CHR 2013 — Proceedings of the 10th International Workshop on Constraint Handling Rules

Henning Christiansen Jon Sneyers

Report CW 641, July 2013

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

CHR 2013 — Proceedings of the 10th International Workshop on Constraint Handling Rules

Henning Christiansen Jon Sneyers

Report CW 641, July 2013

Department of Computer Science, KU Leuven

Abstract

This volume contains the papers presented at CHR 2013, the 10th International Workshop on Constraint Handling Rules held on July 13th 2013 in Berlin (Germany), at the occasion of the Third International Summer School on CHR.

This workshop is the tenth in a series of annual CHR workshops. It means to bring together in an informal setting, people involved in research on all matters involving CHR, in order to promote the exchange of ideas and feedback on recent developments. Previous workshops on Constraint Handling Rules were organized in 2004 in Ulm (Germany), in 2005 in Sitges (Spain) at ICLP, in 2006 in Venice (Italy) at ICALP, in 2007 in Porto (Portugal) at ICLP, in 2008 in Hagenberg (Austria) at RTA, in 2009 in Pasadena (California, USA) at ICLP, in 2010 in Edinburgh (Scotland) at ICLP, in 2011 in Cairo (Egypt) at the second CHR summer school, and in 2012 in Budapest (Hungary) at ICLP.

Keywords : Constraint Handling Rules

CR Subject Classification : D.3.2, D.3.3, F.4.1

Proceedings
of the
Tenth International Workshop
on
Constraint Handling Rules

July 13th, 2013
Berlin, Germany

Preface

This volume contains the papers presented at CHR 2013, the 10th International Workshop on Constraint Handling Rules held on July 13th, 2013 in Berlin. The workshop is organized at the occasion of the 3rd International Summer School on CHR.

There were 10 submissions, one of which was retracted before review process started since it was out of scope. Each remaining submission was carefully reviewed by at least 2, and on the average 3, program committee members or additional reviewers. The program committee maintained a very high standard of quality and decided to accept 6 papers.

This workshop was the tenth in a series of annual CHR workshops. It means to bring together in an informal setting, people involved in research on all matters involving CHR, in order to promote the exchange of ideas and feedback on recent developments. Previous workshops on Constraint Handling Rules were organized in 2004 in Ulm (Germany), in 2005 in Sitges (Spain) at ICLP, in 2006 in Venice (Italy) at ICALP, in 2007 in Porto (Portugal) at ICLP, in 2008 in Hagenberg (Austria) at RTA, in 2009 in Pasadena (California, USA) at ICLP, in 2010 in Edinburgh (Scotland) at ICLP, in 2011 in Cairo (Egypt) at the second CHR summer school, and in 2012 in Budapest (Hungary) at ICLP.

More information about CHR is available on the CHR website.¹ The papers from all previous editions of the CHR workshop (as well as many other CHR-related papers) are also available for download from the CHR website.²

We are grateful to all the authors of the submitted papers, the program committee members, and the reviewers for their time and efforts. We would also like to thank the workshop steering committee and summer school organizers, Thom Frühwirth and Slim Abdennadher.

July 13th, 2013

Henning Christiansen and Jon Sneyers

¹ CHR website: <http://dtai.cs.kuleuven.be/CHR>

² CHR bibliography: <http://dtai.cs.kuleuven.be/CHR/biblio.shtml>

Table of Contents

Implementation of the Operational Semantics for CHR with User-defined Rule Priorities	1
<i>Slim Abdennadher, Ghada Fakhry and Nada Sharaf</i>	
On Failure-Driven Constraint-Based Parsing through CHRg	13
<i>Veronica Dahl, Sinan Egilmez, Joao Martins and Emilio Miralles</i>	
CHR Meets MapReduce	25
<i>Amr Osman, Amira Zaki and Slim Abdennadher</i>	
CHRSM and Probabilistic Argumentation Logic	39
<i>Jon Sneyers, Daniel De Schreye and Thom Fruehwirth</i>	
CHR-IDE: An Eclipse Plug-In for Constraint Handling Rules	53
<i>Amira Zaki, Matthias Rau and Thom Fruehwirth</i>	
Tagging Reasoning System using CHR	61
<i>Nada Sharaf, Injy Hamed and Slim Abdennadher</i>	

Program Committee

Slim Abdennadher	German University of Cairo
María Alpuente	UPV
Henning Christiansen	Roskilde University
Marina De Vos	University of Bath
Gregory Duck	National University of Singapore
François Fages	INRIA Rocquencourt
Thom Frühwirth	University of Ulm
Maurizio Gabbrielli	University of Bologna
Marco Gavanelli	University of Ferrara
Rémy Haemmerlé	Technical University of Madrid
Michael Maher	University of New South Wales, Canberra
Thierry Martinez	INRIA Paris-Rocquencourt
Maria Chiara Meo	Dipartimento di Scienze
Eric Monfroy	UTFSM and LINA
Tom Schrijvers	Ghent University
Jon Sneyers	KU Leuven
Armin Wolf	Fraunhofer FIRST

Additional Reviewers

Eric De La Clergerie
Jacopo Mauro
Christian Theil Have
Alicia Villanueva
Herbert Wiklicky
Amira Zaki

Implementation of the Operational Semantics for CHR with User-defined Rule Priorities

Slim Abdennadher, Ghada Fakhry, and Nada Sharaf

Computer Science and Engineering Department,
The German University in Cairo
{`slim.abdennadher,ghada.fakhry,nada.hamed`}@guc.edu.eg

Abstract. In this paper, we present an implementation of the operational semantics for CHR with user-defined rule priorities CHR^{rp} and CHR with persistent constraints $w_!$. The proposed implementation uses source-to-source transformation to convert programs in CHR^{rp} and $w_!$ into equivalent programs in the CHR refined operational semantics without the need to change the compiler or the runtime system.

1 Introduction

Constraint Handling Rules [5] is a high-level language designed for writing constraint solvers. CHR is a committed-choice language based on multi-headed and guarded rules. Over the past decade, the number of CHR extensions has increased as well as the CHR compilation techniques [9]. Extensions and variants of CHR were proposed with operational semantics different than the refined operational semantics w_r [4] with respect to the execution control, expressivity, declarativity and termination of CHR programs.

This work builds on the tool presented in [1] which introduced a transformation approach that is able to add visualization features to CHR programs without changing the compiler or the runtime system. The presented tool was built with the ultimate goal of having a generic CHR source-to-source transformation workbench for different purposes.

The aim of this work is to introduce an approach to simulate the execution of the different operational semantics of CHR using the refined operational semantics implemented in SWI-Prolog without changing the compiler or the runtime system. The proposed approach uses source-to-source transformation to convert CHR programs written under different operational semantics to equivalent programs that could be used with the refined operational semantics w_r .

In order to reach a generic tool that is able to transform from any operational semantics to the refined operational semantics w_r , different semantics should be implemented with the aim of reaching a generic approach. In this paper, we start by implementing the operational semantics for CHR^{rp} [8]. Having CHR^{rp} running using the refined operational semantics can be further used for implementing the operational semantics $w_!$ for persistent constraints as introduced in [3].

The paper is organized as follows. In section 2, a source-to-source transformation for CHR with user-defined rule priorities is introduced. Section 3 shows how

the same transformation approach is to be used to provide an execution model for CHR with persistent constraints. Finally, we conclude with a summary and a discussion of future work.

2 Transformation for Constraint Handling Rules with User-defined Rule Priorities

CHR^{rp} extends CHR with user-defined rule priorities [8]. Rule priorities improve the expressivity of CHR as it allows certain rules to be applied before other rules according to their priority, resulting in a more flexible execution control. The main idea of CHR^{rp} is that if there is more than one rule to be applied at a certain step, the choice of the rule to be applied is determined by the rule priorities, compared to the textual order of the rules in the refined operational semantics w_r . In [8], a source-to-source transformation approach that uses some of the compiler directives is presented. In [8], constraints are not activated when introduced to the store by the “Activate” or “Reactivate” transition [4]. Instead, they remain passive using the compiler directive `passive/1` and are scheduled for activation with the corresponding rule priority. After trying all the possible matching rules with the constraints in the query, the highest priority scheduled constraint is activated.

In this section, CHR programs written under CHR^{rp} operational semantics are transformed to new CHR programs that are equivalent when executed under the refined operational semantics w_r [4]. Building on the representation used in [1], the rules of the source program rules are also transformed into the “relational normal form” introduced in [7]. This normal form uses special CHR constraints that represent the components of a rule. However, instead of using `head/4`, `head/3` is used, since the information about the constraint identifier is not needed by the transformer. The solver is first parsed. The parser extracts the needed information and represents it in the needed format. The transformer is a CHR solver that runs on the relation normal formal of the source program and writes the new rules into the transformed program file.

The idea of the transformed program is to schedule all the applicable rules at each computational step instead of scheduling the constraints. Afterwards, we choose the rule with the highest priority for application. The idea is implemented similar to the conflict resolution model introduced in [6]. For every rule in the source program there are two rules in the transformed program. The first rule performs the rule matching and the second rule applies the body of the rule. separating the rule matching and rule application allows multiple rules to be matched at a certain computation step. The choice of the rule to be applied among the matched rules depends on the priority conflict resolution strategy. Section 2.1 explains the structure of the transformed program and Section 2.2 explains an explicit propagation history implementation in the transformed program.

2.1 Transformed Program Structure

This section explains the transformation steps of the source program. The rules are written such that the execution of one rule in the source program is done in 4 steps in the transformed program.

In the first step, similar to [8] for every CHR constraint $c(X)$ in the source program, a simplification rule **extend** is added to the transformed program P' :

extend @ $c(X) \Leftarrow c(X, V)$.

so that when executing the transformed program, an extended CHR constraint $c(X, V)$ is created for each CHR constraint $c(X)$, where V is a fresh Prolog variable. V uniquely identifies a constraint and is used in the implementation of the propagation history explained in Section 2.2.

The second step in the transformation adds a propagation rule **match** in the transformed program for each rule in the source program. A **match** rule is a propagation rule with the same head constraints and guards of the source program rule. The body of **match** is a new CHR constraint **cand**/3 whose arguments are the rule name, the list of identifiers of the rule head constraints, and a number p that represent the probability of the rule.

In the CHR refined operational semantics, head constraints are searched from left to right, except for the heads of the simpagation rules. In simpagation rules, the head constraints to be removed are tried before the constraints to be kept [6]. In order to preserve the same head matching of the rules in the source program, removed head constraints are added before the kept head constraints in the **match** transformed rules.

For every CHR rule in the source program P :

$$r_p @ H^k \setminus H^r \Leftrightarrow G \mid B.$$

we will have the following rule in the transformed program P' :

$$\text{match-}r_p @ H^r, H^k, \text{id}(Ni) \Rightarrow G \mid \text{cand}(ri, Ids, p).$$

The operational semantics of CHR^{rp} allows multiple rules matching at each computational step and only one rule application according to the priority. An additional constraint **id**/1 is added to the rule head constraints. Changing the value of the argument of **id**/1 after each rule application allows the propagation rule to be matched with the same constraints instances multiple times. Also, a propagation **trigger** rule is added at the end of the matching rules that adds the CHR constraint **start**/0 which triggers the conflict resolution. **trigger**/0 is a new constraint added to the end of the original query constraints to ensure that it is only activated after all the original query constraints.

trigger @ **trigger**, **id**(Ni) $\Rightarrow$ **start**.

Whenever a rule is applicable, a new constraint **cand**/3 is created for this rule, meaning that this rule could be applied with constraints whose identifiers are in the list. However, in the case of simplification and simpagation rules, the

constraints that were matched in the head of the rule are not removed at the rule matching step. In the execution of the transformed program, the result of the rule matching step is a set of all the applicable rules, each of the candidate rules is represented by the constraint **cand/3**.

In the transformed program, the rule with the highest priority among the set of applicable rules is chosen to be applied. The three rules **start**, **collect**, and **choose** are added to the transformed program to perform the conflict resolution according to the priorities of the rules. The rule **start** initializes an empty priority list such that the applicable rules represented by **cand/3** are added to this list by the **collect** rule. After all **cand/3** constraints are added to the list, the rule **choose** sorts the list and the rule with the highest priority is chosen for application. The chosen rule is represented by a new **CHR** constraint **fire/2**, whose first argument is the rule name and the second argument is a list of identifiers of the rule head constraints. The list of constraint identifiers is added to ensure that the rule will only be applied with the specific combination of constraints. The tail of the list is not used, in the next computation step the priority list will be constructed again; because the set of applicable rules changes at each computation step. In addition, the argument of **id/1** constraint is incremented to allow **match** rules to be tried again.

```

start   @ start   <=> priorityList([]).
collect @ priorityList(L),cand(R,IDs,N)
        <=> priorityList([(N,R,IDs)|L1]).
choose  @ priorityList(L),id(Ni)
        <=> sort(L,[(P,H,IDs)|T]),fire(H,IDs),
           N2 is Ni+1,id(N2).

```

The last part in the transformed file is the rule application. For each rule in the source program, an apply rule **apply** is added to the transformed program. In **apply** rules, a new constraint **fire/2** is added to the head constraints to be removed of the source program rule (if any) to ensure that only this rule will be applied with the specific combination of constraints in the list. The body is the same as the body of the source program rule. For every **CHR** rule in the source program **P**:

$$r_p @ H^k \setminus H^r \Leftrightarrow G \mid B.$$

we will have one of the following two rules in the transformed program **P'**:

$$apply-r_p @ H^k \setminus fire(r_p,Ids) , H^r \Leftrightarrow G \mid B.$$

Figure 1 shows the execution of the transformed program. The transformed program starts by extending the original query constraints, and then the extended constraints try to match the propagation rules in the second part of the transformed file. Then among the set of the applicable rules represented by **cand/3** constraints on rule is chosen according to the rule priority and then applying the chosen rule in the last part. The execution then proceeds by extending the

added constraints after the rule application (if any), matching propagation rules, conflict resolution and so on until reaching a fix point where no more rules are applicable. The transformation depends mainly on the textual order of the rules in the transformed file since it runs using the refined operational semantics implemented in SWI Prolog.

Fig. 1: Transformed Program Structure.

2.2 Propagation History Implementation

CHR^P allows multiple rules matching at each computation step and one rule application, then the set of matched rules that were not applied at one computation step are given a second chance in the next computation step. In order to allow multiple rules matching, the constraint `id/1` is added to the head constraints of the transformed rules. This approach raised a problem with propagation rules; if a propagation rule was chosen to be fired among the set of candidate rules, then in the next computation step the same propagation rule will also be applicable, because the constraints that matched the rule head were not removed, causing a problem of trivial non-termination in the transformed program.

In order to solve this problem, every propagation rule should be fired only once for each specific combination of constraint identifiers. In the proposed approach, a new constraint `history/1` is added to the original query. The argument of `history` is a list, that contains a set of tuples (r, I) where r is the propagation rule name and I is an ordered list of the constraints identifiers that matched the head constraints of rule r . The list is originally empty.

In second step performed by the transformed program, propagation rules are modified such that the rule can be applied only if the tuple containing the rule name and the list of constraint identifiers does not exist in the propagation history. For example, the following rule in the original program:

```
rule1 @ a(X) ==> b(X).
```

is transformed through the second step into:

```
match-rule1 @ a(X,Id1),history(L) ==> \+ member((rule1,[Id1]),L)
                                     |cand(rule1,[Id1],1).
```

In addition, if a propagation rule is applied, a new tuple with the rule name and constraints identifiers list is added to the propagation history. Thus, the propagation rules are modified in step 4 to update the propagation history. The rule in the previous example will be transformed in step 4 into:

```
apply-rule1 @ a(X,Id1)\ fire(rule1,[Id1]),history(L)
               <=> b(X),history([(rule1,[Id1])|L]).
```

2.3 An Example

The following example [8] shows the difference in the execution of the refined operational semantics and CHR^p operational semantics. For the same initial query **a**, the refined operational semantics will apply the rules in the following order: 1,2,4,3. In the refined operational semantics, rules are applied in top-down in the textual order of the program. In addition, body constraints of the rules are added on top of the execution stack. In the beginning, the active constraint **a** matches the head of **r1_1**. Applying **r1_1** adds the constraint **b** on top of the stack. Now the constraint **b** is the active constraint. The constraint **b** tries to match head constraint in rules in the textual order. Accordingly, rule **r2_2** is applied and then **r4_4**. Now **b** becomes passive and **a** is the active constraint. Constraint **a** matches the head of **r3_3** and **r3_3** is applied. While in CHR^p , after applying rules **r1_1** and **r2_2**, rules **r3_3** and **r4_4** could be applied. Since **r3_3** has higher priority than **r4_4**, it is applied. Therefore, the rules will be applied in the following order: 1,2,3. Rule **r4_4** will not be applied anymore because constraint **a** is removed by **r3_3**. The same example is used to illustrate the proposed transformation approach.

```
r1_1 @ a ==> print('rule 1'),b.
r2_2 @ a,b ==> print('rule 2').
r3_3 @ a <=> print('rule 3').
r4_4 @ a,b ==> print('rule 4').
```

The first transformation step extends the source program constraints with fresh Prolog variables. **a/0** and **b/0** will be extended to **a/1** and **b/1**. The following two **extend** rules are added to the transformed program:

```

extend @ a <=> a(_).
extend @ b <=> b(_).

```

The second step transforms source program rules into propagation rules and the body of the rules is replaced by `cand/3`. The first argument of `cand/3` is the rule name, the second argument is a list of the identifiers of the constraints that were matched in the rule head and the third argument of `cand/3` is the respective rule priority. In this example, rules `r1_1`, `r2_2`, and `r4_4` are propagation rules, therefore the propagation history check is applied.

```

match-r1_1 @ a(Id0),id(Ni),history(L) ==> \+ member((r1_1,[Id0]),L)
                                         | cand(r1_1,[Id0],1).

match-r2_2 @ a(Id0),b(Id1),id(Ni),history(L)
                                         ==> \+ member((r2_2,[Id0,Id1]),L)
                                         | cand(r2_2,[Id0,Id1],2).

match-r3_3 @ a(Id0),id(Ni) ==> cand( r3_3,[Id0],3).

match-r4_4 @ a(Id0),b(Id1),id(Ni),history(L)
                                         ==> \+ member((r4_4,[Id0,Id1]),L)
                                         | cand(r4_4,[Id0,Id1],4).

trigger @ trigger, id(Ni) ==> start.

```

The next three rules are added to the transformed program to perform the conflict resolution:

```

start @ start <=> priorityList([]).
collect @ priorityList(L),cand(R,IDs,N)
          <=> priorityList([(N,R,IDs)|L1]).
choose @ priorityList(L),id(Ni)
          <=> sort(L,[(P,H,IDs)|T]),fire(H,IDs),
          N2 is Ni+1,id(N2).

```

In the rule application transformation step, the propagation history is updated for rules `r1_1`, `r2_2`, and `r4_4` by appending a tuple containing the rule name and the list of constraints identifiers that matched the rule head to the propagation history list.

```

apply-r1_1 @ a(Id0) \ fire(r1_1,[Id0]), history(L)
              <=> print('rule 1'),b,
              history([(r1_1,[Id0])|L]).

apply-r2_2 @ a(Id0),b(Id1) \ fire(r2_2,[Id0,Id1]), history(L)
              <=> print('rule 2'),
              history([(r2_2,[Id0,Id1])|L]).

```

```

apply-r3_3 @ fire(r3_3,[Id0]), a(Id0) <=> print('rule 3').

apply-r4_4 @ a(Id0),b(Id1) \ fire(r4_4,[Id0,Id1]), history(L)
                                <=> print('rule 4'),
                                history([(r4_4,[Id0,Id1])|L]).

```

3 Transformation for Constraint Handling Rules With Persistent Constraints

In [3], an operational semantics for CHR with persistent constraints w_l was introduced. The operational semantics w_l solves the problem of trivial non-termination of CHR programs by providing a terminating execution model for propagation rules. The main idea of w_l is based on the concept of persistent constraints, which provide a finite representation of the result of any number of firings of propagation rules. For a propagation rule, the body can be generated any number of times as long as the head constraints of the rule are present in the constraint store. Accordingly, the body of a propagation rule that can be fired many times is considered to be persistent. In addition, the body of a simplification or a simplification rule is considered to be persistent whenever the head constraints of the rule can be matched with completely persistent constraints.

Currently there is no direct implementation for the operational semantics w_l . In [2], source-to-source transformation is used to provide an implementation w_l . The source program P written in w_l is transformed into the program P' in CHR^p such that the execution of P' in CHR^p is equivalent to the execution of P in w_l . The execution model in [2] relies on the existing implementation of CHR^p which uses the compiler directives. In this paper, an implementation of w_l with source-to-source transformation to the refined operational semantics w_r is proposed. Figure 2 shows the general architecture of the transformation.

Fig. 2: Transformed Program Structure.

The Source program P in w_l is first transformed to P' in CHR^{rp} according to the transformation rules presented in [2]. Then the result program P' is transformed to P'' in the refined operational semantics w_r according to the transformation approach presented in Section 2. The final transformed program P'' can then be executed on CHR implemented in SWI Prolog without changing the compiler or the runtime system.

3.1 An Example

Consider the following CHR program for computing the transitive hull of a graph [2]. The edges of the graph are represented by the constraint $e/2$. In the refined operational semantics implementation in SWI Prolog, initial goals containing graphs with cycles will result in non-termination of the program. However, the transitive hull program terminates for all goals in w_l [3].

```
t @ e(X,Y), e(Y,Z) ==> e(X,Z).
```

By applying the transformation rules presented in [2], the program P in w_l will be transformed into P' in CHR^{rp} . The transformation extends the constraints by an argument to indicate the type of the constraint. A constraint with the extended argument l is linear, and p is persistent. The rule priority is concatenated to the rule name. The new program is:

```
r1_3 @ e(l,X,Y),e(l,Y,Z) ==> e(c,X,Z).
r2_3 @ e(l,X,Y),e(p,Y,Z) ==> e(c,X,Z).
r3_3 @ e(p,X,Y),e(l,Y,Z) ==> e(c,X,Z).
r4_3 @ e(p,X,Y),e(p,Y,Z) ==> e(c,X,Z).

r5_3 @ e(p,X,Y) ==> X=Y , Y=Z | e(c,X,Z).

r6_1 @ e(p,X,Y)\e(c,X,Y) <=> true.

constraints @ e(c,X,Y) <=> e(p,X,Y).
```

The next step is to parse the output program P' in CHR^{rp} and run the transformer on the extracted information. The result is the transformed program P'' in w_r . Steps 1 and 2 in the transformed program:

```
extend @ e(V3,V2,V1) <=> e(V3,V2,V1,_).

match-r1_3 @ e(l,X,Y,Id0),e(l,Y,Z,Id1),id(Ni),history(L)
            ==> \+ member((r1_3,[Id0,Id1]),L)
            | cand(r1_3,[Id0,Id1],3).

match-r2_3 @ e(l,X,Y,Id0),e(p,Y,Z,Id1),id(Ni),history(L)
            ==> \+ member((r2_3,[Id0,Id1]),L)
            | cand(r2_3,[Id0,Id1],3).
```

```

match-r3_3 @ e(p,X,Y,Id0),e(l,Y,Z,Id1),id(Ni),history(L)
              ==> \+ member((r3_3,[Id0,Id1]),L)
                  | cand(r3_3,[Id0,Id1],3).

match-r4_3 @ e(p,X,Y,Id0),e(p,Y,Z,Id1),id(Ni),history(L)
              ==> \+ member((r4_3,[Id0,Id1]),L)
                  | cand(r4_3,[Id0,Id1],3).

match-r5_3 @ e(p,X,Y,Id0),id(Ni),history(L)
              ==> \+ member((r5_3,[Id0]),L),X=Y,Y=Z
                  | cand(r5_3,[Id0],3).

match-r6_1 @ e(c,X,Y,Id0),e(p,X,Y,Id1),id(Ni) ==> cand( r6_1,[Id0,Id1],1).

match-r7_2 @ e(c,X,Y,Id0),id(Ni) ==>  cand( r7_2,[Id0],2).

trigger @ trigger, id(Ni) ==>  start.

```

then the conflict resolution:

```

start @ start <=> priorityList([]).
collect @ priorityList(L),cand(R,IDs,N)
          <=> priorityList([(N,R,IDs)|L1]).
choose @ priorityList(L),id(Ni)
          <=> sort(L,[(P,H,IDs)|T]),fire(H,IDs),
              N2 is Ni+1,id(N2).
              fire(H,IDs),id(N2).

```

finally, the rule application step is performed:

```

apply-r1_3 @ e(l,X,Y,Id0),e(l,Y,Z,Id1) \
             fire(r1_3,[Id0,Id1],1),history(L) <=> e(c,X,Z),
             history([(r1_3,[Id0,Id1])|L])).

apply-r2_3 @ e(l,X,Y,Id0),e(p,Y,Z,Id1) \
             fire(r2_3,[Id0,Id1],1), history(L) <=> e(c,X,Z),
             history([(r2_3,[Id0,Id1])|L])).

apply-r3_3 @ e(p,X,Y,Id0),e(l,Y,Z,Id1) \
             fire(r3_3,[Id0,Id1],1), history(L)<=> e(c,X,Z),
             history([(r3_3,[Id0,Id1])|L])).

apply-r4_3 @ e(p,X,Y,Id0),e(p,Y,Z,Id1) \
             fire(r4_3,[Id0,Id1],1), history(L) <=> e(c,X,Z),
             history([(r4_3,[Id0,Id1])|L])).

```

```

apply-r5_3 @ e(p,X,Y,Id0) \ fire(r5_3,[Id0],1), history(L)
                                <=> X=Y,Y=Z | e(c,X,Z),
                                history([(r5_3,[Id0])|L]).

apply-r6_1 @ e(p,X,Y,Id1) \ fire(r6_1,[Id0,Id1],1), e(c,X,Y,Id0)<=> true.

apply-r7_2 @ fire(r7_2,[Id0],1), e(c,X,Y,Id0) <=> e(p,X,Y).

```

Executing the transformed program P'' in the refined operational semantics w_r with initial goal: $e(1,a,b), e(1,b,a)$ will terminate with the following constraints in store: $e(1,a,b), e(1,b,a), e(p,a,a), e(p,a,a), e(p,a,b), e(p,b,a)$. The result state is equivalent to the result of an execution of the original program P in w_l with the same initial goal.

4 Conclusion

This paper introduced a source-to-source transformation approach for implementing the operational semantics of CHR^{rp} . The advantage of the proposed approach over the previous ideas is that the transformation to the refined operational semantics w_r eliminated the need to change the compiler or the runtime system. In addition, the implementation of CHR^{rp} is used to provide an implementation for the operational semantics w_l for persistent constraints in the refined operational semantics w_l . However the computational complexity of the transformed programs is out of the scope of this paper.

For future work, we intend to extend the current transformation approach to allow dynamic rule priorities. In addition to considering the implementation of different CHR operational semantics with the aim of reaching a generic source-to-source transformation tool that simulates the execution of CHR different operational semantics by transforming into the refined operational semantics w_r .

References

1. Slim Abdennadher and Nada Sharaf. Visualization of CHR through Source-to-Source Transformation. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 109–118, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
2. Hariolf Betz, Frank Raiser, and Thom Frühwirth. A complete and terminating execution model for Constraint Handling Rules. volume 10(4–6), pages 597–610. July 2010.
3. Hariolf Betz, Frank Raiser, and Thom Frühwirth. Persistent constraints in Constraint Handling Rules. In A. Wolf and U. Geske, editors, *WLP '09: Proc. 23rd Workshop on (Constraint) Logic Programming*. Universität Potsdam, 2010.

4. Gregory J. Duck, Peter J. Stuckey, Maria Garcia de la Banda, and Christian Holzbaaur. The refined operational semantics of constraint handling rules. In *IN 20TH INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING (ICLP04)*, pages 90–104. Springer, 2004.
5. Thom Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(13):95 – 138, 1998.
6. Thom Frühwirth. *Constraint Handling Rules*. August 2009.
7. Thom W. Frühwirth and Christian Holzbaaur. Source-to-source transformation for a class of expressive rules. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE*, pages 386–397, 2003.
8. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for chr. In Michael Leuschel and Andreas Podelski, editors, *PPDP*, pages 25–36. ACM, 2007.
9. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. 10(1):1–47, 2010.

On Failure-Driven Constraint-Based Parsing through CHR[★]

Veronica Dahl¹, Sinan Eğılmez², João Martins^{2,3} and J. Emilio Miralles¹

¹ Simon Fraser University, Burnaby, BC V5A-1S6, Canada,
veronica@cs.sfu.ca, emiralle@sfu.ca

² CENTRIA and Departamento de Informática, FCT, Universidade Nova de Lisboa

³ Computer Science Department, Carnegie Mellon University, Pittsburgh PA
s.egilmez@campus.fct.unl.pt, jmartins@cs.cmu.edu

Abstract. In this article we adapt and further develop a CHR parsing technique which was initially conceived in the context of grammar induction for Womb Grammars [8]. We also show that applying it to constraint-based linguistic formalisms such as Property Grammars [3] can yield truly direct implementations of specialized parsers that focus on ungrammaticality detection and correction.

1 Introduction

Since the advent of CHR [12] and of its grammatical counterpart CHR_G [7], constraint-based linguistic formalisms can materialize through fairly direct methodologies. In this article we study several advantages of doing so. Efficiency-wise, direct CHR_G renditions of linguistic constraints promote efficiency by their very adherence to the constraint solving framework. We propose to enhance them further through exploiting our novel methodology, arrived at through our work on grammar induction [8], of checking only for falsification of most constraints in PG [3] and similar formalisms.

Constraint-based theories of grammar are by now widespread in computational linguistics, but there is still no general consensus on what comes under that umbrella.

Shieber’s initial characterization of constraint-based theories in terms of common threads such as modularity, declarative constructs and partial information [17] has been influential to this day, partly because it is wide enough to accommodate a variety of grammar theories springing from different fields – linguistics, computational linguistics and artificial intelligence. However such threads are present in many formalisms that would not be viewed as constraint based in the computational science of constraint solving as originally described [18].

The constraint solving paradigm of computing sciences has proved very successful in greatly reducing search spaces by stating a problem in terms of constraints on domain-associated variables, whose possible values are automatically

[★] This paper was supported by NSERC Discovery grant 31611024, and developed during a visit to Universidade Nova de Lisboa

and successively narrowed down through constraint solving into values that represent a solution, if one exists. It would be therefore most interesting from the point of view of efficiency to consider to what extent the so-called constraint based grammar models fit into the constraint solving model strictly speaking.

The candidate constraint-based theories of grammar to consider, if we follow Shieber’s characterization, range widely from augmented transition networks [19] to logic grammars [1], to recent and highly specialized formalisms such as [14], or Womb grammar parsing [8]. Among them, the Property Grammar (PG) framework [3] stands out as an effort to completely describe a grammar in terms of constraints. It defines phrase acceptability in terms of the properties or constraints that must be satisfied by groups of categories (e.g. English noun phrases can be described through a few constraints such as precedence (a determiner must precede a noun), uniqueness (there must be only one determiner), exclusion (an adjective phrase must not coexist with a superlative), and so on). Rather than resulting in either a parse tree or failure, such frameworks characterize a sentence through the list of the constraints a phrase satisfies and the list of constraints it violates, so that even incorrect or incomplete phrases will be parsed to the extent that they can, rather than simply failing.

Because of their sole reliance of constraints, Property Grammars are a main candidate for direct implementation in terms of constraint solving. However the only works which incorporate direct implementation to some extent are [5], [9], [10] and Womb Parsing [8]. Of these, the only one that does not need to calculate all constraints between every pair of constituents is [8]. Instead, it checks constraints only for failure. This works well in the context of grammar induction. In the present paper we examine how to adapt that work to parsing sentences rather than inducing grammars, while retaining both the search space reduction obtained by the failure-driven focus and the direct implementation character (in the constraint-solving sense) of [8].

2 Background

2.1 Property Grammars

The idea of representing a language’s grammar solely through properties between constituents was first proposed as a theoretical formalism by Gabriel Bes [2], and reworked by Philippe Blache into Property Grammars [3,4]. Computationally, it relates to Gazdar and Pullum’s dissociation of phrase structure rules into the two properties of Immediate Dominance (called constituency in the PG literature) and Linear Precedence (called either precedence or linearity in PG) [13]. It presently comprises the following seven categories (we adopt the handy notation of [10] for readability, and the same example):

Constituency $A : S$, children must have categories in the set S

Obligation $A : \triangle B$, at least one B child

Uniqueness $A : B!$, at most one B child

Precedence $A : B \prec C$, B children precede C children

Requirement $A : B \Rightarrow C$, if B is a child, then also C is a child

Exclusion $A : B \not\Leftarrow C$, B and C children are mutually exclusive

Dependency $A : B \sim C$, the features of $C1$ and $C2$ are the same

This paper will handle full sentences, so that we will generally denote determiners by D , nouns by N , personal nouns by PN , verbs by V , noun phrases by NP , verb phrases by VP and sentences by S .

Example 1. For example [11], the context free rules $NP \rightarrow D N$ and $NP \rightarrow N$, which determine what a noun phrase is, can be translated into the following equivalent constraints: $NP : \{D, N\}, NP : D!, NP : \triangle N, NP : N!, NP : D \prec N, D : \{\}, N : \{\}$.

The larger number of constraints allows us to perceive and understand the grammar rules in a more detailed fashion. Furthermore, this finer granularity can be exploited: in some of the literature on PG there is the possibility of declaring some constraints as relaxable. The failure of relaxable constraints is signalled in the output, but does not block the entire sentence’s analysis. Implementations not including constraint relaxation capabilities implicitly consider all properties as relaxable.

2.2 Womb Grammar Parsing

Womb Grammar Parsing is a constraint-based parsing methodology developed in 2012 [8], motivated by the need to aid the world’s linguist uncover the syntax of the many languages that are not being studied for lack of resources. It was designed to induce a target language’s syntax from the known syntax of a source language plus a representative corpus of correct sentences in the target language and the target language’s known lexicon. It was presented in two versions: Hybrid Womb Parsing, in which the source language is an existing language for which the syntax is known, and Universal Womb Parsing, in which the source syntax is a hypothetical universal grammar of the authors’ own devise, which contains all possible properties between pairs of constituents.

Womb Parsing has the originality of addressing through constraint solving a problem which more usually is cast as a machine learning problem. Additionally, it fully applies the technique of using linguistic information from one language for the task of describing another language, which until then had yielded good results only for specific tasks—such as disambiguating the other language [6], or fixing morphological or syntactic differences by modifying tree-based rules [15]—rather than for syntax induction.

3 From Womb Parsing to Direct PG Parsing

3.1 The Main Idea

Womb Parsing works by adjusting the constraints given in the source grammar until they suit the input corpus. Since this corpus is chosen to be correct and

representative, we’re justified e.g. in deleting any constraints that the corpus violates.

For instance, if the source grammar contains the precedence constraint $NP : N \prec ADJ$ and there is an input noun phrase in which an adjective precedes a noun, a CHRG rule will apply and delete the above precedence constraint. Thus, the constraint was checked for violation, not satisfaction.

Each of the properties in PG has similar CHRG rules associated with it. Our proposal here is to adapt these rules into admitting constraint relaxation and signalling failure of non-relaxed constraints that do not satisfy input sentences, rather than adjusting the constraints that failed. Furthermore, the method works on complex sentences rather than just noun phrases.

3.2 Significance

The Womb Parsing method of adapting another language’s grammar constraints until they suit the target language’s representative input corpus can inspire a new way of viewing the PG *parsing problem*, in which constraints are tested only for failure. In contrast, all previous methods exhaustively test each constraint for all constituents that can participate in it. Concretely, a notion not unlike obligation can be used to identify new phrases, and those phrases can be tentatively expanded from nearby constituents.

Example 2. In “john eats an apple”, the noun indicates the existence of a noun phrase. Because noun $NP : \{D, N\}$, then the tentative noun phrase starting at “apple” can include “an” but not “eats”, since it can only be constituted by determiners and nouns. This is the constraint-satisfaction criterion for expansion.

For each tentatively expanded phrase, all other constraints are tested for *failure* only. The phrase is allowed to expand only if either no constraint fails, or all constraints that fail have been declared as relaxable. Exhaustive satisfaction check is thus replaced by a smart guided search for a falsifying assignment. This is appropriate provided that the set of satisfied constraints is the exact complement of the set of failed constraints - an assumption that seems reasonable, and that we make. In this case, it follows that we do not need to check the satisfied constraints. Just as for grammar induction we do not need to touch the constraints of the source grammar that do work for the target grammar, we can for PGs only actively check the constraints that do not hold. Should we need to explicitly output those that hold, they could be inferred from the list of constraints that must be satisfied plus those output as unsatisfied, at less computational cost than the usual practice of evaluating all constraints between every pair of constituents, or of adding heuristics to reduce the search space.

This is significant because deep parsing with Property Grammars is theoretically exponential in the number of categories of the grammar and the size of the sentence to parse [16]. Since all previous approaches to PG parsing (except for Womb Parsing) have to calculate all constraints between every pair of constituents, and since the number of failed constraints will in general be much

smaller than the number of satisfied constraints, any parsing methodology that manages to mostly check the failed ones will have a substantial efficiency advantage. If moreover the failed constraints can be checked in the automatic workings of a truly constraint solving formulation like our CHRG one, by means of letting those rules that apply to each particular input sentence trigger, we now have more realistic hopes of turning (our version of) the PG paradigm, which we call Direct PG, into a useful, directly executable constraint-based theory of language. In this first paper we address this hope from the point of view of direct parsing for ungrammaticality detection.

3.3 Phrase Determination

In [8], where we only dealt with noun phrases, constituency did not need to be checked explicitly, because it followed from a lexicon made only from noun phrase constituents. The core idea was to start the noun phrase from its head noun, and try to expand it. Expansion would fail if any of the grammatical constraints failed.

More concretely, one expands instantiated categories, which are CHRG predicates of the type `iCat(Start, End, CategoryType, Attributes, Tree)`. Instantiated categories simply keep track of the location, attributes (gender and plurality) and parse tree of all phrase types, or *categories*, in a specific sentence (e.g. *N*, *NP*, etc).

Example 3 (Instantiated categories). Take the noun phrase “an apple”. Parsing it results in the following instantiated categories.

```
iCat(0, 1, det, [sing,neutral], det(an))
iCat(1, 2, n,   [sing,neutral], n(apple))
iCat(0, 2, np,  [sing,neutral],
  np(
    iCat(0, 1, det, [sing,neutral], det(an)),
    iCat(1, 2, n,   [sing,neutral], n(apple))
  )
)
```

Notice that the *NP* inherits the attributes of the underlying *N*. This is useful for checking dependency constraints, which require attributes to match. Parse trees are built as a side-effect from parsing, and the trees of a leaf are much simplified.

Since in this work we need to treat whole sentences rather than just noun phrases, some additional considerations are necessary:

- Whereas in [8] it was implicit that noun phrases grew from their head nouns, for full sentences each phrasal category must explicitly have its own *head*. These heads follow closely from the grammar itself and the obligation constraints it implies. In our case, we have:

`head(n, np). head(pn, np). head(v, vp). head(vp, sentence).`

- For full sentences, the extra complexity requires phrases to be treated in a specific ordering. For example, because a *VP* can contain a *NP*, *NPs* should be parsed before *VPs*, and *VPs* should be parsed before *Ss*. The ordering is currently specified as part of the grammar, though it could be induced from the directed constituency graph implied by a grammar. For the sentences we consider, the following order is used:

`parseOrder([np, vp, sentence]).`

After all categories of the type at the head of the list have been parsed and expanded, the algorithm moves on to the next category.

`parseOrder([_|L]) <=> parseOrder(L).`

- While the PG formalism per se does not care about syntactic trees (it just characterizes an input sentence through its lists of satisfied and unsatisfied properties), they are important for complex sentences, and are thus built explicitly by the algorithm. Not only is this handy to the traditional way in which linguists are used to thinking of a parse result, but it is also useful to ensure that the constraints apply on immediate daughters only. For instance, consider the parse tree of “john eats an apple” in Figure 1. It contains two noun phrases, “john” and “an apple”, but only ‘john’ is a direct daughter of the sentence (“an apple” is a direct daughter of the verb phrase “eats an apple”). Therefore, the uniqueness of a noun phrase *NP* in a sentence *S*, *S: NP!*, does not fail, since only the direct daughter noun phrase “john” is identified as the sentence’s noun phrase.

Fig. 1. Parse tree of the sentence.

Taking all these considerations into account, the rule for creating new categories from their heads is the following.

```

iCat(N1, N2, Comp, Attr, Tree), {parseOrder([Cat|_])}
::> head(Comp, Cat), NewTree=..[Cat,iCat(N1,N2,Comp,Attr,Tree)]
| iCat(Cat, Attr, NewTree).

```

This rule is found after the expansion and constraint satisfaction checks. This means it will only fire when `iCat(N1,N2,Comp,Attr,Tree)` is maximally expanded, and when it does not violate unrelaxable constraints. After the rule fires, there will be a new instantiated category, which will in turn be expanded maximally, and checked against the constraints. Once no more checks or expansions can be done on any category of the current type, the parsing moves on to the next category type given by `parseOrder\1`.

The expansion rules are a little more verbose, but the reader should now be acquainted with most predicates used. We present here a simplified version of the left-expansion rule (the right-expansion rule is symmetric).

```

!iCat(N1, N2, Comp, Attr1, Tree1),
iCat(N2, N3, Cat, Attr2, Tree2),
!{tpl(constituency(Cat, L))},!{evil(EL)},
!{parseOrder([Cat|_])}
<:> member(Comp, L),
    buildTree(Cat, iCat(N1,N2,Comp, Attr1, Tree1),
               iCat(N2,N3,Cat , Attr2, Tree2),
               Tree)
| iCat(N1, N3, Cat, Attr2, Tree).

```

Expansion considers two adjacent categories only. The component category will not be removed (e.g. the *D* of an *NP*), whereas the `iCat` of the expanding category is replaced by its expanded version. This rule is guided by constituency, so that senseless expansions do not occur, e.g. *NP* can be expanded to include an adjacent *D* (to the left or right!) but not a *V*. A consequence of this approach is that the constituency constraint is not relaxable.

Notice that the only constraint checked here is constituency. The other constraints are deferred until after the expansion. Whenever an expansion causes some constraint to be falsified, a rule will fire to alert us to that effect. If the falsified constraint is relaxable, the algorithm is allowed to continue. If it is not, however, the expansion that caused the falsification is retracted, and reverts back to its unexpanded predecessor.

We now present a couple of simplified examples of such constraint violation rules for the reader's benefit. We first show the rule that checks for the violation of uniqueness, `Cat : C!`:

```

iCat(N1, N2, C, Attr1, Tree1), % Found a C
..., % and sometime later...
iCat(N3, N4, C, Attr2, Tree2), % Another C
{iCat(N5, N6, Cat, _, Tree)}, % Found a Cat
{tpl(unicity(Cat, C))} % Unicity constraint
::>

```

```

% The C's are within the bounds of Cat
    N5 =< N1, N4 =< N6,
% And they are its direct daughters
    Tree=..[Cat|T],
    member(iCat(N1, N2, C, Attr1, Tree1), T),
    member(iCat(N3, N4, C, Attr2, Tree2), T)
| falsify(uniqueness(Cat,C)):(N1,N4). % Uniqueness falsified!

```

This rule can fire even if a category hasn't been fully expanded - adding more components to an *iCat* is not going to modify uniqueness having been violated. We now present obligation, *Cat* : ΔC , whose rule is structured differently.

```

iCat(N1, N2, Cat, Attr, Tree),      % Found Cat
{tpl(obligation(Cat, C))}           % Cat should have C
::> Tree=..[Cat|T],                % Get children
    not(member(iCat(_,_,C,_,_), T)) % There isn't a child C
| falsify(obligation(Cat, C)).       % Obligation is violated!

```

The obligation rule checks a given category for its direct daughters. Since *iCats* are retracted when they fail, this check only fires after the *iCat* has been fully expanded. Notice the difference with the uniqueness rule discussed above. There is, in fact, an interesting insight regarding the different natures of these rules, which will be expounded upon in Section 3.5.

3.4 Algorithm overview

Since the specifics of the algorithm can get a little involved, it is perhaps helpful to abstract away the rules, their positions and the order in which they fire. In fact, the algorithm is conceptually simple.

1. If there is an *iCat* of the type being handled (e.g. *NP*, *VP*, *S*) that can be expanded without violating unrelaxable constraints, the expansion is carried out. Violated relaxable constraints are stored for reference.
2. If no more expansions are possible but an *iCat* can be a head for the category type being handled, then a new *iCat* of that type is created from the head.
3. If no more expansions are possible and there are no more potential heads, the algorithm proceeds to handle the next category type.

Example 4. Consider our running example of “john eats an apple” again and the following context free grammar (stated in bottom-up fashion), $PN \rightarrow NP$; $D, N \rightarrow NP$; $V, NP \rightarrow VP$; $NP, VP \rightarrow S$, where *PN* is a proper noun and *S* a sentence. *N* and *PN* are the heads of *NP*, *V* of *VP* and *VP* of *S*.

The parse ordering associated with these rules is *NP*, *VP*, *S*. We will consider the constraints induced from the grammar rather than the grammar itself.

The algorithm thus starts by creating the leaf *iCats* for the *PN*, *N*, *D*, and *V* words. Then, the *NPs* “john” and “apple” are created from the respective *PN*

and N heads. The NP “apple” is expandable to “an apple”, and then no more expansions are possible for “john” or “an apple”, since $NP : \{D, N\}$.

The algorithm thus moves to VP s, and create one from the V head “eats”. Since $VP : \{V, NP\}$, the VP is expandable to both left, for “john” and right, for “an apple”. The right expansion succeeds without problems. The left expansion falsifies the precedence $VP : V \prec NP$, and is therefore not carried out.

Since no more expansions are possible for VP s, a S is created from the VP and expanded to “john”, creating the full sentence.

However, suppose that $VP : V \prec NP$ were relaxable. Then, the VP would expand both left and right, encompassing the whole sentence. With no more expansions possible, a sentence S would be created from the VP containing the whole sentence, thereby violating the obligation $S : \triangle NP$ (since the constraint requires NP as a *direct* daughter), and would thus be retracted. This parse would not result in a S , but a NP . If the obligation restriction were, in turn, relaxable, the parse would keep the S instead of retracting it.

If no constraints were relaxable, “john eats apple” would not parse a sentence S (or a VP or even an NP) because nouns must have determiners, i.e. $NP : N \Rightarrow D$. If that were relaxed, however, the entire sentence would be parsed with no further complications! This highlights the flexibility of the approach.

3.5 Types of constraints, and their procedural implications

As we have seen, our parsing algorithm’s reliance on expansion from phrasal heads to incorporate allowable constituents for that phrase places the constraints of constituency and head obligation in the category of non-relaxable, with all other constraints being relaxable.

For relaxable constraints we can identify a further, useful distinction: that between permanent and changeable constraints. Notice that CHR_G rules for constraints such as precedence can be allowed to apply immediately after the two constituents they involve show up in the constraint store, no matter what stage of parsing we are at. This is because as soon as the constraint can be evaluated, its value is *permanent*. Once it fails, adding new constituents to the category cannot make it succeed, since e.g. two unordered elements will remain unordered even when adding more elements around them. Accordingly, our methodology only checks for failure of permanent constraints, regardless of at what stage they fail.

Other constraints, which we call *changing* constraints, can change in their evaluation by the incorporation of one more category into a given phrase for which, without this added category, the constraint had an opposite value. Thus, for “a red apple”, at the point in which “red” and “apple” have been grouped into a noun phrase, this noun phrase will fail the constraint that a noun requires a determiner, $NP : N \Rightarrow D$. However subsequent expansion of the noun phrase into “a red apple” will remedy the failure. This subtype of changing constraints is called *recoverable*. For recoverable constraints, we simply wait until a phrase has been maximally expanded before testing whether they fail. We effect this by applying all expansion rules before checking any recoverable constraints.

For the other type of changing constraints, called *filtering* constraints, the addition of one more element into a phrase might make the constraint fail where it succeeded before. As an example, while unicity of determiner holds for “the book”, it no longer holds for “the the book”. Since the rules that check this type of constraint are triggered only when the constraint fails (e.g. when two determiners show up inside a noun phrase), there is no need for waiting mechanisms, or for any maximal expansion to precede their checking. In this sense our approach is much more focused than previous ones, as a consequence of better fitting the constraint-solving model.

3.6 Constraint Relaxation

Constraint relaxation is indicated by the user as part of the grammar’s definition, through a special system predicate `relaxable(L)`. Any constraints inside the list `L` can be relaxed; the others cannot. If all the *permanent* constraints that are falsified during a phrase’s expansion are relaxable, the constraints are simply added to a list of relaxed constraints and the expansion is accepted. *Changing* constraints are handled slightly differently. As seen, these should only be evaluated once the phrase cannot be expanded anymore. If, at that stage, all falsified changing constraints are relaxable, then the phrase is accepted, and furthermore allowed to be considered as a head of a more complex phrase.

4 Concluding Remarks

We have presented a novel parsing methodology inspired from Womb Grammar Parsing which greatly reduced the combinatorial explosion inherent in PG parsing by three main contributions: a) a truly direct constraint-solving materialization of constraints, b) a great reduction of the number of constraints to be evaluated, through expressing most constraints in terms of failure only, and c) relaxation handling, which is not always present in previous PG renditions.

While a formal comparative complexity analysis would require a previous underlying analysis of the disparate implementation means of the various formulations, we can intuitively trust that our two main contributions above mentioned are bound to enable a much greater degree of efficiency than was previously possible. Roughly speaking, [9] encodes the input PG into a set of CHRG rules that directly interpret the grammar in terms of satisfied or relaxed constraints, which are then propagated while a syntactic tree is built as a side effect. For efficiency, the implementation controls the way in which a constraint is selected for evaluation, but still, the failure or success value of all constraints among every pair of constituents is exhaustively calculated or propagated. [5] resorts to heuristics but remains highly combinatorially explosive, since it involves developing a complete table of satisfied and unsatisfied constraints for every pair of categories involved. [10] is even more combinatorially explosive but interestingly, develops a parsing architecture with full reliance on constraint satisfaction, using classical constraint-based techniques such as branch-and-bound to select and propagate

constraint evaluations. It also allows for the relaxation of constraints, by choosing those solutions that maximize the ratio between satisfied and unsatisfied properties - there is no control over which properties are relaxable, however.

The approach in [9], like our own, also completes original assignments of categories with new categories when they are inferred, and it exploits a similar distinction between permanent and changing constraints to minimize recalculation. However, they need to explicitly inherit permanent constraints at each step, and to recalculate and eventually update at each step the satisfaction value of changing constraints. In contrast, as we have seen, our parsing methodology allows us to simply bypass the need to inherit the value of permanent constraints (through only checking them on maximal phrases), to postpone the checking of recoverable constraints until the phrase has been identified as maximal, and to only check for filtering constraints when and if they do fail. Also in this sense we obtain considerable efficiency gains.

References

1. Abramson, H., Dahl, V.: Logic grammars. Symbolic computation: artificial intelligence, Springer (1989)
2. Bes, G.G., Hagege, C.: Properties in 5p. Tech. rep., Universite de Clermont Ferrand (2001)
3. Blache, P.: Property grammars: A fully constraint-based theory. In: Christiansen, H., Skadhauge, P.R., Villadsen, J. (eds.) CSLP. Lecture Notes in Computer Science, vol. 3438, pp. 1–16. Springer (2004)
4. Blache, P., Balfourier, J.M.: Property grammars: a flexible constraint-based approach to parsing. In: IWPT (2001)
5. Blache, P., Morawietz, F.: Some aspects of natural language processing and constraint programming. Tech. rep., Universitat Stuttgart, Universitat Tubingen, IBM Deutschland (2000)
6. Burkett, D., Klein, D.: Two languages are better than one (for syntactic parsing). In: EMNLP. pp. 877–886. ACL (2008)
7. Christiansen, H.: CHR grammars. TPLP 5(4-5), 467–501 (2005)
8. Dahl, V., Miralles, J.: Womb grammars: Constraint solving for grammar induction. In: Sneyers, J., Frühwirth, T. (eds.) Proceedings of the 9th Workshop on Constraint Handling Rules. vol. Technical Report CW 624, pp. 32–40. Department of Computer Science, K.U. Leuven (2012)
9. Dahl, V., Blache, P.: Directly executable constraint based grammars. Proc. Journees Francophones de Programmation en Logique avec Contraintes (2004)
10. Duchier, D., Dao, T.B.H., Parmentier, Y.: Model-Theory and Implementation of Property Grammars with Features. Journal of Logic and Computation p. 19 (2013), <http://hal.archives-ouvertes.fr/hal-00782398/en/>
11. Duchier, D., Dao, T.B.H., Parmentier, Y., Lesaint, W.: Property grammar parsing seen as a constraint optimization problem. In: de Groote, P., Nederhof, M.J. (eds.) FG. Lecture Notes in Computer Science, vol. 7395, pp. 82–96. Springer (2010)
12. Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. 37(1-3), 95–138 (1998)
13. Gazdar, G.: Phrase structure grammars and natural languages. In: IJCAI. pp. 556–565 (1983)

14. Muresan, S.: A learnable constraint-based grammar formalism. In: Huang, C.R., Jurafsky, D. (eds.) COLING (Posters). pp. 885–893. Chinese Information Processing Society of China (2010)
15. Nicolas, L., Molinero, M.A., Sagot, B., Trigo, E.S., de La Clergerie, E., , Farré, J., Vergés, J.M.: Towards efficient production of linguistic resources: the Victoria Project (2009)
16. van Rullen, T.: Vers une analyse syntaxique à granularité variable. Ph.D. thesis, Université de Provence (2005)
17. Shieber, S.M.: Constraint-based grammar formalisms - parsing and type inference for natural and computer languages. MIT Press (1992)
18. Waltz, D.: Understanding line drawings of scenes with shadows. In: The Psychology of Computer Vision. p. pages. McGraw-Hill (1975)
19. Woods, W.A.: Transition network grammars for natural language analysis. Commun. ACM 13(10), 591–606 (Oct 1970), <http://doi.acm.org/10.1145/355598.362773>

CHR Meets MapReduce

Amr Osman¹, Amira Zaki², and Slim Abdennadher¹

¹ Faculty of Computer Science and Engineering, German University in Cairo, Egypt
`{amr.salaheldin, slim.abdennadher}@guc.edu.eg`

² Faculty of Engineering and Computer Sciences, Ulm University, Germany
`amira.zaki@uni-ulm.de`

Abstract. The robustness, power and abstract syntax of declarative constraint programming has spawned many new research directions [1]. As the industry moves towards more robust, parallel, complex distributed systems and cloud computing for on-demand computation and dynamic scaling of applications, the need to have more intuitive programming languages that provide high-level abstraction from the underlying hardware grows. MapReduce was designed to be one of such programming paradigms. However, it still suffers some drawbacks and it requires developers to have certain skills to efficiently model their problems in the MapReduce paradigm. We argue that CHR could be the next game-changing step in the future cloud computing and we propose a novel work-in-progress for parallelizing execution of CHR programs in MapReduce data-intensive clusters that process BigData.

Keywords: Constraint Handling Rules, MapReduce, Cloud Computing, BigData, Parallel, Concurrent, Stream Computing

1 Introduction

Constraint Handling Rules (CHR) is a declarative logical programming language that relies on a set of constraints to model a specific problem in the form of a program. While different operational semantics and numerous implementations of CHR exist, only a few approaches in the literature attempt to provide parallel and concurrent execution for CHR programs and none of the implementations target multi-node clusters that consume BigData as in cloud computing environments. The power of logic programming in general combined with the ease of declarative constraints could be the next key-enabler to reduce programming complexity in data-intensive clouds and facilitate computations on highly dimensional BigData as in the areas of data mining and Cloud Analytics.

Our work comprises exploiting concurrent execution of CHR programs by firing different constraints in parallel and effectively map the execution pipeline of a CHR program from a sequential way into a MapReduce distributed fashion where different threads per worker per cluster collaborate to run the constraint rules with respect to the input query in order to yield the final result. In a future full of constraint-rich programs and many queries per program streaming into the

service-oriented and data-intensive cloud, traditional sequential processing can no longer scale and maintain a high performance throughput. Another stumbling challenge is avoiding having the constraint store as a single point of failure and storing the constraints in a distributed manner without affecting the consistency of the knowledge base which is updated according to the derivation rules or the behavior of the program.

The rest of the paper is structured as follows: In the next section, we state some preliminary overview about how CHR works and the concept of MapReduce. In section 3, we highlight and review some state-of-the-art approaches in the literature that attempt to provide concurrent CHR implementations on different hardware architectures and platform environments. In the fourth Section, we introduce our proposal to execute CHR concurrently on multi-node clusters and discuss how we suggest to solve various problems that arise from that. Finally, we summarize our approach, discuss its current limitations, lessons learned and our future work suggestions.

2 Background

Before diving into relevant approaches, concurrent CHR and multi-node cluster platforms, we first need to define CHR and provide an overview on the structure of a CHR program, its syntax and how it executes according to operational semantics. Moreover, it is essential to limit our scope to which specific areas of cloud computing we target and how MapReduce generally operates.

2.1 CHR

CHR [2] is a language aimed at developing constraint solvers declaratively by specifying a set of rules which are applied exhaustively to match some input goal constraints and accordingly, remove or add constraints to a multi-set called the constraint store. Unification is used to match any two given constraints with the same arity together. Rule types follow one of the following forms:

```
% rule is optionally tagged by a name before the '@' symbol.

simplification @ HeadReplaced <=> Guard | Body
propagation    @ HeadPersisted ==> Guard | Body
simpagation    @ HeadPersisted \ HeadReplaced <=> Guard | Body
```

The above is also subject to the following specifications:

- Rule heads and goals must consist of only CHR constraints. These, are just FOL predicate symbols that might or might not be unifiable.
- Guards consist of only built-in constraints which are basically a restricted set of functions.
- The Body may be a mix of both: built-in and CHR constraints.

In Simplification rules, the head constraint is replaced by the body if the guard check passes. This is contrary to propagation rules, where head constraints are kept in addition to the body which is further added to the constraint store. Simpagation rules replace constraints after the ‘\’ operator with the body while keeping constraints before it.

It follows that any CHR rule can be generalized in the simpagation rule notation. If ‘HeadPersisted’ is the empty set, then it is equivalent to the simplification rule. While if ‘HeadReplaced’ is empty set then, it is equivalent to the propagation rule. Consequently, any argument proven on a simpagation rule is inherently valid for simplification and simpagation rules.

Various operational semantics have been formalized to govern the constraint transition states and limit the constraint store mutation with respect to the applicability of different rules and unifying goal constraints with other constraints. In brief, abstract semantics addresses the issue of non-termination by preventing running into propagation rule cycles for the same constraints. Additionally, it separates built-in constraints from CHR constraints from goal constraints. The refined semantics were later introduced to deterministically apply rules from top to bottom and unify constraints one by one from left to right. This permits programmers to have more control over the program behavior and re-arrange rules and constraints for efficiency at the expense of limiting execution to be strictly procedural.

2.2 MapReduce

Introduced by Google, MapReduce [3] is a full-stack framework and programming paradigm to abstract distributed processing on large clusters. It relies on a master node to schedule jobs and assign tasks to worker nodes where the execution happens in two phases: A map phase and a reduce phase. During the map phase, nodes that are assigned to map tasks consume chunks of data and emit intermediate results in the form of (Key, Value) pairs. These intermediate results are typically shuffled and then passed to reducer nodes which aggregate the intermediate data grouped by key into a smaller set of results. The framework ensures maximum throughput, fault-tolerance and data locality through different optimization techniques. Since we are only interested in the programming paradigm and the abstraction, we will mathematically formalize MapReduce execution.

MapReduce programming is achieved by introducing two key functions for processing the data: A *map* function and a *reduce* function which execute back to back in the pipeline. We mathematically formalize these two functions as follows: the map function m is represented as $\langle k, v \rangle \xrightarrow{m} \bigcup_{i=1}^n \{ \langle k'_i, v'_i \rangle \}$ whereas the reduce function r which further processes the map function’s output is represented as $\langle k'_i, \vec{V}'' \rangle \xrightarrow{r} \vec{V}'''$ constrained by:

$$\forall \langle k'_i, v'_i \rangle \in m(\langle k, v \rangle) \exists \langle k'_j, v'_j \rangle [\{v'_i, v'_j\} \setminus V'' = \phi] \quad (1)$$

$$\forall \langle k'_i, v'_i \rangle, \langle k'_k, v'_k \rangle \in m(\langle k, v \rangle) k'_i \neq k'_k \Rightarrow v'_i \notin V'' \quad (2)$$

$$|V'''| \leq |V''| \quad (3)$$

where k, v indicate an arbitrary input key along with its corresponding value from the input data respectively and V'', V''' are both lists of values in the form: $\{v_1, v_2, v_3, \dots, v_n\}$. The reduce function is preceded by a grouping function which groups the intermediate output values from $m(\langle k, v \rangle)$ by key. This can be shown in constraints (1) and (2). The reduce function then performs some arbitrary computation and reduces the input set of values V'' into a smaller set V''' as shown by constraint (3).

In terms of CHR, MapReduce can be simulated as follows:

```

mapdone @ map(_,[]) <=> true.
map      @ map(K, [D|DS]) <=> pair(K,D), map(K,DS).

merge    @ pair(K,D1), pair(K,D2) <=> list(K,[D1,D2]).
merge2   @ pair(K,D1), list(K,D2) <=> list(K,[D1|D2]).

% Assume <reduce> sums elements.
reduce   @ list(K,[L|L1]) <=> L2 is L+L1, list(K,L2).
```

Running the above code with the goal $map(1,[2,3,6,10]), map(2,[4,5,7,11])$ under the refined semantics yields the answer $list(2,27), list(1,21)$. The execution flow towards this final result is visualized in Fig. 1. Note that the key value might change in between by the map function itself and that the intermediate values are typically interleaved and not ordered by any means. For simplicity, we let the goal constraint input the initial key values. The constraint **map/2** indicates that a map task with a Key K is to be executed on an input data set $[D|DS]$. The map rule then divides the data in the form of (key,value) pairs as in the constraint **pair/2**. In an intermediate step, the pairs are aggregated and grouped by key forming a list per key as in **list/2**. Finally the intermediate lists are then reduced into a smaller list as in the ‘reduce’ rule which sums the elements of the second argument.

Fig. 1. MapReduce workflow

TwitterStorm [4] is a real-time, MapReduce-inspired, and java-based distributed stream computing framework. Unlike other frameworks which are focused on batch processing and suffer a pipeline stall between the map phase and the reduce phase, it is focused on consuming and propagating data in the form of immutable tuples which are utilized upon arrival. TwitterStorm adds another abstraction layer on top of MapReduce by following a DAG workflow definition. It is based on two simple graph components that are scaled and executed in parallel by all workers: A Spout and a Bolt. Spouts are simply data sources that emit tuples in real-time while Bolts are data sinks that consume input tuples from various streams and emit some new tuples into their corresponding intermediate output streams. This process is demonstrated in Fig. 2.

Fig. 2. TwitterStorm DAG

3 Related Work

Seeking answers to our main challenges, we hereby review the literature in hope to collect and dissect solutions to the following:

1. What are the possible means to have a shared constraint store in a distributed environment?
2. How to fully exploit concurrency and parallelism when executing CHR programs in real-time.
3. How to resolve consistency issues and maintain correctness given that rules will fire out of order and goal constraints might get picked at random.
4. Which approach is ideal and analogous to the MapReduce design pattern?

Previous work [5] introduced a variant of concurrent abstract semantics that allows for multiple parallel removals of constraints from the constraint store. Such issue is a limitation of the abstract semantics under the rules of concurrency and a shared constraint store. Their approach specifically targets a class of constraint objects that are multiplicity independent. In other words duplicating a specific constraint, has no effect on the correctness of the final result. By duplicating these objects, parallel rules can then safely remove the same constraint by removing one of replicated constraints each.

Another paper [6] proposed a variation of the refined semantics that exploits multi-core concurrency using Software Transactional Memory (STM) on top of Haskell. The implementation fully relies on transactions and atomic locks to resolve conflicts of re-writing multiple constraints where a failed transaction can be restarted at any time. It was concluded that even with the somewhat naive implementation, a great speedup was achieved. In a more recent paper [7], the same approach was used and perfected with more optimizations for goal-based parallelism [8] which is argued to scale better than rule-based parallelism for such scenario. The paper makes compelling justifications for different optimization techniques including unique constraint lookup plans to minimize thread conflicts, constraint indexing, breadth-first and depth-first goal execution and lazy goal storage. Thanks to such optimizations, the paper achieves even better speedups for different CHR applications.

General Purpose Graphics Processing Units (GPGPUs) are ideal platforms for massively parallel multi-core stream computing. An ongoing preliminary work [9] demonstrated a method to translate CHR rules and constraints into C++ structures which are then used and computed in parallel during kernel calls on a GPGPU with the shared constraint store inside the device memory. We could later adapt a similar method to port a simple-enough CHR implementation into a MapReduce cluster. Other optimization techniques as aforementioned in previous research could also be incorporated such as constraint indexing and caching utilizing the device shared memory per Symmetric Multi-Processor (SM) on CUDA architectures to make best use of data locality.

AngelicCHR [10] sheds light on a revolutionary approach to bring parallel execution using the abstract semantics. Instead of being committed-choice when

applying CHR rules, it suggests exploring all possible choices and directions according to a proposed formalization of angelic semantics. Moreover, it follows a goal-based approach and visualizes all derivation paths in the form of a derivation network. However, no actual implementation details were provided. It is trivial that under such modification, constraint stores isolation per path are necessary. i.e. no sharing. This could demand more storage. Nevertheless, derivation nets are seemingly the perfect match for MapReduce. The next Section will highlight some key reasons why.

4 Proposed Approach

For maximum concurrency, we base our work on the abstract operational semantics as formalized in [2]. The non-determinism in applying the rules as well as selecting the goal constraints allows for interleaving execution in contrast to the refined semantics which has limited monotonicity where strict rule scheduling and prioritization is enforced by firing rules top-down and matching constraints from left to right. However under a shared constraint store, triggering multiple rules might alter how a certain program behaves and falsify results due to removal of a constraint that could have been matched by some other rule. Similarly, deciding which goal constraint to execute first will have an impact on which of the rules will get fired and the propagation history per state.

Example 1. Consider the following simple CHR program and the goal constraint ‘a’:

```
a <=> b, c.
a ==> d.
```

Clearly if the first rule is executed first, then a will be removed from the constraint store and the result constraints ‘b’ & ‘c’ will be returned since there are no possible rules to match them. On the other hand if the second rule is executed first then, both a and d will be added to the constraint store. Since the propagation rule was marked in the propagation history of the current state, we will avoid executing it for the same constraint ‘a’ again. Now, the 1st rule will fire next replacing ‘a’ with ‘b’ & ‘c’ in the constraint store yielding ‘b’, ‘c’ and ‘d’ as a final result.

Example 2. Consider the CHR program below when prompted with the goal constraints: ‘a, b’ and assuming rules will fire top-down:

```
a <=> true.
a, b <=> x.
```

If we match ‘a’ first, then it will be removed from the constraint store and ‘b’ will be returned as an answer. However if we matched ‘b’ first then, it will match nothing. Consequently we will try to match ‘a’ whilst having ‘b’ in the store which will trigger the second rule adding ‘x’ to the constraint store and removing both: ‘a’ & ‘b’ and therefore, emitting the final result ‘x’.

Note 1. In both examples under the refined operational semantics, the second rule will never fire due to the strict order of execution. It will always be the case that ‘a’ will be removed from the constraint store and replaced by the first rule’s body.

Our main approach is thusly centered around utilizing TwitterStorm and leveraging its real-time distributed stream processing capabilities in order to apply rules on different goal constraints in parallel. This should also be done transparently without imposing any restrictions on the given CHR programs. In Section 3, we provided a high-level review of AngelicCHR. It follows that it lacks implementation and technical aspects of parallel strategies despite the concrete theoretical formalism of Angelic semantics. Precisely, this gap is where our effort is articulated. The concept of derivation nets suits the MapReduce execution strategy in general and TwitterStorm in specific due to the fact that the different derivation paths are shown and that the dependencies between different rule constraints are highlighted in the arcs of the network. Such dependencies easily model the separation between map tasks and reduce tasks respectively. Along the same line of thought, the lack of dependencies is a true indicator of the degree of parallelism of each task.

However, TwitterStorm lacks CHR implementation. One possible way is to follow the footsteps of [9] by simplifying rules in terms of data structures and Java code. Another way is to make use of JCHR [11] in a distributed fashion and ensure that different versions consisting of different constraint permutations are given to each worker to fully explore all paths and then, aggregate the results back during the reduce phase.

Before we start discussing our technical approach, it is worth mentioning that there are many ways to support a shared constraint store in a distributed cloud environment such as using distributed NoSQL, No Single Point of failure in-memory databases which ensure that computation is tightly coupled with the data storage to make best use of locality and avoid the overhead of sending data cross-nodes over the network. Also by relying on in-memory storage, we eliminate another layer of latency during accessing the hard disks. However as we argued before, AngelicCHR relies on isolated constraint stores to fully explore all computation paths. Thus, it is not necessary to have a shared constraint store.

To describe a TwitterStorm topology, we need to define two necessary implementations: A Bolt and a Spout. There could be one or more Bolt implementations depending on the different roles of each one and its position inside the DAG. In CHR, how these two components work can be simulated using the following rules:


```

% A Spout keeps generating new tuples from input data
Spout @ data(X,Y,Z,L,M,N) <=> tuple(X,Z,M),tuple(Y,L,N).

% A Bolt performs arbitrary computation:
%   consumes and emits new tuples
% PS: tuples are immutable.
Bolt  @ tuple(X,Y,Z) ==> X1 is X+Y, newtuple(X1,Y,Z).
Bolt2 @ tuple(X,Y,Z) <=> Y1 is Y+Z, newtuple2(X,Y1,Z).
Bolt3 @ newtuple(X,Y,Z), newtuple2(L,M,N) <=> R is X+Y+Z+L+M+N,
                                         newtuple3(R).

```

The above minimalistic code when executed under the refined semantics simulates the TwitterStorm topology in Fig. 3. (Except for the fact that each Bolt and Spout will be executed by thousands of threads randomly distributed on working nodes in a cluster). We use the constraints `newtuple/3`, `newtuple2/3` and `newtuple3/1` to indicate that such tuples belong to different streams directed at or coming out of different Bolts such that each Bolt can be attached to one ore more streams. We can therefore confirm the following points:

- Tuples are immutable. i.e. their values cannot change but, can be only read.
- Dimensions of tuples throughout the lifecycle of the topology can vary along the stream from one component (Bolt) to another.
- Spouts generate tuple structures from input data while Bolts consume them.
- Different Bolts can consume the same tuples. Or tuple streams can be replicated and distributed according to the definition of the topology. This is why we have rule 2 as a propagation instead of simplification for example.
- Bolts can also aggregate data from different streams. This can be deduced from the last rule.

Fig. 3. A general example of a TwitterStorm topology

As discussed earlier, we decided to use JCHR on all nodes which eliminates the need to implement abstract semantics and CHR rules from scratch at first. However, we discovered that it would be much easier to call the Prolog interpreter via a system command from Java. On Linux, we can also pragmatically direct the input query via bash I/O redirection. This leaves us with the task of rules

re-ordering and parallel execution. We only decided to head this route towards a proof of concept implementation and leave the second route of implementing optimized abstract semantics from scratch for our future work. Table 1 provides a TwitterStorm glossary for some of the built-in functions and terms that will be used.

Table 1. TwitterStorm glossary

Function()/Object	Description
<i>nextTuple()</i>	Automatically called once the Spout needs to emit a new tuple
<i>_collector</i>	An output collector object which emits new tuples onto the out stream
<i>emit(..)</i>	Emits strictly one new tuple onto a specific output stream
<i>Values</i>	An ordered collections of values for a given tuple
<i>Fields</i>	An ordered collection of field names for the corresponding values of a tuple
<i>getIntegerByField()</i>	Gets a specific integer value corresponding to a field name inside a tuple
<i>getStringByField()</i>	Gets a specific String value corresponding to a field name inside a tuple
<i>declareOutputFields()</i>	Called automatically to announce the field names of the values of all output tuples
<i>execute(..)</i>	Called automatically when a Bolt receives an input tuple

Our approach can be summarized using the following algorithm for the Spout:

```
//Assuming we have read the input program and stored the rules.
final String[] rules;
//Input Queries from the user
final String[] queries;
final Random rand = new Random();
int randIndex;
String query;

public void nextTuple() {
    randIndex = rand.nextInt(rules.length);
    query = queries[rand.nextInt(queries.length)];
    final String rule = rules[randIndex];
    _collector.emit(new Values(rules.length, randIndex, rule, query));
}

public void declareOutputFields(OutputFieldsDecelarerer d) {
    d.declare(new Fields("expect", "ruleId", "rule", "query")); }
```

Each Bolt is explicitly told how many rules to expect, a rule identifier for each rule streamed, a random input query and the rule string itself. This is all packaged into the tuple being sent which contains a random rule each time. Our Bolt abstract implementation is as follows:

```

ArrayList<Integer> receivedRules = new ArrayList<Integer> ();
ArrayList<String> rules = new ArrayList<String>();

public void execute(Tuple tuple) {
    int expectedCount = tuple.getIntegerByField("expect");
    int ruleId = tuple.getIntegerByField("ruleId");
    String rule = tuple.getStringByField("rule");
    String query = tuple.getStringByField("query");
    if(rules.size() == expectedCount) {
        String fileName = writePlFile(receivedRules);
        String result = execCHR(query, fileName);
        _collector.emit(new Values(result));
    }
    else if(!receivedRules.contains(ruleId)) {
        rules.add(rule);
        receivedRules.add(ruleId);
    }
}

public void declareOutputFields(OutputFieldsDecelarer d) {
    d.declare(new Fields("result"));
}

```

The correctness and exploring all the derivation paths obviously depends on the degree of parallelism specified for the Bolt. Or, how many threads per worker will execute. These specifications are easily tuned during building the DAG topology and submitting it for execution on the cluster using some provided built-in methods. We could later introduce another Spout to filter out the duplicate results in case the random ordering of the rules was the same at some instance of time or, in case of a confluent program. But, it is not necessary.

In brief, the Spout simply streams a random rule from the input program and a random query from the input queries each time. Every Bolt worker collects and stores rules by the order of arrival and when it reaches the expected number of unique rules as in the original program, it starts executing the random query on the current program version and yields out the result. In the above code, the `writePlFile(..)` function writes the received rules in their order to a valid CHR module in the form of a prolog file. The `execCHR(..)` function will call the prolog interpreter on the written pl file and query it with the current received query using the methodology we highlighted earlier.

Despite that our approach allows multiple streaming queries on a given program in parallel and implements a subset of angelic parallel semantics (More specifically, the rule-based angelism property without deep guards or head decomposability), we still have not fully exploited goal-based parallelism and stream-

ing of constraints in a shared store environment. Similarly, allowing multiple programs with multiple queries to execute in parallel. We leave this and other future optimizations for our coming development.

5 Conclusion and Future work

CHR is an inherently-parallel declarative constraint logic programming language. Relative to other languages, its power in terms of lines of code is significantly higher and thus, it is more abstract and high level. As the cloud seeks more abstraction from the hardware and virtualization all the way to the service-level, declarative constraint programming is potentially the key. Different cloud applications could be abstracted using CHR such as service orchestration [12], multi-policy management, business modeling [13][14]. Many approaches have also been proposed for massively BigData-rich parallel computations as in data mining [15], cloud analytics and scheduling [16]. Other areas such as airplanes runway-routing inside airports, railway scheduling and more [17] remain active fields of interest.

We introduced a very early and draft work-in-progress to bring CHR into the data-intensive cloud and parallelize/scale it inside real-time cluster environments. We further modeled the MapReduce programming paradigm and the TwitterStorm [4] workflow using CHR after our mathematical formalization. Our early implementation follows the same footsteps as AngelicCHR [10], supports streaming multi-queries per program and is headed towards a parallel, shared and distributed constraint store implementation inside the cloud relying on abstract semantics. Exploiting the class of CHR programs that enjoy the online property is additionally on our schedule. For future work, we also plan to use optimization techniques discussed in [8] with the aid of distributed in-memory databases for our implementation and add multi-program support. CHR as a service is indeed the next big step in the era of the service-oriented cloud.

References

1. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint handling rules. *TPLP* **10**(1) (2010) 1–47
2. Frühwirth, T.: Constraint handling rules. Volume 3. Cambridge University Press Cambridge (2009)
3. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. *Communications of the ACM* **51**(1) (2008) 107–113
4. Nathan, M., James, X., Jason, J.: Storm: Distributed real-time computation system, [online]. available: <http://storm-project.net/>. (2012)
5. Raiser, F., Frühwirth, T.: Exhaustive parallel rewriting with multiple removals. In: *Proc. of 24th Workshop on (Constraint) Logic Programming*. (2010)
6. Lam, E.S., Sulzmann, M.: A concurrent Constraint Handling Rules semantics and its implementation with software transactional memory. In: *DAMP '07: Proc. ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. (January 2007)

7. Sulzmann, M., Lam, E.S.: Parallel execution of multi-set constraint rewrite rules. In: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, ACM (2008) 20–31
8. Lam, E.S., Sulzmann, M.: Concurrent goal-based execution of constraint handling rules. *Theory and Practice of Logic Programming* **11**(6) (2011) 841–879
9. Zaki, A., Frühwirth, T., Geller, I.: Parallel execution of constraint handling rules on a graphical processing unit. In: Proceedings of the 9th International Workshop on Constraint Handling Rules. (2012) 82
10. Martinez, T.: Angelic chr. In: Eighth International Workshop. (2011) 19
11. Van Weert, P., Schrijvers, T., Demoen, B.: Ku leuven jchr: a user-friendly, flexible and efficient chr system for java. In: Proceedings of the 2nd Workshop on Constraint Handling Rules. (2005) 47–62
12. Salomie, I., Chifu, V., Harsa, I., Gherga, M.: Web service composition using fluent calculus. *International Journal of Metadata, Semantics and Ontologies* **5**(3) (2010) 238–250
13. Dausend, M., Raiser, F.: Model transformation using constraint handling rules as a basis for model interpretation. In: Proceedings of the Eighth International Workshop on Constraint Handling Rules. (2011) 64–78
14. Raiser, F., Frühwirth, T.: Analysing graph transformation systems through constraint handling rules. *Theory Pract. Log. Program.* **11**(1) (January 2011) 65–109
15. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2008) 204–212
16. Abdennadher, S., Marte, M.: University timetabling using constraint handling rules. In: Proc. JFPLC. Volume 98. (1998) 39–49
17. Langbein, J., Stelzer, R., Frühwirth, T.: A rule-based approach to long-term routing for autonomous sailboats. In: Robotic Sailing 2011, Part V. (2011) 195–204

CHReSM and Probabilistic Argumentation Logic

Jon Sneyers¹, Danny De Schreye¹, and Thom Frühwirth²

¹ Dept. of Computer Science, KU Leuven, Belgium
jon.sneyers@cs.kuleuven.be, danny.deschreye@cs.kuleuven.be

² University of Ulm, Germany
thom.fruehwirth@uni-ulm.de

Abstract. Riveret et al. proposed a framework for probabilistic legal reasoning. Their goal is to determine the chance of winning a court case, given the chances of the judge accepting certain claims and legal rules. We tackle the same problem by defining and implementing a new formalism, called probabilistic argumentation logic (PAL). We implement PAL in CHReSM, and discuss how it can be seen as a probabilistic generalization of Nute’s defeasible logic. Not only does this provide an automation of the — only hand-performed — computations in Riveret et al, it also provides a solution to one of their open problems: a method to determine the initial probabilities from a given body of precedents.

1 Introduction

Riveret et al. [1] proposed a framework of probabilistic legal reasoning based on the argumentation framework of Prakken et al. [2], which provides a dialectical proof theory in the formal setting of Dung [3]. Their goal is to determine the chance of winning a court case, given the probabilities of the judge accepting certain claimed facts to be valid and legal rules to be applicable.

Roth et al. [4] tackle a similar problem, but with the focus on finding legal strategies for the involved parties to maximize their chances of winning. They propose a rather complex framework, based on a logic layer, an argument layer, a dialectical layer, a procedural layer, and finally a probabilistic weighting.

To our knowledge, none of these approaches have been implemented so far. Both papers only contain a hand-performed computation to illustrate their approach on an example. Such an implementation of an ‘argument assistance system’ is left explicitly as future work in [1].

Another issue is how to determine or verify the probabilities, which are assumed to be known in advance. Riveret et al. [1] suggest that maybe somehow a statistical analysis of how the judge has decided in the past could be performed, but they again leave this issue to future work.

In this paper we tackle the problem by defining a new formalism, called probabilistic argumentation logic, which can be seen as a probabilistic generalization of defeasible logic with explicit underlying assumptions.

We formalize our approach and implement it in CHReSM [5]. CHReSM is a rule-based probabilistic logic programming language based on CHR [6] in the

host language PRISM [7]. For reasons of space we will omit an introduction to these formalisms; we assume the reader is already familiar with them. We refer the reader to [6, 8, 9] for CHR and to [5] for CHRiSM. In a companion paper [10], an introduction to CHRiSM can be found, as well as a discussion of the advantages and disadvantages of CHRiSM in this setting.

Our implementation provides an automation of the probability computations that were hand-performed in [1]. The built-in learning algorithm of CHRiSM also provides a solution to the open problem of determining the probabilities.

The remainder of this paper is organized as follows. In Section 2 we introduce the approach of [1] and the running example in their paper. We show how this example can be encoded in CHRiSM. Section 3 introduces our new formalism, probabilistic argumentation logic. We show that it generalizes Nute’s defeasible logic [11] and we give an implementation in CHRiSM. In Section 4 we briefly discuss learning. Finally we conclude in Section 5.

2 The mad cow example and dialogue games

The running example in [1] is the following. John, the proponent, wants to sue Henry, the opponent, claiming compensation for the damage that Henry’s cow caused to him when he drove off the road to avoid the cow. John argues that an animal’s owner has to pay damages caused by their animal, that Henry is the owner of the cow, and that the accident was caused by the need to avoid the cow (argument A). Henry can counterattack in various ways: he can claim that the damage was due to John’s negligence (he did not pay sufficient attention to crossing animals) – argument B – or that it was a case of force majeure: the cow suddenly went crazy and crossed into the street – argument C. The last objection could be replied to by using the debated rule that only exogenous events count as force majeure, and the cow’s madness is endogenous (argument D).

[1] assumes an abstract argumentation framework [3], which consists of a set of arguments and a binary “defeats” relation. They then define the notion of a dialogue game which captures the rules of legal argumentation. A dialogue is a sequence of abstract arguments, in which the proponent and the opponent alternate, each argument defeats the previous argument, and the proponent cannot repeat arguments and his arguments have to strictly defeat the previous argument of the opponent.

Moreover, each of the arguments has some given “construction chance”, which is the probability that the judge will actually accept the argument. The aim is to estimate the overall chance that the case is won.

2.1 Encoding of dialogue games in CHRiSM

In general, [1] assume an abstract argumentation framework [3], which consists of a set of arguments and a binary “defeats” relation. Argument X strictly defeats argument Y if X defeats Y and Y does not defeat X. For example, argument A and B defeat one another, while argument D strictly defeats argument C.

Moreover, each of the arguments has some given “construction chance”, which is the probability that the judge will actually accept the argument. The aim is to estimate the overall chance that the case is won.

They consider dialogue games which capture the rules of legal argumentation. A dialogue is a sequence of arguments, in which the proponent and the opponent alternate, with the following conditions:

- The proponent cannot repeat arguments;
- Every opponent argument defeats the previous (proponent) argument;
- Every proponent argument *strictly* defeats the previous argument;
- If a player cannot make a move, he loses.

The first argument is the main claim of the proponent. If the proponent wins all possible dialogues, he wins the case. Otherwise the opponent wins. Obviously, the outcome depends on which arguments are actually accepted by the judge.

In the mad cow example there are two possible dialogues if the judge accepts all arguments: [A,B] (in which the proponent loses) and [A,C,D] (in which the proponent wins). So if the judge accepts all arguments, the proponent loses (since he has no winning strategy in case the opponent uses argument B).

We can directly encode the rules of the dialogue game in CHReSM, as follows. We use a dummy constraint predicate `begin/0` to initialize a dialogue:

```
begin <=> init_defeats, dialogue([]).
```

The auxiliary predicate `init_defeats/0` initializes the “defeats” relation:

```
init_defeats <=> defeats(argA,argB), defeats(argB,argA),
                 defeats(argA,argC), defeats(argC,argA),
                 defeats(argD,argC).
```

The “strictly defeats” relation is derived from the “defeats” relation:

```
defeats(A,B) ==> strictly_defeats(A,B).
defeats(B,A) \ strictly_defeats(A,B) <=> true.
```

The main constraint predicate is `dialogue/1`, which contains a (reversed) list representing a (partial) dialogue. For example, a dialogue [A,C,D] would be encoded as `dialogue([p-argD,o-argC,p-argA])`. Note the reversed order (for easy access to the last element) and the “p-” and “o-” tags to denote the player.

First of all, we make sure that the construction chances are taken into account. If the construction chance of some argument is p , then we simply prune away a partial dialogue that ends with that argument with probability $1 - p$. For example, suppose the construction chance of argument A is 0.9, then we remove a dialogue ending with A with probability 0.1. These are the values used in [1]:

```
0.1 ?? dialogue([_argA|_]) <=> true.
0.6 ?? dialogue([_argB|_]) <=> true.
0.9 ?? dialogue([_argC|_]) <=> true.
0.85 ?? dialogue([_argD|_]) <=> true.
```

Now we add a rule for the proponent to make his main claim, argument A:

```
dialogue([]) ==> dialogue([p-argA]).
```

The rest of the dialogue is constructed according to the above conditions:

```
dialogue([p-A|D]), defeats(B,A) ==> dialogue([o-B,p-A|D]).
dialogue([o-A|D]), strictly_defeats(B,A) ==>
    \+ member(p-B,D) | dialogue([p-B,o-A|D]).
```

Now comes a tricky part: if a partial dialogue has been extended (but only after it has been extended in all possible ways), we can discard it. If it could not be extended, we have a final dialogue, which was won by the last player. One way to implement this is as follows: (exploiting the refined operational semantics and the `passive` pragma)

```
% dialogue was extended: remove prefix
dialogue([_|D])#passive \ dialogue(D) <=> true.

% dialogue was not extended: last player wins
dialogue([X-_|_]) ==> winner(X).
```

Finally, as soon as there is one dialogue which is won by the opponent, then the proponent cannot be a winner.

```
winner(o) \ winner(p) <=> true.
```

In [1], several pages are used to calculate the probability that the proponent wins the case. In CHRiSM, we can simply use the following query:

```
?- prob begin ==> winner(p).
```

```
Probability of begin ==> winner(p) is: 0.494100000000000
```

Where did the dialog go? In the encoding of the arguments in Section 2, we abstract away the turn-based dialog. We impose no total order on the arguments, nor strict alternation between the players; in fact, we do not even record which party (proponent or opponent) makes which claim. The essential structure of the reasoning is left intact though.

The rules were written as if both the proponent and the opponent make all of their claims and arguments simultaneously at the start. In order to model more accurately that some arguments will only be put forward as a “reaction” to the acceptance of other statements, we can also write the rules in a different way, for example, instead of always claiming that the cow was mad:

```
0.2 ?? begin ==> accept(e,[]).
```

the opponent Henry only makes that claim if the judge is tempted to make him compensate John’s damages:

```
0.2 ?? accept(c,_) ==> accept(e,[]).
```

Similarly, instead of always insisting that cow madness is endogenous:

```
0.3 ?? begin ==> accept(f,[]).
```

the proponent John only makes that claim if the judge has actually accepted that the cow was mad:

```
0.3 ?? accept(e,_) ==> accept(f,[]).
```

With respect to the outcome of the case (whether or not statement *c* gets accepted), there is no difference between both ways of encoding. The only difference is that the “put everything on the table at once” approach can introduce irrelevant claims, while the “reactive” approach more closely models the dialog, since claims are only triggered when they are needed.

2.2 CHRiSM encoding of the mad cow example

In the above, we described a CHRiSM encoding of dialogue games with abstract arguments, in which the arguments are treated as black boxes that only interact through the “defeats” relation. We now proceed with a finer level of granularity.

The arguments in [1] consist of premises and rules, which each have a probability of being accepted by the judge. For example, argument *A* consists of the premises that Henry owns the cow (*a*), and that the accident was caused by the need to avoid the cow (*b*), together with the rule “ $a \wedge b \rightarrow c$ ”, where *c* stands for “Henry has to compensate damages”. If *a* will certainly be accepted, *b* is accepted with a probability of 0.9, and the rule “ $a \wedge b \rightarrow c$ ” is certainly accepted, then the overall construction chance of argument *A* is $1 \times 0.9 \times 1 = 0.9$.

We will call both premises and conclusions “statements” and use ground Prolog terms to denote them. The auxiliary predicate `neg/2` simply negates a literal in a way that avoids double negations.

We use a dummy predicate `begin/0` (to be used as the initial goal). The main constraint predicate is `accept/2`, which indicates that the judge conditionally accepts some statement: `accept(S,C)` denotes that statement *S* is accepted if all conditions *C* (a Prolog list of statements) hold. If *C* is the empty list, the statement is unconditionally accepted; otherwise the acceptance of the statement can still be retracted if one of the conditions turn out to be unacceptable. If a statement is already accepted with conditions *A*, then it is redundant to also accept it with stronger conditions $B \supseteq A$.

```
accept(X,A) \ accept(X,B) <=> subset(A,B) | true.
```

The above rule implies a set semantics for unconditionally accepted statements.

A condition is redundant if it is implied by the other conditions:

```
accept(A,B) \ accept(X,C) <=> select(A,C,D), subset(B,D) | accept(X,D).
```

If a statement Y was accepted with conditions B , but one of those conditions is contradicted (“undercut”) by a statement X with weaker conditions A that are implied by B , then the acceptance of Y has to be retracted — we use a simpagation rule to remove the `accept(Y,B)` constraint:

```
accept(X,A) \ accept(Y,B) <=>
    subset(A,B), neg(X,NX), member(NX,B) | true.
```

Finally, we allow no contradictions:

```
accept(X,A), accept(Y,B) ==> subset(A,B), neg(X,Y) | fail.
```

Now we encode the premises and the rules of the arguments:

```
% Argument A (rule r1, premises a and b):
% "If Henry is the owner of the cow (a) and the accident was caused by the
% need to avoid the cow (b), then Henry has to compensate damages (c)."
```

```
1.0 ?? accept(a,[]), accept(b,[]) ==> accept(c, [app(r1)]).
1.0 ?? begin ==> accept(a,[]).
0.9 ?? begin ==> accept(b,[]).
```

The rule being used is a defeasible rule, so its conclusion c will be accepted with the condition that the rule is actually applicable (`app(r1)`). This condition can be “undercut” by arguments B or C .

```
% Argument B (rule r2, premise d):
% "If John was negligent (d), then r1 is not applicable."
0.8 ?? accept(d,[]) ==> accept(not(app(r1)), []).
0.5 ?? begin ==> accept(d,[]).
```

For example, the judge could accept a , b and d and both rules, to reach the state “`accept(c,[app(r1)]), accept(not(app(r1)), [])`”. Now the undercutting rule removes the conditional acceptance of c , because its condition was contradicted.

```
% Argument C (rule r3, premise e):
% "If the cow was mad (e), it was a case of force majeure.
% so r1 is not applicable."
0.5 ?? accept(e,[]) ==> accept(not(app(r1)), [app(r3)]).
0.2 ?? begin ==> accept(e,[]).
```

Again, the above rule ($r3$) is defeasible, so its conclusion can only be accepted with the condition `app(r3)`, which can be undercut by the final argument:

```
% Argument D (rule r4, premise f):
% "If the cow's madness is endogenous (f), then the 'force majeure'
% rule r3 is not applicable."
0.5 ?? accept(f,[]) ==> accept(not(app(r3)), []).
0.3 ?? begin ==> accept(f,[]).
```

The only thing now left to do, is to resolve the conditions by making assumptions. For example, one possible result of the query `begin` is the following:

```

accept(a, []), accept(b, []), accept(e, []),
accept(c, [app(r1)]), accept(not(app(r1)), [app(r3)])

```

In this case, both `c` and `not(app(r1))` are conditionally accepted. Although `not(app(r1))` undercuts the conditions of `c`, the “undercut” rule is not applicable (yet) because the condition `[app(r3)]` is not weaker than the condition `[app(r1)]`. However, since there is no counter-evidence for `app(r3)`, we can assume this condition to hold, “promoting” `not(app(r1))` to an unconditionally accepted statement, which then causes the acceptance of `c` to be retracted.

To implement this idea, we add the following rules at the end of the program:

```

begin <=> assume.
assume, accept(X,C) ==> select(A,C,D), accept(A,[]) ; true.
assume <=> true.

```

The middle rule nondeterministically selects conditions and accepts them. This can either lead to a contradiction (e.g. if there is counter-evidence), which causes backtracking, or succeed.

This concludes our program. We can now compute the desired probability — which took several pages of manual calculations in [1] — with a simple query:

```

?- prob begin ==> accept(c, []).
Probability of begin ==> accept(c, []) is: 0.494100000000000

```

3 Generalization and Formalization

In order to generalize the running example, we will propose a transformation from an arbitrary probabilistic legal argumentation logic $\mathcal{A}$ — a notion that will be introduced in this section — to a CHReSM program $P_{CHReSM}(\mathcal{A})$.

3.1 Probabilistic Argumentation Logic

We use $lit(A)$ to denote the set of literals over a set of atomic formulas A , i.e. $lit(A) = A \cup \{-a \mid a \in A\}$. We will sometimes denote conjunctions over literals as sets since the order of the conjuncts is irrelevant.

Definition 1 (Probabilistic argumentation logic). *A probabilistic argumentation logic or PAL is a tuple (S, A, R, P) , where S is a set of statements and A is a set of assumptions (with $S \cap A = \emptyset$), R is a set of rules, and P is a function assigning probabilities to each rule, $P : R \mapsto [0, 1]$. The rules in R have the following form:*

$$s_1 \wedge \dots \wedge s_n \Rightarrow c_1 \wedge \dots \wedge c_m \text{ assuming } a_1 \wedge \dots \wedge a_k$$

where the left hand side (the antecedent) is a conjunction of literals ($s_i \in lit(S \cup A)$) which can be empty ($n \geq 0$), the right hand side (the consequent) is a non-empty ($m \geq 1$) conjunction of literals ($c_i \in lit(S \cup A)$), and the part after

“assuming” (the assumption) is a possibly empty ($k \geq 0$) conjunction of assumption literals ($a_i \in \text{lit}(A)$). If the left hand side is empty, the rule is also called a fact and the arrow can be omitted; if the part after “assuming” is empty, the rule is called unconditional and the keyword “assuming” can be omitted.

To illustrate the definition, we now write out the running mad cow example as a formal probabilistic argumentation logic

$$\mathcal{A}_{\text{mc}} := (\{a, b, c, d, e, f\}, \{\text{app}(\mathbf{r1}), \text{app}(\mathbf{r3})\}, R_{\text{mc}}, P_{\text{mc}})$$

where the rules $R_{\text{mc}} = \{r_1, r_2, r_3, r_4, s_a, s_b, s_d, s_e, s_f\}$ are the following:

$$\begin{aligned} r_1 &:= a \wedge b \Rightarrow c \text{ assuming } \text{app}(\mathbf{r1}) \\ r_2 &:= d \Rightarrow \neg \text{app}(\mathbf{r1}) \\ r_3 &:= e \Rightarrow \neg \text{app}(\mathbf{r1}) \text{ assuming } \text{app}(\mathbf{r3}) \\ r_4 &:= f \Rightarrow \neg \text{app}(\mathbf{r3}) \\ \forall x \in \{a, b, d, e, f\} : s_x &:= x \end{aligned}$$

and the probabilities P_{mc} are the following:

$$P_{\text{mc}} := \{(r_1, 1), (r_2, 0.8), (r_3, 0.5), (r_4, 0.5), (s_a, 1), (s_b, 0.9), (s_d, 0.5), (s_e, 0.2), (s_f, 0.3)\}$$

Now we define an interpretation of a PAL as a set of conditional statements.

Definition 2 (Conditional statement). *Given a PAL $\mathcal{A} = (S, A, R, P)$, a conditional statement is a pair (s, C) with $s \in \text{lit}(S \cup A)$ and $C \subseteq \text{lit}(A)$, such that C is not self-contradictory, that is, there is no $c \in C$ and $\neg c \in C$.*

Definition 3 (Interpretation). *Given a PAL $\mathcal{A}$, an interpretation is a set I of conditional statements of $\mathcal{A}$ such that if $(s, C_1) \in I$ and $(\neg s, C_2) \in I$, then $C_1 \not\subseteq C_2$ and $C_2 \not\subseteq C_1$.*

In other words, the conditional statements are not directly contradictory — although both a statement and its negation can be conditionally accepted at the same time, as long as the assumptions are different.

Definition 4 (Partial Ordering of Interpretations). *We say an interpretation I_1 is smaller than an interpretation I_2 , denoted $I_1 \leq_i I_2$, if for all conditional statements $(s, c_1) \in I_1$, there is a corresponding conditional statement $(s, c_2) \in I_2$ such that $c_1 \subseteq c_2$.*

If $I_1 \leq_i I_2$, then the set of statements in I_1 is a subset of the statements in I_2 , so I_1 makes less claims than I_2 (which is why we call it smaller), but the claims in I_1 are in a sense stronger since they have a weaker condition.

We now define the semantics of a PAL, somewhat inspired by the definitions in [11], but extending them to take the explicit conditions into account, as well as the rule selection (which will be needed to introduce the rule probabilities).

Definition 5 (Compliant Interpretation w.r.t. Rule Selection). *Given a PAL $\mathcal{A} = (S, A, R, P)$ and a set of selected rules $R_s \subseteq R$, a compliant interpretation I w.r.t. the selected rules R_s is a minimal (w.r.t. $\leq_i$) interpretation that satisfies the following additional criterion:*

- if R_s contains a rule $r = (S_r \Rightarrow C_r \text{ assuming } A_r)$,
- and $\forall s \in S_r : \exists c : (s, c) \in I$ (the antecedent is conditionally accepted),
- and $\forall a \in A_r : \neg \exists c : (\neg a, c) \in I$ (the assumption is not questioned),
- then for every set $\{(s_1, c_1), \dots, (s_n, c_n)\} \subseteq I$ such that $S_r = s_1 \wedge \dots \wedge s_n$, it must hold that $\forall x \in C_r : \exists y \subseteq A_r \cup \{c_1, \dots, c_n\} : (x, y) \in I$ (the consequent is conditionally accepted).

Definition 6 (Valid Interpretation w.r.t. Rule Selection). *Given a PAL $\mathcal{A} = (S, A, R, P)$ and a set of selected rules $R_s \subseteq R$, a valid interpretation I w.r.t. the selected rules R_s is a compliant interpretation without gratuitous statements, that is, there is no subset $K \subseteq I$ with $K \neq \emptyset$ such that:*

- if R_s contains a rule $r = (S_r \Rightarrow C_r \text{ assuming } A_r)$,
- and $\forall s \in S_r : \exists c : (s, c) \in I \setminus K$, and $\forall a \in A_r : \neg \exists c : (\neg a, c) \in I$,
- then for every set $\{(s_1, c_1), \dots, (s_n, c_n)\} \subseteq I \setminus K$ such that $S_r = s_1 \wedge \dots \wedge s_n$, it must hold that $\forall x \in C_r : \forall y \subseteq A_r \cup \{c_1, \dots, c_n\} : (x, y) \notin K$.

Returning to the mad cow example, consider the rule selection R_{mc} of all rules. The following is the only valid interpretation w.r.t. R_{mc} :

$$\{(a, \emptyset), (b, \emptyset), (d, \emptyset), (e, \emptyset), (f, \emptyset), (\neg \text{app}(r1), \emptyset), (\neg \text{app}(r3), \emptyset)\}$$

This is the only valid interpretation w.r.t. $R' = \{r_1, r_3, r_4, s_a, s_b, s_e, s_f\}$:

$$\{(a, \emptyset), (b, \emptyset), (c, \emptyset), (e, \emptyset), (f, \emptyset), (\neg \text{app}(r3), \emptyset)\}$$

An interpretation like the above, but with $(c, \{\text{app}(r1)\})$ instead of $(c, \emptyset)$ also satisfies the criterion of Def. 5, but it is not compliant because it is not minimal w.r.t. $\leq_i$. The condition for the acceptance of the consequent of applicable rules is allowed to be weaker than (i.e. a subset of) the union of all the conditions arising from the antecedent and assumption. This relaxation serves two goals. Firstly, it means that if a consequent can be derived in different ways such that it would be accepted multiple times with varying conditions, it suffices to have only the weakest conditions in the interpretation. Secondly, because the interpretation has to be minimal w.r.t. $\leq_i$, conditions will only be present in the interpretation to avoid contradiction. The following example illustrates this point.

```

r1 := bird ⇒ flies assuming normal-bird
r2 := tux ⇒ bird ∧ tuxedo-plumage assuming eyes-OK
r3 := tuxedo-plumage ⇒ penguin assuming feathers-make-bird
r4 := penguin ⇒ ¬flies
r5 := tux

```

Consider the selection of all rules. The following interpretation is compliant:

$$\{(\text{tux}, \emptyset), (\neg\text{eyes-OK}, \emptyset)\}$$

but it is not a valid interpretation since the statement $\neg\text{eyes-OK}$ is trivially gratuitous: there is not even a rule that could derive it.

The following interpretation satisfies the criterion of Def. 5:

$$\left\{ \begin{array}{l} (\text{tux}, \emptyset), (\text{bird}, \{\text{eyes-OK}\}), (\text{flies}, \{\text{eyes-OK}, \text{normal-bird}\}), \\ (\text{tuxedo-plumage}, \{\text{eyes-OK}\}), (\text{penguin}, \{\text{eyes-OK}, \text{feathers-make-bird}\}), \\ (\neg\text{flies}, \{\text{eyes-OK}, \text{feathers-make-bird}\}) \end{array} \right\}$$

but it is not minimal w.r.t. $\leq_i$; we can relax some conditions to get a minimal interpretation, for example: (in this case there are three minimal interpretations)

$$\left\{ \begin{array}{l} (\text{tux}, \emptyset), (\text{bird}, \emptyset), (\text{flies}, \{\text{normal-bird}\}), \\ (\text{tuxedo-plumage}, \emptyset), (\text{penguin}, \{\text{eyes-OK}\}), (\neg\text{flies}, \{\text{eyes-OK}\}) \end{array} \right\}$$

Note that some conditions have to be kept in order to avoid a direct contradiction between `flies` and $\neg\text{flies}$. Also note that in the above program, one could replace r_4 with `penguin` $\Rightarrow$ $\neg\text{normal-bird}$, $\neg\text{flies}$ to get rid of these conditions and have a unique minimal interpretation which contains $(\neg\text{flies}, \emptyset)$.

Definition 7 (Plausibility of a statement). *Given a PAL $\mathcal{A} = (S, A, R, P)$, a statement literal $s \in \text{lit}(S)$ is called plausible w.r.t. a rule selection R_s if all valid interpretations w.r.t. R_s contain a conditional statement (s, c) for some c .*

Definition 8 (Acceptability of a statement). *Given a PAL $\mathcal{A} = (S, A, R, P)$, a statement literal $s \in \text{lit}(S)$ is called acceptable w.r.t. a rule selection R_s if it is plausible and there exists a valid interpretation w.r.t. R_s which contains the conditional statement $(s, \emptyset)$.*

In the above example, `tux`, `bird`, and `tuxedo-plumage` are acceptable statements, while `flies`, $\neg\text{flies}$, and `penguin` are plausible but not acceptable. All other literals are not even plausible.

The probability $\text{prob}(R_s)$ of a rule selection $R_s \subseteq R$ is defined as follows:

$$\text{prob}(R_s) = \left(\prod_{r \in R_s} P(r) \right) \left(\prod_{r \in R \setminus R_s} 1 - P(r) \right)$$

which ensures that $\sum_{R_s \in \mathcal{P}(R)} \text{prob}(R_s) = 1$.

Definition 9 (Probability of a statement). *Given a PAL $\mathcal{A} = (S, A, R, P)$, the probability of a statement $s \in \text{lit}(S)$ is defined as the sum of the probabilities of all rule selections $R_s \in \mathcal{P}(R)$ in which s is acceptable.*

3.2 Transformation to CHRSiSM

We now introduce a transformation from an arbitrary probabilistic legal argumentation logic $\mathcal{A} = (S, A, R, P)$ to a CHRSiSM program $P_{CHRSiSM}(\mathcal{A})$. The transformation is a generalization of the example discussed in Section 2.2. We assume the list predicates **member**/2, **subset**/2, and **append**/2 (whose first argument is a list of lists) are already defined in the host language (Prolog).

The transformed program starts with the same three rules as in Section 2.2. These rules insure that **accept**/2 encodes an interpretation, redundant conditional statements are removed, as well as defeated statements. It ends with the rules for the **assume** phase as in Section 2.2.

In between, there are two CHRSiSM rules for each rule of $\mathcal{A}$. Each rule $r \in R$:

$$s_1 \wedge \dots \wedge s_n \Rightarrow c_1 \wedge \dots \wedge c_m \text{ assuming } a_1 \wedge \dots \wedge a_k$$

is transformed into one simple probabilistic CHRSiSM rule:

$$P(r) \text{ ?? begin } ==> \text{selected}(r).$$

and one non-probabilistic CHRSiSM rule:

$$\begin{aligned} &1 \text{ ?? selected}(r), \text{ accept}(s_1, C_1), \dots, \text{ accept}(s_n, C_n) \\ &==> \text{append}([C_1, \dots, C_n, [a_1, \dots, a_k]], NC), \\ &\quad \text{accept}(c_1, NC), \dots, \text{ accept}(c_m, NC). \end{aligned}$$

where all literals of the form $\neg x$ are encoded as **not**(x).

Correctness. It is relatively straightforward to see that when the **assume** phase starts, **accept**/2 encodes an interpretation that satisfies the criterion of Def. 5 w.r.t. the rule selection encoded by **selected**/1. It also does not contain gratuitous statements. However, the interpretation is not necessarily minimal. The **assume** phase searches for minimal interpretations by nondeterministically relaxing the assumptions. Since the relaxed **accept**/2 constraints will cause the transformed PAL rules to be revisited, the criterion of Def. 5 remains satisfied.

3.3 Relationship between PAL and Defeasible Logic

In defeasible logic [11], there are three kinds of rules: strict rules (denoted with $\rightarrow$), defeasible rules (denoted with $\Rightarrow$), and undercutting defeaters (denoted with $\sim$). There is also a precedence relation $<$ over the non-strict rules, to settle conflicting rules. Conflicts are defined using a set of conflict sets, which should contain at least all sets of the form $\{\phi, \neg\phi\}$.

Without loss of generality (cf. Theorem 6 of [12]) we can assume that the precedence relation is empty, there are no undercutting defeaters, and the conflict set are just the atomic formulas and their negations.

Given a closed defeasible theory D with a set of initial facts F and a set of rules $R_S \cup R_D$ (where R_S are the strict rules and R_D are the defeasible rules), we can translate it to a probabilistic argumentation logic $\text{PAL}(D) = (S, A, R, P)$ as follows:

- The set of statements S is the set of atoms appearing in F , R_S and R_D ;
- The set of assumptions A consists of fresh symbols $a_1, \dots, a_{|R_D|}$;
- The probability function P is the constant function 1;
- The rules R are constructed as follows:
 - For every fact $f \in F$, we add a rule $\Rightarrow f$.
 - For every strict rule $(X \rightarrow Y) \in R_S$, we add a rule $X \Rightarrow Y$.
 - For every defeasible rule $(X \Rightarrow Y) \in R_D$, we add a rule $X \Rightarrow Y$ assuming a_i , where i is the number corresponding to the defeasible rule.
 - If the i -th and the j -th defeasible rule are conflicting rules, that is, if rule i is of the form $(X \Rightarrow \psi) \in R_D$ and rule j is of the form $(Y \Rightarrow \neg\psi) \in R_D$, then we add the rules $\psi \Rightarrow \neg a_j$ and $\neg\psi \Rightarrow \neg a_i$.
 - If the i -th defeasible rule $(X \Rightarrow \psi) \in R_D$ conflicts with a strict rule $(Y \rightarrow \bar{\psi}) \in R_S$ (where $\bar{\psi}$ denotes the negation of the literal ψ), then we add the rule $\bar{\psi} \Rightarrow \neg a_i$.

To illustrate this translation, let us look at a trivial example: the defeasible rules $\Rightarrow p$ and $\Rightarrow \neg p$. Since these rules contradict one another, neither p nor $\neg p$ can be inferred. The translated PAL rules are the following:

$$\begin{aligned}
 &\Rightarrow p \text{ assuming } a_1 \\
 &\Rightarrow \neg p \text{ assuming } a_2 \\
 &p \Rightarrow \neg a_2 \\
 &\neg p \Rightarrow \neg a_1
 \end{aligned}$$

This PAL has two valid interpretations (w.r.t. all rules): $\{(p, \emptyset), (\neg a_2)\}$ and $\{(\neg p, \emptyset), (\neg a_1)\}$. As desired, neither p nor $\neg p$ are acceptable.

Conjecture 1 (Weak Completeness). The above translation is complete in the following sense: for all literals p in Lit_D , we have that if $D \vdash p$ (p is defeasibly derivable from D), then p is plausible in $PAL(D)$ (w.r.t. the selection of all rules) and if $D \sim p$ (p is defeasibly refutable in D), then p is not acceptable in $PAL(D)$.

Conjecture 2 (Soundness). The above translation is sound in the following sense: if the probability of p in $PAL(D)$ is 1, then $D \approx p$ (p is defeasibly entailed by D), and if the probability of p in $PAL(D)$ is 0, then $D \not\approx p$ (p is not defeasibly entailed by D).

4 Learning

For now, we have assumed the probabilities to be known in advance. As mentioned in the conclusion of [1], an issue is: *where do these numbers come from?* They suggest a statistical analysis of known precedents. The CHRiSM framework gives us exactly the tools needed to do this.

Instead of using fixed probabilities, we can make some or all probabilities learnable. We can then use a “training set” consisting of the outcomes of earlier similar cases — we call these the *observations* — to find a maximum likelihood probability distribution to fit the observations.

Table 1. Re-discovering the probabilities with CHReSM’s learning algorithm.

<i>Rule</i>	<i>Orig. prob.</i>	<i>Learned prob.</i>
a	1	0.999999723
b	0.9	0.875267302
d	0.5	0.540243696
e	0.2	0.204545455
f	0.3	0.258644714

<i>Rule</i>	<i>Orig. prob.</i>	<i>Learned prob.</i>
r1	1	0.999975929
r2	0.8	0.723429684
r3	0.5	0.547720330
r4	0.5	0.503165649

Some of the observations may be *full* observations, meaning that we not only know the final outcome, but also how exactly the reasoning went: what statements were put forward, what statements were accepted or rejected. More realistically, we only have *partial* observations: e.g. the final outcome is known, but not the intermediate steps. In CHReSM we can use both.

As a proof of concept, let us try to “rediscover” the original probabilities in our running example. Assume for the sake of the example we have a database of 1100 prior rulings, but we do not have the time and resources to read through all of them to find out exactly what the reasoning was. Say that we take 100 random samples and input them as full observations, for example

```
begin <==> accept(not app(r1), []), accept(d, []), accept(b, []), accept(a, []).
```

In this example observation, even though both **a** and **b** were accepted, **c** was not accepted, either because rule **r1** was not applied (remember, we do not know that it should have probability 1), or because its assumption **app(r1)** was refuted because **d** was accepted and rule **r2** was applied.

The remaining 1000 cases are not studied in that much depth: all that was checked is who won the case (i.e. was **c** accepted or not?) and whether or not statement **e** (“the cow was mad”) was accepted. For the four possible combinations, the following counts were recorded:

```
62 times begin ==> accept(c, []), accept(e, []).
432 times begin ==> accept(c, []), ~accept(e, []).
138 times begin ==> ~accept(c, []), accept(e, []).
368 times begin ==> ~accept(c, []), ~accept(e, []).
```

In full observations, the right hand side of the large double arrow has to be exhaustive, so there is no need for explicit negation. In partial observations, the rhs is not an exhaustive enumeration, so explicit negation (denoted by tilde) can be useful, like in the above example.

The full observations above were obtained by simply taking 100 random samples (i.e. running the query **sample begin**) on the original program with explicit probabilities; the counts for the partial observations were obtained by computing the probabilities for each case multiplying them by 1000.

Now that we have a training set, we can use the built-in learning algorithm to find a probability distribution that fits the data. The resulting probabilities are shown in Table 1; they approximate the original probabilities reasonably well.

5 Conclusion

We defined probabilistic argumentation logic (PAL) and showed how it can be used for probabilistic legal reasoning in the style of [1]. We provided an implementation of PAL in CHRiSM through a straightforward encoding. Where previous work only showed by hand-performed examples how the problems could be solved, we provide a formalism and its encoding in CHRiSM that implements the problems and automates the generation of solutions. This automation goes much further than the original problem statement. The resulting CHRiSM program can be used to compute the probabilities of possible outcomes, to obtain random samples, and to learn some or all underlying probabilities, solving an open problem in [1].

It remains a topic for future work to relate PAL to other existing proposals for defeasible reasoning, argumentation, and non-monotonic logic in general. We have already indicated how Nute's defeasible logic can be translated to a fragment of PAL in a sound and weakly complete way, although we postpone the proofs to future work.

References

1. Riveret, R., Rotolo, A., Sartor, G., Prakken, H., Roth, B.: Success chances in argument games: a probabilistic approach to legal disputes. In: JURIX. Volume 165 of *Frontiers in A.I. and Applications.*, IOS Press (2007) 99–108
2. Prakken, H., Sartor, G.: Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics* **7**(1) (1997) 25–75
3. Dung, P.M.: On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. *A.I.* **77**(2) (1995) 321–358
4. Roth, B., Riveret, R., Rotolo, A., Governatori, G.: Strategic argumentation: a game theoretical investigation. In: ICAIL, ACM (2007) 81–90
5. Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., Sato, T.: CHR(PRISM)-based probabilistic logic learning. *TPLP* **10**(4-6) (2010) 433–447
6. Frühwirth, T.: *Constraint Handling Rules*. Cambridge University Press (2009)
7. Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems* **31** (2008) 161–176
8. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. *TPLP* **10**(1) (2010) 1–47
9. Frühwirth, T., Raiser, F., eds.: *Constraint Handling Rules: Compilation, Execution, and Analysis*. BOD (March 2011)
10. Sneyers, J., De Schreye, D., Frühwirth, T.: Probabilistic legal reasoning in CHRiSM. In: ICLP 2013, 29th International Conference on Logic Programming. (August 2013)
11. Nute, D.: Defeasible logic: theory, implementation, and applications. In: INAP 2001, 14th International Conference on Applications of Prolog. (2001) 87–114
12. Maier, F., Nute, D.: Ambiguity propagating defeasible logic and the well-founded semantics. In: JELIA 2006, 10th European Conference on Logics in Artificial Intelligence. Volume 4160 of LNCS., Springer (2006) 306–318

CHR-IDE: An Eclipse Plug-In for Constraint Handling Rules

Matthias Rau, Amira Zaki, and Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
`{matthias-1.rau, amira.zaki, thom.fruehwirth}@uni-ulm.de`

Abstract. Constraint Handling Rules (CHR) is a rule-based programming language which extends a host language such as Prolog, Java or Haskell. Programmers usually write code using generic editors or integrated development environments (IDEs) specific for the host language used. This work presents an IDE developed for the K.U.Leuven CHR(Prolog) system as an Eclipse plug-in. It features a combined editor, executor and visual tracer aimed to enrich a developer’s experience.

Keywords: Constraint Handling Rules, Eclipse Plug-in, Editor, Tracer, Integrated Development Environment

1 Introduction

Constraint Handling Rules (CHR) is a committed-choice rule-based programming language based on a set of multi-headed multi-set rewrite rules [11]. CHR is a language extension which uses its host language for the evaluation of built-in constraints. The most common implementation, is that with Prolog, although implementations with Haskell, Java and C also exist. For a developer writing CHR code, it is common to use a generic editor and execute the code externally. Another option could be the use of an integrated development environment (IDE) associated with the host language, however such environments are not customized for CHR. They do not provide special handling for the CHR syntax, neither do they offer a proper CHR tracing mechanism showing interactions with the constraint store.

The K.U.Leuven CHR System is a state-of-the-art CHR system [13]. Their recommended CHR(Prolog) system is that of SWI-Prolog [14]. In this work, the aim is to provide an open-source platform-independent tool for CHR with SWI-Prolog. The idea is realized by creating an integrated development environment for CHR, which offers an editor functionality, an executor and a visual tracer. The main advantage of the tool is integrating these three functionalities together in one place, for the convenience of the developer. The tool also features a visualization of the rule applications as interactions with the constraint store. Constraints inserted and removed are clearly depicted visually in a step-by-step manner. Furthermore, the tool is a developer-assistant, offering syntax

highlighting and auto-completion while typing the code. The tool will be of particular interest for people learning programming with CHR as it enhances the user experience during writing, executing and tracing CHR.

In this work, an IDE is created for CHR with SWI-Prolog as a plug-in for Eclipse and is presented as a system demo in this paper. The paper follows by presenting the related work in Section 2, then the features of the plug-in developed are listed in Section 3. The paper concludes in Section 4.

2 Related Work

2.1 Plug-In Development

Eclipse is an open source platform that has become highly popular nowadays as an integrated development environment for Java [2]. It allows flexible program development and integration through its plug-in technology. The development of custom plug-ins which support application-specific functionalities is possible through the plug-in construction toolkit. Several applications have used this toolkit to implement editors for various programming languages [6,8,9,10].

2.2 Constraint Handling Rules

Constraint Handling Rules (CHR) is a high-level, committed-choice, rule-based programming language [11,12]. It consists of guarded rules that perform conditional transformation of multi-sets of constraints, known as a constraint store, until a fixed point is reached. CHR is a language extension and depends on a host language for the implementation of built-in constraints; current implementations extend Prolog, Java and Haskell. In this work, CHR is used under SWI-Prolog [14]. CHR utilizes the built-in constraints together with other user-defined CHR constraints. A generalized CHR simpagation rule is given as:

$$[\textit{simpagation-id } \mathbb{Q}] H_k \setminus H_r \Leftrightarrow [G \mid] B.$$

where H_k and H_r are a conjunction of one or more CHR constraints that are kept/removed respectively. The rule consists of an optional conjunction of host language constraints known as the guard and given by G . Following the partitioning $\mid$ is the body of the rule (B), and it consists of built-in constraints and user-defined CHR constraints. Every rule has an optional unique identifier preceding it, followed by an $\mathbb{Q}$.

Two other types of rules exist which are special cases of the generalized simpagation rule, namely simplification and propagation rules. A simplification rule does not keep any head constraints, while a propagation rule does not remove any head constraints. These rules are of the form:

$$\begin{aligned} [\textit{simplification-id } \mathbb{Q}] H_r &\Leftrightarrow [G \mid] B. \\ [\textit{propagation-id } \mathbb{Q}] H_k &\Rightarrow [G \mid] B. \end{aligned}$$

A complete CHR program consists of the CHR rules, in addition to definitions for some user-defined built-in constraints and other built-in constructs. The complete valid syntax of CHR supported by SWI-Prolog and used for this work can be found in [14].

2.3 Development Tools

Currently, several libraries exist for CHR as an extension of Prolog, Haskell, C and Java. Similarly various development tools exist for these extensions, however no tool to our knowledge features both an integrated editor and a visual tracer customized for CHR.

For SWI-Prolog, Gerhard Röhner [1] has developed an integrated Prolog editor in MS-Windows following the conventions of this platform. It is an editor mainly for Prolog, yet it also supports CHR programs. However the editor is not built with specific handling for CHR, furthermore the tracer is only meaningful for Prolog programs and shows a rather useless trace with CHR.

Prolog Development Tools (ProDT) [7] is a Prolog integrated development environment aiming to be as rich in functionality as java's Eclipse IDE. It provides the developer a single environment to control the development of a Prolog project starting from code edition, to test execution and debugging. This project stands on top of Eclipse's projects to take advantage of its already existent features and its extensibility. The work implemented in this paper is quite similar to ProDT, however we exceed the functionality of ProDT by customizing it to the syntax and use of CHR. Furthermore the debugger offered by ProDT is specialized for Prolog programs, and hence does not work well with CHR. It does not show a constraint store, nor visualize the interactions with it.

Another similar work, was performed for JCHR (CHR extending Java) [3]. An IDE was also developed as an Eclipse plug-in for JCHR of the K.U.Leuven system. This IDE provided a editor interface to facilitate the writing of JCHR programs. Moreover it incorporated a compiler capable of compiling the program written and presenting the output in an external window.

An earlier work proposed a Java library known as JACK (JAvA Constraint Kit) [4]. This introduced the use of CHR in Java to write application specific constraint solvers, and provided an interactive tool to visualize the computations. The visualization offered by the tool shows the constraint store as a box, and handles constraints as sub-boxes which are inserted and removed from the box store as the program proceeds. This visualization is similar to the tracing mechanism implemented in our work, however it was implemented on the compiler level.

Another visualization was performed for CHR programs, in order to visualize the execution of the rules [5] through a source-to-source transformation of the CHR programs. Graphical objects are associated with the constraints and the effect of the rule applications are shown on the objects.

3 CHR-IDE

The Eclipse plug-in developed is available online, under <http://pmx.informatik.uni-ulm.de/chr/plugin.zip>. It features three main features which are discussed and illustrated in this section.

3.1 Editor

The aim of the editor is to enhance the user experience whilst programming in CHR. This begins with the addition of a file wizard. The wizard automatically creates a CHR file with the necessary headers. The wizard also allows the programmer to define the user-defined constraints that will be declared with their respective arity. The declaration of these constraints is then automatically written to the created file.

The editor has been enabled with a content assistant functionality. This makes the task of writing CHR rules, much easier. When starting to write any line, it is possible to insert a template for any type of rule. This provides a choice of the three rule types (with and without guards) by inserting the delimiters of the chosen rule type. A snapshot of the editor whilst writing a CHR program is shown in Figure 1.

Additionally an auto-completion feature has been implemented. While typing, one can easily access the list of constraints declared and have them automatically inserted. Auto-completion makes the task of writing CHR rules easier and simpler.

Syntax checking has been enabled of the CHR constructs as listed in the SWI-Prolog manual. The plug-in developed performs a verification of the types code against the CHR grammar. Syntax errors are identified by pattern matching and marked by underlining them. Additionally some warnings are identified, such as assigning a rule identifier to more than one rule.

Furthermore, the editor has been enhanced with a highlighting of the source code written particularly for CHR constructs. The highlighting is done by a rule-based scanner designed precisely for the syntax of CHR. It distinguishes between comments (single-lined and multi-lined ones), rule identifiers, rule delimiters, guard expressions and other code lines which do not contain CHR rules. The syntax highlighting implemented can be seen in Figure 1. The colors chosen can be modified from the preferences of the editor.

An outline of the program is also created continuously as the program evolves and is presented in a side panel. It can be seen in Figure 1 in the top-right corner. Through this outline, it is possible to see the constraints declared with their arity, the rules declared so far and the user-defined built-in predicates defined. Each of these is differentiated from the others by a *C*, *R* and *P* icon respectively (also named rules are marked with a *N*). This enables a programmer at a glance, to have an overview of the written program. It is useful for large programs which contain mixed ordering of constraint declarations and rules, by providing a summary of the program constructs.

Fig. 1. Snapshot of CHR-IDE during writing the greatest common divisor in CHR.

Fig. 2. Snapshot of CHR-IDE in debugging/tracer mode, showing constraints currently in the store and a shorter more readable textual trace.

3.2 Executor

The plug-in features a link to SWI-Prolog as the Prolog system installed on the machine. This includes CHR as an extension based on the K.U.Leuven CHR system [13]. Through a console window, all interactions with the SWI-Prolog console are facilitated. Hence a programmer can “run” the code; i.e. consult it. Then the console acts as an interpreter, executing input queries given. The console is identical to that of SWI-Prolog, meaning that backtracking and other options are also enabled.

3.3 Tracer

Another valuable feature of the plug-in developed, is the implementation of a visual code tracer. It is possible using the SWI-library of CHR, to switch on a tracer of the CHR code, by running the program in a debugging mode. This enables a step by step trace of the interactions with the constraint store. The tracer shows how constraints from the query are inserted into the store, in a left to right order. The constraint store is visible in a panel on the right as shown in Figure 1. Then the tracer highlights rule applications whilst writing which constraints are removed and added due to the firing of the rule (as in Figure 2).

The tracer implemented in the plug-in works by parsing the textual CHR trace obtained from SWI (obtained by `chr_trace`). The lines produced of the trace are always indexed with the word `CHR:` followed by keywords which indicate the action that is performed, these are `Insert`, `Call`, `Apply`, `Remove`. The tracer implemented handles line by line the textual trace obtained, and reflects it visually to a constraint store panel. The panel represents the constraint store during the execution of a query. Whenever a constraint is added to the store, it is inserted in the panel as well. Then whenever a rule application takes place, the matching head constraints are highlighted in a different color. Then the rule application occurs, and the result of removing and adding constraints is reflected to the store. This visualization is quite simplistic, however much more readable and understandable than following the textual trace.

In the textual trace, SWI automatically renames local variables with an internal naming convention (an underscore, a letter then followed by a sequence of numbers). During a trace of multiple variables, it becomes quite tedious to follow the variables. Thus to enhance the readability of the variables in the constraint store, the tracer implemented renames variables to short alphabetical names.

Furthermore to extend the readability, a supplementary shorter and more readable textual trace is displayed in another console (as seen in the bottom on Figure 2). This includes a textual representation of the trace, with the modified variable names and having redundant constructs removed. With experience, it has been seen that this shorter trace is more understandable than the original longer one.

4 Conclusion

In this work, a complete integrated development environment (IDE) was implemented for the K.U.Leuven CHR system offered with SWI-Prolog. The IDE was created as a plug-in for Eclipse, which offers a combined editor, executor and visual tracer functionality. The editor provides customized syntax highlighting, syntax error checking, auto-completion and a content assistant functionality. These features enrich the developer experience whilst writing CHR. The plug-in integrates a means to compile and execute the written programs. It uses the SWI-Prolog installation on the machine. The plug-in also features a tracer, capable of visualizing the execution of the rules by graphically displaying the constraint store and coloring the interactions with it.

The plug-in is still under development and several extensions will be implemented to increase its usefulness. One such feature would be to integrate the compiler detected errors and warnings. Compiler detected syntax errors could be reflected back in the IDE by highlighting the error line, moreover fixes can be suggested for warnings such as removing rules that never fire. Another next step, would be handling type and mode declarations, and integrating some static type checking in the IDE and proposing type decelerations heuristically based on the rules. The tracer could also be enhanced by allowing breakpoints on rules or occurrences of constraints. Furthermore, we aim to integrate the visualization of [5] to depict the constraints and their interactions visually by graphical components through a source-to-source transformation of the code.

References

1. SWI-Prolog-Editor for Windows. <http://lakk.bildung.hessen.de/netzwerk/faecher/informatik/swiprolog/indexe.html>. Accessed: 24.05.2013.
2. The Eclipse Foundation open source community wesbite. <http://www.eclipse.org/>. Accessed: 24.05.2013.
3. Slim Abdennadher and Shehab Fawzy. JCHRIDE: An Integrated Development Environment for JCHR. In Sibylle Schwarz, editor, *WLP '08: Proc. 22nd Workshop on (Constraint) Logic Programming*, pages 1–6, Dresden, Germany, September 2008.
4. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauß. JACK: A Java Constraint Kit. In *WFLP '01: Proceedings of 10th International Workshop on Functional and (Constraint) Logic Programming*, volume 64, pages 1–17, 2002.
5. Slim Abdennadher and Nada Sharaf. Visualization of CHR through Source-to-Source Transformation. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 109–118. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.
6. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Ranjit Majumdar. An eclipse plug-in for model checking. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 251–255, 2004.
7. Claudio Cancinos and Serrano Carolina. Prolog Development Tools (ProDT). <http://prodevtools.sourceforge.net/index.html>. Accessed: 24.05.2013.

8. Gerardo Canfora and Luigi Cerulo. Jimpa: An eclipse plug-in for impact analysis. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 341–342, 2006.
9. Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. Jasper: an eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, eclipse '06, pages 65–69, 2006.
10. Karsten Ehrig, Claudia Ermel, Stefan Hänsen, and Gabriele Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, 2005.
11. Thom Frühwirth. *Constraint handling rules*. Cambridge University Press, 2009.
12. Thom Frühwirth and Frank Raiser, editors. *Constraint Handling Rules: Compilation, Execution, and Analysis*. Books on Demand, March 2011.
13. Tom Schrijvers and Bart Demeo. The k.u.leuven chr system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.
14. Jan Wielemaker, Thom Frühwirth, Leslie De Koninck, Markus Triska, and Marcus Uneson. *SWI Prolog Reference Manual 6.2.2*. Books on Demand, September 2012.

Tagging Reasoning System using CHR

Nada Sharaf, Injy Hamed and Slim Abdennadher

The German University in Cairo,
Faculty of Media Engineering and Technology
{nada.hamed,injy.hamed,slim.abdennadher}@guc.edu.eg

Abstract. Collaborative tagging systems allow their users to mark resources with labels thus providing metadata about shared content. This paper presents a new reasoning engine for collaborative tagging systems. The engine was built using Constraint Handling Rules (CHR). Through the new engine, different application-specific properties are captured. In addition, the engine is able to incorporate generally needed features for reasoning such as tag hierarchies and tag co-occurrences.

Keywords: Constraint Handling Rules, Collaborative Tagging Systems, Tag Reasoning

1 Introduction

Social annotation systems have grown in popularity over the web in the last years. Such systems allow users to attach descriptors (or tags) to resources, such as web page bookmarks in del.icio.us ¹, academic publications in CiteULike ², and photographs in Flickr ³. The great success achieved by these services is solid proof that annotation is a great collaboration tool. It is considered to be an intuitive and easy way for people to describe objects.

The act of annotation serves several purposes. Tags provide precise and meaningful descriptions to items. This abstracted information about the item can be very useful in navigation, browsing, searching, and discovering subjectively interesting items. Tags enable users to organize and categorize the resources they describe as well as to query the system for resources that satisfy certain criteria. For example, a user might search for films that are referred to as **detective** and not **horror**. Tags can also be used to identify similar items in recommender systems. They also provide a novel source of information about the users' interests and preferences. This enables the identification of users with similar interests within a given category.

The success of social annotation systems has given rise to many challenging problems to the research community on how to fully utilize the tags available in the systems. Tag reasoning is used to tackle problems faced by social annotation

¹ <https://www.delicious.com/>

² <http://www.citeulike.org/>

³ <http://www.flickr.com/>

systems such as data sparsity, data validation, resolving ambiguity and building user profiles. In this paper we present an inference system for tag reasoning using Constraint Handling Rules (CHR).

Constraint Handling Rules (CHR) [6, 5] is a concurrent committed-choice constraint programming language that was developed for the implementation of constraint solvers. CHR has evolved from a language designed for writing constraint solvers to a general purpose language. CHR has many implementations. However, the Prolog implementation is the most prominent one. With its declarative semantics, CHR has proved to be useful in various fields of applications. This paper uses CHR in the field of Semantic Web and collaborative tagging systems. After the huge impact that the World Wide Web has had on people and how they communicate [1], the Semantic Web was proposed to make use of content that is understandable by computers [2]. The Semantic Web extends the current web and gives data and information meanings. Through such metadata, computers and people will be able to work together and communicate. In addition, software agents should be able to perform tasks that are hard for users [2]. To have such functionalities, the Semantic Web is built using different layers. Some layers aim at representing and exchanging data and metadata. Other layers are dedicated to querying. The logic layer allows for defining application-specific and declarative rules. The proof layer executes these rules [7, 1]. There are different rule based approaches and formalisms for the Semantic Web such as RuleML [3]. Through this work, we aim to present CHR as the language for the Semantic Web. We aim to prove that with its declarativity, CHR is suitable for the different aspects and needs of the Semantic Web. This paper, however, focuses on annotations in collaborative tagging systems. The presented work is, to the best of our knowledge, the first reasoning engine that uses CHR as the mechanism for tag reasoning. This paper discusses how CHR was used to be the basis of a new reasoning engine for a collaborative tagging system. This paper is divided as follows: in Section 2, CHR is introduced. Section 3 introduces the newly implemented inference system. Finally, conclusions and directions for future work are provided.

2 CHR By Example

The rest of the section provides an example of a CHR solver that defines the partial order relation $\leq$ (`leq/2`). This example is used to introduce the syntax and semantics of CHR. `leq(A,B)` holds if A is less than or equal to B.

```
:- use_module(library(chr)).
:- chr_constraint leq/2.
reflexivity @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y) , leq(Y,X) <=> X=Y.
idempotence @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y) , leq(Y,Z) ==> leq(X,Z).
```

The first rule is called `reflexivity` (rule names are optional). Its head consists of a single constraint. This rule replaces constraints of the form `leq(A,A)` by `true`.

Therefore, through such rule, any constraint of the form `leq(A,A)` is removed from the constraint store. The `antisymmetry` rule, is another simplification rule with two heads. Through this rule, symmetric `leq` constraints are replaced by an equality constraint. Such constraint is handled through the host language and thus Prolog does unification in this case. Simplification rules correspond to logical equivalence, as the syntax suggests.

The third rule is a simplification rule which is used to remove redundant copies of the same constraint. Such rules are needed because of the multi-set semantics of CHR. Finally, the `transitivity` rule is a propagation rule that adds new constraints to the constraint store. Such added constraints could then cause further simplification. Propagation rules correspond to logical implication.

Execution proceeds by exhaustively applying the rules to the input query. For the query `leq(A,B), leq(B,C), leq(C,A)` the transitivity rule first adds `leq(A,C)`. Afterwards, the `antisymmetry` rule is applied, replacing the two constraints `leq(A,C)` and `leq(C,A)` by `A=C`. The `antisymmetry` rule is thus now applicable on the first two constraints of the original query. Now all CHR constraints are eliminated so no further rules can be applied, and thus the answer `A=C, A=B` is returned.

3 Inference System

This section introduces how CHR can be used to enhance the collaborative tagging process through providing a reasoning engine. Through such engine, users can be provided with different tag explanations and implications. In addition, the whole process can be analyzed and studied.

In order to be able to use CHR to reason about tag annotations, such annotations have to first be represented. For this purpose, a new CHR constraint called `annotation` was used. For example the constraint

```
annotation(alex,harrypotter,sciencefiction,'1/2/2012',1)
```

means that the user with the name `alex` annotated the object named `harrypotter` with the tag `sciencefiction` on the date 1/2/2012. The last argument represents the fact that this annotation was the first annotation made by `alex` to the object `harrypotter`. The annotations added by users are thus represented in this format in order to be able to reason about them using the provided CHR engine.

The implemented engine is able to add different properties to the collaborative tagging system. The rest of the section presents these features in details.

3.1 Application-Specific Inference Rules

Applications generally have their own specific rules that have to be taken into consideration when reasoning. An example is the one presented in [4], where in the case of annotating art objects, if at least 95% of the users chose colors as the

first two tags, then the object was considered to be an “abstract art object”. As introduced in Section 2, the semantics of CHR make application-specific rules straightforward to implement using CHR rules. In this section, some examples of application-specific rules are introduced. In addition, the implementation of these rules using CHR is also presented.

Absence In Presence

This section discusses the class of rules where a new tag is added in the case that a specific tag is absent in the presence of another tag. For example as presented in [4], whenever an object has the tag **bug** but the tag **fixed** is missing, then a new tag named **todo** should be added to the object. This denotes the fact that there is a bug that was not fixed yet and has to be added to a **todo** list.

The special information regarding the tags to be monitored and the new tags to be added has to be represented in a specific format understandable by the reasoning engine. Consequently, a CHR constraint named **absenceinpresenceimplies/3** is used. For example, the constraint **absenceinpresenceimplies(fixed,bug,todo)** is used to represent the case that whenever an object is annotated with the tag **bug** and the tag **fixed** is missing, for the same object, then a new tag named **todo** should be added to the object. The CHR rules used to infer the new tag from the old ones are given below:

```
absinpresimplies(F,B,TD),annotation(U,I,F,_,_) ==> safe(I,F,B,TD).
absinpresimplies(F,B,TD),annotation(U,I,B,_,_) ==> dangerous(I,F,B,TD).

dangerous(I,F,B,TD),safe(I,F,B,TD)<=>true.
absinpresimplies(F,B,TD)\dangerous(I,F,B,TD) <=>
    annotation(automatic,I,TD,automatic,-1).
```

For example, whenever an item *I* is tagged with **fixed**, then the constraint **safe(I,F,B,TD)** is added to the constraint store. The second rule adds the constraint **dangerous(I,F,B,TD)** whenever the item *I* is tagged with **bug**. The third rule is a simplification rule that represents the fact that when an item is annotated with **bug** and **fixed**, then no additional annotation is to be added. The last rule is thus only applied whenever the item is only tagged with **bug** and the new annotation with the tag **todo** is added.

With the data set containing the following input information:

```
annotation(injy,item,bug,'1/2/2012',1).
absenceinpresenceimplies(fixed,bug,todo).
```

the engine adds to the constraint store the following CHR constraint

```
annotation(automatic,item,todo,automatic,-1)
```

This constraint encodes the information that the annotated object is a bug that was not fixed and was thus added to a **todo** list. In this specific example, the user *injy* annotated the object with **bug** and the constraint


```
absenceinpresenceimplies(fixed,bug,todo)
```

represented the fact that an item tagged with **bug** with no corresponding **fixed** annotation has to be added to the **todo** list.

Tag Percentage

Another application-specific rule introduced in [4], is concerned with the case when the class of the annotated object can be identified by the properties and the frequency of the already used tags. In more details, the offered example was connected with tagging art objects. The specific rule is that if at least 95% of the users use colors as their first two tags, then the tagged object is an “abstract art object”. The offered engine can account for such cases. The colors are represented using a special constraint. Afterwards, the percentage of collected tags is calculated and compared with the threshold (which is 95% in this case). The new tag can, therefore, be added after the comparison is done. For example if the initial data set included the following annotations:

```
annotation(injy,artobj,red,'1/2/2012',1).
annotation(injy,artobj,blue,'1/2/2012',2).
annotation(injy,artobj,sf,'1/2/2012',3).
annotation(nada,artobj,green,'1/2/2012',1).
annotation(nada,artobj,blue,'1/2/2012',2).
```

then the engine produces the following annotation

```
annotation(automatic,artobj,'abstract art',automatic,-1)
```

since 100% of the users had colors as their first two annotations for this object.

3.2 General Inference Rules

This section discusses the general inference rules the engine is able to handle. These rules are thus not application-specific and could be used for any tagging system.

Inconsistent Tags

Inconsistency of tags is a problem that arises when an object is annotated with conflicting tags. It might be useful to identify such objects for further analysis. The provided engine is able to perform this functionality through the use of CHR rules. The inconsistent tags are enumerated using the constraint **inconsistent/2**. Afterwards, through a propagation rule, whenever an object is tagged with two tags that are inconsistent, the object is marked as having inconsistent tags using the constraint **itemInconsistent/1**. The propagation rule used is:

```
inconsistent(Tag1,Tag2),annotation(_,I,Tag1,_,_),
    annotation(_,I,Tag2,_,_) ==> itemInconsistent(I).
```

In this case **Tag1** and **Tag2** are inconsistent and the item **I** is tagged with both **Tag1** and **Tag2** and thus is considered to have a source of inconsistency. In addition to identifying objects with inconsistencies, it is useful to identify if a specific user has tagged an object with inconsistent tags. In this case, such user can be identified to be a source of noise if this behavior persists. Such users are represented using the constraint **userInconsistent/1**. To identify such users, a similar propagation rule is also used:

```
inconsistent(Tag1,Tag2),annotation(U,I,Tag1,_,_),
    annotation(U,I,Tag2,_,_) ==> userInconsistent(U,I).
```

For example, given the following data set:

```
inconsistent(nice,ugly).
annotation(john,artobj,nice,'1/2/2012',1).
annotation(john,artobj,blue,'1/3/2012',2).
annotation(john,artobj,sf,'1/4/2012',3).
annotation(alex,artobj,ugly,'1/2/2012',1).
annotation(alex,artobj,nice,'1/3/2012',2).
```

In this case, the engine produces the constraints **itemInConsistent(artobj)** and **userInConsistent(alex, artobj)**. The two tags **nice** and **ugly** were identified to be inconsistent through the constraint **inconsistent(nice,ugly)** in the data set. Since the item **artobj** was annotated with both tags **nice** and **ugly**, the constraint **itemInConsistent(artobj)** was produced. In addition, the user **alex** annotated the same object (**artobj**) with both constraints **nice** and **ugly** and thus the constraint **userInConsistent(alex, artobj)** was produced.

Tag Hierarchy

The number of annotations given by users to resources depends on their motivation to annotate. It is usually the case that this number is small, as annotation is a tedious task. Hierarchies between tags can be used to infer non-existing tags that are related to a resource. This adds tags to poorly annotated resources as well as improves search recall. For example, if the resource **I, Robot** is assigned the tag **hard science fiction**, then according to the tags hierarchy shown in Figure 1, the tags **science fiction** and **fiction** will be added to the resource **I, Robot**. The ancestor relation between two tags is represented using the constraint **ancestor/2**. The CHR rules used for inference are shown below. The first rule covers the transitivity closure of the ancestor relation. Afterwards, propagation is used to infer the annotations to be added to the system.

Fig. 1: Tag Hierrachy.

```

ancestor(X,Y),ancestor(Y,Z) ==> ancestor(X,Z).
ancestor(X,Y), annotation(U,I,Y,D,_) ==> annotation(automatic,I,X,D,-1).

```

Co-occurrences

The co-occurrence technique is one of the famous techniques known from the Social Network Analysis [10]. A co-occurrence graph is defined as an undirected graph $G = (V; E)$. V is the set of nodes in the graph, where each node corresponds to a tag. E is the set of edges, where $E(t1, t2)$ exists if both tags co-occur in at least one item. The weight of each edge is defined as the number of items in which the two tags co-occur. There exist several ways to construct the co-occurrence graph [8, 9]. The different methods mainly differ in the weight value assigned to the edges. However, the general assumption is that, the more often two tags are used in combination, the stronger the relationship between them is. The co-occurrence graph is used to tackle several problems, such as tag recommendations, resolving tag ambiguity, learning user profiles and inferencing tag hierarchies. The provided engine infers tag co-occurrences using the constraint `association/3`. The constraint `association(pyramid, egypt, 5)` means that the tags `pyramid` and `egypt` co-occurred in 5 resources. The following are the rules used in the engine:

```

annotation(_,I,T1,_,_),annotation(_,I,T2,_,_) ==>
    T1@<T2 | association(T1,T2,1).
association(T1,T2,X),association(T1,T2,Y) <=>
    Z is X+Y, association(T1,T2,Z).

```

For example the following data set:

```

annotation(injy,item1,'pyramids','1/2/2012',1).
annotation(injy,item1,'history','1/2/2012',2).
annotation(injy,item1,'Ancient Egyptians','1/2/2012',3).
annotation(nada,item2,'history','3/2/2012',1).
annotation(nada,item2,'Ancient Egyptians','3/2/2012',2).
annotation(alex,item3,'Egyptian museum','1/2/2012',1).
annotation(alex,item3,'Ancient Egyptians','1/2/2012',2).

```

corresponds to the co-occurrence graph given in Figure 2.

Fig. 2: Tag Co-occurrences graph.

Through the use of the reasoning engine, the following constraints are produced: `association('Ancient Egyptians', 'Egyptian museum', 1)`, `association('Ancient Egyptians', 'history', 2)`, `association('Ancient Egyptians', 'pyramids', 1)`, `association('history', 'pyramids', 1)` which correspond to the previous graph. For example, the two tags 'Ancient Egyptians' and 'pyramids' co-occurred one time together through the annotations `annotation(injy, item1, 'pyramids', '1/2/2012', 1)` and `annotation(injy, item1, 'history', '1/2/2012', 2)`.

4 Conclusion and Future Work

CHR provided a simple and yet powerful method to implement the needed inference rules for application-specific and general purpose rules. CHR was suitable as the basis of a reasoning engine for collaborative tagging systems due to its declarativity. With CHR, the added rules are intuitive thus resulting in simple short solvers. An example of that is the solver presented for computing co-occurrences. The solver had only two rules. A Java program for the same purpose would, on the other hand, be rather lengthy.

For future work, CHR rules can be added to use tag co-occurrences to learn tag hierarchies which can enhance searching and browsing collaborative tagging systems. Moreover, tag clustering could be introduced to overcome some of the problems that occur with tags such as tag ambiguities and similarities. In addition, the engine can be extended with different new rules that are able to overcome many of the problems faced in tagging systems such as validating tags and removing any source of ambiguity. The engine can be also extended to be used for recommendations and for profiling users.

References

1. Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA, 2004.
2. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.
3. Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale for RuleML: A Markup Language for Semantic Web Rules. In Isabel F. Cruz, Stefan Decker, Jérôme Euzenat, and Deborah L. McGuinness, editors, *SWWS*, pages 381–401, 2001.

4. François Bry and Jakub Kotowski. Towards reasoning and explanations for social tagging. In Thomas Roth-Berghofer, Stefan Schulz, David B. Leake, and Daniel Bahls, editors, *ExaCt*, pages 118–128, 2008.
5. Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.
6. Thom W. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.
7. Marja-Riitta Koivunen and Eric Miller. W3C Semantic Web Activity. In *Proceedings of the Semantic Web Kick-off Seminar in Finland*, November 2001. <http://www.w3.org/2001/12/semweb-fin/w3csw>.
8. Elke Michlmayr. Learning User Profiles from Tagging Data and Leveraging them for Personal(ized) Information Access. In *Proceedings of the Workshop on Tagging and Metadata for Social Information Organization, 16th International World Wide Web Conference (WWW2007)*, 2007.
9. Elke Michlmayr, Steve Cayzer, and Paul Shabajee. Add-A-Tag: Learning Adaptive User Profiles from Bookmark Collections. In Natalie S. Glance, Nicolas Nicolov, Eytan Adar, Matthew Hurst, Mark Liberman, and Franco Salvetti, editors, *ICWSM*, 2007.
10. S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, 1994.

Author Index

A

Abdennadher, Slim 1, 25, 61

D

Dahl, Veronica 13

De Schreye, Daniel 39

E

Egilmez, Sinan 13

F

Fakhry, Ghada 1

Fruehwirth, Thom 39, 53

H

Hamed, Injy 61

M

Martins, Joao 13

Miralles, Emilio 13

O

Osman, Amr 25

R

Rau, Matthias 53

S

Sharaf, Nada 1, 61

Sneyers, Jon 39

Z

Zaki, Amira 25, 53