1 Terminological Reasoning with Constraint Handling Rules

Chapter 19 in Principles and Practice of Constraint Programming,
V. Saraswat and P. Van Hentenryck (Eds.), MIT Press, 1995.

Thom Frühwirth[1] and Philipp Hanschke[2]

1.1 Abstract

Terminological knowledge representation formalisms in the tradition of KL-ONE enable one to define the relevant concepts of a problem domain and their interaction in a structured and well-formed way. In recent years, a wealth of literature has appeared on this topic, e.g. [Baj93]. Sound and complete inference algorithms for terminological logics have been developed using constraint calculi. These terminological reasoning services can be reduced to a single algorithm checking consistency.

We propose constraint handling rules (CHRs) as an implementation language for terminological reasoning. We will give an introduction into the language. CHRs are a flexible means to implement 'user-defined' constraints on top of existing host languages like Prolog and Lisp. In particular, inference rules, which are often used to define constraint calculi, can usually be written as CHRs with little modification. The result of using CHRs for terminological reasoning is an incremental and concurrent consistency checking algorithm.

The implementation provides a natural combination of three layers: (*i*) a constraint layer that reasons in well-understood domains such as rationals or finite domains, (*ii*) a terminological layer providing a tailored, validated vocabulary on which (*iii*) the application layer can rely. As an application example, a configuration problem is modeled. The flexibility of the approach will be illustrated by extending the formalism, its implementation and the example with attributes, a new quantifier and concrete domains.

[1]Address: ECRC, Arabellastrasse 17, D-81925 Munich, Germany, thom@ecrc.de.
Partially supported by ESPRIT Project 5291 CHIC.
[2]Current address: sd&m, Schimmersfeld 7a, D-40880 Ratingen, Germany, hanschke@uni-duesseldorf.de. Partially supported by BMFT Projects ARC-TEC (Grant ITW 8902 C4) and IMCOD/VEGA (Grant 413 5839 ITW 9304/3). Work was done while at DFKI, Postfach 2080, D-67608 Kaiserslautern, Germany.

1.2 Introduction

A reader interested in constraint-based programming may consider this chapter as a case study on user-defined constraints using the CHRs approach. A reader with background in terminological reasoning may learn about a flexible way to implement terminological languages. We tried to tackle the difficulties of serving both audiences. The final judgement is to you, the reader.

Terminological formalisms based on KL-ONE [BS85] are used to represent the terminological knowledge of a particular problem domain on an abstract logical level. To describe this kind of knowledge, one starts with atomic concepts and roles, and then defines new concepts and their relationship in terms of existing concepts and roles. For example,

```
grand_father isa male and some child is parent.
```

Concepts can be considered as unary relations which intensionally define sets of objects (similar to types). Roles correspond to binary relations over objects (not necessarily of the same kind - properties like `color` can be roles as well). Then, assertions are added, describing actual objects of the application, e.g.

```
john:male.  (john,jack):child.  jack:parent.
```

Terminological formalisms have a straightforward embedding in first-order logic, so it seems natural to implement them as logic programs. Moreover, the limited expressiveness of terminological formalisms allows for decision procedures for a number of interesting reasoning problems. These problems include consistency of assertions and classification of concepts. The key idea of [Hol90, ScSm91, BDS93]) for constructing such inference algorithms is to reduce all reasoning services to consistency checking. This essential algorithm can be considered as constraint solving, where concepts and roles are the constraints.

We therefore aim at implementing reasoning with terminologies as constraint logic programs (CLP). In this way, an instance of the CLP scheme results, and we can carry over the declarative and operational semantics from CLP. *Constraint logic programming* [JaLa87, VH89, HS90, VH91, F*92, JaMa] combines the advantages of logic programming and constraint solving. In logic programming, problems are stated in a

declarative way using rules to define relations (predicates). Problems are solved by the built-in logic programming engine (LPE) using chronological backtrack search. In constraint solving, efficient special-purpose algorithms are employed to solve problems involving distinguished relations referred to as constraints.

A practical problem remains: Constraint solving is usually 'hard-wired' in a built-in constraint solver (CS) written in a low-level language. While efficient, this approach makes it hard to build a CS over a new domain like terminologies, let alone verify its correctness. We proposed *constraint handling rules* (CHRs) [Fru92] to overcome this problem. CHRs are a language *extension* providing a declarative means to introduce *user-defined* constraints into a given high-level host language. In this chapter the host language is Prolog, a CLP language with equality over Herbrand terms as the only built-in constraint. CHRs define *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence, e.g.

```
X>Y,Y>X <=> false.
```

Propagation adds new constraints which are logically redundant but may cause further simplification, e.g.

```
X>Y,Y>Z ==> X>Z.
```

When repeatedly applied by a constraint handling engine (CHE) the constraints are incrementally solved as in a CS, e.g.

`A>B,B>C,C>A` results in `false.`

CHIP was the first CLP language to introduce constructs (demons, forward rules, conditionals) [VH89] for user-defined *constraint handling* (like constraint solving, simplification, propagation). These various constructs have been generalized into CHRs. CHRs are based on guarded rules, as can be found in concurrent logic programming languages [Sha89], in the Swedish branch of the Andorra family [HaJa90], Saraswats cc-framework of concurrent constraint programming [Sar93], and in the 'Guarded Rules' of [Smo91]. However all these languages (except CHIP) lack features essential to define non-trivial constraint handling, namely for handling conjunctions of constraints and defining constraint propa-

gation. CHRs provide these two features using multi-headed rules and propagation rules.

This chapter first introduces CHRs, then terminological reasoning. Next it is shown that CHRs indeed can serve as a flexible implementation layer for an incremental and concurrent version of the consistency test for terminologies. Moreover, the implementation effort turns out to be minimal, as the CHRs directly reflect the inference rules of the tuned tableaux calculus that is used for the consistency test.

Last but not least we illustrate that extensions to the basic terminological formalism proposed in the literature carry over to the implementation with CHRs in a painless manner. One such extension allows to parameterize terminologies with *concrete domains*, e.g. linear constraints over rational numbers [BaHa91, Han92]. Concrete domains can be either implemented by CHRs or provided as built-in constraints of the host language. In this way we obtain a fairly natural combination of three knowledge representation layers - the constraint, terminological and application layer - on a common implementational basis.

1.3 CLP with Constraint Handling Rules

Here we assume that constraint handling rules extend a a given constraint logic programming language. The syntax and semantics given reflect this choice. It should be stressed, however, that the host language for CHRs need not be a CLP language. Indeed, the work presented here has been done at DFKI in the context of LISP [Her93].

1.3.1 Syntax

A CLP+CH program is a finite set of clauses from the CLP language and from the language of CHRs. Clauses are built from atoms of the form $p(t_1, ...t_n)$ where p is a predicate symbol of arity n ($n \geq 0$) and $t_1, ...t_n$ is a n-tuple of terms. A term is a variable, e.g. X, or of the form $f(t_1, ...t_n)$ where f is a function symbol of arity n ($n \geq 0$) applied to a n-tuple of terms. Function symbols of arity 0 are also called constants. In this chapter, predicate and function symbols start with lowercase letters while variables start with uppercase letters. Infix notation may be used for specific predicate symbols (e.g. $X = Y$) and functions symbols (e.g. $-X + Y$). There are two classes of distinguished atoms, built-in

constraints and user-defined constraints. In most CLP languages there is a built-in constraint for syntactic equality over Herbrand terms, `=`, performing unification. The built-in constraint `true`, which is always satisfied, can be seen as an abbreviation for `1=1`. `false` (`1=2`) is the built-in constraint representing inconsistency.

A *CLP clause* is of the form

$$H \texttt{ :- } B_1, \ldots B_n. \quad (n \geq 0)$$

where the head H is an atom but not a built-in constraint, the body $B_1, \ldots B_n$ is a conjunction of literals called *goals*. The empty body ($n = 0$) of a CLP clause may be denoted by the built-in constraint `true`. A *query* is a CLP clause without head.

There are two kinds of CHRs[3]. A *simplification* CHR is of the form

$$H_1, \ldots H_i \texttt{ <=> } G_1, \ldots G_j \mid B_1, \ldots B_k.$$

A *propagation* CHR is of the form

$$H_1, \ldots H_i \texttt{ ==> } G_1, \ldots G_j \mid B_1, \ldots B_k. \quad (i > 0, j \geq 0, k \geq 0)$$

where the multi-head $H_1, \ldots H_i$ is a conjunction of user-defined constraints and the guard $G_1, \ldots G_j$ is a conjunction of literals which neither are, nor depend on, user-defined constraints.

1.3.2 Semantics

Declaratively, CLP programs are interpreted as formulas in first order logic. Extending a CLP language with CHRs preserves its declarative semantics. A CLP+CH program P is a conjunction of universally quantified clauses.

A CLP clause is an implication[4]

$$H \leftarrow B_1 \wedge \ldots B_n.$$

A simplification CHR is a logical equivalence provided the guard is true in the current context

$$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \leftrightarrow B_1 \wedge \ldots B_k).$$

A propagation CHR is an implication provided the guard is true

[3] A third, hybrid kind is described in [B*94].

[4] For simplicity of presentation, we do not use Clark's completion.

$$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \rightarrow B_1 \wedge \ldots B_k).$$

Procedurally, a CHR can fire if its guard allows it. A firing simplification CHR *replaces* the head constraint by the body, a firing propagation CHR *adds* the body to the head constraints. No theorem proving in the general sense is required to reason with the formulas expressed by CHRs.

The *operational semantics* of CLP+CH can be described by a transition system.

A *computation state* is a tuple

$$< Gs, C_U, C_B >,$$

where Gs is a set of goals, C_U and C_B are constraint stores for user-defined and built-in constraints respectively. A *constraint store* is a set of constraints. A set of atoms represents a conjunction of atoms.

The *initial state* consists of a query Gs and empty constraint stores,

$$< Gs, \{\}, \{\} >.$$

A *final state* is either *failed* (due to an inconsistent built-in constraint store represented by the unsatisfiable constraint `false`),

$$< Gs, C_U, \{\texttt{false}\} >,$$

or *successful* (no goals left to solve),

$$< \{\}, C_U, C_B >.$$

The union of the constraint stores in a successful final state is called *conditional answer* for the query Gs, written $answer(Gs)$.

The built-in constraint solver (CS) works on built-in constraints in C_B and Gs, the user-defined CS on user-defined constraints in C_U and Gs using CHRs, and the logic programming engine (LPE) on goals in Gs and C_U using CLP clauses. The following *computation steps* are possible to get from one computation state to the next.

Solve
$< \{C\} \cup Gs, C_U, C_B > \;\longmapsto\; < Gs, C_U, C'_B >$
if $(C \wedge C_B) \leftrightarrow C'_B$

The built-in CS updates the constraint store C_B if a new constraint C was found in Gs. To *update* the constraint store means to produce a

new constraint store C'_B that is logically equivalent to the conjunction of the new constraint and the old constraint store.

We will write $H =_{set} H'$ to denote equality between the sets H and H', i.e. $H = \{A_1, \ldots, A_n\}$ and there is a permutation of H', $\mathrm{perm}(H') = \{B_1, \ldots, B_n\}$, such that $A_i = B_i$ for all $1 \leq i \leq n$.

Introduce
$< \{H\} \cup Gs, C_U, C_B > \longmapsto < Gs, \{H\} \cup C_U, C_B >$
if H is a user-defined constraint

Simplify
$< Gs, H' \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, C_B >$
if $(H$ <=> $G \mid B) \in P$ and $C_B \rightarrow (H =_{set} H') \wedge answer(G)$

Propagate
$< Gs, H' \cup C_U, C_B > \longmapsto < Gs \cup B, H' \cup C_U, C_B >$
if $(H$ ==> $G \mid B) \in P$ and $C_B \rightarrow (H =_{set} H') \wedge answer(G)$

The constraint handling engine (CHE) applies CHRs to user-defined constraints in Gs and C_U whenever all user-defined constraints needed in the multi-head are present and the guard is satisfied. A guard G is *satisfied* if its local execution does not involve user-defined constraints and the result $answer(G)$ is entailed (implied) by the built-in constraint store C_B. Equality is entailed between two terms if they match. To *introduce* a user-defined constraint means to take it from the goal literals Gs and put it into the user-defined constraint store C_U. To *simplify* user-defined constraints H' means to replace them by B if H' matches the head H of a simplification CHR H <=> G | B and the guard G is satisfied. To *propagate from* user-defined constraints H' means to add B to Gs if H' matches the head H of a propagation CHR H ==> G | B and G is satisfied.

Unfold
$< \{H'\} \cup Gs, C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$:- $B) \in P$.

The logic programming engine (LPE) unfolds goals in Gs. To *unfold* an atomic goal H' means to look for a clause $H :- B$ and to replace the H' by $(H = H')$ and B. As there are usually several clauses for a goal, unfolding is nondeterministic and thus a goal can be solved in different ways using different clauses. There can be CLP clauses for user-

defined constraints. Thus they can be unfolded as well. This unfolding is called *(built-in) labeling*. The transition below is somewhat simplified, the details can be found in [Fru92, B*94].

Label
$< Gs, \{H'\} \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$:- $B) \in P$.

Note that any constraint solver written with CHRs will be *incremental* and *concurrent*. By "incremental" we mean that constraints can be added to the constraint store one at a time using the "introduce"-transition. Then CHRs may fire and simplify the user-defined constraint store. The rules can be applied concurrently, even using chaotic iteration (i.e. the same constraint can be simplified by different rules at the same time), because correct CHRs can only replace constraints by equivalent ones or add redundant constraints.

1.3.3 Implementation

The operational semantics are still far from the actual workings of an efficient implementation. At the moment, there exist two implementations, one prototype in LISP [Her93], and one fully developed compiler in a Prolog extension.

The compiler for CHRs is available as a library of ECLiPSe [B*94], ECRC's advanced constraint logic programming platform, utilizing its delay-mechanism and built-in meta-predicates to create, inspect and manipulate delayed goals. All ECLiPSe documentation is available by anonymous ftp from ftp.ecrc.de, directory /pub/eclipse/doc. In such a sequential implementation, the transitions are tried in the textual order given above. To reflect the complexity of a program in the number of CHRs, at most two head constraints are allowed in a rule[5]. This restriction also makes complexity for search of the head constraints of a CHR linear in the number of constraints on average (quadratic in the worst case) by using partitioning and indexing methods. Termination of a propagation CHR is achieved by never firing it a second time with the same pair of head constraints.

The CHRs library includes a debugger and a visual tracer as well as a

[5] Two is the minimal number such that a rule with more head constraints can be rewritten into several number-restricted rules.

full color demo using geometric constraints in a real-life application for wireless telecommunication. 18 constraint solvers currently come with the release, for booleans, finite domains (similar to CHIP [VH89]), also over arbitrary ground terms, reals and pairs, incremental path consistency, temporal reasoning (quantitative and qualitative constraints over time points and intervals [Fru93]), solving linear polynomials over the reals (similar to CLP(R) [J*92]) and rationals, for lists, sets, trees, terms and the solver described in this chapter. The average number of rules in a constraint solver is as low as 24. Typically it took only a few days to produce a reasonable prototype solver, since the usual formalisms to describe a constraint theory, i.e. inference rules, rewrite rules, sequents, first-order axioms, can be expressed as CHRs programs in a straightforward way. Thus one can directly express how constraints simplify and propagate without worrying about implementation details. Starting from this executable specification, the rules then can be refined and adapted to the specifics of the application.

On a wide range of solvers and examples, the run-time penalty for our declarative and high-level approach turned out to be a constant factor in comparison to dedicated built-in solvers (if available). Moreover, the slow-down is often within an order of magnitude. On some examples (e.g. those involving finite domains with the element-constraint), our approach is faster, since we can exactly define the amount of constraint simplification and propagation that is needed. This means that the solver are intentionally made as incomplete as the application requires it. Some solvers (e.g. disjunctive geometric constraints in the phone demo) would be very hard to recast in existing CLP languages.

1.4 Terminological Reasoning

In this chapter we will recall the concept language $\mathcal{ALC}$ [ScSm91] as our basic terminological logic (TL) and show its implementation in CHRs. Section 1.5 will then proceed with some useful extensions of this formalism demonstrating the flexibility of the CHRs approach.

1.4.1 Terminology

A terminology (T-box) consists of a finite, cycle free set of *concept definitions*

C `isa` s,

where C is the newly introduced concept name and s is a concept term constructed from concept names and roles. Inductively, *concept terms* are defined as follows:

1. Every concept name C is a concept term.
2. If s and t are concept terms and R is a role name then the following expressions are concept terms:

   s `and` t (conjunction),

   s `or` t (disjunction),

   `nota` s (complement),

   `every` R `is` s (value restriction),

   `some` R `is` s (exists-in restriction).

Although there is an established notation for terminologies, we use a more verbose syntax so that readers not familiar with the topic can read the expressions easily as sentences in (almost) natural language.

An *interpretation* $\mathcal{I}$ with a set $\mathrm{dom}_\mathcal{I}$ as domain interprets a concept name C as a set $C^\mathcal{I} \subseteq \mathrm{dom}_\mathcal{I}$ and a role name R as a set $R^\mathcal{I} \subseteq \mathrm{dom}_\mathcal{I} \times \mathrm{dom}_\mathcal{I}$. In other words, roles are interpreted as an arbitrary binary relation over $\mathrm{dom}_\mathcal{I}$. It can be lifted to concept terms in a straightforward manner: Conjunction, disjunction, and complement are interpreted as set intersection, set union, and set complement wrt $\mathrm{dom}_\mathcal{I}$, respectively, and

$a \in$ (`every` R `is` s)$^\mathcal{I}$ iff, for all $b \in \mathrm{dom}_\mathcal{I}$, $(a,b) \in R^\mathcal{I}$ implies $b \in s^\mathcal{I}$, and

$a \in$ (`some` R `is s`)$^\mathcal{I}$ iff, there is some $b \in \mathrm{dom}_\mathcal{I}$ such that $(a,b) \in R^\mathcal{I}$, $b \in s^\mathcal{I}$.

An interpretation is a *model* of a terminology T if $C^\mathcal{I} = s^\mathcal{I}$ for all (C `isa` s) $\in T$.

Example: The domain of a configuration application comprises at least devices, interfaces, and configurations. The following concept definitions express that these are disjoint concepts.

```
primitive device.[6]
interface isa nota device.
configuration isa nota (interface or device).
```

Let us assume that a simple device has at least one interface. We introduce a role `connector` which relates devices to interfaces and employ the exists-in restriction.

```
role connector.
simple_device isa device and
some connector is interface.
```

□

1.4.2 Assertions and Reasoning Services

Objects are (Herbrand) constants or variables. Let a, b be objects, R a role, and C a concept term. Then $b : C$ is a *membership assertion* and $(a, b) : R$ is a *role-filler assertion.* An *A-box* is a collection of membership and role-filler assertions.

Example (contd): We introduce instances of devices and interfaces.

```
pc:device.  rs231:interface.  (pc,rs231):connector.
```

□

An *interpretation of an A-box* $\mathcal{A}$ is a model of the underlying terminology that, in addition, maps Herbrand constants to elements of $\mathrm{dom}_{\mathcal{I}}$. For these constants we adopt the unique name assumption. An A-box $\mathcal{A}$ is *consistent* if there is an interpretation $\mathcal{I}$ and a variable assignment $\sigma : \mathit{objects} \to \mathrm{dom}_{\mathcal{I}}$ such that all assertions of $\mathcal{A}$ are satisfied, i.e., $(a\sigma^{\mathcal{I}}, b\sigma^{\mathcal{I}}) \in R^{\mathcal{I}}$ and $b\sigma^{\mathcal{I}} \in C^{\mathcal{I}}$, for all $(a, b) : R$ and $b : C$ in $\mathcal{A}$. An object a is a *member of a concept* C iff for all models $\mathcal{I}$ of the terminology all assignments $\sigma : \mathit{objects} \to \mathrm{dom}_{\mathcal{I}}$ that satisfy $\mathcal{A}$ also satisfy $a : C$. A concept C_1 *subsumes* a concept C_2 iff for all models $\mathcal{I}$ of the terminology $C_1^{\mathcal{I}} \supseteq C_2^{\mathcal{I}}$. Figure 1.1 shows the subsumption graph of the terminology developed in sections 1.4 and 1.5.

The *consistency test* is the central reasoning service for terminological knowledge representation systems with complete inference algorithms. Various other services can be reduced to this test [Hol90]. In particular, the subsumption (and similarly membership) services can be implemented on the basis of the consistency test of A-boxes:

[6] This declaration introduces a concept name that is not defined any further.

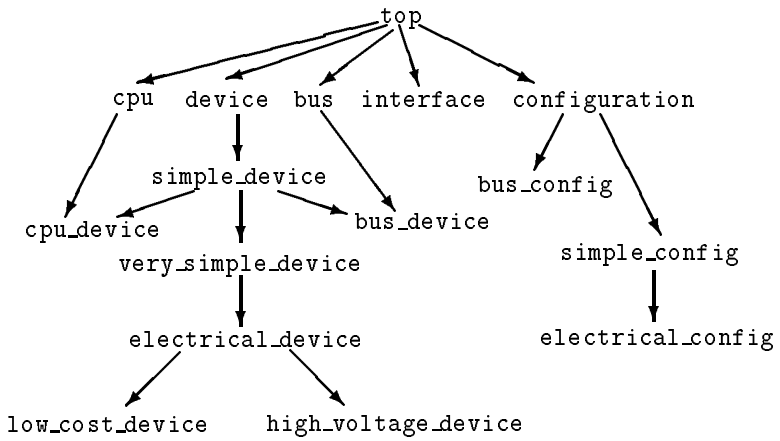

Figure 1.1
Subsumption Graph of the Example Terminology

1. A concept C_1 subsumes a concept C_2 iff an A-box consisting just of the membership assertion $a : C_2$ `and nota` C_1 is inconsistent.
2. An object a is a member of C wrt the A-box $\mathcal{A}$ iff $\{a :$ `nota` $C\} \cup \mathcal{A}$ is inconsistent.

1.4.3 CLP+CH(TL)

Roughly, the consistency test of A-boxes works as follows.

1. Simplify and propagate the assertions in the A-box to make the knowledge more explicit.
2. Look for obvious contradictions ("clashes") such as "`X:C`, `X:nota C`".

The consistency test can be implemented with CHRs by regarding assertions as user-defined constraints. CHRs similar to the ones that follow can be found as the unfolding and completion rules in [ScSm91] and the propagation rules in [BDS93]. However, the former work does not provide an incremental algorithm and the latter does not simplify constraints.

First we treat the complement operator:

```
I:nota nota S  <=>  I:S.
I:nota (S or T)  <=>  I:(nota S and nota T).
I:nota (S and T)  <=>  I:(nota S or nota T).
I:nota every R is S  <=>  I:some R is nota S.
I:nota some R is S  <=>  I:every R is nota S.
```

These simplification CHRs show how the complement operator `nota` can be pushed towards the leaves of a concept term.

The conjunction rule generates two new, smaller assertions:

```
I:S and T  <=>  I:S,I:T.
```

An exists-in restriction generates a new object (i.e. variable) that serves as a "witness" for the restriction:

```
I:some R is S  <=>  (I,J):R, J:S.
```

The time-complexity of executing the above CHRs is linear in the size of the concept term.

A value restriction has to be propagated to all role fillers:

```
I:every R is S, (I,J):R  ==>  J:S.
```

In the implementation this propagation rule will be applied only once per matching pair of membership and role-filler assertions. The time-complexity is quadratic in the number of assertions.

Disjunction is treated lazily by two CLP clauses (introducing a disjunction for built-in labeling):

```
I:S or T :- I:S.
I:S or T :- I:T.
```

These are the two rules where the exponential complexity of consistency tests for terminologies surfaces.

The unfolding rules expand concept names to their definitions:

```
I:C  <=>  (C isa S) | I:S.
I:nota C  <=>  (C isa S) | I:nota S.
```

Since concept definitions do not contain cycles, the above two CHRs clearly terminate.

For $\mathcal{ALC}$ we need only the following single clash rule, one may need more for extensions of the formalism.

```
I:nota S, I:S  <=> false.
```

The above solver simplifies terminological constraints until a normal form is reached. In the normal form, the only constraints are `I:C`, `I:nota C`, `I:S or T`, `I:every R is S`, `(I,J):R`, where `C` is a primitive concept name. The solver is *complete*, i.e. it detects all inconsistencies through the clash rule independent of the order in which constraints are added and CHRs are applied. All CHRs except the clash rule have pairwise disjoint heads (`C` can only stand for concept names). Therefore for completeness we have to show that the clash rule can still be applied even after an inconsistent pair of membership assertions is reduced by other rules. For example, the inconsistent constraints

```
I:nota every R is S, I:every R is S
```

can be simplified by pushing `nota` down in the first constraint

```
I:some R is nota S, I:every R is S  ⟼  (some-rule)
(I,J):R, J:nota S, I:every R is S  ⟼  (every-rule)
(I,J):R, J:nota S, I:every R is S, J:S
```

and the clash rule applies to `J:nota S, J:S` leading to `false`. □

1.5 Extensions

In a number of papers (e.g., [Hol90, ScSm91, BDS93]) the above idea of a tableaux based consistency test as the central reasoning service has been successfully applied to terminological logics with various other language constructs. This flexibility carries over to extensions of our implementation.

1.5.1 Functional Roles

Attributes (also called features) are functional roles, i.e., their interpretation is the graph of a partial function $\mathrm{dom}_{\mathcal{I}} \rightharpoonup \mathrm{dom}_{\mathcal{I}}$. Assuming declarations of attributes of the form `attribute` F, F an attribute name, we just have to extend our implementation by

```
(I,J1):F, (I,J2):F  ==> attribute F | J1=J2.
```

Example (contd): Now we are ready to define a simple configuration which consists of two distinguished simple devices:

```
attribute component_1.
attribute component_2.
simple_config isa configuration and
    some component_1 is simple_device and
    some component_2 is simple_device.
```

Asking the query

```
:- config1:simple_config, (config1,dev1):component_1,
  (config1,dev2):component_2,
```

the membership service can derive that `dev1` and `dev2` are simple devices. The reason is that the attribute-rule constrains the witness for `some component_1 is simple_device` and the second argument of the role `(config1,dev1):component_1` to be equal (analogously for `dev2`).

□

A more local way to specify functionality of roles is provided through concept terms of the form "`at_most_one` R", R a role name.[7] An $a \in \mathrm{dom}_{\mathcal{I}}$ is an element of (`at_most_one` R)$^{\mathcal{I}}$ if there is at most one R-role filler for a. This is implemented through

```
I:at_most_one R, (I,J1):R, (I,J2):R  ==> J1=J2.
```

An object does not belong to `at_most_one` R if, and only if, there are at least two different role fillers:

```
I:nota at_most_one R  <=>
  (I,J1):R, (I,J2):R, different(J1,J2).
```

The constraint `different` is unsatisfiable if its arguments are identical.

```
different(X,X)  <=> false.
```

Example (contd):

```
very_simple_device isa simple_device and
  at_most_one connector.
```

□

1.5.2 Concrete Domains

In [Han92] restricted forms of quantification over predicates of a *concrete domain* $\mathcal{D}$ have been suggested as concept forming operators. Examples

[7] The `at_most_one` construct is a restricted form of the *number restriction* [BS85].

of concrete domains are Allen's temporal interval relations, rational (natural) numbers with comparison operators and linear polynomials over the reals (all of which have been implemented by CHRs). An *admissible* concrete domain has to be closed under complement (since we have to propagate the complement operator `nota`) and has to provide a satisfiability test for conjunctions of predicates. Unlike in [FrHa93], we present here an untyped and thus simpler version of the concrete domains extension for space reasons.

The syntax for the new operators in the extension TL($\mathcal{D}$) of the concept language is as follows:

`every` w_0 `and` ...`and` w_n `is p`
`some` w_0 `and` ...`and` w_n `is p`,

where w_i is of the form "R_{i1} `of` ...`of` R_{ik_i}", R_{ij} are role names, $n > 0$, $k_i \geq 0$, $i = 1, \cdots, n$, and p is an n-ary concrete predicate (constraint) of $\mathcal{D}$. These constructs are inspired by the value restriction and the exists-in restriction. The semantics of the new operators is as follows:

$a \in$ (`every` w_0 `and` ...`and` w_n `is p`)$^{\mathcal{I}}$
iff, for all $b_1, \cdots, b_n \in \mathrm{dom}_{\mathcal{I}}$: $(a, b_i) \in w_i^{\mathcal{I}}$, for $i = 1, \cdots, n$, implies $(b_1, \cdots, b_n) \in p^{\mathcal{I}}$

$a \in$ (`some` w_0 `and` ...`and` w_n `is p`)$^{\mathcal{I}}$
iff, there are $b_1, \cdots, b_n \in \mathrm{dom}_{\mathcal{I}}$ such that $(a, b_i) \in w_i^{\mathcal{I}}$, for $i = 1, \cdots, n$, and $(b_1, \cdots, b_n) \in p^{\mathcal{I}}$

The denotation of $w_i^{\mathcal{I}}$ is defined inductively similar to relational product: $(a, c) \in [R$ `of` $S]^{\mathcal{I}}$ iff there exists a b such that $(a, b) \in S^{\mathcal{I}}$ and $(b, c) \in R^{\mathcal{I}}$.

Note that with R matching the expression "w_0 `and` ...`and` w_n" the complement-rules are also applicable to the new constructs for concrete domains. We just have to introduce a CHR to complement predicates over the concrete domain:

```
(X1,···,Xn):nota P  <=>
   concrete_complement(P,Q) | (X1,···,Xn):Q,
```

where `concrete_complement` is a two place predicate belonging to $\mathcal{I}$ that associates each predicate of the concrete domain with its complement wrt to $\mathrm{dom}_{\mathcal{I}}^n$.

Analogous to the value restriction (resp., the exists-in restriction) CHR we have to collect (resp., generate) objects satisfying the concrete domain predicate. This can be implemented through a fixed, finite set of CHRs that collect (resp., generate) objects according to the roles and attributes occurring in the operators "from left to right" and then restrict the collected (resp., generated) objects with the concrete predicate. To simplify the presentation we give two schemata of propagation (resp., simplification) rules. For each term of the form `every` w_0 `and` ...`and` w_n `is P` that occurs in the knowledge base or in a query we introduce a rule

$$\begin{array}{l}\texttt{X:every } w_0 \texttt{ and } \ldots\texttt{and } w_n \texttt{ is P},\\ (\texttt{X},\texttt{X}_{11}) : \texttt{R}_{11}, \cdots, (\texttt{X}_{1k_1-1}, \texttt{X}_{1k_1}) : \texttt{R}_{1k_1},\\ \cdots\\ (\texttt{X},\texttt{X}_{n1}) : \texttt{R}_{n1}, \cdots, (\texttt{X}_{nk_n-1}, \texttt{X}_{nk_n}) : \texttt{R}_{nk_n} \texttt{ ==> } (\texttt{X}_{1k_1}, \cdots, \texttt{X}_{nk_n})\texttt{:P}.\end{array}$$

Analogously, for a term `some` w_0 `and` ...`and` w_n `is P` we introduce a rule

$$\begin{array}{l}\texttt{X:some } w_0 \texttt{ and } \ldots \texttt{ and } w_n \texttt{ is P } \texttt{ <=>}\\ \qquad\qquad (\texttt{X}_{1k_1}, \cdots, \texttt{X}_{nk_n})\texttt{:P},\\ \qquad\qquad (\texttt{X},\texttt{X}_{11}) : \texttt{R}_{11}, \cdots, (\texttt{X}_{1k_1-1}, \texttt{X}_{1k_1}) : \texttt{R}_{1k_1},\\ \qquad\qquad \cdots\\ \qquad\qquad (\texttt{X},\texttt{X}_{n1}) : \texttt{R}_{n1}, \cdots, (\texttt{X}_{nk_n-1}, \texttt{X}_{n,k_n}) : \texttt{R}_{nk_n}.\end{array}$$

For example, an expression `every f of r1 and r2 is p` leads to a rule

$$\begin{array}{l}\texttt{X:every f of r1 and r2 is P, (X,X}_{11}\texttt{):r1, (X}_{11}\texttt{,X}_{12}\texttt{):f,}\\ \texttt{(X,X}_{21}\texttt{):r2 ==> (X}_{12}\texttt{,X}_{21}\texttt{):P}.\end{array}$$

As a simple example of a concrete domain we take inequalities over rational numbers. The reasoning in the concrete domain itself is implemented through the following rules which find all contradictions (but do not perform all possible simplifications).

```
X > Y <=>  Y < X.
X >= Y <=>  Y =< X.
X =< X <=>   true.
X < X <=>   false.
X =< Y <=>  number_>(X,Y) |  false.
```

```
X < Y <=>  number_>=(X,Y) |  false.
X < Y, Y < Z ==>   X < Z.
X =< Y, Y < Z ==>   X < Z.
X < Y, Y =< Z ==>   X < Z.
X =< Y, Y =< Z ==>   X =< Z.
```

The guard `number_>(X,Y)` (resp., `number_>=(X,Y)`) is true if `X` and `Y` are bound to numbers x and y and $x > y$ (resp., $x \geq y$). The predicate `concrete_complement` associating concrete predicates with their complements is defined by the following clauses:

```
concrete_complement(<,>=).
concrete_complement(>=,<).
concrete_complement(=<,>).
concrete_complement(>,=<).
```

The CHRs for the new operator generates atoms of the form $(x, y) : \langle comparison\ operator \rangle$ and $x : (\langle comparison\ operator \rangle \langle number \rangle)$[8]. These atoms have to be translated to the infix syntax of the concrete domain:

```
X:(< N) <=>   X < N.          (X,Y):< <=>   X < Y.
X:(> N) <=>   X > N.          (X,Y):> <=>   X > Y.
X:(>= N) <=>   X >= N.        (X,Y):=< <=>   X =< Y.
X:(=< N) <=>   X =< N.        (X,Y):>= <=>   X >= Y.
```

Example (contd): Now we can associate price and voltage with a device and require that in an electrical configuration the voltages have to be compatible.

```
attribute price.
attribute voltage.
electrical_device isa very_simple_device and
   some voltage is > 0 and some price is > 1 .
low_cost_device isa electrical_device and
   every price is < 200.
high_voltage_device isa electrical_device and
   every voltage is > 15.
electrical_config isa simple_configuration and
```

[8] The latter enables the use of numbers in concept terms.

```
   every component_1 is electrical_device and
   every component_2 is electrical_device and
   every voltage of component_1 and
      voltage of component_2 is >=.
```
□

The new operator can also be used to specify upper bounds. This is illustrated by a configuration where several CPUs are plugged onto a bus with the side condition that the maximal frequency of the CPUs must be less than the frequency of the bus.

```
attribute frequency.
primitive bus.
bus_device isa simple_device and bus and
  some frequency is > 0 .
primitive cpu.
cpu_device isa simple_device and cpu and
  some frequency is > 0 .
role main_device.
role sub_device.
bus_config isa configuration and
   some main_device is bus_device and
   every component is cpu_device and
   every frequency of main_device and
   frequency of sub_device is > .
```
□

1.5.3 CLP+CH(TL($\mathcal{D}$))

If we apply the CLP scheme of Höhfeld und Smolka [HS90] in a straightforward manner to A-boxes of TL($\mathcal{D}$), we obtain a CLP language with three representation and reasoning layers [AH93].

Example (contd): The following CLP clauses specify the catalog of devices and describe possible configurations that are based on this catalog.

```
catalog(dev1) :- dev1:electrical_device,
  (dev1,10):voltage, (dev1,100):price.
catalog(dev2) :- dev2:electrical_device,
  (dev2,20):voltage, (dev2,1000):price.
possible_config(C) :-
```

```
catalog(D1), (C,D1):component_1,
catalog(D2), (C,D2):component_2.
```

The following queries enumerate possible configurations satisfying the requirements.

```
:- possible_config(C).
:- possible_config(C), C:electrical_config.
:- possible_config(C), C:electrical_config,
   (C,D1):component_1, D1:low_cost_device,
   (C,D2):component_2, D2:high_voltage_device.
```

The first query enumerates all possible electrical configurations comprising two devices based on the catalog, i.e., configurations consisting of two devices `dev1`, two `dev2`, or `dev1` and `dev2`. The second query allows for all configurations involving `dev1` and `dev2`, except the one where `dev1` is component one and `dev2` is component two. Finally, the third query has no solution, because the catalog lists only one low-cost device and there is no high-voltage device with a compatible voltage. □

1.6 Conclusions

Constraint handling rules (CHRs) are a language extension for implementing user-defined constraints. Rapid prototyping of novel applications for constraint techniques is encouraged by the high level of abstraction and declarative nature of CHRs.

In this chapter we investigated terminological reasoning as constraint solving with CHRs. The terminological constraint system is related to other term domains like [Sow91, Smo92, APG93], which we currently implement in a similar way. The overall language has some similarities with LOGIN [AiNa86]. Flexibility was illustrated by extending the formalism and its implementation with attributes, a special quantifier and concrete domains. Applicability was illustrated by sketching a generic, hybrid knowledge base for solving configuration problems.

Bibliography

[AiNa86] H. Ait-Kaci and R. Nasr, Login: A Logic Programming Language with Built-In Inheritance, Journal of Logic Programming, 3:185-215, 1986.

[APG93] H. Ait-Kaci, A. Podelski and S. C. Goldstein, Order-Sorted Feature Theory Unification, DEC PRL Research Report 32, May 1993, DEC Paris Research Laboratory, France.

[AH93] A. Abecker and P. Hanschke. TaxLog: A flexible architecture for logic programming with structured types and constraints. In Notes of the Workshop on Constraint Processing held in conjunction with CSAM'93, Petersburg, 1993.

[B*94] ECLiPSe 3.4 Extensions User Manual, ECRC Munich, Germany, July 1994.

[Baj93] R. Bajcsy (ed.), Terminological Logic I-IV, Sessions in Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambery, France, Morgan Kaufmann, 1993, pp. 662-717.

[BaHa91] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In Proceedings of the 12^{th} International Joint Conference on Artificial Intelligence, 1991.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. Cognitive Science, 9(2):171–216, 1985.

[BDS93] M. Buchheit, F. M. Donini and A. Schaerf. Decidable Reasoning in Terminological Knowledge Representation Systems. Journal of Artificial Intelligence Research, 1(1993):109-138.

[F*92] T. Frühwirth et al, Constraint Logic Programming - An Informal Introduction, Logic Programming Summer School, Zurich, Switzerland, September 1992, Springer LNCS 636, 1992 (also Technical Report ECRC-93-05, ECRC Munich, Germany, February 1993).

[FrHa93] Terminological Reasoning with Constraint Handling Rules, Workshop on Principles and Practice of Constraint Programming, Newport, Rhode Island, USA, April 1993 (revised version as Technical Report ECRC-94-06, ECRC Munich, Germany, February 1994, available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-06.ps.Z).

[Fru92] T. Frühwirth, Constraint Simplification Rules (former name for CHRs), Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992 (revised version of Internal Report ECRC-LP-63, October 1991), available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-92-18.ps.Z.

[Fru93] T. Frühwirth, Temporal Reasoning with Constraint Handling Rules, Technical Report ECRC-94-05, ECRC Munich, Germany, February 1994. (first published as CORE-93-08, January 1993), available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-05.ps.Z.

[HaJa90] S. Haridi and S. Janson, Kernel Andorra Prolog and its Computation Model, Seventh International Conference on Logic Programming, MIT Press, 1990, pp. 31-46.

[Han92] P. Hanschke. Specifying role interaction in concept languages. In Third International Conference on Principles of Knowledge Representation and Reasoning (KR '92), October 1992.

[Han93] P. Hanschke. A declarative integration of terminological, constraint-based, data-driven, and goal-directed reasoning. PhD Thesis at University of Kaiserslautern, Germany, July 1993.

[Her93] Eine homogene Implementierungsebene fuer einen hybriden Wissensrepraesentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, April 1993.

[Hol90] B. Hollunder. Hybrid inferences in KL-ONE-based knowledge representation systems. In 14th German Workshop on Artificial Intelligence (GWAI-90), volume 251, pages 38–47. Springer, 1990.

[HS90] M. Höhfeld and G. Smolka, Definite Relations over Constraint Languages. LILOG Report 53, IBM Deutschland, West Germany, October 1988.

[J*92] J. Jaffar et al., The CLP(R) Language and System, ACM Transactions on programming Languages and Systems, Vol.14:3, July 1992, pp. 339-395.

[JaLa87] J. Jaffar and J.-L. Lassez, Constraint Logic Programming, ACM 14th POPL 87, Munich, Germany, January 1987, pp. 111-119.

[JaMa] J. Jaffar and M. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming, 1994:19,20:503-581.

[Sar93] V. A. Saraswat, Concurrent Constraint Programming, MIT Press, Cambridge, 1993.

[ScSm91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. In Journal of Artificial Intelligence, 47, 1991.

[Sha89] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[Smo91] G. Smolka, Residuation and Guarded Rules for Constraint Logic Programming, Digital Equipment Paris Research Laboratory Research Report, France, June 1991.

[Smo92] G. Smolka, Feature Constraint Logics for Unification Grammars, Journal Of Logic Programming, 12:51-87 (1992).

[Sow91] J. Sowa (ed.), Principles of Semantic Networks, Morgan Kaufmann, 1991.

[VH89] P. Van Hentenryck, Constraint satisfaction in Logic Programming, MIT Press, Cambridge, 1989.

[VH91] P. Van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.