t(v_0) : v \mapsto e$ With balanced binary trees, logarithmic time cost for each vertex.
- To check if a binary hyperarc e_0 between v_0 and v_1 introduces a conflict:
  1. choose one of the predecessor vertex, say v_0, (preferably the one with least ancestors)
  2. let $t \leftarrow t(v_1) + (v_1 \mapsto e_0)$
  3. begin with $v \leftarrow v_0$,
  4. for each predecessor vertex v' of each predecessor hyperarc e of v,
  5. if $t(v')$ is defined, succeeds if $t(v') = e$ or v' in trivial cycle, else fails,
  6. if not, let $t \leftarrow t + (v' \mapsto e)$ and recursively go to 4 for $v \mapsto v'$.

In worst case, logarithmic cost (table search) for each ancestor.

In practice, either hyperarcs between neighbours ($t(v_0)$ is often defined) or hyperarcs between a vertex and a top-level ask (with few ancestors).

This algorithm gives a polynomial construction for completing the derivation net at each execution step.

2.3 Derivation nets in presence of Propagation Rules

As far as the abstract semantics is concerned, propagation rules $n@H \Rightarrow G|B_b, B_u$ are shortcuts for $n@H \Leftrightarrow G|H, B_b, B_u$: the head is restored after firing the rule. This non-consumption leads to trivially infinite computation paths that can be avoided with the $\omega_!$ semantics [11]. In terms of derivation nets, distinguishing the set of vertices between linear and persistent constraints make construction rules of derivation nets being refined to prevent this trivial non-termination. The strict output vertices of a hyperarc are marked as persistent if and only if all the input vertices are persistent or if it is a propagation (that is to say, if all input vertices are in a trivial cycle).

3 Angelic Programming

3.1 Head Decomposability

In the abstract semantics, a CHR rule can only observe the presence of a user constraint, not the absence: firing is monotonic relatively to the store. As a consequence, if observations are restricted to the side effects of the firing of some rules, silent partial consumption of the head of these rules cannot be observed. Such an observable is motivated by the fact that the body of fired rules is the place where side effects can happen. In particular, multiple headed rules can be rewritten in 2-headed rules by the introduction of fresh intermediary user constraints (carrying the context variables if any).

The rule

```
a, b, c, d ⇔ print("side effect")
```

and the set of rules

```
a ⇔ f1
f1, b ⇔ f2
f2, c ⇔ f3
f3, d ⇔ print ("side effect")
```

are equivalent provided that `f1, f2, f3` are fresh user constraints that do not appear elsewhere neither in the program nor in the initial goal.

This equivalence does not hold in general with a committed-choice scheduler since premature consumptions of `a` and `b` can prevent other rules to be fired even if `c` and `d` never appear. On the contrary, premature consumptions in angelic settings will only lead to blocking computation branches that will not prevent other branches to be explored. This property can benefit to the implementation: only 2-headed rules have to be considered. More precisely, all CHR rules can be translated to rules with two heads where one of them is an intermediary user constraint.

Such a translation makes trivial cycles of simpagations become non-trivial: the algorithm presented above can nevertheless be adapted in this case, since all hyperarcs involved in the cycle only introduce intermediary user constraints. Therefore, a hyperarc cannot be unreachable due to the consumption of such constraints by another rules.

3.2 Controlling the Angelism

User constraints can be introduced to explicitely sequence the execution of rules. The following program produces as side-effect a unspecified permutation of `a`, `b`, `c` when launched with the goal start.

```
start ⇔ a, b, c.
a ⇔ print("a").
b ⇔ print("b").
c ⇔ print("c").
```

The order can be fixed by the introduction of fresh intermediary constraints to mark the step of the sequence (carrying the context variables if any).

```
start ⇔ s0, a, b, c.
a, s0 ⇔ s1, print("a").
b, s1 ⇔ s2, print("b").
c, s2 ⇔ print("c").
```

Operationaly, such an explicit sequencing forces the derivation net to have a linear path instead of branching hyperarcs. Formally, the sequence is coded declaratively in the logic instead of being left to the conventional implementation of the comma sequence operator.

4 Applications

4.1 Angelism for CHR∨

Angelic execution can be seen as a search among scheduling. Therefore, CHR∨ rules where there can be multiple bodies, leading to a search for a successful one, can be encoded as multiple rules with the same head.

The rule `N @ H ⇔ G | B1 ; ...; Bn.` is encoded as the set of n rules `N1 @ H ⇔ G | B1.`, ..., `Nn @ H ⇔ G | Bn.` Angelic execution ensures that consequences of `B1`, ..., `Bn` are explored.

4.2 Meta-interpreters

The decomposability of heads allow CHR meta-interpreters to be conveniently written. Suppose that a rule is coded with a user constraint `rule(N @ H ⇔ G | B)` where `H` is a list of heads and with a proper encoding for the body `B` where user constraints are marked with the functor `ucstr`, then the following rules code a meta-interpreter.

```
first_head @ rule(_N @ H ⇔G, B), ucstr(H0) ⇔
  copy_term((H, G, B), ([H0 | T0], G0, B0)) |
  match(T0, G0, B0).
matching_end @ match([], G, B) ⇔call(G) |
  call(B).
matching_cont @ match([H | T], G, B), ucstr(H) ⇔
  match(T, G, B).
```

This meta-interpreter uses the decomposability of rules with match as intermediary user constraint.

Example 2. The scalar product example (Example 1) is encoded into the following constraints. A derivation net for the meta-interpretation of this program is given in Fig. 4.

```
rule(init @ [scalar(X1, Y1, X2, Y2, P)] ⇔true |
  ucstr(product(X1, X2, X)),
  ucstr(product(Y1, Y2, Y)),
  ucstr(sum(X, Y, P))).
rule(product @ [product(A, B, C)] ⇔true |
  V is A * B,
  ucstr(value(C, V))).
rule(sum @ [sum(A, B, C), value(A, VA), value(B, VB)] ⇔true |
  V is VA + VB,
  ucstr(value(C, V))).
```

4.3 User-defined Indexation

Thanks to the dependency book-keeping operated by derivation nets, rules can reformulate user constraints to another form while keeping the dependency relation between the original user constraint and the reformulation. For instance, suppose that the underlying implementation only makes indexing on the principal functor of the user constraint arguments. The following rule decomposes a nested term in a user constraint into another user constraint with this term as root argument.

```
c(f(X)) ⇔ cf(X) .
```

Then consuming `cf(X)` in another rule is equivalent to consuming `c(f(X))`: therefore, rules having heads matching on `c(X)` in general and rules having heads matching on `cf(X)` in particular can coexist and consume observationnaly the same user constraints.

4.4 Deep Guards

In a previous paper [15], we proposed a framework to extend the built-in guard language in CHR with a mechanism allowing to trigger CHR computation during guard entailment checking. This framework suffers from the trigerred computation having to satisfy sanity conditions not to pollute the store in case of non-entailment. Deep guards, that is to say CHR rules whose guards involve arbitrary CHR computations, can be trivially implemented with angelism. In the general form, a rule with a deep guard is written as follows.

```
H ⇔ (G => C) | B.
```

with the convention that the rule can only be fired if, once `H` has been consumed, the CHR goal `G` entails the CHR store `C`. Such a rule can be rewritten as follows, where mark is fresh (carrying the context variables if any).

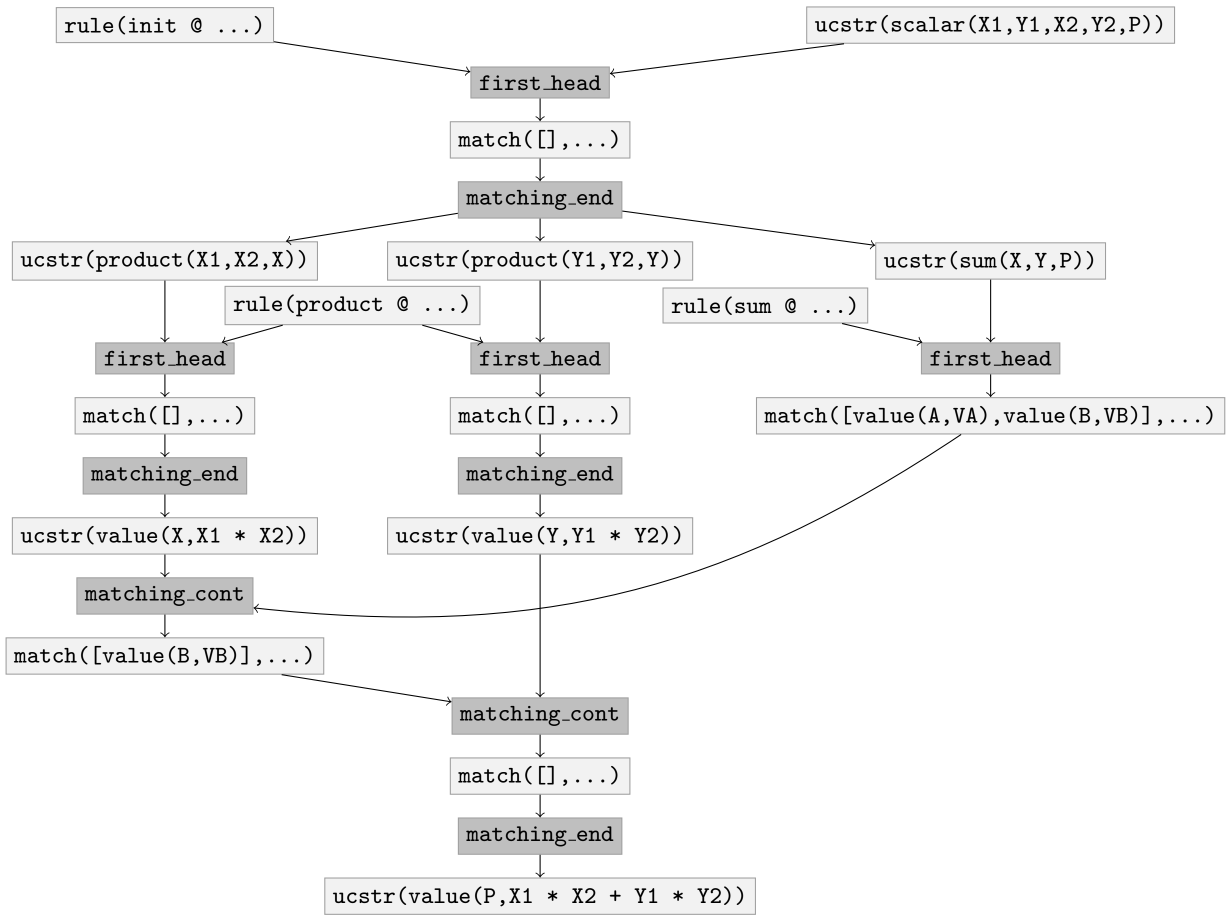

Fig. 4. Meta-interpretation of the query `ucstr(scalar(X1,Y1,X2,Y2,P))` with the meta-interpreted program given in Example 2

```
H ⇔ G, mark.
mark, C ⇔ B.
```

This is only correct in the case of an angelic execution since `H` is consumed before the execution of `G`: if this execution does not lead to `C`, other branches of execution should be explored.

5 Conclusion

We have described a new execution model for CHR, the angelic semantics, which computes all the reachable configuration from an initial CHR goal. We introduced a notion of derivation nets for graphical represention of CHR execution paths. These derivation nets allow the description of sharing strategies which make the angelic semantics tractable in practice. In these settings, we illustrate how angelic semantics can result to a fully declarative language, where control is captured by the logical interpretation. Proposed applications give natural solutions in the angelic execution model to questions which are still open with committed-choice: the existence of CHR meta-interpreters, the redefinition of specific user constraint representations, for indexation in particular, and more generally the interleaving of arbitrary computation between head consumption, allowing deep guards. The implementation still has to be done in the light of what has been already implemented for LCC. This work can begin with a simple meta-interpreter of CHR written in angelic LCC. However, even if we believe that such an implementation is possible, a lot of work remains to be explored in terms of compilation techniques to make the performance competitive with committed-choice implementations. This work is a move forward to more declarativity, for reducing the gap between the logical interpretation and the effective implementation of the semantics. We hope for more theoretical development of algorithms taking benefits of the angelic semantics, as well as progress for implementation efficiency.

References

1. Frühwirth, T.W.: Constraint Handling Rules. Cambridge University Press (2009)
2. Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: Constraint Handling Rules – a survey of CHR research between 1998 and 2007. Theory and Practice of Logic Programming **2** (January 2010)
3. Duck, G.J., Stuckey, P.J., Banda, M.G.D.L., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: In 20th International Conference on Logic Programming (ICLP'04, Springer-Verlag (2004) 90–104
4. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In: Proceedings of PPDP'07, International Conference on Principles and Practice of Declarative Programming, Wroclaw, Poland, ACM Press (2007) 25–36
5. Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., Sato, T.: CHR(PRISM)-based probabilistic logic learning. Theory and Practice of Logic Programming **10**(4-6) (2010) 433–447

6. Betz, H., Frühwirth, T.W.: A linear-logic semantics for constraint handling rules. In: Proceeding of CP 2005, 11th. (2005) 137–151
7. Girard, J.Y.: Linear logic. Theoretical Computer Science **50(1)** (1987)
8. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of constraint handling rules. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(2) (2009) 1–42
9. Martinez, T.: Semantics-preserving translations between linear concurrent constraint programming and constraint handling rules. In: Proceedings of PPDP'10, International Conference on Principles and Practice of Declarative Programming, Edinburgh, UK, ACM (2010) 57–66
10. Jagadeesan, R., Shanbhogue, V., Saraswat, V.A.: Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Parc (1991)
11. Betz, H., Raiser, F., Frühwirth, T.: A complete and terminating execution model for Constraint Handling Rules. Theory and Practice of Logic Programming **10**(4-6) (2010) 597–610
12. Betz, H.: Relating coloured petri nets to constraint handling rules. In: Proceedings of the forth Constraint Handling Rules Workshop CHR'07. (2007) 33–47
13. Mayr, E.W.: An algorithm for the general petri net reachability problem. In: Proceedings of the thirteenth annual ACM symposium on Theory of computing. STOC '81, New York, NY, USA, ACM (1981) 238–246
14. Lipton, R.J.: The reachability problem requires exponential space. Technical Report 62, New Haven, Connecticut: Yale University, Department of Computer Science, Research (January 1976)
15. Fages, F., de Oliveira Rodrigues, C.M., Martinez, T.: Modular CHR with ask and tell. In Frühwirth, T., Schrijvers, T., eds.: Proceedings of the fifth Constraint Handling Rules Workshop CHR'08. (2008)