International Workshop on Description Logics, Rome, Italy; June 1995.

Implementing and Testing Expressive Description Logics: a Preliminary Report

Paolo Bresciani, Enrico Franconi and Sergio Tessaris
Knowledge Representation and Reasoning group
IRST, I-38050 Povo TN, Italy
{bresciani|franconi|tessaris}@irst.itc.it

Abstract

The aim of the CRACK project is the research and the development of a knowledge representation system based on description logics. The CRACK language differentiates itself from other knowledge representation systems for its high expressivity and its provably sound and complete reasoning procedures. With respect to other systems available in the research community CRACK is more expressive, it is expandable to new constructs, it treats the conceptual and individual levels in a homogeneous way, it is modular, it is comparably fast. However, CRACK algorithms do not work in polynomial space, i.e. in the (arguably rare in practice) worst cases they may require exponential memory. The performance of the system has been tested against several different classes of random knowledge bases, characterized by an order parameter generating phase transitions in the satisfiability probability space.

1 Introduction

In this paper we present the knowledge representation system CRACK, an implementation of an expressive description logic. In the full paper [Bresciani *et al.*,1995] more details on CRACK could be found. The greatest motivation for the CRACK project is the possibility of having a modular system, where the expressivity of the language could be extended, without re-implementing the control part of the system. Thus, experimentation with many operators and, if a very expressive language is implemented, with different sub-languages can be done.

Algorithms in CRACK are based on the tableaux calculus technique, according to the many recent theoretical works in description logics. However, differently from other implementations, rules of tableaux are directly implemented, allowing for a high grade of modularity by just adding/removing rules. As a further consequence of this choice, it turns out that the conceptual and the individual levels are treated in a uniform manner. The architecture of the system has been organized in such a way that different possible extensions of the language – involving many different condition testing for the rules, different basic data management, or different clash checking – can be easily added on top of the core system.

The final part of the paper is dedicated to the topic of testing such a system, by considering the problem of generating random *average* knowledge bases. Stimulated by the studies in the literature of the hardness of the satisfiability problems for some classes of random propositional formulae, we have tried to explore the probability space of satisfiability in the case of description logics. Such tests are just at the beginning, but we believe that it is a good direction to explore.

2 The $\mathcal{ALCRIFO}$ Description Logic

In the following, we will consider only $\mathcal{ALCRIFO}$, a significative fragment of the language currently implemented in CRACK, because it embeds the main peculiarities of the system. We will not go into details of the language, since it is just a collection of standard constructors in description logics. $\mathcal{ALCRIFO}$ extends the propositionally complete concept language $\mathcal{ALC}$ [Schmidt-Schauß and Smolka,1991] with role conjunction ($\mathcal{R}$)[Hollunder *et al.*,1990], inverse roles ($\mathcal{I}$) [Donini *et al.*,1991], feature selection, agreement and disagreement ($\mathcal{F}$) [Hollunder and Nutt,1990], enumerated types ($\mathcal{O}$) [Schaerf,1994]. Section 4 describes the full CRACK language and some of its possible extensions. Figure 1 defines syntax and semantics of $\mathcal{ALCRIFO}$.

Knowledge base satisfiability is the basic reasoning task, since all the relevant reasoning tasks – like concept satisfiability, concept subsumption, instance checking, retrieval – are reducible to it. Checking for KB satisfiability is deciding whether there is at least one model for a knowledge base Σ, made out of TBox axioms and ABox statements.

The tableaux-based calculus to decide the satisfiability of $\mathcal{ALCRIFO}$ knowledge bases operates on constraints. A constraint is an expression of the type $x : C$, xRy, xpy and $x \neq y$, where x, y are objects, i.e. either individuals or variables belonging to a predefined set of variable symbols. Starting from a system S associated to a knowledge base – i.e. obtained by directly translating into constraints every ABox statement – the *propagation rules* are applied, until a contradiction (a *clash*) is generated or a model of Σ is explicitly obtained: the propagation rules preserve satisfiability. A clash is a system having one of the forms: $\{x : \bot\}$; $\{x : A,\ x : \neg A\}$; $\{x \neq x\}$;

$C, D \rightarrow$	$A \mid$	
	$\top \mid$	$\Delta^{\mathcal{I}}$
	$\bot \mid$	$\emptyset$
	$\neg C \mid$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
	$C \sqcap D \mid$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
	$C \sqcup D \mid$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
	$\forall R.\, C \mid$	$\{i \in \Delta^{\mathcal{I}} \mid \forall j.\, R^{\mathcal{I}}(i,j) \rightarrow C^{\mathcal{I}}(j)\}$
	$\exists R.\, C \mid$	$\{i \in \Delta^{\mathcal{I}} \mid \exists j.\, R^{\mathcal{I}}(i,j) \wedge C^{\mathcal{I}}(j)\}$
	$p\uparrow \mid$	$\Delta^{\mathcal{I}} \setminus \mathrm{dom}\, p^{\mathcal{I}}$
	$p : C \mid$	$\{i \in \mathrm{dom}\, p^{\mathcal{I}} \mid C^{\mathcal{I}}(p^{\mathcal{I}}(i))\}$
	$p \stackrel{\downarrow}{=} q \mid$	$\{i \in \mathrm{dom}\, p^{\mathcal{I}} \cap \mathrm{dom}\, q^{\mathcal{I}} \mid p^{\mathcal{I}}(i) = q^{\mathcal{I}}(i)\}$
	$p \stackrel{\downarrow}{\neq} q \mid$	$\{i \in \mathrm{dom}\, p^{\mathcal{I}} \cap \mathrm{dom}\, q^{\mathcal{I}} \mid p^{\mathcal{I}}(i) \neq q^{\mathcal{I}}(i)\}$
	$\{\ a_1 \ldots a_n\ \}$	$\{a_1^{\mathcal{I}} \ldots a_n^{\mathcal{I}}\}$
$R, Q \rightarrow$	$P \mid$	
	$R^{-1} \mid$	$\{(i,j) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid R^{\mathcal{I}}(j,i)\}$
	$R \sqcap Q$	$R^{\mathcal{I}} \cap Q^{\mathcal{I}}$
$p, q \rightarrow$	$f \mid$	
	$p \circ q$	$p^{\mathcal{I}} \circ q^{\mathcal{I}}$

Figure 1: Syntax and semantics for the $\mathcal{ALCRIFO}$ description logic.

$\{x : f\uparrow,\ xfy\}$; $\{xfa,\ xfb\}$ if a and b are individuals; $\{a : \{\ a_1 \ldots a_n\ \}\}$ with $a \neq a_i$ for all i.

Figure 2 lists a complete set of propagation rules for $\mathcal{ALCRIFO}$; they are a simple variant of the standard rules found in the literature. A role $R = R_1 \sqcap \ldots \sqcap R_k \sqcap ({R_{k+1}}^{-1} \sqcap \ldots \sqcap {R_n}^{-1})$ is written as the pair $\langle R^+, R^- \rangle$ where $R^+ = \{R_1, \ldots, R_k\}$ and $R^- = \{R_{k+1}, \ldots, R_n\}$. We say that xRy *holds in* a constraint system S if xQy is in S and Q specializes R – i.e. $R^+ \subseteq Q^+$ and $R^- \subseteq Q^-$. For the calculus, we consider only concepts in a sort of normal form [Bresciani *et al.*,1995]. An arbitrary $\mathcal{ALCRIFO}$ concept can be transformed in polynomial time into an equivalent concept in the normal form.

Propagation rules are either deterministic – they yield a uniquely determined constraint system – or nondeterministic (e.g. $\rightarrow_{\sqcup}$) – yielding several possible constraint systems. A constraint system is said to be *complete* if no propagation rule is applicable to it. Because of the presence of nondeterministic rules, several complete systems can be derived from the starting one. A constraint system is satisfiable if and only if there exist a clash free complete constraint system which can be derived from it by applying the propagation rules.

3 Implementing Tableaux Calculus

In this section we will show a general architecture for implementing reasoning systems based on tableaux calculus; in our case, it will be a procedure for deciding satisfiability of $\mathcal{ALCRIFO}$ knowledge bases.

The main idea is to implement directly the rules of tableaux calculus, as opposed to the currently implemented tableaux-based description logics relying on a *functional algorithm*, like e.g. KRIS [Baader and Hollunder,1991]. A functional algorithm exploits the property of independency between *traces* of a satisfiability proof. It assumes that a completed system can be partitioned into traces, where the computation can be performed independently – i.e. a clash can be generated only by constraints belonging to the same trace. The functional algorithm for $\mathcal{ALCR}$ can be sketched as follows [Hollunder *et al.*,1990]:

$S \rightarrow_{\sqcap}$ $\{x : C_1,\ x : C_2\} \cup S$
if $x : C_1 \sqcap C_2$ in S,
and both $x : C_1$ and $x : C_2$ not in S

$S \rightarrow_{\sqcup}$ $\{x : D\} \cup S$
if $x : C_1 \sqcup C_2$ in S,
and neither $x : C_1$ nor $x : C_2$ in S,
and $D = C_1$ or $D = C_2$

$S \rightarrow_{\exists}$ $\{xRy,\ y : C\} \cup S$
if $x : \exists R.C$ in S,
and there is no z s.t. both xRz and $z : C$ in S,
and y is a new variable

$S \rightarrow_{\forall}$ $\{y : C\} \cup S$
if $x : \forall R.C$ in S, and xRy holds in S,
and $y : C$ not in S

$S \rightarrow_{-1}$ $\{yPx\} \cup S$
if $xP^{-1}y$ in S,
and yPx not in S

$S \rightarrow_{+1}$ $\{yP^{-1}x\} \cup S$
if xPy in S,
and $yP^{-1}x$ not in S

$S \rightarrow_{:}$ $\{xpy,\ y : C\} \cup S$
if $x : (p : C)$ in S,
and there is no z s.t. both xpz and $z : C$ in S,
and y is a new variable

$S \rightarrow_{\stackrel{\downarrow}{=}}$ $\{xpy,\ xqy\} \cup S$
if $x : p \stackrel{\downarrow}{=} q$ in S,
and there is no z s.t. both xpz and xqz in S,
and y is a new variable

$S \rightarrow_{\stackrel{\downarrow}{\neq}}$ $\{xpy,\ xqz,\ y \neq z\} \cup S$
if $x : p \stackrel{\downarrow}{\neq} q$ in S,
and there are no y', z' s.t. xpy', xqz' and $y' \neq z'$ in S,
and y, z are new variables

$S \rightarrow_{\circ}$ $\{xpz,\ zqy\} \cup S$
if $x(p \circ q)y$ in S,
and there is no w s.t. both xpw and wqy in S,
and z is a new variable

$S \rightarrow_{funct}$ $[y/z]S$
if xfy and xfz in S,
and y is not an object,
and y is not equal to z

$S \rightarrow_{\{\}}$ $[x/a_i]S$
if $x : \{\ a_1 \ldots a_n\ \}$ in S,
and x is not an object

$S \rightarrow_{\neg\{\}}$ $\{x \neq a_1 \ldots x \neq a_n\} \cup S$
if $x : \neg\{\ a_1 \ldots a_n\ \}$ in S

Figure 2: Propagation rules for the $\mathcal{ALCRIFO}$ description logic.

$sat(S) =$ **if** S includes a clash
  then false
  elseif $C \sqcap D \in S$ **and** $C \notin S$ **or** $D \notin S$
    then $sat(S \cup \{C, D\})$
  elseif $C \sqcup D \in S$ **and** $C \notin S$ **and** $D \notin S$
    then $sat(S \cup \{C\})$ **or** $sat(S \cup \{D\})$
  else forall $\exists R.\, C \in S$
      $sat(\{C\} \cup \{D \mid \forall Q.\, D \in S \wedge R \sqsubseteq Q\})$

Unfortunately, it seems that expressive languages do not have this nice property. For example, inverse roles and one-of introduce interactions between traces. The unsatisfiability of the concept:

$$\exists\texttt{CHILD}.(\texttt{Man} \sqcap \{\ \texttt{peter}\ \}) \sqcap \exists\texttt{CHILD}.(\neg\texttt{Man} \sqcap \{\ \texttt{peter}\ \})$$

can not be detected with a trace-based algorithm. The two traces generated by the two existential quantifications on `CHILD` are independently satisfiable, but are globally unsatisfiable, since both existential variables should be co-referenced to the individual `peter`.

Thus, our general satisfiability algorithm does not assume any independency between traces, since it has to be used for very expressive description logics. This allows for deductions like that in the above example. However, there is a price to pay for this choice: sub-languages where the complexity of satisfiability is PSPACE-complete are not handled properly with such a non-trace algorithm. In fact, the plain non-deterministic application of the propagation rules may generate an exponential number of variables. For example, the completed system generated by the concept (of dimension n) [Donini *et al.*,1992a]:

$$\exists R.C_1 \sqcap \exists R.C_2 \sqcap \forall R.\boxed{(\exists R.C_1 \sqcap \exists R.C_2 \sqcap \forall R.\boxed{(\ldots \boxed{\ldots_n}\ )_2}\ )_1}$$

has an exponential number of variables with respect to n, as shown below:

$$x : \exists R.C_1 \sqcap \exists R.C_2 \sqcap \forall R.(\exists R.C_1 \sqcap \exists R.C_2 \sqcap \forall R.(\ldots))$$

$$\boxed{\begin{array}{l} xRx_1,\ x_1 : C_1 \\ \quad x_1 : \exists R.C_1 \sqcap \exists R.C_2 \sqcap \forall R.(\ldots) \\ \qquad \boxed{\begin{array}{l} \ldots \\ \ldots \end{array}} \\ xRx_2,\ x_2 : C_2 \\ \quad x_2 : \exists R.C_1 \sqcap \exists R.C_2 \sqcap \forall R.(\ldots) \\ \qquad \boxed{\begin{array}{l} \ldots \\ \ldots \end{array}} \end{array}}$$

We believe that such worst cases are unlikely to happen in real cases, just as exponential in time worst cases do not happen in real and average knowledge bases; for a preliminary study on the meaning of *average knowledge base* refer to the section 5 of the paper.

Another peculiarity of our approach is that the computation is driven by the constraints, and not by the non-deterministic choice of applicable rules. At the beginning of a satisfiability proof, constraints are inserted in a queue and processed in a sequential manner. A constraint triggers the application of only one rule, and thereafter it could be either inserted in the database of permanent constraints – if the constraint itself could be necessary in the future for checking the application of some other rule triggered by some other constraint or for detecting a clash – or otherwise discarded. The application of a rule may generate other new constraints, which are pushed in the queue and further elaborated. Figure 3 presents the CRACK architecture.

This architecture may deal with very expressive languages, and it can be extended to handle new propagation rules without having to reconsider the control mechanism of the system. Moreover, the treatment of new types of clashes or new types of operations on constraints can be easily added to the system by changing

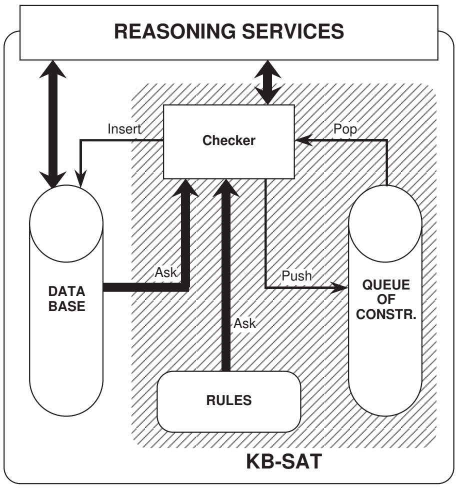

Figure 3: The CRACK architecture.

only the affected modules. In the following, the formal operational semantics describing the behavior of the system for handling $\mathcal{ALCRIFO}$ will be introduced, the data structures will be analyzed, and an abstract overview to the satisfiability algorithm itself will be given.

3.1 Operational Semantics

The operational semantics is defined through a formal system that allows to prove assertions of the form:

$$\langle S, DB \rangle \vdash v$$

where S is a sequence of constraints, DB is a database and v is either a database or the symbol **FAIL**. An assertion of this kind states that an "evaluation" of constraints in S over the database DB produces the result v. If v is **FAIL** the constraint set with respect to the database is unsatisfiable, otherwise the result is a new database *DB'* from which a model may be derivable.

A database is a pair $\langle \Sigma, \rho \rangle$, where Σ is a set of constraints and ρ is a rename function mapping objects into objects. Due to the unique name assumption, the rename function restricted to individuals must be the identity. The rename function may change as new constraints are added to the constraint system by the propagation rules; for this reason, we introduce the notation $\rho[z, r]$ – where ρ is a rename function – denoting a new rename function ρ' equal to ρ except that $\rho'(r) = z$. We also impose some restrictions on the structure of role and feature constraints in the database. For every pair of related objects there must be at most one role constraint – conjoining all the asserted role constraints – and if a role binds a pair of objects its inverse should bind the inverted pair of objects. A similar restriction is applied also for feature constraints (and their inverses). According to this, adding role or feature constraints involves some extra operations on the set; for sake of simplicity, will will ignore this in the following.

There are five types of constraints: instance, relation, attribute, inequality and equality constraints

($x : C$, $x\,R\,y$, $x\,f\,y$, $x \neq y$, $x = y$). Only *persistent* constraints are stored in a database. They are those which may be used more than once during a satisfiability proof. For example, a universal constraint is processed any time a new filler for the role (i.e. a new role constraint) is added to the constraint system; on the contrary, an existential constraint is processed only once. It can be easily verified that persistent constraints are only of the kind: $x : A$, $x : \neg A$, $x : \forall R.C$, $x(R^{+} \sqcap R^{-})y$, xfy, $xf^{-1}y$, $x \neq y$, $x : f\uparrow$, where R^{+}, R^{-} are conjunctions of roles and inverse roles respectively.

The formal system of operational semantics is given in form of pre-conditions which should be verified in order to prove the truth value of every assertion. Operationally, the control of the "executor" is fixed by the sequence of pre-conditions to check. Let's see, for example, the conditional assertion about the `and` concept:

$$\frac{\langle \pi(\rho x : D)(\pi(\rho x : C)S), \langle \Sigma, \rho \rangle \rangle \vdash v}{\langle <(x : C \sqcap D)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

This formula states that the result of the evaluation on a sequence, whose head is the conjunction of two concepts, over a database ($\langle \Sigma, \rho \rangle$) is the same as the evaluation on a sequence obtained by adding two new constraints ($\rho x : C$ and $\rho x : D$) to the rest of the original sequence (S) over the same database. Please note the *strategy* two-place function π: it maps a constraint and a sequence of constraints into a new sequence of constraints. Intuitively, the strategy function inserts a new constraint into a sequence determining a "good" place for it. The strategy function does not modify a sequence when trying to add an already present constraint.

In the following the set of the conditional assertions for $\mathcal{ALC}$ is listed – the complete set for $\mathcal{ALCRIFO}$ can be found in [Bresciani *et al.*,1995]. The domains of the operational semantics are: objects names ($\mathcal{O}$), including individuals names ($\mathcal{J}$) and variables names ($\mathcal{V}$); Concept Names ($\mathcal{C}_{\mathrm{nam}}$); Role Names ($\mathcal{R}_{\mathrm{nam}}$); Feature Names ($\mathcal{F}_{\mathrm{nam}}$); Concept Expressions ($\mathcal{C}$); Role Expressions ($\mathcal{R}$); Feature Expressions ($\mathcal{F}$). The terminology function $\mathcal{T}$ maps a concept name into its definition; if a concept name does not have a definition, the terminology function returns the name itself. The normal form function Nf maps a concept expression into its normal form.

Termination on Success

- $$\frac{}{\langle \Box, \langle \Sigma, \rho \rangle \rangle \vdash \langle \Sigma, \rho \rangle}$$

Bottom Concept

- $$\frac{}{\langle <(x : \bot)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash \mathbf{FAIL}}$$

Atomic concept

- $$\frac{\neg A(\rho x) \in \Sigma}{\langle <(x : A)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash \mathbf{FAIL}}$$

- $$\frac{\begin{array}{l} \neg A(\rho x) \notin \Sigma \\ \mathcal{T}(A) = A \\ \Sigma' = \Sigma \cup \{A(\rho x)\} \\ \langle S, \langle \Sigma', \rho \rangle \rangle \vdash v \end{array}}{\langle <(x : A)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

- $$\frac{\begin{array}{l} \neg A(\rho x) \notin \Sigma \\ \mathrm{Nf}(\mathcal{T}(A)) = C \qquad C \neq A \\ \Sigma' = \Sigma \cup \{A(\rho x)\} \\ \langle \pi(\rho x : C)S, \langle \Sigma', \rho \rangle \rangle \vdash v \end{array}}{\langle <(x : A)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

Negated Atomic Concept

- $$\frac{A(\rho x) \in \Sigma}{\langle <(x : \neg A)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash \mathbf{FAIL}}$$

- $$\frac{\begin{array}{l} A(\rho x) \notin \Sigma \\ \mathcal{T}(A) = A \\ \Sigma' = \Sigma \cup \{\neg A(\rho x)\} \\ \langle S, \langle \Sigma', \rho \rangle \rangle \vdash v \end{array}}{\langle <(x : \neg A)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

- $$\frac{\begin{array}{l} A(\rho x) \notin \Sigma \\ \mathrm{Nf}(\neg\mathcal{T}(A)) = C \qquad C \neq \neg A \\ \Sigma' = \Sigma \cup \{\neg A(\rho x)\} \\ \langle \pi(\rho x : C)S, \langle \Sigma', \rho \rangle \rangle \vdash v \end{array}}{\langle <(x : \neg A)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

And concept

- $$\frac{\langle \pi(\rho x : D)(\pi(\rho x : C)S), \langle \Sigma, \rho \rangle \rangle \vdash v}{\langle <(x : C \sqcap D)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

Or concept

- $$\frac{\begin{array}{l} \langle \pi(\rho x : C)S, \langle \Sigma, \rho \rangle \rangle \vdash v \\ v \neq \mathbf{FAIL} \end{array}}{\langle <(x : C \sqcup D)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

- $$\frac{\begin{array}{l} \langle \pi(\rho x : C)S, \langle \Sigma, \rho \rangle \rangle \vdash \mathbf{FAIL} \\ \langle \pi(\rho x : D)S, \langle \Sigma, \rho \rangle \rangle \vdash v \end{array}}{\langle <(x : C \sqcup D)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

Some concept

- $$\frac{\begin{array}{l} y \text{ new variable} \\ \rho' = \rho[y, y] \\ \langle \pi(y : C)(\pi(\rho x\,R\,y)S), \langle \Sigma, \rho' \rangle \rangle \vdash v \end{array}}{\langle <(x : \exists R.C)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

Universal concept

- $$\frac{\begin{array}{l} S' = \min \mu \supseteq S \ s.t. \\ \quad \forall Q(\rho x, y) \in \Sigma.\, Q \sqsubseteq R \Rightarrow \mu = \pi(y : C)\mu \\ \langle S', \langle \Sigma, \rho \rangle \rangle \vdash v \end{array}}{\langle <(x : \forall R.C)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

Role

- $$\frac{\begin{array}{l} S' = \min \mu \supseteq S \ s.t. \\ \quad (\forall(\forall Q.C)(\rho x) \in \Sigma.\, R \sqsubseteq Q \Rightarrow \mu = \pi(\rho y : C)\mu) \wedge \\ \quad (\forall(\forall Q.C)(\rho y) \in \Sigma.\, R^{-1} \sqsubseteq Q \Rightarrow \mu = \pi(\rho x : C)\mu) \\ \langle S', \langle \Sigma, \rho \rangle \rangle \vdash v \end{array}}{\langle <(x\,R\,y)\,.\,S>, \langle \Sigma, \rho \rangle \rangle \vdash v}$$

3.2 The Data Structures

A special data structure is used to implement the database, in order to allow for efficient computation of the applicability conditions of the rules, for a fast detection of clashes, and for an effective insertion of new constraints in the database itself. Moreover, the data structure is capable to cope with the non-determinism of the database updating, and with the problem of variable renaming.

The data structure has been organized in a way that new rules can be added without reimplementing the database structure. Thus, the expressivity of the language can be extended by just adding new rules, being not necessary to change the whole system architecture or the data structures.

The *Constraint System* is the structure for the database, holding descriptions of persistent constraints. A constraint system is represented by a stack of *frames*. Each frame is a backtrack point for a non-deterministic rule: when a non-deterministic rule is applied, a new empty frame is pushed on the top of the stack. The top frame is called *active* frame. The active frame is the location where the information is stored at a certain point of the non-deterministic computation.

Slot name	Content	Indexed by
name	A	*none*
preceding-entry	$\longmapsto$ entry	*none*
is-variable	{T|F}	*none*
Constraints:		
isa	$\langle A_1, \ldots\rangle$	concept name
isnt	$\langle A_1, \ldots\rangle$	concept name
universals	$\langle [R_1 \,.\, C_1], \ldots\rangle$	*none*
roles	$\langle$ [x_1 . direct $\longmapsto$ role-struct; inverse $\longmapsto$ role-struct] , ...$\rangle$	related object
features	$\langle$ [x_1 . $\longmapsto$ feature-struct] , ...$\rangle$	related object
inv-features	$\langle$ [x_1 . $\longmapsto$ feature-struct] , ...$\rangle$	related object
not-equal	$\langle x_1, \ldots\rangle$	object name
undefinedness	$\langle f_1, \ldots\rangle$	feature name

Figure 4: The data structure for an entry.

Backtracking pops the top frame from the constraint system, such that the preceding frame becomes active.

A *Frame* is the data structure for a deterministic part of a database. It is a pair consisting of a table of *entries* indexed by objects (either variables or individuals) and a mapping from variables to objects – implementing a rename function. Note that an index may appear, obviously, at most once within a frame; however, the same object may be an index for different entries in different frames. The mapping binds variables to objects and allows to collapse two or more objects without an explicit change in the frame structure. The rename function is defined only through the mapping in the active frame.

An *Entry* is the data structure for persistent constraints indexed by objects – see figure 4 for details. It maintains information about constraints which must be stored during a satisfiability proof. Associated to each object, there are the persistent constraints containing the object itself. A role (or feature) structure contains the information about the relation between two objects – see figure 5 for details.

In the constraint system there is always only one active role-structure between the same pair of objects: the last one added. When a new relation is established between two objects already related in a non-active frame, the old relation is copied from the newest of the non-active frames and updated in the current active frame. On the contrary, information on attributes is distributed over the frame chain. The difference between an attribute and a role is that the latter needs an efficient handling of the *holds* test predicate.

Every instance of the above described structures is local to a single frame; thus, entries of different frames

+	$\langle R_1, \ldots\rangle$	$\leftarrow$ direct relations (R^+)
−	$\langle R_1, \ldots\rangle$	$\leftarrow$ inverse relations (R^-)

$\langle f_1, \ldots\rangle$ $\leftarrow$ conjunction of features

Figure 5: The role and feature data structures.

do not share any structure. This is necessary for the backtracking mechanism, since it is necessary to discard the last frame without any change in the previous ones.

3.3 The Satisfiability Abstract Algorithm

During the computation, a structure called *constrack* maintains constraints before their use. The satisfiability checker iterates over such constraints, applies the propagation rules and inserts persistent constraints in the constraint system. The constrack structure is indexed by object names and it is simpler than the constraint system structure, since it does not allow for complex indexing between objects and concepts.

The function `SAT` takes a set of constraints and a constraint system structure and returns the completed constraint system, if the set of constraints are satisfiable, nil otherwise.

```
function sat(ConstraintsSet,database)
  constrack := set2constrack(ConstraintsSet)
  new-dbase := database
  foreach object in constrack
    new-dbase :=
      eval-object(object,constrack,new-dbase)
    if new-dbase = NIL then EXIT_WITH_FAILURE
  endloop
  EXIT_WITH_SUCCESS
endfunction
```

The function `EVAL-OBJECT` evaluates all the constraints related to a given object. It modifies the constrack, by inserting new constraints, and the database. If a clash is found, the it returns NIL, otherwise it returns the modified database.

```
function eval-object(obj,constrack,dbase)
  new-dbase := dbase
  loop until there are no constraints for obj
    new-dbase :=
      (OR eval-and-constraints
            (obj,constrack,new-dbase)
          ...
          eval-role-constraints
            (obj,constrack,new-dbase))
    if new-dbase = NIL return(NIL)
  endloop
```

```
  return(new-dbase)
endfunction
```

There is a EVAL-...-CONSTRAINTS function for each type of constraint; functions for $\mathcal{ALC}$ are listed below – the complete listing for $\mathcal{ALCRIFO}$ can be found in [Bresciani *et al.*,1995]. Those functions apply the corresponding rule to the constrack and the database; they return NIL if a clash is found; otherwise they return the modified database. They change the constrack, removing used constraints and possibly adding new constraints.

```
function
  eval-and-constraints(object,constrack,dbase)
  foreach concept of and-constraint for object
    if concept is simple
      then if (concept = bottom) or
                (:instance object (:not concept))
              in database
             then return(NIL)
             else add (:instance object concept)
                    to dbase
      else add (:instance object concept)
             to constrack
  endloop
  return(dbase)
endfunction

function
  eval-or-constraints(object,constrack,dbase)
  foreach concept of or-constraint for object
    new-db := sat(copy(constrack) +
                    (:instance object concept),
                  add-frame(dbase))
    unless new-db=NIL return(new-db)
  endloop
  return(NIL)
endfunction

function
  eval-some-constraints(object,constrack,dbase)
  foreach (:some role concept)
          of some-constraint for object
    new-object := create-variable
    add (:related object new-object role) and
        (:instance new-object concept)
      to constrack
  endloop
  return(dbase)
endfunction

function
  eval-role-constraints(object,constrack,dbase)
  foreach (:related object object1 role)
          of role-constraint for object
    if (:instance object (:all role1 concept))
         in dbase and
       role specializes role1
      then add (:instance object1 concept)
             to constrack
    if (:instance object1 (:all role1 concept))
         in dbase and
       (:inverse role) specializes role1
      then add (:instance object concept)
             to constrack
    add (:related object object1 role) to dbase
  endloop
  return(dbase)
endfunction
```

4 Extensions to the language

The CRACK system currently implements complete reasoning for a larger language than $\mathcal{ALCRIFO}$. Roles and features have been syntactically unified, so that the existential and universal quantifiers can be uniformly used with them. Composition, range and domain have been added for roles. Existential agreement between role paths, singleton variable and fill operators have been added for concepts. The CRACK language can express, for example, the *schema-views* language for querying object-oriented databases – with the exception of cyclic definitions in the schema [Buchheit *et al.*,1994].

We plan to add the possibility of expressing general inclusion statements – as in [Buchheit *et al.*,1993]. Of course, in this case only the subset $\mathcal{ALCR}$ of the language should be used in order to guarantee the termination of the satisfiability procedure. A more ambitious plan is the implementation of $\mathcal{CIF}$ [De Giacomo and Lenzerini,1994] – a description logic in correspondence with an extension of *converse-PDL*, i.e. propositional dynamic logic with the converse operator. De Giacomo and Lenzerini have proved the decidability of satisfiability in $\mathcal{CIF}$, which includes $\mathcal{ALC}$, functional restrictions, and disjunction, composition, transitive closure, inverse, and identity for roles. In $\mathcal{CIF}$, full terminological axioms can be expressed. Number restrictions, which are present in KRIS and in other systems, can be naively implemented within CRACK, using the same basic modules (like the one for variable renaming) and data structures; however, the efficiency of such a straightforward implementation should be tested. Perhaps, more sophisticated data structures are needed to handle efficiently number restrictions. Two other possible extensions we have taken into consideration are concrete domains [Baader and Hanschke,1991] and plural entities [Franconi,1993].

CRACK supports also an efficient way of management of ABox constraints, i.e. constraints where only individuals are involved. From an abstract point of view, the architecture described so far already supports ABox constraints, since they are uniformly processed as a special type of constraint. Since no variables but only individuals are involved, ABox constraints may be considered static – i.e. they may be stored permanently, being the same for every computation. This avoids redundant applications of the same rules. Thus, a *permanent frame* containing pre-completed ABox constraints is maintained in the constraint system, as the bottom frame. A pre-completion is the processing of the deterministic part of a constraint, which is necessarily equal in every completion of the initial constraint system. The non-deterministic ABox constraints are stored in the entries of the permanent frame, such that they could be processed in later stages. Moreover, dependencies between individuals is maintained, in order to consider only connected parts of the ABox in the processing. Of course, revision of ABox constraints is disallowed without canceling the permanent frame (or at least the connected part of it) and recomputing the whole pre-completion. Preliminary, but not sufficiently complete, tests show

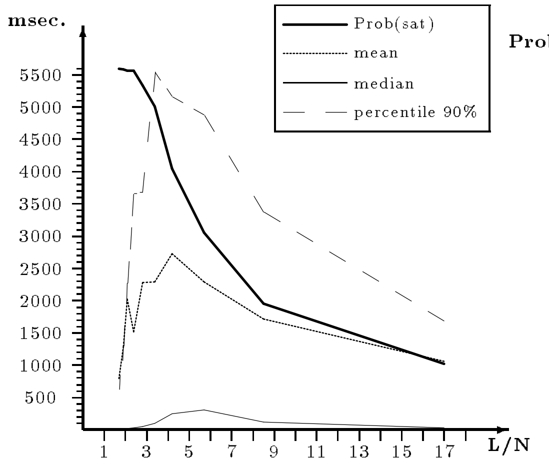

Figure 6: Random propositional 3-SAT transition. N is the number of propositional variables (i.e. primitive concepts); L is the number of `and` branches.

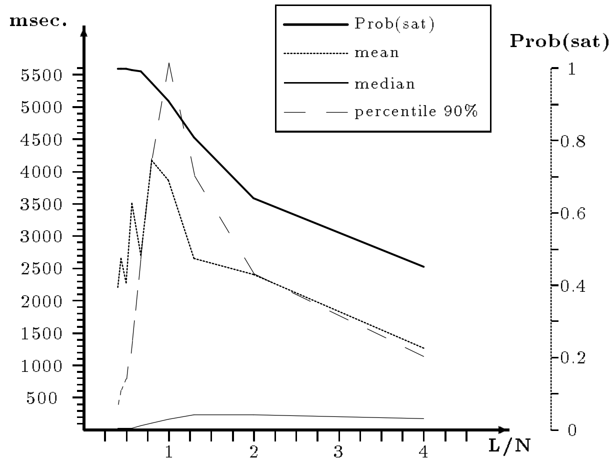

Figure 7: Random modal 2-SAT transition (degree 1). N is the number of propositional variables (i.e. primitive concepts); L is the number of `and` branches.

that ABox reasoning is quite efficient. We are still working on optimizing this task. It should be noted, however, that the presence of enumerated types – i.e., `one-of` – yields to implicit assertions which can not be easily treated as ABox constraints in the general architecture.

A powerful query language for retrieval has been implemented in CRACK. Apart from the possibility of querying an ABox with the usual open world semantics, it is possible to *efficiently* query an ABox under a closed world assumption. Closed queries make sense because give to the knowledge base a *database* interpretation, which is the preferred one in most real applications. The answer to such a query is computed as if the knowledge base had a closed world semantics; it is equivalent to a query on the knowledge base having the usual open world semantics, where each atomic role and concept is prefixed by the epistemic operator [Donini *et al.*,1992b]. Individuals are indexed with respect to the taxonomic structure of concept terms, in order to optimize the retrieval task. Two different indices sets are pre-computed, using the one for open queries and the other for closed queries. The query is classified and the obtained indexed individuals are checked; thus, the number of instance checking problems to solve is considerably reduced.

5 Testing Description Logics

In this section we present some preliminary results on testing the hardness of satisfiability in description logics. The first problem in such an effort is to find a reasonable way of generating random knowledge bases to be used in the test. It turns out that it is not even an easily defined task. In fact, it is not clear how a *random* distribution in the multi-dimensional space of well formed formulae of description logics (knowledge bases) could be expressed. Which parameters are the relevant ones? Which structures are representative of the average element of the space? Does such a distribution include both easy and hard problems?

It is possible, like it has been done in [Baader *et al.*,1994], to generate random knowledge bases[1] according to a predefined structure, which should resemble the one of real knowledge bases used in different applications based on description logics. A more general idea would be to generate knowledge bases according to the grammar of the language itself, where each transition has the same probability. However, an extensive testing proved that the generated knowledge bases do not include the real hard cases, and, moreover, they are almost always satisfiable cases (more than 95%, for $\mathcal{ALC}$ knowledge bases).

For these reasons, we looked for particular structures of knowledge bases that could be easily characterized in terms of an order parameter with respect to satisfiability, and showing some neat difficult case in correspondence to a phase transition in the satisfiability probability space. [Cheeseman *et al.*,1991] conjecture that all NP-complete problems have at least one order parameter and the hard to solve problems are around a critical value (phase transition) of this order parameter; the phase transition separates the over-constrained from the under-constrained regions of the problem space. There are many papers – e.g. refer to [Gent and Walsh,1994] – giving for propositional logic quite extensive case studies and some theoretical results. Figure 6 points out the random propositional 3-SAT transition obtained with the propositional sublanguage of CRACK. Of course, this is not a surprising result, since theoretical studies foresee the easy-hard-easy pattern for satisfiability, within a sat/unsat transition in 3-SAT problems, at around the critical empirical value of 4.2 L/N. The sat/unsat transition appears smooth since we did not consider formulae with more that 20 conjuncts for our limited computing resources[2]. Extensive tests show that average cases in *propositional* 3-SAT are 3 orders of magnitude harder

[1] In this section, knowledge bases mean just TBoxes.

[2] Satisfiability checkers tailored for propositional calculus are much more efficient than our tableaux-based architecture.

than average cases in *full* $\mathcal{ALC}$ with random grammar: this gives strong evidence that the random grammar based generator does not capture any of the hard cases, where the testing is more meaningful.

It is interesting to notice that in the (*easy*) $\mathcal{ALC}$ cases CRACK is 3 times faster than KRIS, whereas in the *hard* propositional cases (with 4.2 L/N) the ratio is inverted. This is due to the way CRACK handles in general nondeterministic rules (like the disjunction rule: $\rightarrow_{\sqcup}$): CRACK naively generates a fresh new frame at each nondeterministic point of the computation. We are re-implementing the basic resource manager in order to reuse unused frames instead of generating new ones.

Then, we tried to extend the study to the $\mathcal{ALC}$ case – which is in correspondence to the propositional modal logic $\boldsymbol{K}_{(m)}$ [Schild,1991; Halpern and Moses,1985]. It turned out that it is not easy to find a parameter leading to a phase transition; moreover, no results are presented in the literature for satisfiability in PSPACE-complete languages – like $\mathcal{ALC}$. Nonetheless, we have found some interesting cases[3]. For example, figure 7 indicates that even in the modal case a phase transition can be found; formulae are in conjunctive normal form, with L conjuncts and 2 disjuncts, where each literal could be either an existential or a universal expression, whose restriction is a L-2-CNF formula with ground literals. Again, also in this experiment the transition appears smooth. The same transition has been found using both CRACK and KRIS. For the same reasons as above, CRACK is slower than KRIS in the hard cases. More experimental and theoretical studies are needed in order to generalize this preliminary result.

Acknowledgments

Many people contributed to the effort of developing the ideas behind CRACK. We would like to thank Francesco M. Donini, Andrea Schaerf, Werner Nutt, Bernhard Hollunder, Roberto Sebastiani for the invaluable discussions we had with them. We thank also the anonymous referees who carefully read previous versions of the paper. All the errors of the paper are, of course, our own.

References

[Baader and Hanschke, 1991] Franz Baader and Philipp Hanschke. A scheme for integrating concrete domains into concept languages. In *Proc. of the 12 th IJCAI*, pages 452–457, Sidney, Australia, 1991.

[Baader and Hollunder, 1991] F. Baader and B. Hollunder. A terminological knowledge representation system with complete inference algorithm. In *Proc. of the Workshop on Processing Declarative Knowledge*, pages 67–86, Kaiserslautern, Germany, 1991.

[Baader *et al.*, 1994] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H. J. Profitlich. An empirical analysis of optimization techniques for terminological representation systems. *Applied Intelligence*, 4(2):109–132, April 1994. Kluwer Academic Publishers. Special Issue on Knowledge Base Management. Edited by John Mylopoulos.

[Bresciani *et al.*, 1995] Paolo Bresciani, Enrico Franconi, and Sergio Tessaris. Implementing and testing expressive description logics: a preliminary report. Long Version. Submitted, February 1995.

[Buchheit *et al.*, 1993] Martin Buchheit, Francesco M. Donini, and Andrea Schaerf. Decidable reasoning in terminological knowledge representation systems. In *Proc. of the 13 th IJCAI*, pages 704–709, Chambery, France, August 1993.

[Buchheit *et al.*, 1994] Martin Buchheit, Manfred A. Jeusfeld, Werner Nutt, and Martin Staudt. Subsumption between queries to object-oriented databases. *Information Systems*, 19(1):33–54, 1994.

[Cheeseman *et al.*, 1991] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proc. of the 12th IJCAI*, pages 331–337, 1991.

[De Giacomo and Lenzerini, 1994] Giuseppe De Giacomo and Maurizio Lenzerini. Boosting the correspondence between description logics and propositional dynamic logics. In *Proc. of AAAI-94*, 1994.

[Donini *et al.*, 1991] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. Tractable concept languages. In *Proc. of the 12 th IJCAI*, pages 458–465, Sidney, Australia, August 1991.

[Donini *et al.*, 1992a] F. M. Donini, B. Hollunder, M. Lenzerini, A. Marchetti Spaccamela, D. Nardi, and W. Nutt. The complexity of existential quantification in concept languages. *Artificial Intelligence*, 53:309–327, 1992.

[Donini *et al.*, 1992b] F. M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, and W. Nutt. Adding epistemic operators to concept languages. In *Proc. of the 3 rd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 342–353, Cambridge, MA, 1992.

[Franconi, 1993] Enrico Franconi. A treatment of plurals and plural quantifications based on a theory of collections. *Minds and Machines*, 3(4):453–474, November 1993. Kluwer Academic Publishers. Special Issue on Knowledge Representation for Natural Language Processing.

[Gent and Walsh, 1994] Ian P. Gent and Toby Walsh. The SAT phase transition. In *Proc. of the 11th ECAI*, pages 105–109, 1994.

[Halpern and Moses, 1985] J. Y. Halpern and Y. Moses. A guide to the modal logics of knowledge and belief: Preliminary draft. In *Proc. of the 9 th IJCAI*, pages 480–490, Los Angeles, CA, 1985.

[Hollunder and Nutt, 1990] B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. Research Report RR-90-04, DFKI, April 1990.

[Hollunder *et al.*, 1990] B. Hollunder, W. Nutt, and M. Schmidt-Schauß. Subsumption algorithms for concept description languages. In *Proc. of the 9 th ECAI*, pages 348–353, Stockholm, Sweden, 1990.

[Schaerf, 1994] Andrea Schaerf. Reasoning with individuals in concept languages. *Data and Knowledge Engineering*, 13(2):141–176, 1994.

[Schild, 1991] Klaus D. Schild. A correspondence theory for terminological logics: preliminary report. In *Proc. of the 12th IJCAI*, pages 466–471, October 1991.

[Schmidt-Schauß and Smolka, 1991] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[3]Thanks to Roberto Sebastiani.