First-order ASP programs as CHR programs

Igor Stéphan
Univ Angers, LERIA, SFR MATHSTIC
F-49000 Angers, France
igor.stephan@univ-angers.fr

ABSTRACT

We present in this paper a way to use the paradigm of Answer Set Programming (ASP) into the Constraint Handling Rules (CHR) paradigm. We present a translation of the ASP language to the Constraint Handling Rules language.

The committed-choice principle of the CHR paradigm leads to choose the rule-oriented approach of answer set computation. Since CHR is a first-order logic programming paradigm, the initial grounding phase of most of the ASP solvers is not required.

Our implementation compiles an ASP program to a CHR(Prolog) program or to a CHR(C++) program. Preliminary experiments of the latter present some interesting results on ASP programs with some large sets of facts.

Since Constraint Handling Rules is a paradigm developed for the implementation of user-defined constraints, we show how some extensions of ASP may be easily implemented in CHR: we show this by example for the choice rule.

CCS CONCEPTS

• **Theory of computation → Constraint and logic programming**.

KEYWORDS

Constraint Handling Rules, Answer Set Programming

ACM Reference Format:
Igor Stéphan. 2018. First-order ASP programs as CHR programs. In *SAC '21,March 22 – March 26, 2021, Gwangju, South Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Constraint Handling Rules (CHR) [13, 14] is a committed-choice language consisting of multiple-heads guarded rules that rewrites constraints into simpler ones until they are solved. CHR is a special-purpose language concerned with defining declarative constraints in the sense of *Constraint logic programming* [20, 21]. CHR is a language extension that allows to introduce *user-defined* constraints, i.e. first-order predicates, into a given host language as Prolog or imperative host languages like C/C++ or Java. CHR defines for example *propagation* over user-defined constraints that adds new constraints that are logically redundant but may cause further simplifications. For example, the user-defined constraint $(. \leq .)$ denotes the less-than-or-equal constraint. The rule $((X \leq Y), (Y \leq Z) \Rightarrow (X \leq Z))$ expresses the transitivity of less-than-or-equal constraint and means "if constraints $(X \leq Y)$ and $(Y \leq Z)$ (the multiple-head) are present then constraint $(X \leq Z)$ is logically entailed". Answer Set Programming (ASP) is a very convenient paradigm to represent knowledge in Artificial Intelligence and to encode combinatorial problems: a first-order logic program with default negation is elaborated in such a way that the *answer sets* (*stable models* in the seminal [19]) of the program represent the solutions of the initial problem.

Example 1.1 (3-coloring problem). The following answer set program P_{col} encodes a simple 3-coloring problem:

$vertex(1).$ $vertex(2).$ $vertex(3).$
$color(blue).$ $color(red).$ $color(green).$
$edge(1,2).$ $edge(1,3).$ $edge(2,3).$
$colored(V,C)$
$\leftarrow vertex(V), color(C), not\ ncolored(V,C).$
$ncolored(V,C)$
$\leftarrow colored(V,C\_), color(C), C\_ != C.$
$\bot \leftarrow edge(V,V\_), colored(V,C), colored(V\_,C).$

The three first lines represent a graph with three vertices, three colors and three edges between them (nine facts). The rule:
$colored(V,C) \leftarrow vertex(V), color(C), not\ ncolored(V,C).$
is a guess rule that chooses if a vertex V is colored with a certain color C or not. The rule:
$ncolored(V,C) \leftarrow colored(V,C\_), color(C), C\_ != C.$
means that a vertex V that is colored with a certain color C is not colored with another color $C\_$. The rule:
$\bot \leftarrow edge(V,V\_), colored(V,C), colored(V\_,C).$
is an ASP constraint that expresses that two adjacent vertices cannot be of the same color.

Since ASP is convenient to encode combinatorial problems, our first question is : how can we encode ASP programs as CHR programs?

An `ASPeRiX` computation [22] is a forward chaining process to compute answer sets of an ASP program that instantiates and fires one unique rule at each iteration according to two kinds of inference: a monotonic step of propagation and a nonmonotonic step of choice. This forward chaining process is very close to the usual operational semantics of CHR: the body of an ASP rule corresponds to the multi-head of a CHR rule.

In this article, we propose a translation from a first-order ASP program to a CHR program with "don't know" nondeterminism ($CHR^{\vee}$ [2]) complete and sound (under some conditions) thanks to `ASPeRiX` computations. It is valuable to define such a translation since, we may use ASP programs as constraints in CHR programs

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3441963

and we may also implement user-defined constraints that are not in the ASP paradigm with some user-defined CHR constraints.

Our first question leads us to a second one: how this translation may offer an efficient solution to the bottleneck of the grounding phase for ASP programs with a huge Herbrand universe?

For most of the ASP solvers the computation is done actually only on a propositional program. ASP atom-oriented solvers like `Clingo` [17] have two-phases algorithm : a grounding phase and a resolution phase. A grounder computes such propositional program by instantiating the variables of a first-order program by the elements of the Herbrand universe. For an infinite Herbrand universe, the grounding phase is sometimes infinite and the solver cannot compute the finite answer sets.

Example 1.2 (Infinite Herbrand universe with a finite answer set). This is the case for the following ASP program P_∞:
$q(s(X)) \leftarrow p(X), not\ r(X).$
$p(s(X)) \leftarrow p(X), not\ q(s(X)).$
$r(s(0)).$
$p(0).$
that has an infinite Herbrand universe and a unique finite answer set $\{p(0), r(s(0)), q(s(0))\}$.

But `ASPeRiX` computations are based on an on-the-fly grounding principle: There is no more grounding phase but interleaving of propagations and instantiations like in CHR.

Our translation is implemented in Prolog and generates CHR rules for CHR(Prolog) or CHR(C++). The latter means that the CHR rules are embedded in an C++ program, those rules are compiled to a C++ code by a dedicated compiler that generates a pure C++ program, this program is compiled by a classical gcc compiler. Experimental results presented in Section 5 show that

- the execution time of an ASP program considered as a CHR constraint in an C++ host is comparable to and in most case outperforms the execution time of the ASP solver with on-the-fly grounding `ASPeRiX` [23] and
- on some benchmarks with very large sets of facts (and with a huge Herbrand universe), the execution time is also comparable to the execution time of the state-of-the-art ASP solver with grounding phase, the solver `Clingo` [17].

Since CHR is a Turing-complete paradigm developed for the implementation of user-defined constraints solvers, CHR is well adapted to extend ASP with the extensions of [10] or completely new features as, for example, quantifiers[1]. We take the example of the *choice rule*.

The paper is organized as follows. In Section 2, we recall the backgrounds about ASP (and `ASPeRiX` computations) and about CHR (and CHR$^\vee$). In Section 3, we present our translation from ASP to CHR$^\vee$ thanks to the `ASPeRiX` computations. In Section 4, we describe some related works. In Section 5, we present some experimental results. In Section 6, we show how CHR allows to extend ASP to the choice rule. In Section 7, we conclude and draw some perspectives.

[1][8] shifts the QCSP (Quantified Constraint Satisfaction Problems) framework to the QCHR (Quantified Constraint Handling Rules) framework by enabling dynamic binder and access to user-defined constraints.

2 BACKGOUNDS

Answer Set Programming. An *answer set program* is a set of *rules* like
$(c \leftarrow\ a_1, \dots, a_n, not\ b_1, \dots, not\ b_m.)\ (n \geq 0, m \geq 0)$
where $c, a_1, \dots, a_n, b_1, \dots, b_m$ are atoms built from predicate symbols, constant symbols, variables and function symbols. PS_P (or simply PS) denotes the set of predicate symbols of an ASP program P. For any predicate symbols p, $ar(p)$ denotes its arity. For a rule r (or by extension for a rule set), we note $head(r) = c$ its *head*, $b^+(r) = \{a_1, \dots, a_n\}$ its *positive body*, $b^-(r) = \{b_1, \dots, b_m\}$ its *negative body*, and $\mathcal{V}(r)$ denotes the set of variables of an ASP rule r. A rule with an empty body is a *fact*. If necessary, ASP rules are considered numbered. $facts(P)$ denotes the set of the facts of an ASP program P. When the negative body of a rule is not empty we say that this rule is *non-monotonic*. A *ground substitution* is a mapping from the set of variables to the set of the ground terms (terms without any variable). If t is a term (resp. a an atom, r an ASP rule) and σ a ground substitution, $\sigma(t)$ (resp. $\sigma(a)$, $\sigma(r)$) is a *ground instance* of t (resp. a, r). A program P can be seen as an intensional version of the propositional program $ground(P) = \bigcup_{r \in P} ground(r)$ where $ground(r)$ is the set of all fully instantiated rules that can be obtained by substituting every variable in r by every term of the Herbrand universe of P. $\mathcal{A}_P$ denotes the Herbrand base of P, ie. the set of all the ground atoms built from the predicate symbols of P.

The following definitions are from [9, 22]. An `ASPeRiX` *computation* for a program P is defined as a process on a computation state based on a *partial interpretation* that is a pair (IN, OUT) of disjoint atom sets included in the Herbrand base of P. The notion of partial interpretation defines different status for rules (r a rule, σ a ground substitution and $I = (IN, OUT)$ a partial interpretation): if $b^+(\sigma(r)) \subseteq IN$ then $\sigma(r)$ is *supported* w.r.t. I ; if $b^-(\sigma(r)) \cap IN \neq \emptyset$ then $\sigma(r)$ is *blocked* w.r.t. I ; if $b^-(\sigma(r)) \subseteq OUT$ then $\sigma(r)$ is *unblocked* w.r.t. I ; if $\sigma(r)$ is supported and not blocked then $\sigma(r)$ is *applicable* w.r.t. I.

The following definitions set ground rules that can be fired by propagation (Δ_{pro}) and those that are applicable (Δ_{cho}) and can be chosen to be unblocked or to be blocked[2] (P a program, $I = (IN, OUT)$ a partial interpretation and R a set of ground rules, σ a substitution, $\sigma(r) \in ground(P) \setminus R$):
$\Delta_{pro}(P, I, R) = \{\sigma(r) \text{ supported and unblocked}\}$
$\Delta_{cho}(P, I, R) = \{\sigma(r) \text{ supported and not blocked}\}$

The specific case of ASP constraints (ie. rules with $\perp$ as head) is treated by adding $\perp$ to the OUT set. By this way, if an ASP constraint is fired (violated), $\perp$ should be added to IN and thus, (IN, OUT) would not be a partial interpretation.

An `ASPeRiX` *computation* [9, 22] for an answer set program P is a sequence $\langle R_j, K_j, I_j \rangle_{j=0}^{\infty}$ of `ASPeRiX` *states* constituted of ground rule sets R_j, ground ASP constraint sets K_j and partial interpretations $I_j = (IN_j, OUT_j)$ that satisfies the following conditions[3] ($K_0 = \emptyset$, $R_0 = \emptyset$ and $I_0 = (\emptyset, \{\perp\})$):

- $(Revision)$ Three possible cases ($\forall j \geq 1$),
  - $(Propagation)$ with $r_j \in \Delta_{pro}(P, I_{j-1}, R_{j-1})$:

[2]To fire a rule means to add the head of the rule in the IN set.
[3]In this context, r_j is a ground instance of a rule of the ASP program.

$K_j = K_{j-1}, R_j = R_{j-1} \cup \{r_j\}$,
$I_j = (IN_{j-1} \cup \{head(r_j)\}, OUT_{j-1})$

– ($Rule\ Choice$) if $\Delta_{pro}(P \cup K_{j-1}, I_{j-1}, R_{j-1}) = \emptyset$ and with $r_j \in \Delta_{cho}(P, I_{j-1}, R_{j-1})$:
$K_j = K_{j-1}, R_j = R_{j-1} \cup \{r_j\}$,
$I_j = (IN_{j-1}, OUT_{j-1} \cup b^-(r_j))$

– ($Rule\ Exclusion$) if $\Delta_{pro}(P \cup K_{j-1}, I_{j-1}, R_{j-1}) = \emptyset$ and with $r_j \in \Delta_{cho}(P, I_{j-1}, R_{j-1})$:
$K_j = K_{j-1} \cup \{\bot \leftarrow \cup_{b \in b^-(r_j)} not\ b.\}, R_j = R_{j-1}$,
$I_j = I_{j-1}$

- or ($Stability$) $K_j = K_{j-1}, R_j = R_{j-1}$ and $I_j = I_{j-1}$.

If there exists $i \geq 0$ such that $\Delta_{cho}(P \cup K_i, I_i, R_i) = \emptyset$ then the computation is said to *converge*.

The only syntactic restriction required by this methodology is that every rule of an ASP program must be *s*afe. That is, all variables occurring in the head and all variables occurring in the negative body of a rule occur also in its positive body. Note that this condition is already required by all standard evaluation procedures.

The following theorem [22] establishes a connection between the results of any ASPeRiX computation that converges and the answer sets of an answer set program:

Theorem 2.1 ([22]). *Let P be an answer set program and X be a finite atom set. Then, X is a finite answer set of P if and only if there is an* ASPeRiX *computation* $S = \langle R_j, K_j, I_j \rangle_{j=0}^{\infty}$, $I_j = (IN_j, OUT_j)$, *for P such that S converges and* $IN_\infty = X$.

Constraint Handling Rules. CHR is a declarative language extension especially designed for writing user-defined constraints (relations or predicates). CHR is essentially a language consisting of multi-headed garded rules that rewrite constraints into simpler ones until they are solved. Starting from an initial store, a set of user-defined constraints, the following kind of rules may be non deterministically applied: *simplification* replaces constraints by simpler ones while preserving logical equivalence ; *propagation* adds new constraints that are logically redundant but may cause further simplifications ; *simpagation* mixes and subsumes simplification and propagation. CHR allows to use guards that are sequences of host language statements. The CHR formalism is defined as follows : a CHR rule is a rule of the form ($K_1, \ldots, K_m, D_1, \ldots, D_n, B_1, \ldots, B_p$ some constraints):

- [Simpagation Rule] $r@(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B)$ with $n > 0, m > 0$ or
- [Propagation rule] $r@(K_1, \ldots, K_m \Rightarrow B)$ with $m > 0$ or
- [Simplification rule] $r@(D_1, \ldots, D_n \Leftrightarrow B)$ with $n > 0$

and $B = B_1, \ldots, B_p$ with $p > 0$ or $true$ or $fail$[4] (two reserved symbols), where $K_1, \ldots, K_m, D_1, \ldots, D_n$ are the multiple heads ($D_1, \ldots, D_n$ are deleted from the store of constraints by the application of the rule while $K_1, \ldots, K_m$ are kept), B the body, and r is the name of the CHR rule. CHR$^\vee$ [2] introduces the *don't know* nondeterminism: the body B may contain disjunctions whose disjuncts are separated, like in Prolog, by ";".

From the very beginning, [13] gives a declarative semantics in terms of first-order classical logic: Simplification rules are considered as logical equivalences and propagation rules as implications.

[4]we use $fail$ instead of $false$ to be compatible with CHR(Prolog)

But [13] gives also a first *abstract* (or *high-order* or *theoretical*) operational semantics ω_t based on a transition system over sets (with some extensions to avoid the trivial nontermination of propagation rules [1])[5]. Since the formal semantics for CHR$^\vee$ is only useful for the proofs of soundness and completeness of our translation of ASP programs to CHR$^\vee$ programs (see Subsection 3.7), which are, due to the lack of space, not included into this article, the formal semantics of CHR$^\vee$ is also omitted (see [2, 14–16]).

3 TRANSLATION FROM ASP TO CHR$^\vee$.

In this section we show how we translate all the mechanisms of ASPeRiX computation to CHR$^\vee$ mechanisms: each ASP rule is translated into a sequence of CHR rules that encode ($Propagation$) for monotonic and nonmonotonic rules and ($Rule\ Choice$) and ($Rule\ Exclusion$) for nonmonotonic rules. Non-determinism of the ASPeRiX computation is managed by the non-determinism of CHR$^\vee$.

The IN set of ASPeRiX computation is maintained by the store of constraints: each atom of an ASP program P is translated into a constraint of same predicate symbol and same arity. The OUT set of ASPeRiX computation is also maintained by the store of constraints: each atom of predicate symbol p of an ASP program P is translated into its *out version*: a constraint of predicate symbol *out_p* with same arity. O denotes the set of the out predicate symbols of a program P: $\{out\_p \mid r \in P, p(t_1, \ldots, t_n) \in b^-(r)\}$. Function *out* is such that $out(sa)$ replaces in a sequence sa of atoms any atom considered as constraint by its out version if its predicate symbol is into the O set.

3.1 Translation of facts

In ASP, a fact is an ASP rule with an empty body. Since facts are supported and unblocked they are applied during the first propagation phase and inserted into the IN set and hence into the initial store of constraints. In what follows all the ASP rules are rules with a non-empty body.

3.2 Propagation

The property of the ground rule to be supported is encoded by the *sup_i* CHR constraint. A ground rule $\sigma(r_i)$ is supported if $b^+(\sigma(r_i)) \subseteq IN$. This is encoded by a membership test for $b^+(\sigma(r_i))$ into the store of constraints. The following function ST generates the CHR rules that maintain this property:

$ST(r_i) =$
if $head(r_i) = \bot$ then $[(b^+(r_i) \Rightarrow fail)]$ else
if $b^+(r_i) \neq \emptyset$ then $[(b^+(r_i) \Rightarrow sup\_i(\mathcal{V}(r_i)))]$ else [ ]

Instead of adding $\bot$ into the OUT set, we directly use the atom $fail$ of CHR in the translation.

For first order ASPeRiX computation, the grounding of ASP rules is maintained by the parameter passing mechanism of CHR. For an ASP rule r_i with an empty positive body and a non-empty negative body, since the ASP rules are safe, $\mathcal{V}(r_i)$ is also empty and the associated CHR constraint *sup_i* is of arity 0. Since the positive body is empty these ASP rules are automatically supported ($b^+(\sigma(r_i)) = \emptyset \subseteq IN$) and the associated CHR constraints *sup_i* are inserted

[5]The *refined* operational semantics ω_r [12] is finer than the previous one w.r.t. to the classical implementations of CHR.

into the initial store of constraints. Function SPT collects for an ASP program those CHR constraints (ie. $SPT(P) = \{sup\_i \mid r_i = (head(r_i) \leftarrow not\ b_1, \ldots, not\ b_m.) \in P\}$).

The $(Propagation)$ step of ASPeRiX computation applies one supported and unblocked ASP rule. A ground rule $\sigma(r_i)$ is supported if $\sigma(sup\_i(\mathcal{V}(r_i)))$ is into the store of constraints and is unblocked if $b^-(\sigma(r_i)) \subseteq OUT$. This is encoded by a membership test of $\sigma(out(b^-(r_i)))$ into the store of constraints. The following function PT generates the CHR rules that apply $(Propagation)$ step:

$PT(r_i) =$
if $head(r_i) = \bot$ then [] else
if $out(b^-(r_i)) = \emptyset$ then $[sup\_i(\mathcal{V}(r_i)) \Leftrightarrow head(r_i)]$
else $[out(b^-(r_i)) \backslash sup\_i(\mathcal{V}(r_i)) \Leftrightarrow head(r_i)]$

First case is for monotonic rules while second case is for nonmonotonic rules.

Example 3.1. (Example 1.1 continued)
$ST(r_0) = [colored(V, C), colored(V\_, C), edge(V, V\_) \Rightarrow fail]$
$ST(r_1) = [color(C), vertex(V) \Rightarrow sup\_1(V, C)]$
$PT(r_1) = [out\_ncolored(V, C) \backslash sup\_1(V, C) \Leftrightarrow colored(V, C)]$

3.3 Choice and exclusion

Revision by $(Rule\ Exclusion)$ step is not necessary to characterize answer sets. It adds the possibility to block a rule from Δ_{cho} instead of firing it. To block a rule is to add an ASP constraint with the negative atoms of the rule body. This possibility restricts rule choice in Δ_{cho} and thus forbids some ASPeRiX computations. $(Rule\ Choice)$ and $(Rule\ Exclusion)$ steps are combined into a CHR$^\vee$ choice point[6]. The following function CT generates the CHR rules that apply this combination of $(Rule\ Choice)$ and $(Rule\ Exclusion)$ steps (r an ASP rule):

$CT(r_i) =$
if $b^-(r_i) = \emptyset$ then [] else
$[(sup\_i(\mathcal{V}(r_i)), next\_it \Leftrightarrow out(b^-(r_i)), head(r_i), next\_it\ ;$
$excl\_i(\mathcal{V}(r_i)), next\_it)]$

First part of the disjunct encodes $(Rule\ Choice)$: The adding of $b^-(\sigma(r_i))$ to OUT_{j-1} is encoded by inserting $out(b^-(\sigma(r_i)))$ constraints into the store of constraints. Second part of the disjunct encodes $(Rule\ Exclusion)$: The ASP constraints of K_j are encoded by the $excl\_i$ CHR constraint. The mechanism is similar to that of the sup_i CHR constraint. The ASP constraint $(\bot \leftarrow\ b^-(\sigma(r_i)).) \in K_j$ is fired if $b^-(\sigma(r_i)) \subseteq OUT$. The following function ET generates the CHR rules that maintain this property:

$ET(r_i) =$ if $b^-(r_i) = \emptyset$ then []
else $[excl\_i(\mathcal{V}(r_i)), out(b^-(r_i)) \Leftrightarrow fail]$

These CHR rules will stop any further ASPeRiX computation with $b^-(\sigma(r_i))$ into the OUT set.

The *next_it* (for *next iteration*) constraint maintains the general loop of the algorithm: it forces application of $(Rule\ Choice)$ and $(Rule\ Exclusion)$ when Δ_{pro} is empty (and $(Propagation)$ is no more possible) and Δ_{cho} is not empty.

[6] Δ_{pro} have to be empty: all the CHR rules that define supported ASP rules and that encode Δ_{pro} have to be before the CHR rules that encode Δ_{cho}. See Subsection 3.6 for the order of the CHR rules in the program.

Example 3.2. (Example 1.1 continued)
$CT(r_1) = [sup\_1(V, C), next\_it \Leftrightarrow$
$out\_ncolored(V, C), colored(V, C), next\_it;$
$excl\_1(V, C), next\_it]$
$ET(r_1) = [excl\_1(V, C), out\_ncolored(V, C) \Leftrightarrow fail]$

$(Rule\ Exclusion)$ step introduces some ASP constraints that have to be checked at the end of an ASPeRiX computation to be sure that the excluded ASP rules are not applicable ($\Delta_{cho}(P \cup K_i, I_i, R_i) = \emptyset$). The *next_it* CHR constraint controls at the end of the ASPeRiX computation that this is truly the case (r_i a nonmonotonic rule):

$EC(r_i) = [next\_it, excl\_i(\mathcal{V}(r_i)) \Leftrightarrow fail]$

Finally, if an ASP constraint of a K_j set is blocked then it must not be kept in Δ_{cho}. The following function CLT generates the CHR rules that maintain this property (r_i an ASP rule):

$CLT(r_i) = [(b \backslash excl\_i(\mathcal{V}(r_i)) \Leftrightarrow true) \mid b \in b^-(r_i)]$

Example 3.3. (Example 1.1 continued)
$EC(r_1) = [next\_it, excl\_1(V, C) \Leftrightarrow fail]$
$CLT(r_1) = [ncolored(V, C) \backslash excl\_1(V, C) \Leftrightarrow true]$

3.4 Set properties for the IN and OUT sets

Set property of the IN set is maintained by the following sequence of CHR rules generated by the function *in_is_a_set* (S a set of predicate symbols):

$in\_is\_a\_set(S) =$
$[(p(\vec{X}) \backslash p(\vec{X}) \Leftrightarrow true)$
$\mid p \in S, ar(p) = n, \vec{X}$ a sequence of n variables$]$

Set property of the OUT set is maintained in a similar way by the function *out_is_a_set*.

The disjoint set property for the IN and OUT sets is encoded by the following rules generated by the function *disjoin* (S a set of predicate symbols):

$disjoin(S) =$
$[(p(\vec{X}), out\_p(\vec{X}) \Leftrightarrow fail)$
$\mid p \in S, ar(p) = n, \vec{X}$ a sequence of n variables$]$

Example 3.4. (Example 1.1 continued)
$in\_is\_a\_set(\{ncolored, colored\}) =$
$[colored(V, C) \backslash colored(V, C) \Leftrightarrow true,$
$ncolored(V, C) \backslash ncolored(V, C) \Leftrightarrow true]$
$disjoin(\{ncolored\}) =$
$[ncolored(V, C), out\_ncolored(V, C) \Leftrightarrow fail]$

3.5 Two simple optimizations

If the head of a ground rule is already into the IN set, it is not useful to keep it in Δ_{pro} then the $sup\_i(\mathcal{V}(r))$ constraint is consumed. The following function UT generates the CHR rules that maintain this property:

$UT(r_i) =$ if $head(r_i) = \bot$ then []
else $[(head(r_i) \backslash sup\_i(\mathcal{V}(r_i)) \Leftrightarrow true)]$

In a same way, if a ground rule is supported but blocked, it is not useful to keep it in Δ_{cho}. The following function BT generates the CHR rules that maintain this property:

$BT(r_i) = [(b \backslash sup\_i(\mathcal{V}(r_i)) \Leftrightarrow true) \mid b \in b^-(r_i)]$

Example 3.5. (Example 1.1 continued)
$UT(r_1) = [colored(V, C) \backslash sup\_1(V, C) \Leftrightarrow true]$
$BT(r_1) = [ncolored(V, C) \backslash sup\_1(V, C) \Leftrightarrow true]$

3.6 Translation of an ASP program

The translation of an ASP rule that is not a fact is then the sequence of the following CHR$^\vee$ rules ($\oplus$ stands for concatenation of sequences, r_i an ASP rule):

$RT(r_i) =$
if r_i is a nonmonotonic rule then
$(ST(r_i) \oplus UT(r_i) \oplus PT(r_i) \oplus BT(r_i) \oplus$
$ET(r_i) \oplus CLT(r_i), CT(r_i), EC(r_i))$
else $(ST(r_i) \oplus UT(r_i) \oplus PT(r_i), [], [])$

The result of the function RT is a triplet: rules generated by the CT and EC functions are isolated to be inserted together at the end of the CHR program (see the following subsection).

The translation of a set R of ASP rules that are not facts is then the following sequence of CHR$^\vee$ rules[7] (H denotes the set of the predicate symbols that appear in at leat one head of a rule of R):

$RulesT(R) =$
$in\_is\_a\_set(H) \oplus out\_is\_a\_set(O) \oplus disjoin(O) \oplus$
$(\bigoplus_{r_i \in R} fst(RT(r_i))) \oplus$
$(\bigoplus_{r_i \in R} snd(RT(r_i))) \oplus$
$(\bigoplus_{r_i \in R} rd(RT(r_i)))$

Finally we define our translation $ASPT$ from ASP program to CHR$^\vee$ program according to ASPeRiX computation as a pair constituted of a CHR$^\vee$ program $RulesT(P) \oplus [next\_it \Leftrightarrow true]$ and an initial store of constraints $facts(P) \oplus SPT(P) \oplus [next\_it]$ (see subsection 3.2 for $SPT(P)$) as follows:

$ASPT(P) =$
$(RulesT(P) \oplus [next\_it \Leftrightarrow true],$
$facts(P) \oplus SPT(P) \oplus [next\_it])$

Example 3.6. (Example 1.1 continued) Table 1 reports $ASPT(P_{col})$ for the program P_{col} with

$$facts(P_{col}) \oplus SPT(P_{col}) \oplus [next\_it] = [vertex(1), vertex(2), vertex(3), color(blue), color(red), color(green), edge(1,2), edge(1,3), edge(2,3), next\_it]$$

for CHR(Prolog) since $SPT(P_{col}) = []$.

We obtained the six intended answer sets.

Table 2 reports the trace of computation of this first answer set, that is for the solution $= \{colored(1, green), colored(2, red), colored(3, blue)\}$. Rule $ct(r_1)$ at lines resp. 10, 21 and 30 adds to the IN set, resp. `colored(1,green)`, `colored(2,red)` and `colored(3,blue)` and by backtrack at lines resp. 18, 20 and 29, resp. `excl_1(2,green)`, `excl_1(3,green)` and `excl_1(3,red)`. Those three exclusions are confirmed at lines resp. 24, 33 and 36 by the adding to the IN set of `ncolored(1,green)`, `ncolored(2,red)` and `ncolored(3,blue)` at lines 22, 23, 31, 32, 34 and 35 by application of $st(r_2)$ and $pt(r_2)$. At last, `ncolored(1,red)`, `ncolored(1,blue)` and `ncolored(2,blue)` are added to the IN set by application of $st(r_2)$ and $pt(r_2)$ at lines 11, 12, 14, 15, 25 and 26.

Example 3.7. (Example 1.2 continued) Table 3 reports $ASPT(P_\infty)$ for the program P_∞ with

$facts(P_\infty) \oplus SPT(P_\infty) \oplus [next\_it] = [r(s(0)), p(0), next\_it]$
for CHR(Prolog) since $SPT(P_\infty) = []$.

[7] Functions fst, snd and rd are such that for a triplet (x, y, z), $fst((x, y, z)) = x$, $snd((x, y, z)) = y$, and $rd((x, y, z)) = z$.

```
:- use_module(library(chr)).
:- chr_constraint sup_1/2, sup_2/3, excl_1/2,
   next_it/0, edge/2, colored/2, color/1,
   vertex/1, ncolored/2, out_ncolored/2.
colored(V,C) \ colored(V,C) <=> true.
ncolored(V,C) \ ncolored(V,C) <=> true.
ncolored(V,C),out_ncolored(V,C) <=> fail.
% st(r_0) = // Example 1
colored(V,C),colored(V_,C),edge(V,V_) ==> fail.
% st(r_1) = // Example 1
color(C),vertex(V) ==> sup_1(V,C).
% st(r_2) =
color(C_),colored(V,C) ==>
   C \= C_ | sup_2(V,C_,C).
% ut(r_1) = // Example 5
colored(V,C) \ sup_1(V,C) <=> true.
% ut(r_2) =
ncolored(V,C) \ sup_2(V,C,_) <=> true.
% pt(r_1) = // Example 1
out_ncolored(V,C) \ sup_1(V,C) <=>
   colored(V,C).
% pt(r_2) =
sup_2(V,C,_) <=> ncolored(V,C).
% bt(r_1) = // Example 5
ncolored(V,C) \ sup_1(V,C) <=> true.
% et(r_1) = // Example 2
excl_1(V,C), out_ncolored(V,C) <=> fail.
% clt(r_1) = // Example 3
ncolored(V,C) \ excl_1(V,C) <=> true.
% ct(r_1) = // Example 2
next_it,sup_1(V,C) <=>
   out_ncolored(V,C),colored(V,C),next_it ;
   excl_1(V,C),next_it.
next_it, excl_1(_,_) <=> fail. % Example 5
next_it <=> true.
```

Table 1: $ASPT(P_{col})$

The Herbrand universe of this program is infinite but the unique answer set $\{p(0), r(s(0)), q(s(0))\}$ is computed by CHR(Prolog).

3.7 Soundness and completeness of the translation

The following theorems express completeness and soundness of our translation. We show that any ASPeRiX computation of a program P can be translated into a CHR$^\vee$ execution of the CHR program $ASPT(P)$ and conversely. Completeness and soundness follow from Theorem 2.1.

Theorem 3.8 (completeness). *Let P be an ASP program. Let $(CHR\_Program, Initial\_Store) = ASPT(P)$. Let X be a set of atoms considered also as constraints.*

If X is an answer set of P then there exists a CHR$^\vee$ execution under the ω_t semantics of the CHR program CHR_Program with an initial store of constraints Initial_Store and a final store S such that $S \cap \mathcal{A}_P = X$.

Soundness is only insured under two conditions that insure that $\Delta_{pro}(P, I_{j-1}, R_{j-1})$, $\Delta_{cho}(P, I_{j-1}, R_{j-1})$, $\Delta_{pro}(P \cup K_{j-1}, I_{j-1}, R_{j-1})$

```
1.  st(r_1) with C=blue and V=3
2.  st(r_1) with C=blue and V=2
3.  st(r_1) with C=blue and V=1
4.  st(r_1) with C=red and V=3
5.  st(r_1) with C=red and V=2
6.  st(r_1) with C=red and V=1
7.  st(r_1) with C=green and V=3
8.  st(r_1) with C=green and V=2
9.  st(r_1) with C=green and V=1
10. ct(r_1) with C=green and V=1
11. | st(r_2) with C=green, C_=red and V=1
12. | pt(r_2) with C=red and V=1
13. | bt(r_1) with C=red and V=1
14. | st(r_2) with C=green, C_=blue and V=1
15. | pt(r_2) with C=blue and V=1
16. | bt(r_1) with C=blue and V=1
17. | ct(r_1) with C=green and V=2
18. | | st(r_0) with C=green, V=1 and V_=2 // Fail
19. | | ct(r_1) with C=green and V=3
20. | | | st(r_0) with C=green, V=1 and V_=3 // Fail
21. | | | ct(r_1) with C=red and V=2
22. | | | | st(r_2) with C=red, C_=green and V=2
23. | | | | pt(r_2) with C=green and V=2
24. | | | | clt(r_1) with C=green and V=2
25. | | | | st(r_2) with C=red, C_=blue and V=2
26. | | | | pt(r_2) with C=blue and V=2
27. | | | | bt(r_1) with C=blue and V=2
28. | | | | ct(r_1) with C=red and V=3
29. | | | | | st(r_0) with C=red, V=2 and V_=3 // Fail
30. | | | | | ct(r_1) with C=blue and V=3
31. | | | | | | st(r_2) with C=blue, C_=green and V=3
32. | | | | | | pt(r_2) with C=green and V=3
33. | | | | | | clt(r_1) with C=green and V=3
34. | | | | | | st(r_2) with C=blue, C_=red and V=3
35. | | | | | | pt(r_2) with C= and V=3
36. | | | | | | clt(r_1) with C=red and V=3
37. | | | | | | next_it <=> true.
```

Table 2: Trace for the first answer set for $ASPT(P_{col})$

```
:- use_module(library(chr)).
:- chr_constraint sup_0/1, sup_1/1, excl_0/1, excl_1/1,
   next_it/0, q/1, out_q/1, r/1, out_r/1, p/1.
% r_0 = p(s(X)):-p(X),not(q(s(X))
% r_1 = q(s(X)):-p(X),not(r(X))
q(X) \ q(X) <=> true.
p(X) \ p(X) <=> true.
q(X), out_q(X) <=> fail.
r(X), out_r(X) <=> fail.
% st(r_0) =
p(X) ==> sup_0(X).
% st(r_1) =
p(X) ==> sup_1(X).
% ut(r_0) =
q(s(X)) \ sup_0(X) <=> true.
% ut(r_1) =
p(s(X)) \ sup_1(X)<=> true.
% pt(r_0) =
out_r(X) \ sup_0(X) <=> q(s(X)).
% pt(r_1) =
out_q(s(X)) \ sup_1(X)<=> p(s(X)).
% bt(r_0) =
r(X) \ sup_0(X) <=> true.
% bt(r_1) =
q(s(X)) \ sup_1(X) <=> true.
% et(r_0) =
excl_0(X), out_r(X) <=> fail.
% et(r_1) =
excl_1(X), out_q(s(X)) <=> fail.
% clt(r_0) =
r(X) \ excl_0(X) <=> true.
% clt(r_1) =
q(s(X)) \ excl_1(X) <=> true.
next_it, sup_0(X) <=>
   (out_r(X), q(s(X)), next_it ; excl_0(X), next_it).
next_it,sup_1(X) <=>
   (out_q(s(X)), p(s(X)), next_it ; excl_1(X), next_it).
next_it, excl_0(_) <=> fail.
next_it, excl_1(_) <=> fail.
```

Table 3: $ASPT(P_\infty)$

and $\Delta_{cho}(P \cup K_i, I_i, R_i)$ are computed in this order (and when the previous one is empty).

Theorem 3.9 (soundness). *Let P be an ASP program. Let $(CHR\_Program, Initial\_Store) = ASPT(P)$. Let X be a set of atoms considered also as constraints.*

If there exists a CHR$^\vee$ execution under the ω_t semantics of the CHR program CHR_Program with an initial store of constraints Initial_Store and a final store S such that $S \cap \mathcal{A}_P = X$ under the following two conditions:

- *the constraint next_it is introduced into the store of constraints only when it is the only constraint of the goal and*
- *the order of the CHR rules of $ASPT(P)$ for a program ASP P is preserved.*

Then X is an answer set of P.

As far as we know all the (non parallel) implementations of CHR$^\vee$ comply with those two conditions.

4 RELATED WORKS

Some works use also forward chaining of rules that are instantiated as and when required: GASP[26] and more recently OMiGA [11] and alpha[28]. GASP is implemented in Prolog and Constraint Logic Programming over finite domains. Each rule instantiation and propagation is realized by building and solving a CSP. OMiGA is implemented in Java and uses an underlying Rete network for instantiation and propagation. ASPeRiX is based on the ASPeRiX computation and is implemented in C++. alpha blends lazy-grounding and search procedures based on conflict-driven nogood learning (CDNL). In the next section, we do not compare to GASP or OMiGA that also realize grounding on the fly since they are always comparable or dominated by ASPeRiX or alpha.

In [4, 5] , a translation from first-order ASP programs to first-order sentences on finite structures is proposed. The translation is done through an ordered completion, which is a modification of Clark's completion. The solver $GROC$ first translates an ASP program to its ordered completion, then grounds this first-order sentence, and finally calls an SMT solver.

In [3], the solver s(CASP) is presented as a constraint answer set programming solver without grounding phase. It is based on the s(ASP) solver [24] that is a top-down, goal-driven interpreter of ASP programs written in Prolog. The top-down evaluation makes the grounding phase unnecessary. There are many other works about hybridization of ASP and constraints [6, 7, 18, 25] but none treats ASP programs as a CHR programs and none allows to extend ASP with user-defined constraints.

5 EXPERIMENTAL RESULTS

The $ASPT$ translation follows Section 3 plus the must-be-true optimization described in [22] that optimizes the ASPeRiX computation when the body of the rule contains only one negated atom. This translation is implemented in Prolog [8]. This implementation generates some pseudo-code that can generate a CHR(Prolog) program, like for Example 3.6 or Example 3.7, or some CHR(C++) codes. The prototype that generates a CHR(C++) program is called CHR^{ASP}. It translates an ASP program into a CHR program embedded into a C++ host code. This program is given to the CHR compiler CHR++ [8] [9] that produces a C++ program. This last program is compiled with the C++ compiler gcc. CHR++ is an efficient integration of CHR in the C++ programming language, based on and implements most of the well-known optimizations (indexed constraint stores, grounded variables, late storage, etc.). CHR++ is not complete w.r.t. the semantics of the $\mathrm{CHR}^{\vee}$ language but it implements a simple version of the disjunction that is sufficient to encode the choice points of an ASPeRiX computation.

In the following, we give some results of the evaluation of CHR^{ASP} compared to ASPeRiX 0.2.5 and alpha that are the state-of-the-art rule-oriented ASP solvers with a grounding on the fly. We compare also with Clingo 5.2.2 [17] that is the state-of-the-art atom-oriented ASP solver even if we do not want to prove that CHR^{ASP} is an efficient ASP solver but a useful way to integrate the nonmonotonic reasoning into the CHR paradigm and a promising approach for ASP programs with very large sets of facts.

All the systems have been run on an Intel Core-i7-4900MQ with 8 cores at 2.9GHz and about 4GB RAM running Linux Ubuntu 16.04 64 bits. We compute all the answer sets. The time for the first and second steps of the pipeline is negligible (ie. lower than 0.1 second) for CHR^{ASP}. The following table reports the execution time in seconds. The C++ file is compiled in nearly constant time and and this time must be added to the results of the following table for CHR^{ASP} (Schur-* : 18.2 seconds, 3-coloring-wheel-* : 11.1 seconds, n-queens :10.1 seconds, cutedge-*: 20.9 seconds). Rules may be compiled separately from facts only one. For the cutedge family [11] of benchmarks, the number of facts is very large (for example, the cutedge-200-50 instance contains 19958 facts).

The other benchmarks of [22] need some function symbols that we do not have yet in the CHR(C++) version.

On all the benchmarks, CHR^{ASP} is better than ASPeRiX. Clingo and alpha are better than CHR^{ASP} on all the benchmarks but on the cutedge problem. The instances of this problem have huge sets of facts. CHR^{ASP} is better than Clingo and alpha on all the instances of this problem (OM is for "Out of Memory"). Compilation time (20.9 seconds) becomes insignificant compared to the execution time. All the 19957 answer sets of `cutedge-200-50` are computed only by CHR^{ASP} in less than 3600 seconds.

Problem instances	CHR^{ASP}	ASPeRiX	Clingo	alpha
Schur-10	1.1	8.8	<0.1	1.0
Schur-11	4.0	31.4	<0.1	1.0
Schur-12	12.6	108.4	<0.1	1.0
Schur-13	41.5	394.7	<0.1	1.0
3-coloring-wheel-12	3.9	7.7	<0.1	0.7
3-coloring-wheel-13	13.9	25.8	<0.1	0.7
3-coloring-wheel-14	41.3	87.8	<0.1	0.7
7-queens	1.8	46.5	<0.1	0.9
8-queens	20.5	509.1	<0.1	1.4
cutedge-100-30	31.4	210.1	42.2	OM
cutedge-100-50	**109.3**	876.6	170.2	OM
cutedge-100-60	**173.3**	>3600.0	297.5	OM
cutedge-200-30	**897.4**	>3600.0	2246.4	OM
cutedge-200-50	**2678.4**	>3600.0	>3600.0	OM

Table 4: Execution times for some benchmarks

The `cutedge` family is particularly interesting because we think that Clingo is faced with the bottleneck of the grounding phase and alpha is faced with the bottleneck of the conflict-driven nogood learning.

6 ASP CHOICE RULE AS CHR CODE

We show in this section how one can translate a choice rule of ASP in a user-defined CHR constraint. A choice rule allows us to use disjunction in the head of a rule. Choice rules allow us to define a generative space and are what lead to the possibility of multiple answer sets. In [10], the semantics of choice rules is defined by a translation to normal rules.

Due to the lack of space, we only show it for the choice rule (`Min {p(X) : q(X)} Max.`) with no body that indicates that an answer set must contain exactly between `Min` and `Max` atoms of the form `p(X)` when `q(X)` is also into the answer set.

We suppose that there exist the following functions. [10] :

- *init* of arity 3 is such that $init(Min, Max, Size)$ initializes 3 CHR constraints : $size(Size)$ with $Size = 0$, $min(Min)$ and $max(Max)$,
- *inc* of arity 1 is such that $inc(Size) = Size + 1$,
- inf of arity 2 is such that $inf(N, M) = (N < M)$,

[8]The examples of this article may be obtained at http://www.info.univ-angers.fr/pub/stephan/Research/ASP_to_CHR/Examples/ASP_to_CHR.html
[9]it can be downloaded at https://gitlab.com/vynce/chrpp

[10]For swipl, we use the following definitions that use mutable terms of Sicstus Prolog:

```
:- expects_dialect(sicstus).
:- chr_constraint p/1, q/1, choice/2, after/0,
 min/3, max/3, size/3, minimum/0, split/3.
init(Min,Max,Size) :-
 create_mutable(0,Size), size(Size), min(Min), max(Max).
inc(Size) :- get_mutable(N, Size), N_ is N+1, update_mutable(N_, Size).
inf(N,M) :- (integer(N), !, N_=N ; get_mutable(N_, N)),
 (integer(M), !, M_=M ; get_mutable(M_, M)), N_<M_.
equal(N,M) :- (integer(N), !, N_=N ; get_mutable(N_, N)),
 (integer(M), !, M_=M ; get_mutable(M_, M)), N_=M_.
```

- *equal* of arity 2 is such that $equal(N, M) = (N = M)$.

The user-defined CHR constraint *choice* of arity 2 is added to ASP language to represent the choice rule[11]. Since this choice rule has no body, constraint $choice(Min, Max)$ is added to the initial store. This first CHR rule expresses that there cannot be any answer set if there is an atom of the form `q(X)` when there should be none `p(X)` (ie. $Max = 0$).

```
q(_X), choice(_Min,0) <=> fail.
```

The following CHR rules express that the choice rule is awaken when an atom of the form `q(X)` is added to the IN set of an `ASPeRiX` computation or there are already some atoms of the form `q(X)` into the store and the choice rule is introduced into the store. (The `split` CHR constraint realises the possible combinations: in the first part of the disjunction, `p(X)` is added while in the second part, no.)

```
q(X) \ choice(Min,Max) <=> init(Min,Max,Size),
   (inc(Size), p(X), split(X) ; split(X)).
```

The following three CHR rules compute the combinations for the atoms of the form `q(X)` already in the IN set.

```
max(Max), size(Size) \ split(_) <=>
   equal(Max,Size) | after.
split(X), max(Max), q(Y), size(Size) ==>
   inf(Size,Max), X \= Y | inc(Size), p(Y) ; true.
split(_) <=> after.
```

The CHR constraint `after` waits for new atoms of the form `q(X)` to generate new combinations or to stop the process.

```
max(Max), size(Size) \ after <=>
   equal(Max,Size) | true.
q(X), max(Max), after, size(Size) ==>
   inf(Size,Max), inc(Size), p(X) ; true.
```

The two last CHR rules check that at the end of an `ASPeRiX` computation the minimum of atoms of the form `p(X)` is exceeded[12]

```
minimum \ min(Min), size(Size) <=>
   inf(Size,Min) | fail.
minimum \ min(Min), size(Size) <=>
   (inf(Min,Size) ; equal(Min,Size)) | true.
```

7 CONCLUSION

We have presented in this article a translation form ASP to $\mathrm{CHR}^{\vee}$ thanks to `ASPeRiX` computations. This translation offers the possibility to use nonmonotonic reasoning in CHR paradigm but also to add some user-defined constraints in ASP. Since the grounding is done on-the-fly, the approach can compute the finite answer sets of some programs with infinite Herbrand universes. The experimental results show that it is a useful approach since it is a competitive approach compared to one of the rule-oriented state-of-the-art solver with on-the-fly grounding, `ASPeRiX`, and a promising approach for ASP programs with very large sets of facts. We plan to improve and refined our translation and to implement new ASP features like aggregates [27] in CHR. We plan also to compare our approach to the quite different approach that hybridizes Answer Set Programming and Constraint Programming [6, 7, 18, 25] .

[11] In a complete translation, an identifier must be added to distinguish between the different choice rules.

[12] To avoid redundant answer sets, $after$, max, min and $size$ have to be declared as passive.

REFERENCES

[1] S. Abdennadher. 1997. Operational Semantics and Confluence of Constraint Propagation Rules. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*. 252–266.

[2] S. Abdennadher and H. Schütz. 1998. CHR: A Flexible Query Language. In *Proceedings of the 3^{rd} International Conference on Flexible Query Answering Systems*. 1–14.

[3] J. Arias, M. Carro, E. Salazar, K. Marple, and G. Gupta. 2018. Constraint Answer Set Programming without Grounding. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 337–354.

[4] V. Asuncion, Y. Chen, Y. Zhang, and Y. Zhou. 2015. Ordered completion for logic programs with aggregates. *Artificial Intelligence* 224 (2015), 72–102.

[5] V. Asuncion, F. Lin, Y. Zhang, and Y. Zhou. 2012. Ordered completion for first-order logic programs on finite structures. *Artificial Intelligence* 177-179 (2012), 1–24.

[6] M. Balduccini and Y. Lierler. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *Theory and Practice of Logic Programming* 17, 4 (2017), 462–515.

[7] M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17, 4 (2017), 408–461.

[8] V. Barichard and I. Stéphan. 2019. Quantified Constraint Handling Rules. In *Proceedings 35th International Conference on Logic Programming (Technical Communications), (ICLP'19)*. 210–223.

[9] C. Béatrix, C. Lefèvre, L. Garcia, and I. Stéphan. 2016. Justifications and Blocking Sets in a Rule-Based Answer Set Computation. In *Proceedings of the 32nd International Conference on Logic Programming (ICLP'16), Technical communication*. 6:1–6:15.

[10] F. Calmeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub. 2020. ASP-Core-2 Input Language Format. *Theory and Practice of Logic Programming* 20, 2 (2020), 294âĂŞ309. https://doi.org/10.1017/S1471068419000450

[11] M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl. 2012. OMiGA : An Open Minded Grounding On-The-Fly Answer Set Solver. In *Proceedings of the 13th European Conference on Logics in Artificial Intelligence (JELIA'12)*. 480–483.

[12] G.J. Duck, P.J. Stuckey, M.G. de la Banda, and C. Holzbaur. 2004. The Refined Operational Semantics of Constraint Handling Rules. In *Proceedings of the 20th International Conference on Logic Programming (ICLP'04)*. 90–104.

[13] T.W. Frühwirth. 1994. Constraint Handling Rules. In *Constraint Programming: Basics and Trends*. 90–107.

[14] T.W. Frühwirth. 2009. *Constraint Handling Rules*. Cambridge University Press.

[15] T.W. Frühwirth and S. Abdennadher. 2003. *Essentials of Constraint Programming*. Springer-Verlag.

[16] T.W. Frühwirth and F. Raiser (Eds.). 2011. *Constraint Handling Rules: Compilation, Execution, and Analysis*.

[17] M. Gebser, B. Kaufmann, and T. Schaub. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187 (2012), 52–89.

[18] M. Gebser, M. Ostrowski, and T. Schaub. 2009. Constraint Answer Set Solving. In *Proceedings of the 25th International Conference on Logic Programming (ICLP'09)*. 235–249.

[19] M. Gelfond and V. Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference on Logic Programming (ICLP'88)*. 1070–1080.

[20] P. Van Hentenryck. 1991. Constraint logic programming. *Knowledge Engineering Review* 6, 3 (1991), 151–194.

[21] J. Jaffar and M.J. Maher. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming* 19/20 (1994), 503–581.

[22] C. Lefèvre, C. Béatrix, I. Stéphan, and L. Garcia. 2017. *ASPeRix*, a First Order Forward Chaining Approach for Answer Set Computing. *Theory and Pratice of Logic Programming* 17, 3 (2017), 266–310.

[23] C. Lefèvre and P. Nicolas. 2009. A First Order Forward Chaining Approach for Answer Set Computing. In *Proceedings of the 10th Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. 196–208.

[24] K. Marple, E. Salazar, and G. Gupta. 2017. Computing Stable Models of Normal Logic Programs Without Grounding. *CoRR* abs/1709.00501 (2017).

[25] V. Mellarkod, M. Gelfond, and Y. Zhang. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence* 53, 1-4 (2008), 251–287.

[26] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. 2009. Answer Set Programming with Constraints using Lazy Grounding. In *Proceedings of the 25th International Conference on Logic Programming (ICLP'09)*. 115–129.

[27] T.C. Son and E. Pontelli. 2006. A Constructive Semantic Characterization of Aggregates in ASP. *CoRR* (2006).

[28] A. Weinzierl. 2017. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *Proceedings of the 14^{th} Conference on Logic Programming and Nonmonotonic Reasoning - 14th International (LPNMR'17)*. 191–204.