Representing Argumentation Schemes with Constraint Handling Rules (CHR)

Thomas F. Gordon, Horst Friedrich
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
Berlin, Germany

ABSTRACT

We present a high-level declarative programming language for representing argumentation schemes, where schemes represented in this language can be easily validated by domain experts, including developers of argumentation schemes in informal logic and philosophy, and serve as *executable specifications* for automatically constructing arguments, when applied to a set of assumptions. Since argumentation schemes are defeasible inference rules, both premises and conclusions of schemes can be second-order schema variables, i.e. without a fixed predicate symbol. Thus, while particular schemes can be and have been implemented in computer programs, in general argumentation schemes cannot be represented as executable specifications using logic programming languages based on first-order logic, such as Prolog. Moreover, even if the conclusion (head) of Prolog rules could be second-order variables, a depth-first, backward-chaining search strategy, as typically used in logic programming, would usually cause such programs to not terminate, since every goal would match the head of such a scheme, including all goals created by instantiating the body of the same scheme. The language for representing argumentation schemes presented here, for the purpose of automatically constructing arguments, uses Constraint Handling Rules (CHR), a declarative, Turing complete, forwards-chaining, rule-based programming language introduced by Thom Fruhwirth in 1991. We also report on a legal pilot application, called DUCK, which uses the system in a web application to help cloud service providers comply with the new European General Data Protection Regulation (GDPR).

KEYWORDS

argumentation schemes, argument generation, rule-based systems, Constraint Handling Rules, logic programming, data protection

1 INTRODUCTION

Argumentation schemes [23] serve at least two functions:

(1) They provide normative standards for critically evaluating arguments, by matching arguments to schemes to see if they fit acceptable patterns of argumentation, to identify missing premises, and to facilitate the asking of critical questions.
(2) They provide guidance for making (constructing, inventing, generating) good arguments in the first place, i.e. arguments that will satisfy the normative standards specified by the schemes.

Computational models of argument can model either or both of these functions of argumentation schemes. In this paper, we focus on the second function. Whereas some prior work on computational models of argumentation schemes consists of procedural programs for generating arguments for specific schemes, e.g [2], our aim is to develop a high-level declarative programming language for representing, ideally, *all* argumentation schemes, where schemes represented in this language can be easily validated by developers of argumentation schemes in informal logic and philosophy and serve as *executable specifications* for automatically constructing arguments, when applied to a set of assumptions.

Argumentation schemes [23] are defeasible inference rules. Most, like argument from expert opinion, are to some extent domain-dependent, because they include predicates intended to be interpreted in a particular, domain-dependent way. Some, like defeasible modus ponens, are more generic. Let's take a closer look at these two schemes.

First, here is a simplified version of the scheme for argument from expert witness testimony, which is the prototypical argumentation scheme most often used to introduce the concept of argumentation schemes.

ARGUMENT FROM EXPERT WITNESS TESTIMONY

Premises:

- E is an expert
- E asserts P

Conclusion: P

The expert witness scheme makes use of two domain-dependent predicates:

- is-an-expert/1
- asserts/2

The numbers indicate the arity of each predicate. In the presentation of the scheme, the predicates are shown using an infix notation close to natural language.

In the scheme, E and P are scheme variables. P is a *second-order* variable, ranging over propositions. Notice that the conclusion of the scheme is P. The conclusion does not mention a particular predicate. When the scheme is applied, P is instantiated with a particular proposition, with a particular predicate, and an argument for the proposition P is constructed. The argument constructed can be attacked in the usual ways, with a *rebuttal* (an argument for some proposition which cannot be accepted if P is accepted, for example $\neg P$), an *undercutter* (an argument against the applicability of this argument, for example an argument for E being biased), or a *premise defeater* (an argument for some proposition contrary to a premise, for example an argument con "E is an expert.").

The second argumentation scheme we want to discuss is defeasible modus ponens.

DEFEASIBLE MODUS PONENS

Premises:

- if P then Q
- P

Conclusion: (Presumably) Q

Defeasible modus ponens has the same form as modus ponens, except that the conclusion is only presumably true, rather than necessarily true. (The modality, "presumably", is made explicit in the example only for emphasis. The conclusions of all argumentation schemes are presumably true.) An argument constructed by instantiating this scheme can be attacked in the usual ways, for example by a rebuttal, an argument for $\neg Q$.

Notice that the major premise of defeasible modus ponens, "if P then Q.", is not an atomic proposition, but rather a compound proposition built by connecting two propositions, P and Q, using an "if-then" (implication) operator. Thus, P and Q here are once again second-order variables ranging over propositions. The conclusion of the defeasible modus ponens scheme is also a second-order variable, as is the conclusion of the expert witness testimony scheme. In addition, the defeasible modus ponens scheme has a minor premise, P, which is a second-order variable.

Argumentation schemes like these, with second-order variables, are quite common. It is particularly common for the conclusion of schemes to be a second-order variable, as in both of these examples. Other examples include schemes for argument from abduction, analogy, credible source, established rule, ethos, ignorance, position to know, and precedent.

We are aware of no *computational* models of argumentation schemes which are capable of automatically constructing (inventing, generating, deriving) arguments by instantiating second-order schemes such as these. Prior computational models of argumentation schemes are more limited. Either they are used to check whether existing arguments match the form of a given scheme, as in Aracauria [19], are restricted to propositional (fully instantiated) schemes, as in ArguMed [22], are not defined in sufficient detail to know whether second-order variables are supported, e.g. Pollock's OSCAR system [17], are mathematical models which leave too many details unspecified to be sufficient as a specification for implementing an inference engine, such as ASPIC+[1] [18], or are based on logic programming methods enabling only first-order argumentation schemes to be used to automatically construct arguments, such as Assumption-Based Argumentation (ABA) [7] and earlier versions of Carneades [10].[2]

To understand more clearly the difficulties in representing argumentation schemes using Horn clause logic, the subset of first-order logic used by logic programming languages such as Prolog, let us see how far we can get in representing the scheme for arguments from expert witness testimony in Prolog. Let us first represent, as Prolog "facts", the following assumptions about a case:

```
expert(john).
asserts(john, caused_by(global_warming, humans)).
```

Given these facts, the challenge is to represent the expert witness scheme as a single Prolog rule (Horn clause), in such a way that the following query can be proven by Prolog, answering `yes`:

```
?- caused_by(global_warming, humans).
```

The above representation of the assumptions already shows how one hurdle can be overcome. Although Horn clause logic is a subset of first-order logic, it is possible to represent second-order propositions about atomic formulas, such as global warming being caused by humans here, by reifying such atomic formulas as terms. So far, so good.

But how can the expert witness scheme be represented? Here is one approach, suggested to me by Trevor Bench-Capon but also used by ABA [7, 200–201]:

```
holds(P) :- asserts(E,P), expert(E).
```

The idea here is to represent the second-order conclusion, P, of the scheme with a first-order atomic formula, `holds(P)`, by introducing a unary `holds` predicate. This approach attempts to reduce general inference rules, with premises and second-order variables, to first-order axioms, i.e. inference rules with no premises and only first-order variables in the conclusion. That is, strictly speaking there are no premises in this Horn clause representation of the inference rule, because a Horn clause is a first-order formula. The `:-` symbol in the clause represents the material conditional logical connective, not a deducibility relation.

While axioms can be viewed as a basic form of inference rule, the attempt to represent more general inference rules using a `holds` predicate like this has severe limitations. Since we will want to be able to chain arguments together, by using argumentation schemes to construct arguments for the premises of other arguments, we need some way to convert atoms of the form `holds(P)` to `P`, so that premises can be matched (unified) with `P`. It may seem that one way to achieve this would be to add an additional rule for each predicate, as in the following examples:

```
expert(P) :- holds(expert(P)).
asserts(E,P) :- holds(asserts(E,P)).
caused_by(X,Y) :- holds(caused_by(X,Y)).
```

From a knowledge-representation point of view, this seems rather verbose and cumbersome, but presumably these additional rules could be generated automatically, using some kind of preprocessor. However this approach suffers from a more serious problem: Such a rule would need to be generated for *every* predicate in the application domain, and thus *every* goal would match the conclusion of *every* argumentation scheme with a second-order variable as its conclusion, due to Prolog's goal-directed, backwards-chaining control strategy, causing the search space to become infinite. Thus many (not all) queries will cause the inference engine to enter an endless loop and fail to terminate, depending on the order of facts and rules. Suppose, for example, the Prolog program consists only of the following rules, without any facts:

```
holds(P) :- asserts(E,P),expert(E).
expert(E) :- holds(expert(E)).
asserts(E,P) :- holds(asserts(E,P)).
caused_by(X,M) :- holds(caused_by(X,M)).
```

With these clauses, the following query causes an endless loop and runs out of stack space:

[1] The TOAST implementation of ASPIC+ [20] is propositional. Its rules are fully instantiated axioms, with no variables.

[2] These earlier versions of Carneades allowed argumentation schemes with second-order variables to be represented and used to manually construct arguments, by filling in forms, and to check whether arguments correctly instantiate schemes, but not to automatically construct arguments from a set of assumptions.

```
?- caused_by(global_warming,humans).
ERROR: Out of local stack
```

It is clear why this happens: The query causes an endless loop between the `holds` and `asserts` rules:

(1) `caused_by(global_warming,humans)`
(2) `holds(caused_by(global_warming,humans))`
(3) `asserts(E,caused_by(global_warming,humans))`
(4) `holds(asserts(E,caused_by(global_warming,...)))`
(5) ...

Since this encoding of argumentation schemes requires the definition of every predicate to have an additional `holds` rule, many queries will not terminate in this way, making this encoding useless in combination with Prolog's simple depth-first, backwards-chaining control strategy. Notice that the example query will not terminate no matter how the clauses of the program are ordered, because the program contains no facts. While the program can be made to terminate for this particular goal (query) by adding sufficient facts before the rules, this would not be a general solution to the control problem, not even for other queries using the same rules, because one cannot assume that there are sufficient facts to answer every query affirmatively.

The event calculus [15] uses a `holds` predicate for a similar but more limited purpose, for reasoning about the effects of actions using Prolog. It does not suffer from the control issues discussed here, because the `holds` predicate is used in a more focused way only for a subset of the predicates, called *fluents*, which are state-dependent in the domain model. These fluents are queried *only* using the `holds` predicate. They are never mapped to object-level predicates in the way suggested above.

There is one final and fatal problem with representing argumentation schemes directly in Prolog this way that is important to mention: no arguments are constructed! Thus there is no way to resolve conflicts among arguments, to balance arguments or to use the arguments to help understand or explain the results, for example using argument diagrams.

All of these problems might be overcome by writing a meta-interpreter for argumentation schemes in Prolog, but this would be using Prolog in its capacity as a general-purpose programming language, rather than as an inference engine for Horn clause logic. Some expert system shells, in particular APES [14], were implemented as meta-interpreters in Prolog. APES was able to generate explanations which can be viewed as arguments from the traces of rule applications [3]. However rules in APES were Horn clauses and could not represent argumentation schemes with second-order variables, for the reasons discussed above, and also did not generate counterarguments or use a structured model of argument to resolve attack relations among arguments. The alternative approach we investigate in this paper, using Constraint Handling Rules to represent argumentation schemes, can also use Prolog as an implementation language. Indeed several implementations of Constraint Handling Rules in Prolog exist and we make use of the one provided by SWI Prolog.

As suggested in the previous paragraph, this paper explores the idea of representing argumentation schemes using another kind of rule-based programming, Constraint Handling Rules, introduced by Thom Frhwirth in 1991 [9], to overcome all of the problems identified above by meeting the following requirements:

- Allow second-order variables in the premises and conclusions of schemes
- Not require additional rules for bringing second-order propositions down to the object-level.
- Generate arguments as output
- Guarantee termination

The rest of this article is organized as follows. The next section introduces Constraint Handling Rules, including examples. This is followed by a section showing one way to represent argumentation schemes using Constraint Handling Rules, in such as way as to generate arguments and overcome the other problems identified in this introduction. The section also briefly describes two implementations of this approach, one using the Constraint Handling Rules interpreter provided as a library by SWI Prolog and the second based on our custom implementation of Constraint Handling Rules in the Go programming language. Next is a section about a pilot application based on the approach, called DUCK (Data Use statement Compliance checKer), where domain-dependent argumentation schemes are used to model data use statements and data protection regulations in a web-application for helping cloud service providers to comply with these regulations. The final section presents our conclusions and summarizes the main results.

2 CONSTRAINT HANDLING RULES

Constraint Handling Rules (CHR) is a *declarative*, forwards-chaining rule language originally developed by Thom Frhwirth in 1991 [9].[3]. Forwards-chaining rule engines, based on production rules, have been used from the beginning in expert systems [4, 5]. Production rules are condition-action rules, where the conditions of rules are matched against data structures in *working memory*, a conflict-resolution strategy is used to select a matching rule, and then the action of the selected rule is executed, possibly modifying the working memory and performing side effects, such as outputting data to a file. This process is then repeated until no rule matches the state of the working memory.

While production rule systems have been widely and successfully used for expert systems and implementing so-called "business rules", they do not have a *declarative* semantics. Declarative programming languages are used to describe *what* the problem is, rather than a procedure or algorithm stating *how* to solve the problem. Declarative programming languages, or rather interpreters or compilers for these languages, are clever enough to figure out how to solve the problem on their own, from a description of the problem. Declarative programming languages are typically based on well-founded theories of mathematical functions and/or logic. Prolog [6], based on the Horn clause subset of first-order logic, is perhaps the most prominent of these declarative languages.

One of the achievements of CHR is to realize a forwards-chaining rule language, similar to production rule languages, but with a declarative semantics. CHR is so-named, because the language was initially intended to be used to implement constraint solvers. A constraint solver takes as input a set of relationships among variables, called constraints, and derives further information about these variables.

[3] See also the CHR homepage at https://dtai.cs.kuleuven.be/CHR/

Early constraint solvers were for particular domains, for example propositional constraints over Boolean variables, or equations and inequalities over integers. CHR is more general purpose. It enables constraint solvers for a variety of domains to be specified, using rules.

To make this clearer, let us take a look at the standard example used to illustrate CHR, which defines rules for partial orderings:

```
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.
```

The predicate `leq` is intended to mean "less than or equal to". The words to the left of the `@` symbol in these four rules are identifiers, naming the rules. The `reflexivity`, `antisymmetry`, and `transitivity` rules specify the axioms of partial orderings, in a form close to their usual expression in mathematics. The first rule, for reflexivity, states that for all x, $x = x$. The `idempotence` rule allows the second instance of `X leq Y` to be deleted from the constraint store, since it is redundant.

There are three kinds of rules in CHR: simplification, propagation and "simpagation' rules, where simpagation rules are a hybrid kind of rule combining the features of simplification and propagation rules. All three kinds of rules are illustrated in the example. The reflexivity and antisymmetry rules are simplification rules; the transitivity rule is a propagation rule; and the indempotence rule is a simpagation rule.

Operationally, CHR rules are applied to a multiset of constraints (similar to Prolog facts) in a data structure called the *constraint store*, which serves the same function as the *working memory* in production rule systems. Since the constraint store is a multiset, the same fact may occur multiple times in the store.

When simplification rules, such as the reflexivity and antisymmetry rules, are applied ("fired"), the constraints matching the patterns on the left-hand side of the `<=>` symbol, called the *head* of the rule, are replaced by the constraints on the right-hand side, called the *body* of the rule. For people familiar with Prolog, this terminology may be somewhat confusing, because in Prolog the head of a rule represents its conclusion and the body its antecedents, opposite the convention of CHR. (Moreover, unlike Prolog, a CHR rule may have multiple conclusions.) But CHR's use of the terms "head" and "body" is nonetheless consistent with how these terms are used in Prolog, because in both languages atoms are matched against patterns in the head and, if the match is successful, replaced by atoms on the right. The difference is that Prolog is a goal-directed, backwards-chaining language, which reduces a goal by replacing it with new goals, whereas CHR, on the other hand, as a forwards-chaining rule language, applies rules to derive constraints, adding them to the constraint store.

Simplification and simpagation rules also *delete* constraints from the constraint store. Simplification rules replace the constraints matching the head of the rule with the constraints matching the body of the rule. Simpagation rules, similarly, replace the constraints to the right of the backslash symbol, `\`, in the head of the rule, with the constraints on the right. The matching constraints to the left of the backslash symbol in the head are not deleted.

It might seem counterintuitive at first that a declarative language is allowed to delete constraints from the store. But in CHR this is done in principled way, in a way which does not change the meaning of the constraints in the store. Simplification rules and simpagation rules are used to simplify constraints, as their names suggest, by replacing constraints matching the head with fewer constraints having the same meaning. Consider the indempotence rule, for example. Since the constraint store is a multiset, it may contain duplicate, redundant constraints. The indempotence rule simplifies the constraint store by removing duplicate constraints of the form `X leq Y`.

In addition to heads and bodies, CHR rules may also include, in so-called "guards", further *built-in* constraints. Which built-in constraints are available depends on the particular implementation of CHR. Guards are not illustrated here.

To get an idea of how the CHR inference engine works, let us see what CHR derives when applying the rules defining partial orderings above to the following "query", i.e. giving the initial state of the constraint store:

```
leq(A,B)
leq(B,C)
leq(C,A)
```

First, the transitivity propagation rule is fired and adds `leq(A,C)` to the store. Next, the antisymmetry simplification rule is fired, causing `leq(A,C)` and `leq(C,A)` to be removed and replaced by `A=C`. CHR has built-in support for equality reasoning, which is then used to derive `leq(C,B)` from `leq(A,B)`. Now the antisymmetry simplification rule is applied to `leq(C,B)` and `leq(B,C)`, causing these constraints to be replaced with `B=C`. No further rules can be applied, so the process terminates and returns the constraint store with `A=C` and `B=C`. Thus, CHR was able to infer that all three variables are equal.

In addition to supporting forwards-chaining, CHR has some other properties which may be desirable, depending on the application:

- Turing completeness: Any computable function can be represented using CHR rules.
- Every algorithm can be implemented in CHR with the best known time and space complexity [21].
- CHR rules can be executed concurrently [16].
- The execution of CHR rules can be interrupted and restarted at any time, with intermediate results approximating the final solution.
- Constraints can be input incrementally as they become known, during rule execution, without requiring recomputation.
- Inference rules, rewrite rules, sequents, proof rules, and logical axioms can be directly written in CHR [1].

The last three properties, in particular, appear attractive for representing and implementing argumentation schemes. Argumentation typically takes place in dialogs, with evidence and arguments brought forward and asserted by the participants incrementally, during the course of the dialog. It would be useful if CHR could be used to incrementally and efficiently construct arguments from evidence during dialogs. Moreover, since argumentation schemes are (defeasible) inference rules, the ability of CHR to represent inference rules directly would appear to be quite useful.

3 REPRESENTATION AND IMPLEMENTATION OF ARGUMENTATION SCHEMES

In this section we show how to represent argumentation schemes using CHR rules, and present an overview of the implementation of the component for generating arguments with argumentation schemes represented using CHR in this way, provided by Version 4 of the Carneades argumentation system.[4]

First we need a notation for argumentation schemes. Let us use the syntax for schemes we have developed for Carneades 4, which is based on the YAML markup language[5], which in turn is syntactic sugar for JSON,[6], to make it easier to read and write. Here is a version of the scheme for arguments from expert opinion using this concrete syntax:

```
id: expert_opinion
meta:
  title: Argument from Expert Opinion
  source: >
    Douglas Walton, Appeal to Expert Opinion,
    The Pennsylvania University Press,
    University Park, Albany, 1997, p.211-225.
variables: [E,D,P]
premises:
  - expert(E,D)
  - in_domain(P,D)
  - asserts(E,P)
exceptions:
  - untrustworthy(E)
  - inconsistent_with_other_experts(P)
assumptions:
  - based_on_evidence(asserts(E,P))
conclusions:
  - P
```

This representation of argumentation schemes is, we claim, very high level and quite close to the usual way schemes are represented in informal logic. We have good evidence, from our collaboration with Doug Walton, that informal logicians are able to read, understand and validate schemes represented in this form.

There are a few things to notice about this syntax. First, the schema variables are declared explicitly. This may seem burdensome, but is useful for checking for misspelled variables in schemes, among other purposes. Second, as proposed in [11], the two types of critical questions are represented by exceptions and assumptions. Thirdly, argumentation schemes may now have more than one conclusion, though there is only one in this example. This change was motivated by the desire to support the full CHR rule language. There can be multiple conclusions in the body of CHR rules. But it has the further advantage of reducing the number of schemes required when several conclusions can be derived from the same premises. Fourthly, note that this rule is an example of a scheme having a second-order variable as its conclusion, `S` here. Finally, the example shows how arbitrary meta-data about the scheme can be expressed. The various meta-data properties, such as `title` and `source` in this example, are not predefined and can be freely selected.

We now show, by way of this example, how argumentation schemes are translated into CHR rules. The expert opinion scheme is translated into the following rule:

```
expert_opinion @
    expert(W,D),
    in_domain(S,D),
    asserts(W,S)
    ==>
    S,
    based_on_evidence(asserts(W,S)),
    argument(expert_opinion,[W,D,S]).
```

As illustrated here, each argumentation scheme is translated into a single CHR rule, in this example a propagation rule, with the same name (identifier). The premises of the scheme are translated into constraints in the head of the rule. The conclusions of the scheme are translated into constraints in the body of the rule. Moreover, each of the assumptions of the scheme are also added to the body of the CHR rule, allowing them to be used to derive further information by applying other rules. (The assumptions can be questioned and retracted later, when evaluating the arguments constructed.) Finally, an additional constraint is added to the end of the body of the rule, of the form `argument(<id>,[<variable>,...])`, to keep a record of the argument to be generated when applying the scheme.

Notice that the exceptions of a scheme are not translated and do not appear in the resulting CHR rule. To understand how exceptions are handled, we first need to explain the steps in the process for generating and evaluating arguments:

(1) The argumentation schemes are translated into CHR rules, as illustrated above.
(2) A set of assumptions, represented as ground atomic formulas, are translated into CHR constraints and added to the initial state of the constraint store.
(3) The CHR inference engine is run, repeatedly applying the rules to the constraint store until no rules match or until the `fail` constraint, signaling failure, is derived.
(4) The argument constraints in the store, i.e. the constraints of the form

    `argument(<id>,[<variable>,...])`

    are then translated into Carneades arguments and added to the argument graph.
(5) Assumptions of arguments are added to the assumptions of the argument graph.
(6) When adding arguments to the graph, undercutting arguments are also added for any exceptions of the schemes applied. This information can be looked up using the identifier of the scheme included in the record of the argument in the constraint store.
(7) Finally, the arguments are evaluated, using the formal model of structured argument in [13], to weigh and balance the arguments, resolve attack relations among arguments and label the statements in the argument graph `in`, `out`, or `undecided`.

[4] https://carneades.github.io/Carneades/
[5] http://yaml.org/
[6] http://json.org/

In addition to supporting multiple conclusions in schemes, we have extended argumentation schemes in further ways, in order to support the full expressiveness of Constraint Handling Rules. To illustrate one of these extensions, supporting simplification, here is a reconstruction of the CHR rules for partial orders, represented as argumentation schemes:

```
- id: reflexivity
  variables: [X]
  deletions:
    - leq(X,X)
  conclusions:
    - true

- id: antisymmetry
  variables: [X,Y]
  deletions:
    - leq(X,Y)
    - leq(Y,X)
  conclusions:
    - X=Y

- id: transitivity
  variables: [X,Y,Z]
  premises:
    - leq(X,Y)
    - leq(Y,Z)
  conclusions:
    - leq(X,Z)

- id: idempotence
  variables: [X,Y]
  premises:
    - leq(X,Y)
  deletions:
    - leq(X,Y)
  conclusions:
    - true
```

All of the premises which are to be deleted from the constraint store when the scheme is applied are listed in a `deletions` block of the scheme. Thus, similar to CHR simpagation rules, argumentation schemes here combine the features of CHR simplification and propagation rules.

One caveat is order: Although all CHR rules *can be* expressed in Carneades, it is not possible in Carneades to formulate *the query* needed to reproduce the example presented in Section 2. This is because CHR queries are represented by Carneades assumptions and assumptions are restricted to ground atomic formulas. This restriction assures that all arguments are fully instantiated. That is all premises, conclusions, exceptions and assumptions of arguments are assured to be ground atomic formulas.

Carneades can be configured to use one of two different implementations of CHR for generating arguments from argumentation schemes using the method presented above: the implementation which comes with CHR pre-installed with SWI Prolog[7], and a new implementation of CHR in the Go programming language, by the second author of this paper.[8]

Our new implementation of CHR is still under development but nearing completion. While we do not expect it to have the performance and maturity of the SWI Prolog implementation, it offers several advantages for Carneades: it eliminates a dependency on another system, making it easier to install and administer a Carneades server, and enables us to experiment with CHR extensions. Two extensions have already been implemented:

(1) Since CHR is Turing complete, termination of CHR programs cannot in general be guaranteed. Our implementation of CHR allows the user to set a maximum number of rule firings, to assure termination within roughly predictable time limits, and returns the arguments constructed before the limit was reached. The engine can be restarted to generate further arguments. This is very much in line with the purpose and spirit of argumentation, as a rational method for problem solving and decision-making when information is inconsistent or incomplete.

(2) The second example argumentation scheme in the introduction, for defeasible modus ponens, cannot be implemented using the SWI Prolog version of CHR. While it allows second-order variables in the body (conclusion) of rules, it does not allow them in the head (premises). We are not sure whether this is a limitation of the SWI Prolog implementation of CHR, or the CHR specification. Either way, our implementation of CHR removes this restriction and allows second-order variables in both the head and body of rules, enabling defeasible modus ponens to be represented.

4 THE DUCK DATA PROTECTION APPLICATION

In the Data Usage Compliance Checker (DUCK) pilot project sponsored by Microsoft, we have applied the Carneades 4 inference engine for argumentation schemes, based on CHR, to develop a web-based regulatory compliance system to help cloud service providers to use the ISO/IEC 19944 standard, entitled Information technology — Cloud computing — Cloud services and devices : data flow, data categories and data use, to develop data use documents which are compliant with data protection regulations, in particular the new European General Data Protection Regulation (GDPR). The DUCK system is open source software, available on Github.[9]

Most argumentation schemes are to some extent domain-dependent, because they make use of predicates with particular, intended meanings, such as 'expert'. We take this idea further by adopting the position, expressed in [10, 12], that legal norms can be modeled as domain-dependent argumentation schemes or, more strongly, **are** argumentation schemes, because legal norms regulate not only how to act, but also how to argue about legal issues. The knowledge base of the DUCK system, following this approach, currently consists of more than 100 domain-dependent argumentation schemes. Most of these schemes model the taxonomy of concepts (ontology) defined

[7] http://www.swi-prolog.org/

[8] https://github.com/hfried/GoCHR

[9] https://github.com/Microsoft/DUCK

by the ISO/IEC 19944 standard.[10] The remaining schemes model provisions of the GDPR.

To give a brief impression of the GDPR part of the knowledge base, consider the following three schemes:

```
- id: s1  # default
   weight:
     constant: 0.1
   variables: [S]
   premises:
     - dataUseStatement(S)
   conclusions:
     - consentRequired(S)

- id: s2
   variables: [S]
   premises:
     - notPii(S)
   conclusions:
     - notConsentRequired(S)

- id: s3
   variables: [S]
   premises:
     - pii(S)
     - li(S)
   conclusions:
     - notConsentRequired(S)
```

The first scheme, `s1`, says that data use statements presumably require the explicit consent of the user. Requiring consent is the default. The burden of proof is on the cloud service provider to show that consent is not required. The second scheme says that consent is not required if the data use statement does not make use of personally identifiable information (`pii`). The first scheme has been assigned low weight, 0.1 on a scale of 0.0 to 1.0. If no weight has been specified for a scheme, it has the weight of 0.5. Thus, an argument constructed from `s2` will have more weight than an argument constructed from `s1` and will rebut it when they are evaluated. Argument weights have not been discussed in this article, because it is focused on showing how CHR is used to model argumentation schemes for the purpose of generating arguments. See [13] for further information about how weights are handled in Version 4 of Carneades.

The third scheme, `s3`, provides a second way to construct an argument for consent not being required. It says that even if the data use statement makes use of personally identifiable data, consent is not required if the cloud service provider has a legitimate interest (`li`) in the data.

While these three schemes only scratch the surface of the DUCK knowledge base, they are sufficient to illustrate all of the features of Carneades argumentation schemes which have been applied in this project. For every property of a data use statement, such as requiring consent or not, a scheme with low weight has been defined for the default value of the property. Other schemes, with greater weight, are then used to construct rebuttals overriding the default. Unlike in Prolog, where negation as failure is used to always make the negated proposition the default, the Carneades language for argumentation schemes enables either a proposition or its contrary to be the default, on a predicate by predicate basis. More generally, any position of an issue can be the default. There can be more than two positions (options) for resolving an issue.

5 CONCLUSIONS

Our experiments with using Constraint Handling Rules (CHR) to represent argumentation schemes for the purpose of generating arguments have been encouraging.

We have successfully implemented twenty-five of the most widely used argumentation schemes of [23], including their critical questions.[11] Ten of these twenty-five schemes have conclusions which are second-order variables. Only one of the schemes, defeasible modus ponens, has a second-order variable as a premise. Our implementation of CHR has been extended to allow second-order variables in the premises of schemes.

Moreover, the approach has been successfully validated in the DUCK project, which uses our CHR-based implementation of argumentation schemes to generate arguments, in a system designed to help cloud service providers to develop data use documents compliant with data protection regulations.

Using CHR as a foundation for implementing argumentation schemes provided us with an opportunity to extend the concept of an argumentation scheme in various ways, to make it possible to represent any CHR rule as an argumentation scheme. This enables the Carneades language for argumentation schemes to inherit all of the attractive features of CHR, including Turing completeness, the possibility of concurrent execution, support for stopping and restarting computation at anytime, with intermediate results available for use, and support for inputting further information incrementally during dialogues and other argumentation processes.

Conversely, the synthesis of CHR and argumentation provided by Carneades provides additional benefits not provided by CHR alone. CHR has no concept of negation. Carneades issues can model negation or, more generally, a set of conflicting positions of issues. Moreover, CHR provides no built-in support for defeasible reasoning. We use CHR to generate pro and con arguments, which are then evaluated in a post-process, using a model of structured argument, to support defeasible reasoning by weighing and balancing arguments and resolving attack relations among arguments. Most importantly, our system produces arguments which can be used to explain and understand CHR inferences, for example by visualizing the arguments in argument maps.

While the method presented here for generating arguments using Constraint Handling Rules was developed for the latest version of the Carneades model of structured argument [13], it can be adapted for use in any model of argument in which arguments are constructed by instantiating argumentation schemes. We leave it for future research by others to adapt the method to other models of structured argument.

[10] Interestingly, CHR is expressive enough to efficiently implement, in a straightforward way, a correct and complete reasoner for Description Logic, the decidable subset of first-order predicate logic which is the formal foundation of the Web Ontology Language (OWL) commonly used to define ontologies in the context of the Semantic Web.[8] The ISO/IEC 19944 standard used in DUCK does not require this expressivity, but this may prove useful later in the project.

[11] https://github.com/carneades/carneades-4/blob/master/examples/AGs/YAML/walton.yml

ACKNOWLEDGEMENTS

This work was partially funded by the Canadian Social Sciences and Research Council (SSHRC) in the Carneades project, by Microsoft, in the DUCK project, and by the European Union Horizon 2020 research and innovation programme under grant agreement No 732189 in the AEGIS project.

REFERENCES

[1] Slim Abdennadher, Thom Frühwirth, and Christian Holzbaur. Introduction to the special issue on constraint handling rules. *Theory and Practice of Logic Programming*, 5(4-5):401–402, 2005.

[2] Trevor Bench-Capon, Katie Atkinson, and Adam Wyner. Using argumentation to structure e-participation in policy making. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XVIII*, pages 1–29. Springer, 2015.

[3] Trevor Bench-Capon and Marek Sergot. Toward a Rule-Based Representation of Open Texture in Law. In Charles Walther, editor, *Computer Power and Legal Language*, pages 39–60. 1988.

[4] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming in OPS5: an Introduction to Rule-Based Programming*. Addison-Wesley Longman, Boston, 1985.

[5] Bruce G Buchanan and Edward H Shortliffe, editors. *Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, Mass., 1984.

[6] W F Clocksin and C S Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[7] Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. Assumption-based argumentation. In Iyad Rahwan and Guillermo R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 199–218. Springer, 2009.

[8] Thom Frühwirth and Philipp Hanschke. Terminological reasoning with constraint handling rules. In *International Conference on Principles and Practice of Constraint Programming*, pages 361–381, 1995.

[9] Thom Frhwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[10] Thomas F Gordon. Constructing Arguments with a Computational Model of an Argumentation Scheme for Legal Rules – Interpreting Legal Rules as Reasoning Policies. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Law*, pages 117–121. ACM Press, 2007.

[11] Thomas F. Gordon, Henry Prakken, and Douglas Walton. The Carneades model of argument and burden of proof. *Artificial Intelligence*, 171(10-11):875–896, 2007.

[12] Thomas F Gordon and Douglas Walton. Legal Reasoning with Argumentation Schemes. In Carole D Hafner, editor, *12th International Conference on Artificial Intelligence and Law (ICAIL 2009)*, pages 137–146, New York, NY, USA, 2009. ACM Press.

[13] Thomas F. Gordon and Douglas Walton. Formalizing balancing arguments. In *Proceeding of the 2016 conference on Computational Models of Argument (COMMA 2016)*, pages 327–338. IOS Press, 2016.

[14] Peter Hammond. APES: A user manual. Technical report, 1983.

[15] Robert Kowalski and Marek Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4:67–95, 1986.

[16] Edmund S. L. Lam and Martin Sulzmann. A concurrent constraint handling rules implementation in Haskell with software transactional memory. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 19–24, New York, NY, USA, 2007. ACM.

[17] John L Pollock. Defeasible reasoning in OSCAR. *The Computing Research Repository*, 2000.

[18] Henry Prakken. An abstract framework for argumentation with structured arguments. *Argument & Computation*, 1:93–124, 2010.

[19] Chris Reed and Glenn Rowe. Araucaria: Software for puzzles in argument diagramming and XML. Technical report, Department of Applied Computing, University of Dundee, 2001.

[20] Mark Snaith and Chris Reed. Toast: Online aspic+ implementation. *COMMA*, 245:509–510, 2012.

[21] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. *ACM Trans. Program. Lang. Syst.*, 31(2):8:1–8:42, February 2009.

[22] Bart Verheij. *Virtual Arguments*. TMC Asser Press, The Hague, 2005.

[23] Douglas Walton, Chris Reed, and Fabrizio Macagno. *Argumentation Schemes*. Cambridge University Press, 2008.