Argument-based Constraint Logic-Programming in Satisfiability Modulo CHR

Francesco Santini

Dipartimento di Matematica e Informatica,
Università di Perugia, Italy
francesco.santini@dmi.unipg.it

Abstract. We revise classical results in Argumentation-based Logic Programming, e.g., Defeasible Logic Programming (DeLP), under the umbrella of Satisfiability Modulo CHR. Through this language it is possible to reason on arguments and have an underlying theory solver to implement resolution of conflicts. Strict and defeasible rules, as well as certainty scores associated with such rules (e.g., Possibilistic DeLP), can be cast to SMCHR rules, which act as conflict "disentanglers". At the same time, we inherit several built-in theory solvers, as unification or linear arithmetic, usable even in combination. SMCHR can be used also to straightforwardly solve Argumentation based on Classical Logic, by selecting a *sat* solver.

1 Introduction

This paper links *Argumentation-based Logic Programming* [10,24,14,28] (*ALP*) to *Satisfiability Modulo Theories* (*SMT*), with the purpose to have declarative and powerful tools to reason in case of conflict, and reach a justifiable conclusion with a support in Argumentation-based reasoning.

To accomplish such goal, we use *Satisfiability Modulo Constraint Handling Rules* (*SMCHR*) [9], which in turn exploits the declarativeness of a rule-based language as CHR [12,13], and binds it to SMT. Solving CHR constraints in other propositional contexts typically relies on some external machinery. For example, Prolog CHR implementations such as K.U.Leuven CHR system [27] use Prolog's default backtracking search to handle disjunction. The execution algorithm for CHR is based on constraint rewriting and propagation over a global store of constraints. CHR solvers are incremental: when a new constraint c is asserted, we check c and the store against the rules in order to find a match. If there is a match, we fire that rule, possibly generating new constraints in the store. Otherwise c is simply added to the global store.

SMT solvers have applications such as program verification, program analysis, model checking, theorem proving, and constraint programming [20]. Most SMT solvers support a mixed set of first-order theories, such as linear arithmetic over the reals, arrays, uninterpreted functions, and so on. SMCHR is much more flexible, as it can support any theory implementable in CHR.

In the following, we first show how to model Argumentation based on Classical Logic [3,4] by using SMCHR, hence exploiting its *Boolean Satisfiability* solver (*sat*). For instance, the notions of *argument* and *defeater* are represented by using SMCHR clauses. Then we provide a general overview on how to program constraint-propagators on top of solvers (as the *sat* one), with the purpose to resolve conflicts between arguments, e.g., $arg \wedge \neg arg$. In general, an *unsatisfiable* result (i.e., the asked goal is not satisfiable) from a solver points to an inconsistency in the knowledge base: such conflict can be overcome by writing an ad-hoc propagator to solve it by removing either *arg* or $\neg arg$ from the constraint store. This decision can be taken by considering qualitative/quantitative preference scores associated with arguments, which define a total/partial order among arguments. A conflict resolution-procedure favours the preferred argument between two. In this sense, an example of naturally weighted frameworks is *Possibilistic Defeasible Logic Programming* (*P-DeLP*) [1], where ALP is mixed with belief scores.

The use of propagators brings us to refer to *Argument-based Constraint Logic Programming* (*ACLP*). Constraint Logic Programming [15] is a form of Constraint Programming [26], where Logic Programming is extended to include concepts from constraint satisfaction. A Constraint Logic Program is a Logic Program that contains constraints in the body of clauses, besides the literals. An example clause is $A(X,Y) :- X > Y, B(X), C(Y)$: where $X > Y$ is a constraint, and $A(X,Y)$, $B(X)$, $C(Y)$ are literals, as in regular Logic Programming. During the evaluation of such programs, encountered constraints are placed in a constraint store; if this set is found to be unsatisfiable, the interpreter backtracks, to find an alternative valid solution.

Several different proposals have been crafted to express ALP in ad-hoc logics and settle such conflicts [10,24,14,28]. One of our goals is to offer the features of SMCHR and propagators as a general means to resolve them. We take as an example the *Defeasible Logic Programming* framework (*DeLP*) [14] with the purpose to show how SMCHR can be used to model and solve various reasoning processes in a particular instance of such logics. For example, we are able *i)* to check the "correctness" of an argument structure (following its definition), *ii)* to check if one argument is the counter-argument of another, and *iii)* if it is a proper or a blocking defeater for it [14]. Then we show that similar considerations hold for the possibilistic extension of DeLP, i.e., P-DeLP.

To summarise, the main motivations behind this paper are to:

- have a unifying solving framework in which to solve all the ALP proposals [10,24,14,28], independently developed;
- link constraint-based representation and solving techniques, as the design of propagators, to help argument-based reasoning in an efficient way. Note in this work we focus on non-Abstract Argumentation Frameworks [10], where, on the contrary, AI-based techniques have been already successfully applied [7]: e.g., *sat*, constraints, Answer Set Programming (ASP);
- design propagators (which collectively implement a "Theory") on top of different built-in solvers, and then to check their satisfiability (from this,

"Satisfiability Modulo Theory"). This unlocks the use of weights, which represent quantitative preferences on arguments, or some additional information to be taken into account during the reasoning. Possibility scores in P-DeLP are such an example. Efficient underlying solvers, as the *bounds* solver or the simplex algorithm (see Sec. 3.2), optimise the search procedure in case of complex debates, e.g., hundreds of arguments (supports and claims) and complex constraints over them. This is clearly not possible by using boolean solvers only, as a "plain" *sat* solver where values are not considered to check satisfaction.

The paper is organised as follows: Section 2 opens the paper by introducing the related work, since our proposal generalises them towards a single framework. Then, in Sec. 3 we summarise the necessary background-notions to understand SMCHR, its rewriting rules, and underlying solvers. Section 4 shows how to model Argumentation based on Classical Logic, while Sec. 5 suggests how constraint propagators can be used to resolve conflicts. Section 6 shows how SMCHR can represent strict and defeasible rules, and some reasoning processes of DeLP, as checking counter-arguments or defeaters. Such processes can be also reproduced in case of weighted rules, i.e., in P-DeLP. Finally, Sec. 7 wraps up the paper and hints directions for future work.

2 Related Work

In this section we revise some of the most important proposals that combine Argumentation with Logic Programming.

One of the first attempt made for integrating Logic Programming and Argumentation is [22], where Donald Nute introduces a formalism called *Logic for Defeasible Reasoning* (*LDR*). The proposed language has three different types of rules: *strict*, *defeasible*, and *defeaters*. Even if LDR is not a defeasible formalism, its implementation in d-Prolog is enhanced with comparison criteria between rules.

In his seminal work [10] on Abstract Argumentation, Dung shows how that argumentation can be viewed as a special form of logic programming with negation as failure, e.g. "a logic program can be seen as a schema to generate arguments". Then, he introduces a general logic-programming based method to generate meta-interpreters for argumentation systems.

Two years later, inspired by legal reasoning, Prakken and Sartor [24] present a semantics (given by a fixed point definition) and a proof theory of a system for defeasible reasoning, where arguments are expressed in a logic-programming language with both strong and default negation. Conflicts between arguments are decided with the help of priorities associated with rules; such priorities can be defeasibly derived as conclusions within the system.

Defeasible Logic Programming (*DeLP*) is another formalism introduced in [14], where Logic Programming is extended with two types of rules: *strict*, and *defeasible*. As in [24], the language encompasses both strong and default negation.

A query is satisfied, that is warranted from a DeLP program, if it is possible to assemble an argument that supports the query and this argument is found to be undefeated. This process implements an exhaustive dialectical analysis that involves the construction of the arguments that support or attack a query.

In [28] the authors formulate a variety of notions of attack for extended logic programs from combinations of undercuts and rebuts; moreover, they define a general hierarchy of argumentation semantics, which is parametrised by the notions of attack chosen by proponent and opponent. The proposed language is proved to correspond to the one in [24]. Finally, they prove the equivalence and inclusion relationships between the semantics, and they examine some essential properties concerning consistency and coherence principles, which relate default and explicit negation.

Since in Sec. 6.1 we also investigate possibilistic frameworks (i.e., P-DeLP [1]), it is worth mentioning other proposals as probabilistic [19], weighted [11,18], fuzzy [16], or more general semiring-based ones [5] (however, the last three proposals are based on Abstract Argumentation).

One more work, supporting the claim that Argumentation theory is a suitable framework for uncertain reasoning, is [21]. The authors propose an approach based on possibilistic ASP. The specification language is able to capture incomplete information and incomplete states of a knowledge-base at the same time. By considering the evidence of each argument (e.g., *maybe*, *likely*, *certain*), it is presented a conflict managing approach between possibilistic arguments, dealing with the inconsistency of a possibilistic knowledge-base.

Other papers where the authors formalise argument-based decision-making under uncertainty are [17,2,6].

3 CHR and Satisfiability Modulo CHR

3.1 Constraint Handlng Rules

Constraint Handling Rules (CHR) [12,13] is essentially a committed-choice language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. CHR rules define simplification of, and propagation over, multi-sets of relations interpreted as conjunctions of constraint atoms. Simplification rewrites constraints to simpler constraints while preserving logical equivalence (e.g. $X < Y, Y < X <=> \mathit{false}$). Propagation adds new constraints, which are logically redundant but may cause further simplification (e.g. $X \leq Y, Y \leq Z ==> X \leq Z$). Repeatedly applying the rules incrementally solves constraints (e.g. $A \leq B, B \leq C, C \leq A$ leads to $A = B \wedge A = C$). In the following we show the formal syntax of such basic rules: r is the optional unique-name of a rule, each H (and $H_k \backslash H_r$) is the (multi-) head of a rule, and it consists in a conjunction of one or more defined constraints indicated by commas ($H = h_1, \ldots, h_n$), G is the guard being a conjunction of built-in atoms, and B the body being a conjunction of constraints:

- **Simplification:** `[r@] H <=> [G|] B.`

– **Propagation:** `[r@] H ==> [G|] B.`
– **Simpagation:** `[r@]` $\mathtt{H}_k$ `\` $\mathtt{H}_r$ `<=> [G|] B.`

The @ symbol assigns a name r to a rule. A constraint (also built-in, as =) is a predicate of First Order Logic. Rules are tried and (in case) fired in the order they are written in the program (from top to bottom). For each rule, one of its head constraints is matched against the last constraint added to the store. Matching succeeds if the constraint is an instance of the head. If matching succeeds and the rule has more than one head constraint, the constraint store is searched for partner constraints that match the other head constraints. A guard is a precondition on the applicability of a rule: it is basically a test that either succeeds or fails. If the firing rule is a simplification rule, the matched constraints are removed from the store and the body of the rule is executed by adding the constraints in the body. Similarly for firing a simpagation rule, except that the constraints that match the head-part preceding \ (i.e., $\mathtt{H}_k$) are kept in the store; a simpagation rule can be seen as a short hand for $\mathtt{H}_k$`,` $\mathtt{H}_r$ `==> B,` $\mathtt{H}_r$. If the firing rule is a propagation rule, its body is executed without removing any constraint. The rule is remembered with the purpose to not fire it again with the same constraints.

Basically, rules are applied to an initial conjunction of constraints (syntactically, a goal) until exhaustion, i.e., until no more change happens. An initial goal is called *query*. The intermediate goals of a computation are stored in the so-called store. A final goal, to which no more rule is applicable, represents the answer (or result) of a computation. Figure 1 shows four rules to reason on the $\leq$ relation: by posting the goal $A \leq B, B \leq C, C \leq A$ to the store we obtain a final store containing the result $\{A = B, A = C\}$.

```
reflexivity   @ X leq X <=> true.
antisymmetry  @ X leq Y, Y leq X <=> X = Y.
transitivity  @ X leq Y, Y leq Z ==> X leq Z.
idempotence   @ X leq Y \ X leq Y <=> true.
```

Fig. 1. Four rules that implement a solver for a less-or-equal constraint.

3.2 Satisfiability Modulo CHR

SMCHR[1] [8,9] is essentially a Satisability Modulo Theories (SMT) solver where a theory T is implemented in CHR.

SMCHR follows the theoretical operational-semantics of CHR. No assumptions should be made about the ordering of rule applications. The SMCHR system also treats deleted constraints differently from CHR. A deleted constraint stays deleted "forever", i.e. it is not possible to re-generate a copy of the same constraint: for instance, a the program $p(x) <=> p(x)$ always terminates.

[1] `http://www.comp.nus.edu.sg/~gregory/smchr/`

The following list introduces all the solvers that can be plugged into SMCHR at the time of writing [8,9]:

- *eq*: an equality solver based on union-find. This solver is complete for (dis-) equality constraints.
- *linear*: a linear arithmetic solver over the integers based on the simplex algorithm over the rationals. This solver is incomplete if there exists a rational solution for the given goal.
- *bounds*: a simple bounds-propagation solver over the integers. This solver is incomplete.
- *dom*: a simple solver that interprets the constraint int $dom(x, l, u)$ as an integer domain/range constraint $x \in [l..u]$. This, in combination with the *bounds* solver, forms a *Lazy Clause Generation* (LCG) finite domain solver. This solver is incomplete.
- *heaps*: a heap solver for program reasoning based on some of the ideas from *Separation Logic*. This solver is complete.
- *sat*: the boolean satisfiability solver. This solver is complete and is always enabled by default.

The SMCHR system assumes all solvers are incomplete, and therefore will always answer "UNKNOWN" even if the goal is satisfiable. However, if the user knows that a given solver combination is complete for a given goal, then the answer UNKNOWN can be re-interpreted as a "SAT" response. SMCHR returns the answer "UNSAT" to indicate that the goal is unsatisfiable. Some performance benefits in using the *sat* solver are that it is possible to inherit all the advantages of no-good clause learning, non-chronological back-jumping, and unit propagation during computation [8].

4 SMCHR and Argumentation based on Classical Logic

In order to formalise argumentation, it is possible to use any logic to define the logic entailment of the claim from the support (e.g., Modal Logic). In this introductory section, the aim is to briefly show how basic concepts in argumentation can be represented in classical logic, and, ultimately, how such problems can be encoded and solved in SMCHR. We represent atoms by lower case roman letters (a, b, c, d, ...), formulas by greek letters (α, β, ...), and use $\wedge$, $\vee$, $\rightarrow$, and $\neg$ to denote the logical connectives as conjunction, disjunction, negation, and implication (respectively). We also use $\bot$ to denote a contradiction. We use $\vdash$ to denote the classical consequence relation, and so if Δ is a knowledge-base (we assume Δ to be a finite set of formulas), and α is a formula, then $\Delta \vdash \alpha$ denotes that Δ entails α. $\Delta \vdash \bot$ denotes that Δ is contradictory (or equivalently inconsistent). Thus, Δ can represent facts, beliefs, views, etc.: the formulas in Δ can represent certain or uncertain information, and they can represent objective, subjective, or hypothetical statements. Indeed, Δ is not expected to be consistent. Definition 1 describes a valid argument.

Definition 1 (Argument). *An argument is a pair $\langle \Phi, \alpha \rangle$ such that 1) $\Phi \nvdash \bot$, 2) $\Phi \vdash \alpha$, and 3) Φ is a minimal subset of Δ satisfying 2.*

Argument supports are in Δ (i.e., $\Phi \subseteq \Delta$), while a claim α is generally not in Δ. For instance, we can have $\Delta = \{a, a \to b, c \to \neg b, c, d, d \to b, \neg a, \neg c\}$, and an argument $\langle \Phi, \alpha \rangle = \langle \{a, a \to b\}, b \rangle$.

Condition 1 in Def. 1 can be checked in SMCHR by invoking `./smchr` and just verifying the solvability of Φ. Asking the goal $\mathtt{a} \wedge (\mathtt{a} \to \mathtt{b})$ returns $\mathtt{a} \wedge \mathtt{b}$ and UNKNOWN as result, which, being *sat* the complete solver loaded by default, means that $\Phi \nvdash \bot$. Condition 2 in Def. 1 can be verified by passing $\Phi \wedge \alpha$, i.e., $\mathtt{a} \wedge (\mathtt{a} \to \mathtt{b}) \wedge \mathtt{b}$, which prints the same outcome as for 1). Therefore, $\Phi \vdash \alpha$, i.e., $\mathtt{a} \wedge (\mathtt{a} \to \mathtt{b}) \vdash \mathtt{b}$. Minimality (i.e., condition 3) is not a mandatory requirement in general: here we suppose reasons for a claim to be exactly identified [25, pp. 137], which means that they do not incorporate irrelevant information. However, a rough procedure to check it is to repeat checking condition 2 after removing each clause-item from the support (see Sec. 7).

Informally, an argument that disagrees with another argument is described as a counter-argument, thus highlighting points of contention. In logic-based approaches, *defeaters* are arguments whose claim refutes the support of another argument [22,24].[2] Formally:

Definition 2 (Defeater). *A defeater for an argument $\langle \Phi, \alpha \rangle$ is an argument $\langle \Psi, \beta \rangle$ such that $\beta \vdash \neg(\phi_1 \wedge \cdots \wedge \phi_n)$ for some $\{\phi_1, \ldots, \phi_n\} \subseteq \Phi$.*

Let $\Delta = \{\neg a, a \vee b, a \leftrightarrow b, c \to a\}$. Then, $\langle \{a \wedge b, a \leftrightarrow b\}, a \wedge b \rangle$ is a defeater for $\langle \{\neg a, c \to a\}, \neg c \rangle$. Checking if $\langle \Psi, \beta \rangle$ is a defeater of $\langle \Phi, \alpha \rangle$ can be accomplished in SMCHR by invoking the `./smchr` interpreter and just asking the goal $(\mathtt{a} \wedge \mathtt{b}) \wedge (\neg\, \mathtt{a}) \wedge (\mathtt{c} \to \mathtt{a})$, i.e., $\Psi \wedge \beta$. Being UNSAT the returned result, we can conclude that $(\mathtt{c} \to \mathtt{a}) \vdash \neg((\mathtt{a} \wedge \mathtt{b}) \wedge (\neg\, \mathtt{a}))$.

5 An Overview of the Approach

In this section we suggest a general approach about how to use SMCHR to solve conflicts like: $claim \wedge \neg claim$.

While the previous section only takes advantage of the *sat* solver, here we program ad-hoc CHR propagators (still used in conjunction with the underlying *sat* solver) in order to resolve conflicts between two conflicting claims: a propagator is expressed as a rule like $claim \wedge \neg claim \Rightarrow \mathbf{?}$, where **?** can be decided by considering different factors, as a certainty score associated with *claim* and $\neg claim$ [1], or taking into account if claims are the outcome of strict or defeasible (or weak) rules (see Sec. 6 and Sec. 2). Such propagators are used by the solver to compute a fixed point of constraint propagation. Therefore, we detach

[2] In some frameworks, a defeater can also refute the claim of an argument or the rationale being used (e.g., logical connections in the support are not clear or sound), not only its support.

the Theory for Argumentation from the solver beneath, which can be selected among those in Sec. 3.2.

We show how SMCHR can represent information in the form of strict and weak rules in a declarative manner. Weak rules are the key element for introducing defeasibility [23], and they are used to represent a relation between pieces of knowledge that could be defeated when everything has been considered. For instance, we can mark weak rules by tagging the produced claims with $@w$ (@ is used for SMCHR atom-constants):

$$supp_1 \wedge supp_2 \Longrightarrow claim(@w)$$

while a tag $@s$ points to a strict rule instead:

$$supp_3(X) \Longrightarrow X >= 4 \mid \neg claim(@s)$$

$X >= 4$ is a guard, and it constrains the firing of this rule when variable X is greater than 4 only. Afterwards, we need to envisage rules to resolve possible conflicts in a store of constraints. By firing the previous two rules, then the store contains both *claim* and $\neg$*claim* (one weak and one strict), that is an argument and its counter-argument at the same time.

$$claim(X) \wedge \neg claim(Y) \Longleftrightarrow X = @w \wedge Y = @s \mid \neg claim(Y)$$
$$claim(X) \wedge \neg claim(Y) \Longleftrightarrow X = @s \wedge Y = @w \mid claim(X)$$

In this case, our constraint store contains both *claim* labelled with "weak" and $\neg$*claim* labelled with "strict", the first between the two rules will be fired and the result will be the deletion of *claim* from the store, and the keeping of $\neg$*claim*.

Finally we consider the classical example described in Fig. 2, where we list facts and strict/weak rules (i.e., $\rightarrow_{\mathrm{s}}$ and $\rightarrow_{\mathrm{w}}$ respectively).

$bird(tweety)$	$penguin(tweety)$
$bird(X) \rightarrow_{\mathrm{w}} fly(X)$	$penguin(X) \rightarrow_{\mathrm{s}} \neg fly(X)$

Fig. 2. The classical Tweety example.

In Fig. 3 we implement a possible conflict resolution in SMCHR for the example in Fig. 2. By querying $bird(tweety) \wedge penguin(tweety)$, the generated final constraint-store is $bird(tweety) \wedge penguin(tweety) \wedge \neg fly(tweety, @s)$. Hence we correctly obtain that *tweety* is not capable of flying.

6 DeLP Solved in SMCHR

In this section we choose DeLP [14] among all Argument-based Logic Programming frameworks (see Sec. 2) in order to show how SMCHR propagators can

```
bird(x) ⟹ fly(x,@w);
penguin(x) ⟹ not fly(x,@s);
fly(x,b) ∧ not fly(x,c) ⟹ b = @w, c = @s | not fly(x,c);
```

Fig. 3. SMCHR rules coding the Tweety example.

effectively model such plethora of systems. DeLP variables are represented with constraint variables. A DeLP program is a set of *i)* facts, *ii)* strict rules, and *iii)* defeasible rules.

- *Facts* are ground literals representing atomic information or its negation.
- *Strict rules* represent non-defeasible rules, and they are represented as $L_0 \leftarrow L_1, \ldots, L_n$.
- *Defeasible rules* represent tentative information, in the form of rules like $L_0 \leftarrowtail L_1, \ldots, L_n$.

In words, a defeasible rule is used to represent tentative information that may be used if nothing could be posed against it. On the contrary, the information that represents a strict rule, or a fact, is not tentative. A DeLP-program is denoted by a pair (Π, Δ) distinguishing a subset Π of facts and (only two) strict rules, which represent non-defeasible knowledge, and a subset Δ of (eight) defeasible rules. In Fig. 4 we consider the same example given in [25, Ch. 2]. $\Pi \cup \Delta$ collect information and reason on three rooms a, b, and c, linking their illumination to the day (working or holiday), to the switch position (on/off), to the electricity presence (yes/no), and to the time of the day (day/night): for instance, the first defeasible rule states that "it is reasonable to believe that if the switch of a room is on, then the lights of that room are on". This rule is in Δ because its conclusion can be defeated in case, for instance, there is no electricity.

In Fig. 5 we show a possible encoding to SMCHR of the DeLP program in Fig. 4. All the rules in Fig. 4 (both in Π and Δ) are encoded in a SCMHR propagation-rule in Fig. 5, one by one. Facts, which represent pieces of indefeasible information (i.e., they are in Π as well), are not represented through rules, but with the constraints

$$switchOn(a), switchOn(b), switchOn(c), not\ electricity(b), not\ electricity(c), emergencyLights(c), night(a), night(b), night(c), sunday(a), sunday(b), sunday(c), deadline(a), deadline(b), deadline(c).$$

Such set of F constraints (or part of it) can be passed to the SMCHR interpreter (in "and") as part of a global goal. If we ask the entire F we obtain an UNSAT result, meaning that not all of them can be warranted, i.e., we have contradictions in our knowledge-base.

A DeLP-query is a ground literal Q that a DeLP program tries to warrant. Our SMCHR can straightforwardly check it. There are several queries that succeed with respect to the program in Fig. 5 because they are warranted, e.g., $illuminated(a) \wedge switchOn(a)$. File `argCheck.chr` stores our "Theory" as a set

$$\Pi \begin{cases} night. & switch\_on(a). \\ \sim day \leftarrow night. & switch\_on(b). \\ \sim dark(Y) \leftarrow illuminated(X). & switch\_on(c). \\ sunday. & \sim electricity(b). \\ deadline. & \sim electricity(c). \\ & emergency\_lights(c). \end{cases}$$

$$\Delta \begin{cases} light\_on(X) \prec switch\_on(X). \\ \sim lights\_on(X) \prec \sim electricity(X). \\ lights\_on(X) \prec \sim electricity(X), emergency\_lights(X). \\ dark(X) \prec \sim day. \\ illuminated(X) \prec \sim lights\_on(X), \sim day. \\ working\_at(X) \prec illuminated(X). \\ \sim working\_at(X) \prec sunday. \\ working\_at(X) \prec sunday, deadline. \end{cases}$$

Fig. 4. Π collects facts and strict rules in DeLP, while Δ provides defeasible rules.

```
/* Strict rules */
night(x) ⟹ not day(x);
illuminated(x) ⟹ not dark(x);

/* Defeasible rules */
switchOn(x) ⟹ lightsOn(x);
not electricity(x) ⟹ not lightsOn(x);
not electricity(x) ∧ emergencyLights(x) ⟹ lightsOn(x);
not day(x) ⟹ dark(x);
not day(x) ∧ lightsOn(x) ⟹ illuminated(x);
illuminated(x) ⟹ workingAt(x);
sunday(x) ⟹ not workingAt(x);
sunday(x) ∧ deadline(x) ⟹ workingAt(x);
```

Fig. 5. *argcheck.chr*: a SMCHR propagator coding the DeLP example in Fig. 4.

of propagation rules, and it is shown in Fig. 5. If we call `./smchr --solver argCheck.chr` and we ask the goal $Q = illuminated(a) \wedge switchOn(a)$ then we obtain UNKNOWN, which, being the default *sat* solver complete, can be reinterpreted as SAT (Q is then warranted). The output of the `smchr` interpreter is shown in Fig. 11, where we can also see the number of generated constraints (three: $lightsOn(a)$, $not\ dark(a)$ and $workingAt(a)$), and other information related to search, as the number of backtracks. Other queries cannot be warranted instead: for instance $Q = switchOn(a) \wedge not\ lightsOn(a)$ returns UNSAT, because $switchOn(a)$ propagates to $lightsOn(a)$, which however conflicts with part of Q. Indeed, also two contradictory constraint in a query, as *day(a)* $\wedge$ *not day(a)*, trivially disagree, and the answer is UNSAT as well.

A derivation using Π only is called a *strict derivation*, that is only the first two rules in Fig. 5. A *defeasible derivation* of a literal Q by using (Π, Δ), denoted by $(\Pi, \Delta) \mathrel{|\!\sim} Q$, is a finite sequence of ground literals in form of $L_1, L_2, \ldots, L_n$ with $L_n = Q$, where *i)*, L_i is a fact in Π, or *ii)* there exists a strict or defeasible

```
LOAD solver "sat"
LOAD solver "argcheck.chr"

> illuminated(a) /\switchOn(a)
UNKNOWN:
illuminated(a) /\
lightsOn(a) /\
not dark(a) /\
switchOn(a) /\
workingAt(a)
TIME 0
CONSTRAINTS 3
BACKTRACKS 0
CLAUSES 3
DECISIONS 0
PIVOTS 0
```

Fig. 6. Output of `archCheck.chr` with $illuminated(a)$ as query.

rule in (Π, Δ) with head L_i and body $B_1, \ldots, B_k$ and each B is an L_j element of $L_1, L_2, \ldots, L_n$, with $j < i$. A derivation is defeasible if at least one defeasible rule is used. This brings us to define a valid argument structure:

Definition 3 (Argument Structure [14]). *Let H be a ground literal, (Π, Δ) a DeLP-program, and $A \subseteq \Delta$. The pair $\langle A, H \rangle$ is an argument structure if:*[3]

1. *there exists a defeasible derivation for H from (Π, A),*
2. *there is no defeasible derivation from (Π, A) of contradictory literals.*

With respect to Fig. 4, if we consider $A = \{\sim lights\_on(X) \leftarrow\!\!\!\prec \sim electricity(X)\}$ and $H = \sim lights\_on(b)$, we can check if $\langle A, H \rangle$ is a valid argument structure by using the SMCHR rules in Fig. 7, which collect all the strict rules in Π plus A, as required by 2) in Def. 3. If we set as goal all the facts F in Π, i.e.,

$$Q = F = night(a) \wedge night(b) \wedge night(c) \wedge switchOn(a) \wedge switchOn(b) \wedge switchOn(c) \wedge sunday(a) \wedge sunday(b) \wedge sunday(c) \wedge deadline(a) \wedge deadline(b) \wedge deadline(c) \wedge not\ electricity(b) \wedge not\ electricity(c) \wedge emergencyLights(c)$$

then the obtained output is UNKNOWN and the new (six) constraints generated in the store are

$$defeasibleNotLightsOn(b), defeasibleNotLightsOn(c), not\ day(a), not\ day(b), not\ day(c), strictLightsOn(b).$$

Having constraint $defeasibleNotLightsOn(b)$ in the store means that we generated $H = \sim lights\_on(b)$ by using at least one defeasible rule. Thus, 1) in Def. 3 is satisfied, as well as 2), since a result of UNKNOWN (i.e., SAT because the solver is complete) means that no contradictory literals have been generated. Note that the same holds for $H = \sim lights\_on(c)$, as it can be appreciated from the same final store of constraints above.

[3] We do not consider here a third property, i.e., that there is no proper subset A' of A such that A' satisfies 1) and 2).

```
/* Strict rules */
night(x) ⟹ not day(x);
illuminated(x) ⟹ not dark(x);

/* Defeasible rules */
not electricity(x) ⟹ not lightsOn(x) ∧ defeasibleNotLightsOn();

/*1*/ defeasibleNotLightsOn(x) ∧ lightsOn(x) ⟹ strictLightsOn(x);
/*2*/ defeasibleNotLightsOn(x) ∧ defeasibleLightsOn(x) ⟹ false;
```

Fig. 7. A SMCHR program to check the validity of an argument structure (see Def. 3) by using (Π, A), where $A = \{\sim lights\_on(X) \leftarrowtail \sim electricity(X)\}$ and Π is supposed as taken from Fig. 4.

Rule 2 in Fig. 7 is used to have an UNSAT response in case there are two defeasible derivations leading to the contradiction that lights are on and off at the same time. Therefore, we use this rule to check property 2) in Def. 3. Rule 1 in Fig. 7 is used to add a constraint (in this case, $strictLightsOn(b)$) warning that there is also a strict derivation contradicting our defeasible derivation for b. If we wish to remove such a conflict, we only need to add the following (simpagation) rule to Fig. 7, which can remove the constraint $strictNotLightsOn(b)$ from the store:

```
strictLightsOn(x) \ defeasibleNotLightsOn(x) <=> strictLightsOn(x);
```

Now we can turn our attention to model and check counter-arguments and defeaters in DeLP:

Definition 4 (Counter-Argument [14]). *$\langle B, S\rangle$ is a counter-argument for $\langle A, H\rangle$ at literal P, if there exists a sub-argument $\langle C, P\rangle$ of $\langle A, H\rangle$ such that P and S disagree, that is, there exist two contradictory literals that have a strict derivation from $\Pi \cup \{S, P\}$.*

Consider two valid argument-structures $\langle \{illuminated(X) \leftarrowtail \sim lights\_on(X), \sim day, light\_on(X) \leftarrowtail switch\_on(X)\}, illuminated(b)\rangle$ and $\langle \{dark(X) \leftarrowtail \sim day\}, dark(b)\rangle$. We can accomplish this check by writing a new SMCHR program with the first two rules of Fig. 5; then we ask a query $Q = F \wedge \{illuminated(b), dark(b)\}$, which contains all the facts F in Π in conjunction with $S \cup P$. As answer we get UNSAT, meaning that one is the counter-argument of the other (obtained by only using strict derivations).

In DeLP the argument comparison criterion between two arguments is modular [14]. For this reason, Def. 5 abstracts away from the comparison criterion, assuming there exists one (denoted by $\succ$):

Definition 5 (Proper/Blocking Defeaters [14]). *Let $\langle B, S\rangle$ be a counter-argument for $\langle A, H\rangle$ at point P, and $\langle C, P\rangle$ the disagreement sub-argument. If $\langle B, S\rangle \succ \langle C, P\rangle$ (i.e., $\langle B, S\rangle$ is "better" than $\langle C, P\rangle$) then $\langle B, S\rangle$ is a proper defeater for $\langle A, H\rangle$. If $\langle B, S\rangle$ is unrelated by the preference relation to $\langle C, P\rangle$, (i.e., $\langle B, S\rangle \not\succ \langle C, P\rangle$, and $\langle C, P\rangle \not\succ \langle B, S\rangle$) then $\langle B, S\rangle$ is a blocking defeater for $\langle A, H\rangle$.*

```
type night(var of atom, num); type illuminated(var of atom, num); type
    dark(var of atom, num);

illuminated(x, y) \ dark(x, z) <=> y $> z ∧ r:= (y − z) | not dark(x,
    r) ∧ pDefeaterIlluminatedDark(x);
illuminated(x, y) \ dark(x, z) <=> y $= z ∧ r:= (y − z) | not dark(x,
    r) ∧ bDefeaterIlluminatedDark(x);
```

Fig. 8. *dark* ∧ *not dark*: resolving conflicts with scores.

In SMCHR, preference can be computed and/or constrained in the guard of a rule, thus allowing us to even represent dynamic preferences, i.e. preferences that are subject to some conditions, as suggested in [24] or in implemented systems as GORGIAS[4]. Therefore, in the approach adopted below we opt to associate a score with each argument, and we compute a new numeric result by resolving a conflict, for instance subtracting the strength of a support from another:

$$supp(X) \wedge \neg supp(Y) <=> X >= Y \wedge Z := X - Y \mid claim(Z)$$
$$supp(X) \wedge \neg sup(Y) <=> X < Y \wedge Z := Y - X \mid \neg claim(Z)$$

In these two rules, contradictory constraints $supp(X)$ and $supp(Y)$ are the same argument but with a different preference score. They are removed from the store through a simplification rule, and the result is the a claim with a different preference score Z, computed in the guard of the rules. If $X \geq Y$ ($ >= in SMCHR), *supp* wins over $\neg$*supp*, otherwise the store contains $\neg$*claim*. In both of the cases, the final preference Z is the difference between X and Y.

If we import such conflict-resolution method in our running-case (Fig. 4), we can model it via the program in Fig. 8. The type of constraints is defined at the beginning of Fig. 8 in order to let the program correctly manage operations on their arguments. We set the query Q to $dark(b, 2) \wedge illuminated(b, 3)$. Hence, argument $illuminated(b)$ is associated with a preference value equal to 3, and $dark(b)$ to 2. The program in Fig. 8 states that $illuminated(b)$ propagates to *not* $dark(b)$, if the first preference score is greater/equal than the preference score of $dark(b)$, and the preference of *not* $dark(b)$ is their difference. The first rule generates a proper defeater in the store when the preference is strictly better. If the two scores are the same, the second rule generates a blocking defeater instead. With our query Q, we fire the first rule and we obtain $illuminated(b, 3) \wedge$ *not* $dark(b, 1) \wedge pDefeaterIlluminatedDark(b)$ in the store.

6.1 A Bridge from Constraints Towards P-DeLP

Finally, we show how SMCHR and constraints have an impact on weighted extensions of ALP (see also Sec. 2). *Possibilistic Defeasible Logic Programming* (*P-DeLP*) [1] is an extension of DeLP in which defeasible rules are attached with weights, belonging to the real unit interval [0..1], in the following discretised to

[4] `http://www.cs.ucy.ac.cy/~nkd/gorgias/`

```
type sw1(num); type sw2(num); type sw3(num); type pumpClog(num); type
    pumpFuel(num); type pumpOil(num); type oilOk(num); type fuelOk(num);
     type engineOk(num); type heat(num); type lowSpeed(num);

/* Strict */
pumpClog(x) ⟹ not fuelOk(x);

/* Defeasible */
sw1(x) ⟹ x $<= 60 | pumpFuel(x);
sw1(x) ⟹ x $> 60 | pumpFuel(60);
pumpFuel(x) ⟹ x $<= 30 | fuelOk(x);
pumpFuel(x) ⟹ x $> 30 | fuelOk(30);
sw2(x) ⟹ x $<= 80 | pumpOil(x);
sw2(x) ⟹ x $> 80 | pumpOil(80);
pumpOil(x) ⟹ x $<= 80 | oilOk(x);
pumpOil(x) ⟹ x $> 80 | oilOk(80);
oilOk(x) ∧ fuelOk(y) ⟹ x $<= y ∧ x $<= 30 | engineOk(x);
oilOk(x) ∧ fuelOk(y) ⟹ y $<= x ∧ y $<= 30 | engineOk(y);
oilOk(x) ∧ fuelOk(y) ⟹ 30 $<= x ∧ 30 $<= y | engineOk(30);
heat(x) ⟹ x $<= 95 | not engineOk(x);
heat(x) ⟹ x $> 95 | not engineOk(95);
heat(x) ⟹ x $<= 90 | not oilOk(x);
heat(x) ⟹ x $> 90 | not oilOk(x);
lowSpeed(x) ∧ pumpFuel(y) ⟹ x $<= y ∧ x $<= 70 | pumpClog(x);
lowSpeed(x) ∧ pumpFuel(y) ⟹ y $<= x ∧ y $<= 70 | pumpClog(y);
lowSpeed(x) ∧ pumpFuel(y) ⟹ 70 $<= x ∧ 70 $<= y | pumpClog(70);
sw2(x) ⟹ x $<= 80 | lowSpeed(x);
sw2(x) ⟹ x $> 80 | lowSpeed(80);
sw3(x) ∧ sw2(y) ⟹ x $<= y ∧ x $<= 80 | not lowSpeed(x);
sw3(x) ∧ sw2(y) ⟹ y $<= x ∧ y $<= 80 | not lowSpeed(y);
sw3(x) ∧ sw2(y) ⟹ 80 $<= x ∧ 80 $<= y | not lowSpeed(80);
sw3(x) ⟹ x $<= 60 | fuelOk(x);
sw3(x) ⟹ x $> 60 | fuelOk(80);
```

Fig. 9. SMCHR rules coding the RP-DeLP example in [1].

[0..100] since SMCHR works with integer numbers only. Such score expresses the relative belief or preference strength of arguments. Each fact p_i is associated with a certainty value that expresses how much the relative fuzzy-statement is believed in terms of necessity measures. Weights are aggregated in accordance to $(p_1 \wedge \cdots \wedge p_k \rightarrow q, \alpha)$ iff $(p_1, \beta_1), \ldots, (p_k, \beta_k)$ with $(q, \min(\alpha, \beta_1, \ldots, \beta_k))$. Such computational evaluation can be naturally encoded into SMCHR, as we show in the following example.

The program in Fig. 9 encodes in SMCHR an example provided in [1]. We suppose to have an intelligent agent controlling an engine with three switches *sw1*, *sw2* and *sw3*. These switches regulate different features of the engine, such as the pumping system, speed, etc. Figure 9 shows certain and uncertain knowledge an agent has about how this engine works.

By querying $sw1(100)$ we obtain UNKNOWN and $fuelOk(30)$, thus correctly deriving *pumpFuel* with a certainty score equal to 0.3, that is the minimum value among all the constraints in the store: $sw1(100)$, $pumpFuel(60)$, and $fuelOk(30)$.

By switching the first two switches on, i.e., $sw1(100) \wedge sw2(100)$, the agent knows that the engine works with a certainty score equal to 0.3. The result is UNKNOWN, and the final constraint store is:

$$engineOk(30) \wedge fuelOk(30) \wedge lowSpeed(80) \wedge not\ fuelOk(60) \wedge oilOk(80) \wedge$$

```
type goal1(var of num, var of num, num); type goal2(var of num, num);
    type goal3(var, num);

goal1(x, y, c)  ==> not int_gt(x, y) ∧ not int_gt_c(y,c);
goal2(z, d) ==> int_gt_c(z,d);
goal1(x, y, c) \ goal2(z, d) ∧ int_gt_c(z,d)  <==> d $>= c ∧ k:= c - 2 |
    goal3(z,k) ∧ int_gt_c(z,k);
```

Fig. 10. An example of negotiation using *goal* arguments and the *bounds* propagator.

$$pumpClog(60) \wedge pumpFuel(60) \wedge pumpOil(80) \wedge sw1(100) \wedge sw2(100)$$

6.2 Argumentation and solvers different from SAT

With a small example, in this section we would like to justify the use of solvers different from *sat*, and consequently justify why it is interesting to keep the Theory and the satisfiability as separated, i.e., why to use SMCHR. In the example in Fig. 10 we show two arguments supporting different goals, that is *goal1* and *goal2*. If they are in conflict, *goal2*, belonging to a different agent, is withdrawn, and *goal3* is added, thus reaching a final (consistent) conclusion. The solution is found by calling `./smchr.macosx --solver ex.chr,bounds`: therefore, we add a (incomplete) *bounds* solver, in order to bind the solution variables. The arguments support constraints on the variables, which are object of negotiation: for instance, *int_gt(x,y)* imposes $x > y$, and *int_gt_c(x,c)* imposes that x has to be greater than a constant c. These are two examples of several primitive built-in constraints directly supported by SMCHR. If *goal2* supports that $y > d$ and *goal1* supports $y \not> c$, if $d > c$ a conflict arises. This conflict is resolved by the third rule in Fig. 10: *goal2* is withdrawn from the store and a new argument *goal3* is added, supporting the constraint $y > c - 2$. A possible output is shown in Fig. 3.1, using $Q = goal1(x, y, 5) \wedge goal2(y, 9)$.

```
> goal1(x,y,5) /\ goal2(y,9)
UNKNOWN:
y > 3 /\
goal1(x,y,5) /\
goal3(y,3) /\
int_lb(y,4) /\
not x > y /\
not y > 5 /\
not int_lb(x,6) /\
not int_lb(y,6)
```

Fig. 11. A possible output for the program in Fig 10, given $Q = goal1(x, y, 2) \wedge goal2(y, 9)$. $int\_lb(y, 6)$ states that 4 is the lower (reachable) bound of y, and not $int\_lb(x, 6)$ and *not* $int\_lb(y, 6)$ are the (unreachable) upper bounds of x and y respectively.

7 Conclusion

We have presented how a constraint propagator as SMCHR can be used to fast prototyping different reasoning problems linked to Argumentation-based Logic Programming. The use of constraints becomes interesting when resolving conflicts depends on relations among arguments and/or their preference value, as in P-DeLP. Such methodology can use different solvers, e.g. *sat* or *bounds*. The ideas in this paper suggest the potentiality of having such a powerful declarative tool, paving the way for Argumentation-based Constraint Logic Programming.

The future goal is to have an automatised SMCHR-based framework where to model also dynamic reasoning over argumentation lines (where each argument structure in a sequence is a defeater of the predecessor), and dialectical trees (where each path from the root to a leaf corresponds to a different acceptable argumentation line). Clearly, arguments can be iteratively added to such tree during a debate. Some reasoning side-procedures, as checking the minimality of an argument support, can be programmed on top of SMCHR by, for instance, embedding SMCHR into an imperative language.

References

1. Alsinet, T., Chesñevar, C.I., Godo, L., Simari, G.R.: A logic programming framework for possibilistic argumentation: Formalization and logical properties. Fuzzy Sets and Systems 159(10), 1208–1228 (2008)
2. Amgoud, L., Prade, H.: Using arguments for making decisions: A possibilistic logic approach. In: UAI '04, Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence. pp. 10–17. AUAI Press (2004)
3. Besnard, P., Hunter, A.: A logic-based theory of deductive arguments. Artificial Intelligence 128(1-2), 203–235 (2001)
4. Besnard, P., Hunter, A.: Elements of Argumentation. The MIT Press (2008)
5. Bistarelli, S., Santini, F.: A common computational framework for semiring-based argumentation systems. In: ECAI 2010 - 19th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 131–136. IOS Press (2010)
6. Bonet, B., Geffner, H.: Arguing for decisions: A qualitative model of decision making. In: Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence. pp. 98–105. UAI'96 (1996)
7. Charwat, G., Dvoŕák, W., Gaggl, S.A., Wallner, J.P., Woltran, S.: Methods for solving reasoning problems in abstract argumentation: A survey. Artificial Intelligence 220(0), 28 – 63 (2015)
8. Duck, G.J.: SMCHR: Satisfiability modulo constraint handling rules. TPLP 12(4-5), 601–618 (2012)
9. Duck, G.J.: Satisfiability modulo constraint handling rules (extended abstract). In: IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence (2013)
10. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artificial Intelligence 77(2), 321–358 (1995)

11. Dunne, P.E., Hunter, A., McBurney, P., Parsons, S., Wooldridge, M.: Weighted argument systems: Basic definitions, algorithms, and complexity results. Artificial Intelligence 175(2), 457–486 (2011)
12. Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. 37(1-3), 95–138 (1998)
13. Frühwirth, T.W.: Constraint Handling Rules. Cambridge University Press, New York, NY, USA, 1st edn. (2009)
14. García, A.J., Simari, G.R.: Defeasible logic programming: An argumentative approach. Theory Pract. Log. Program. 4(2), 95–138 (Jan 2004)
15. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 111–119. POPL '87, ACM (1987)
16. Janssen, J., De Cock, M., Vermeir, D.: Fuzzy argumentation frameworks. In: Information Processing and Management of Uncertainty in Knowledge-based Systems. pp. 513–520 (2008)
17. Krause, P., Ambler, S., Elvang-Goransson, M., Fox, J.: A logic of argumentation for reasoning under uncertainty. Computational Intelligence 11(1), 113–131 (1995)
18. Leite, J., Martins, J.: Social abstract argumentation. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three. pp. 2287–2292. IJCAI'11, AAAI Press (2011)
19. Li, H., Oren, N., Norman, T.J.: Probabilistic argumentation frameworks. In: Proceedings of the First International Conference on Theory and Applications of Formal Argumentation. pp. 1–16. TAFA'11, Springer-Verlag (2012)
20. Moura, L.D., Bjørner, N.: Satisfiability modulo theories: Introduction and applications. Commun. ACM 54(9), 69–77 (Sep 2011)
21. Nieves, J.C., Cortés, U., Osorio, M.: Possibilistic-based argumentation: An answer set programming approach. In: Proceedings of the 2008 Mexican International Conference on Computer Science. pp. 249–260. ENC '08, IEEE Computer Society (2008)
22. Nute, D.: Defeasible reasoning: a philosophical analysis in prolog. In: Aspects of Artificial Intelligence, pp. 251–288. Springer (1988)
23. Pollock, J.L.: Cognitive Carpentry: A Blueprint for How to Build a Person. MIT Press, Cambridge, MA, USA (1995)
24. Prakken, H., Sartor, G.: Argument-based extended logic programming with defeasible priorities. Journal of applied non-classical logics 7(1-2), 25–75 (1997)
25. Rahwan, I., Simari, G.R.: Argumentation in Artificial Intelligence. Springer Publishing Company, Incorporated, 1st edn. (2009)
26. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
27. Schrijvers, T., Demoen, B.: The k.u.leuven chr system: Implementation and application. In: First Workshop on Constraint Handling Rules: Selected Contributions. pp. 1–5 (2004)
28. Schweimeier, R., Schroeder, M.: A parameterised hierarchy of argumentation semantics for extended logic programming and its application to the well-founded semantics. Theory Pract. Log. Program. 5(1-2), 207–242 (Jan 2005)