Description Logic and Rules the CHR Way
Extended Abstract

Thom Frühwirth

Fakultät für Ingenieurwissenschaften und Informatik
University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. The challenges of the Semantic Web endeavour in knowledge representation and reasoning prompted a wealth of research in combining description logic (DL) as ontology languages (e.g. OWL) with logic programming for rule-based reasoning. General issues of combining and integrating formalisms have to be faced such as the type of combination, conceptual simplicity and tractability. Even though constraint-based programming has a tradition of tackling these questions, constraint-based rule formalisms such as constraint logic programming, concurrent constraint programming, constraint databases and constraint handling rules (CHR) have not explicitely been considered for combination with DL yet. The same holds for concurrency, which is an essential characteristic of the internet, but to the best of our knowledge has not been related to DL so far. Since CHR is a very expressive declarative concurrent constraint-based programming language with optimal performance guarantee and other interesting properties, we explore in this speculative paper what a CHR-based approach would look like in comparison to recent approaches for integrating OWL and rules.

1 Introduction

In recent years, prompted by research in the Semantic Web, there is a renewed interest in knowledge representation and reasoning by rules based on logical formalisms. In 2004, the Web Ontology Language OWL [3] has been proposed for knowledge representation. OWL is based on Description Logic (DL), a well-founded research area with a long tradition [4]. Also other variants of DL have been considered. For reasoning, various logic programming formalisms have been considered, such as Prolog-style Horn clauses, Datalog from deductive databases, answer set programming, F-Logic, and concurrent constraint handling rules (CHR) [10, 5, 6].

Most of these rule-based formalisms have been considered for combination and integration with OWL or other description logics. In the Semantic Web Rule Language (SWRL) [7], DL was extended with material first-order implication between conjunctions of DL atoms to provide for rules.

Constraint programming has a tradition of investigating the tradeoff between expressiveness and computational complexity, and in concentrating on not only decidable, but efficient theories.

Even though reasoning in description logic is more and more considered and understood as constraint solving, constraint-based rule formalisms such as constraint logic programming, concurrent constraint programming, constraint databases and CHR have not explicitely been considered for combination with DL yet. Even less is known about concurrency and DL, while concurrency is clearly an important aspect in any web-based application.

On the other hand, description logics and concurrent CHR have already been related as early as 1992 [11] by implementing DL as a constraint solver in CHR. This solver is online at WebCHR `http://chr.informatik.uni-ulm.de/~webchr/`.

Since DL is a well-developed flexible knowledge representation formalism that can embed many other approaches and since CHR subsumes essential aspects of many rule-based approaches under the umbrelle of concurrent rules on constraints, such as all those mentioned before as well as production rules, term rewriting, multi-set transformation, Petri nets, event-condition-action rules, it seems worthwhile to consider a combination of CHR rules and DL.

CHR admits a layered approach where constraint solvers may be stacked onto each other in that it distinguishes between CHR-defined constraints (relations, predicates, atoms) and already defined, so-called built-in constraints. Besides a hierarchical use, this also allows to integrate and use different given constraint solvers side-by-side, where communication is implicit and concurrent via shared variables on which common built-in constraints are imposed.

Overview of the paper. In the next section we quickly introduce CHR. Then we show how to implement the basic DL $\mathcal{ALC}$ and extensions in CHR and what optimisations are straightforward. In Section 4 we discuss how and to what extent CHR rules can encode state-of-the-art proposal for the integration of DL and rules. We end with a discussion and conclusions.

2 Constraint Handling Rules (CHR)

CHR is a declarative concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multisets of constraints (relations, predicates, atoms).

Algorithms are often specified using inference rules, rewrite rules, sequents, proof rules, or logical axioms that can be directly written in CHR. Yet, CHR is no theorem prover, but an efficient general-purpose programming language with a clear declarative and a clear operational semantics. CHR supports rapid prototyping by giving the programmer efficiently executable specifications as we will see in this paper.

CHR programs have a number of desirable properties guaranteed and can be analyzed for others. Any CHR program will automatically implement a *concurrent anytime (approximation) and online (incremental) algorithm. Confluence* of rule applications and operational equivalence of programs is decidable for terminating CHR programs. We do not know of any other programming language in

practical use where the latter is also the case. Confluence is also essential to enable *declarative concurrency (logical parallelism)*, which means that a confluent program can be run concurrently without any modification and without many of the problems that usually plague concurrent programs.

There is also a kind of *optimal performance guarantee*. It has been proven that CHR can implement any algorithm with best known time and space complexity [13], something that is not known to be possible in other pure declarative languages. The efficiency of the language is also empirically demonstrated by optimal and elegant CHR programs for algorithms like union-find, shortest paths and Fibonacci heaps. The remaining constant factor performance penalty of using a very high-level declarative language versus an imperative language has been reduced to an order of magnitude by recent optimizing CHR compilers.

Free CHR libraries exist for most Prolog systems, several for Java, Haskell and Curry. Standard constraint systems as well as novel ones such as temporal, spatial, or description logic constraints have been implemented, many programs are available online. CHR is also available as WebCHR for online experimentation with more than 40 constraint solvers, including one for DL.

Besides constraint solvers, applications of CHR can be found in computational logic, in agent programming, multiset rewriting and production rule systems. In computational logic, it integrates deduction and abduction, bottom-up and top-down execution, forward and backward chaining, tabulation and integrity constraints. The several hundred publications [12] mentioning CHR cover such diverse applications as type system design for Haskell, time tabling for universities, optimal sender placement, computational linguistics, spatio-temporal reasoning, chip card verification, semantic web information integration, and decision support for cancer diagnosis.

2.1 Syntax and Semantics

CHR manipulates conjunctions of constraints (relations, predicates, atoms) that reside in a constraint store. Let H, C and B denote conjunctions of constraints. There are three types of rules as given in Fig. 1. The declarative, logical reading (meaning) of a rule is a logical equivalence provided the guard holds. The sequence $\bar{y}$ contains the variables that appear only in the body B of the rule.

Simplification rule:	$H \Leftrightarrow C \mid B$	$\forall \bar{x}\ (C \rightarrow (H \leftrightarrow \exists \bar{y}\ B))$
Propagation rule:	$H \Rightarrow C \mid B$	$\forall \bar{x}\ (C \rightarrow (H \rightarrow \exists \bar{y}\ B))$
Simpagation rule:	$H_1 \backslash H_2 \ \Leftrightarrow\ C \mid B$	$\forall \bar{x}\ (C \rightarrow (H_1 \wedge H_2 \leftrightarrow \exists \bar{y}\ H_1 \wedge B))$

Fig. 1. Types of CHR Rules and their Logical Reading

Operationally, a simplification rule replaces instances of the CHR constraints H by B provided the guard test C holds. A propagation rule instead just adds B

to H without removing anything. The hybrid simpagation rule removes matched constraints H_2 but keeps constraints H_1.

The *standard operational semantics* of CHR is given by a transition system where states are conjunctions of constraints (cf. Figure 2).

if $H \Leftrightarrow G \mid B$ is a copy of a rule $H \Leftrightarrow G \mid B$ with new variables $\bar{X}$
and $CT \models \forall(C_b \rightarrow \exists\bar{X}(H = H' \wedge G))$
then $(H' \wedge C) \longmapsto (B \wedge G \wedge H = H' \wedge C)$

Fig. 2. State transition for simplification rules

The constraints of the store comprise the state of an execution. Starting from an arbitrary initial store (called *query*), CHR rules are applied exhaustively until a fixpoint is reached. A rule is applicable, if its head constraints are matched by constraints in the current store one-by-one and if, under this matching, the guard of the rule is logically implied by the constraints in the store. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice. Trivial non-termination of a propagation rule application is avoided by applying it at most once to the same constraints. Almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program [9].

Search in CHR is usally provided by the host language, e.g., by the built-in backtracking of Prolog or by search libraries in Java. In all Prolog implementations of CHR, the disjunction of Prolog can be used in the body of CHR rules. This was formalized in the language CHR$^\vee$ [1, 2], where the operational semantics of CHR is extended by the transition that distributes the disjunction over conjunction similar to Prolog.

3 A CHR Constraint Solver for DL

3.1 Concepts and Roles, A-Box and T-Box

We use a basic variant of description logic, $\mathcal{ALC}$, and use concrete syntax.

Concept terms are defined inductively: Every *concept (name)* c is a concept term. If s and t are concept terms and r is a *role (name)*, then the following expressions are also concept terms:

`not` s (negation, complement),
s `and` t (conjunction, intersection),
s `or` t (disjunction, union),
`all` r `is` s (value restriction),
`some` r `is` s (exists-in restriction, existential quantification).

Objects (individuals) are constants or variables. Let a, b be objects. Then $a : s$ is a *membership assertion (constraint)* and $(a, b) : r$ is a *role-filler assertion*

(constraint). An *A-box (assertional knowledge)* is a conjunction of membership and role-filler assertions. A *T-box (terminological knowledge)* is a finite set of acyclic *concept definitions* c `isa` s, where c is a concept name and s is a ground concept term.

Each concept has at most one definition. In the basic formalism we assume that a concept cannot be defined in terms of itself directly or indirectly, i.e., concept definitions are acyclic. This implies that there are concepts without definition, they are called *primitive*.

3.2 Implementation of DL as Constraint System in CHR

Reasoning problems of description logics include consistency of assertions, query answering, checking if an individual is an instance of a concept term and classification of concepts by subsumption. Subsumption and consistency can express each other. Usually, one reduces such reasoning services to consistency checking (satisfiability) [4].

Basic description logics have a straightforward embedding in the decidable two-variable fragment of first-order logic (FOL). The theory for $\mathcal{ALC}$ with T-boxes and its implementation in CHR [11] are straightforward, see Figure 3.2. The axioms and T-box are translated into CHR rules, while the A-box is considered as query.

I : `not`$S \leftrightarrow \neg(I : S)$	`I:not S, I:S <=> false` $(*)$
$I : S_1$ `and` $S_2 \leftrightarrow I : S_1 \wedge I : S_2$	`I:S1 and S2 <=> I:S1, I:S2`
$I : S_1$ `or` $S_2 \leftrightarrow I : S_1 \vee I : S_2$	`I:S1 or S2 <=> (I:S1 ; I:S2)`
I : `some` R *is* $S \leftrightarrow \exists J((I, J) : R \wedge J : S)$	`I:some R is S <=> (I,J):R, J:S`
I : `all` R *is* $S \leftrightarrow ((I, J) : R \rightarrow J : S)$	`I:all R is S, (I,J):R ==> J:S`
C `isa` $S \leftrightarrow (I : C \leftrightarrow I : S)$	`I:C <=> I:S,  I:not C <=> I:not S`

$(*)$ Plus CHR rules to produce the Negation Normal Form, see text.

Fig. 3. FOL Constraint Theory and CHR Rules for $\mathcal{ALC}$

One should contrast the CHR implementation with the common completion (transformation) rules for $\mathcal{ALC}$ in abstract syntax given in Figure 3.2. The executable specification in CHR is as concise and as compact as the abstract formulation in logic.

The CHR constraint solver for description logics simplifies and propagates assertions in the A-box by decomposing concept terms and by using the definitions in the T-box to unfold them while looking for obvious contradictions of the form `X : C` and `X : not C`. This is achieved by the so-called clash rule given as first CHR rule in Figure 3.2.

To ensure completeness, we need the rules that turn an arbitrary concept term into Negation Normal Form (NNF). The following simplification rules push the complement operator `not` down to the leaves of a concept term in the obvious way:

`and`: **if** $x : C_1 \sqcap C_2 \in \mathcal{A}$ and $\{x : C_1, x : C_2\} \not\subseteq \mathcal{A}$
then $\mathcal{A} \rightarrow_{\sqcap} \mathcal{A} \cup \{x : C_1, x : C_2\}$

`or`: **if** $x : C_1 \sqcup C_2 \in \mathcal{A}$ and $\{x : C_1, x : C_2\} \cap \mathcal{A} = \emptyset$
then $\mathcal{A} \rightarrow_{\sqcap} \mathcal{A} \cup \{x : D\}$ for some $D \in \{C_1, C_2\}$

`some`: **if** $x : \exists R.D \in \mathcal{A}$ and there is no y with $\{(x, y) : R, y : D\} \subseteq \mathcal{A}$
then $\mathcal{A} \rightarrow_{\exists} \mathcal{A} \cup \{(x, y) : R,\ y : D\}$ for a fresh individual y

`all`: **if** $x : \forall R.D \in \mathcal{A}$ and there is a y with $(x, y) : R \in \mathcal{A}$ and $y : D \notin \mathcal{A}$
then $\mathcal{A} \rightarrow_{\forall} \mathcal{A} \cup \{y : D\}$

Fig. 4. The completion rules for $\mathcal{ALC}$

```
I:not not S <=> I:S.
I:not (S1 and S2) <=> I:not S1 or not S2.
I:not (S1 or S2) <=> I:not S1 and not S2.
I:not (some R is S) <=> I:all R is not S.
I:not (all R is S) <=> I:some R is not S.
```

The so-called completion (transformation) rules as specified in Fig. 3.2 and implemented by CHR rules in Fig. 3.2 work as follows: The conjunction rule generates two new, smaller assertions. An exists-in restriction generates a new variable that serves as a witness for the restriction. Such a generation of a new, implictely existentially quantified variable is no problem for logic programming languages. A value restriction has to be propagated to all role fillers using a propagation rule. To achieve completeness of disjunction, search must be employed. The operator `;` is inherited from Prolog and denotes search by chronological backtracking. This extension of CHR with disjunction is called CHR$^{\vee}$ [2]. Implementations of CHR in languages other than Prolog may employ different search methods.

Finally, for each concept definition c `isa` s the *unfolding* rules replace concept names by their definitions. There also has to be a rule for the complement case, since CHR as a programming language does not provide reasoning by contrapositives.

Logical Correctness and Solved Normal Form The logical reading of the CHR rules (Fig. 1 immediately shows their logical correctness with as consequences of the constraint theory for $\mathcal{ALC}$(Fig. 3.2).

The solved normal form is either `false` (inconsistent), `true` (tautological) or contains one or more constraints of the forms
`I:C, I:not C, I:S or T, I:all R is S` and `(I,J):R`,
where `C` is a primitive concept name. There are no clashes and the value restriction has been propagated to every successor object. It is easy to show by contradiction that exhaustive application of the CHR rules produces this normal form: To any constraint that is not in solved form, at least one of the rules of the solver is applicable.

Anytime and Online Algorithm Property Any CHR program will automatically implement a concurrent anytime (approximation) and online (incremental) algorithm.

Anytime (approximation) algorithm means that we can interrupt the program at any time and restart from the intermediate result without the need to recompute from scratch. Also, the intermediate result approximates the final result. Anytime algorithms are useful to guarantee response times for real-time and embedded systems or when hard problems are solved. In CHR, we can interrupt the computation after any rule application, and the intermediate results are the states in the computation. These states are meaningful, since they have a logical reading; and they approximate the final state.

Online (incremental) algorithm means that we can add additional constraints while the program is already running without the need for redoing the computation from scratch. The program will actually behave as if the newly added constraints were present from the beginning of the computation but had been ignored so far. Online algorithms are useful for constraint solving and interactive, reactive and control systems, including agents.

Clearly, our DL solver has these anytime and online properties. We can stop the computation and restart it anytime while we get closer to the solved normal form and we can add assertions while the program runs without affecting correctness.

Confluence Confluence means that the result of a computation is the same, no matter which applicable rules are applied in which order. In CHR, simplification rules that apply to the same constraints can destroy confluence. Since all CHR rules for DL except the clash rule have pairwise disjoint heads, non-confluence could only result when the clash rule is involved. For terminating, CHR programs, confluence can be automatically checked by considering a finite number of prototypical, minimal conjunctions of constraints which can be built by overlapping rule heads. For example, the critical overlap

```
I:not all R is S, I:all R is S
```

either leads to false using the clash rule or is simplified by pushing `not` down in the first constraint

```
I:some R is not S, I:all R is S      ↦  (some-rule)
(I,J):R, J:not S, I:all R is S       ↦  (all-rule)
(I,J):R, J:not S, I:all R is S, J:S  ↦  (clash-rule)
false,
```

hence leading to failure as well. In this way, it can be shown that the DL rules are confluent.

Concurrency Confluence not only implies consistency of the logical reading of the rules, but also means that the CHR program can be executed in parallel as it is. This property is called declarative concurrency or logical parallelism. Indeed it is easy to see that each constraint can be handled in its own thread by and-parallelism (and or-parallelism for concept union). The only sychronisation necessary is when the clash rule and the propagation rule for value restrictions

are applied. But since the program is confluent, it does not matter when these rule apply. Moreover, given enough threads, all applicable propagation rules can be applied simultaneously, and the same holds for the clash rule since it will immediately lead to failure.

These observations indicate that the main sources of intractability of basic DL, disjunction and value restrictions might not show up in a parallel implementation.

Termination. The only CHR constraints that are rewritten by the rules are membership assertions with given, ground concept terms. Hence, it suffices to show that in each rule, the membership assertions in the body are strictly smaller than the ones in the head. In that case, the propagation rule for value restrictions can only generate a finite number of smaller and smaller membership assertions. A concept term is larger than its proper subterms, and a atomic concept is larger than its defining concept term. Since concept definitions are acyclic and finite by definition, the order is well-founded.

3.3 Complexity and Optimizations

Optimizing CHR compilers nowadays at least support indexing so that given one constraint that matches the head of a multi-headed rule, the other, so-called partner constraints can be found quickly. If we place an index on the first argument and role name (if present) of membership and role-filler assertions, all rules of our DL program can be applied in constant time. This is a strong indication that there is no performance penalty in using a CHR implementation, and indeed this is the case [6].

A CHR programmer will immediately see potential sources of intractability by looking at the rules. He knows that using disjunction and multi-headed propagation rules (for the value restriction) may easily lead to exponential worst case time complexity, and that the introduction of a new unbound variable in the rule for the exists-in restriction could even cause nontermination.

But he also has a number of generic remedies at hand. First of all, CHR naturally support *graceful degradation* in the sense that you pay only for the features that you actually use. If we do not use the expensive rules, the reasoning problem becomes tractable. This means no disjunction (and consequently no negation that can lead to disjunction) and no value restriction (at least not together with the exists-in restriction that generates roles for the value restriction).

Indeed, without these constructs, the remaining rules have a linear worst-case time complexity in the size of the unfolded A-box, where defined concepts have the size of their definition in the T-box. Even though this size measure can be exponential in the syntactic size of the T-box, there are optimizations that bring the complexity down to polynomial.

One effective means is to enforce a set-based semantics of the assertions, i.e. to remove duplicates using the rule

```
IJ:CR \ IJ:CR <=> true.
```

Another standard constraint-programming technique is to employ search only if no other rule is applicable. This is called *labeling* in constraint programming. We just have to introduce an auxiliary constraint `label` whose presence is required by the search rule and to make sure that it is only executed if no other constraint is active. This is achieved in most implementations by putting it last in a query.

In general, we may restrict the applicability of expensive rules by adding more constraints to the head (and guard).

We may transform `some` only if `forall` is present:
`I:all R is D \ I:some R is S <=> (I,J):R, J:S`.

We may unfold a concept only if there is another concept for that variable:
`X:C1 \ X:C <=> X:D` given `C isa D`,
assuming that the T-box has already been checked for consistency.

Standard DL optimisations like cashing, blocking and the trace technique are also possible but are omitted for space reasons.

3.4 Extensions

In Figure 5 we list some standard extensions of description logic languages. We assume that `=` is a built-in equality constraint and some extensions will use additional built-in constraints, namely `feature/1, distinct/1, primitive/1`. We remark that these three built-ins could also be implemented as CHR constraints by modifying the rules accordingly (i.e. moving them from the guard to the head of the rule).

4 DL Rules in CHR

Integration of DL with rules in the context of the Semantic Web usually starts from the OWL languages. OWL [3] actually has three layers of languages with increasing expressiveness and difficulty: The classic DL reasoning problems are in EXPTIME for OWL-Lite and NEXPTIME for OWL-DL while OWL-Full is undecidable. OWL-DL is a W3C recommendation language for ontology representation in the Semantic Web. It provides full negation, disjunction and restricted forms of universal and existential quantification of variables.

The reasons to combine such DL's with logic-based rules are manifold: first of all, the common declarative semantics of FOL, and then the boost in expressiveness by allowing for polyadic predicates and arbitrary conjunctions of DL atoms which are not expressible by concept terms. This is typically the case if the role-filler assertions do not have a tree structure and may even be acyclic. The by now classical example of a non-tree structure is the definition of the `uncle` role as a male sibling of a person's father.

Expressive DL's like OWL include axioms of inclusion between concept terms, written $C \sqsubseteq S$. These axioms can be translated to CHR propagation rules:
`I:C ==> I:S.` `InotS ==> I:not C`,

Top (universal) and bottom (empty) concepts:
```
X:top <=> true.     X:bot <=> false.
```
Allsome quantifiers, e.g. `parent isa allsome child is human`:
```
I:allsome R is S <=> I:all R is S, I:some R is S
```
Role chains (nested roles), e.g. `grandfather isa father of father`:
```
(I,J):A of B <=> (I,K):A, (K,J):B
```
Inverse Roles
```
(I,J):inv(R) ==> (J,I):R.     (I,J):R ==> (J,I):inv(R).
```
Transitive Roles
```
(I,K):R, (K,J):trans(R) ==> (I,J):trans(R)
```
Functional roles (features, attributes):
```
(I,J):F, (I,K):F ==> feature(F) | J=K.
```
Distinct, disjoint primitive concepts:
```
I:C1, I:C2 ==> distinct(C1), primitive(C2) | C1=C2.
```
Nominals (named individuals, singleton concepts)
```
X:{I} ==> X=I.
```
Concrete domains (constraints from other domains):
```
(I,J):smaller ==> I<J.
(I,J):nota smaller ==> I>=J.
(I,A):f1, (I,B):f2 ==> flight(A,B).
```

Fig. 5. Common Extensions of $\mathcal{ALC}$and their CHR Rules

where `InotS` is the solved normal form (SNF) of `I:not S`. So, in some sense, DL's themselves provide already some rules through the T-box.

In the Semantic Web Rule Language (SWRL) [7], OWL is extended with material first-order implication between conjunctions of DL atoms to provide for rules. Together with equality and disequality, these implications are strictly more general than concept inclusion axioms. For example, they can already express the `uncle` definition.

The SWRL approach is conceptually simple and results in a tight integration with OWL. SWRL has no disjunction, negation, no predicates other than those from DL, no nonmonotonic features such as negation-as-failure or defaults (that would require a semantic other than first order logic). Still, SWRL is already undecidable since it can simulate role value maps. SWRL goes beyond Horn clauses because it allows existentials in the consequent.

For mapping into CHR, nothing changes. We are still using propagation rules, and of course they can introduce new variables in the consequent and they will be existentially quantified by definition. So the `uncle` example yields to CHR rule:
```
male(Z), hassibling(Y,Z), hasparent(X,Y) ==> hasuncle(X,Z).
```

The approach of [8] for OWL-DL can be seen as an extension of SWRL in that non-DL atoms can occur in a rule. The rule now has the general form:
$A_1 \vee \ldots \vee A_n \leftarrow B1 \wedge \ldots B_m$,
where the A's and B's are atoms. Again, such a rule can obviously be written

as a CHR propagation rule with disjunction:
`B1,...Bn ==> (A1 ; ...; An)`
this kind of reasoning is intended, then additional CHR rules may be necessary for a faithful translation.

So the current rule-based DL approaches translate to CHR propagation rules. With such rules, CHR basically performs a bottom-up closure. While the operational semantics of CHR rules is relatively straightforward, the rule extension proposals for DL use sophisticated translations into FOL and related logics that are subjected to a theorem prover. This likely makes reasoning about such rule programs more difficult than in CHR.

As a programming language, CHR does not use contrapositives, but their effect can be achieved, as we already have seen with concept definitions and concept inclusions. In general additional CHR rules are necessary to capture contrapositives. Here are some concrete examples from the literature. Consider a translated DL concept definition:
`uncle(X) <=> hasuncle(Y,X).`
It can only be used to simplify `uncle` to `hasuncle`, but cannot draw conclusions the other way round. But if we replace the above CHR rule by two propagation rules, we have regained reasoning power:
```
uncle(X) ==> hasuncle(Y,X).
hasuncle(Y,X) ==> uncle(X).
```
The other example concerns the rule:
`beer(X) ==> happy(sean).`
where we can add the contrapositive:
`not happy(sean) ==> not beer(X).`
Another approach that would bring the CHR implementation close to that using a theorem prover would be to use a clausal form representation. But we wonder if this is actually what the user wants and can comprehend.

Decidability is an issue, and the combination of OWL-DL with the above kind of rules stays decidable for *DL-safe* programs over finite domains of values that lead to ground variables: Each variable in the head is required to occur in a non-DL-atom in the rule body. This is not really a restriction, since one can add unary predicate that ranges over all individuals, but this effects efficiency and unfortunately also the semantics, i.e. results of reasoning, as the `badchild` example in [8] shows.

Using a programming language like CHR for rules, we take undecidability for granted, and we do not need to restrict ourselves to finite domains, so `Romulus` will be a `badchild`, no matter if he is a variable or individual. The examples can be found in the DL implementation accompanying this paper.

5 Discussion and Conclusions

The implementation of a DL reasoner (solver) for $\mathcal{ALC}$ with T-boxes and many extensions by simply encoding the FOL theory of the DL in CHR results in a concise and compact set of rules with performance guarantees. The resulting

program is correct, confluent, and concurrent. It produces a solved normal form and gives an anytime and online algorithm for consistency checking of DL. Many optimizations are possible, both taken from constraint-programming and DL implementations.

The observations on concurrent execution of the DL reasoner indicate that the main sources of intractability of basic DL, disjunction and value restrictions might not show up in a parallel implementation.

We have seen that the rule-based extensions for OWL can be expressed as CHR propagation rules without further ado. Of course, CHR as a programming language is Turing-complete, so for decidability one would have to consider certain syntactic subsets. On the other hand, CHR programs have a number of desirable properties guaranteed and can be analyzed for others. In particular, expressive DL's, concrete domains, unbound variables over arbitrary domains and unsafe rules of a very general form are no problem for the CHR implementation.

The code presented in this paper is online at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/dlrules.pl` and can be run online with WebCHR `http://chr.informatik.uni-ulm.de/~webchr/`.

Future work will investigate the nonmonotonic aspects of rules with DL in CHR.

References

1. S. Abdennadher and H. Schütz. Model generation with existentially quantified variables and constraints. In *6th International Conference on Algebraic and Logic Programming*, LNCS 1298. Springer, 1997.
2. S. Abdennadher and H. Schütz. CHR$^\vee$: A flexible query language. In *Flexible Query Answering Systems*, LNAI 1495. Springer, 1998.
3. M.K. Smith, C. Welty, and D.L. McGuinness. OWL web ontology language guide, W3C recommendation.
4. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
5. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
6. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, to appear.
7. I. Horrocks, P. Patel-Schneider, S. Bechhofer, D. Tsarkov. OWl Rules: A Proposal and Prototype Implementation. *Journal of Web Semantics*, 3, 2005.
8. B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. *Journal of Web Semantics*, 3:41–60, 2005.
9. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *20th International Conference on Logic Programming (ICLP)*, LNCS. Springer, 2004.
10. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, 1998.

11. T. Frühwirth and P. Hanschke. Terminological reasoning with constraint handling rules. In P. V. Hentenryck and V. Saraswat, editors, *International Conference on Principles and Practice of Constraint Programming*, pages 361–381, Cambridge, Mass., 1995. MIT Press.
12. T. Schrijvers and T. Frühwirth. CHR Website, `www.cs.kuleuven.ac.be/~dtai/projects/CHR/`, 2006.
13. J. Sneyers, T. Schrijvers, and B. Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.