Automated Reasoning with Boolean ABoxes

Bernadette Martínez Hernández

Amsterdam, November 25, 2002

Supervisors
Maarten Marx
Carlos Areces

Members of the committee
Dick de Jongh
Maarten de Rijke
Breanndán Ó Nualláin

Acknowledgements

I would like to thank my parents and friends for the support given during the elaboration of this thesis. I want to give an special recognition to my supervisors for their guidance and patience. It is a great pleasure to thank the staff of ILLC, without its help this work would not have been possible. My deep gratitude goes to Nuffic, the organization that gave me the opportunity to study in the Netherlands. I do not mention names, because I do not want to make any omission, but all the people who offered me a shoulder to lean on will be always in my heart.

Contents

Chapter 1

Introduction

Description Logics (DL) is a family of knowledge representation (KR) formalisms tailored to represent the knowledge of an application domain by first defining the basic and derived concepts of the domain, and then using these concepts to specify properties of objects and individuals in the domain. Therefore a DL knowledge base (KB) is made up of two parts, the terminological part (TBox) where the definitions of the basic and derived notions are stored, and the assertional part (ABox), which records facts about individuals.

A common property of a group of individuals is described by a concept. Concepts can be considered as unary predicates and interpreted as sets of objects. Roles are interpreted as binary relations between objects. Thus, an ABox resembles superficially a relational database with only unary and binary relations. However a database represents only one interpretation, while an ABox encodes a set of interpretations, namely all its models. The ABox does not assume its information to be complete, but in a database the absence of information is regarded as negative information. The partial knowledge of an ABox comes out not only from lack of information, but also from disjunctive assertions like $i : C \sqcup D$.

Such possibility of partial knowledge is very restricted, it only allows conjunction of assertions. A less restricted representation is usually needed to deal with interesting problems as planning and diagnosis. To be useful it would have to have good expressivity to describe partial knowledge by means of disjunctions or boolean constraints.

One way to tackle this problem is to use the Boolean ABoxes (BABoxes), in which boolean combinations of ABox assertions are permitted. Given a TBox T and a BABox B, to find whether (T, B) has a model we treat all the assertions of the BABox as propositions. Then, all the propositional models, which are ABoxes, can be obtained. If one of these models A, together with T has a model; then (T, B) has a model. This process may be expensive because of the exhaustive search. Therefore, a suitable translation from BABox to a regular KB could make it better.

In [ABM02] such translation has been proposed. However it is not clear if this translation is really more efficient than an optimized method which tries out all propositional models. The research reported in this thesis was done in order to clarify this issue.

Our main results are that the translation approach is an adequate option for BABoxes containing propositionally non trivial information, specifically when the BABoxes are modally constrained. We give examples of DL's in which the translation approach is always needed.

Our work has several important consequences. We created a number of examples which turned out to be revealing. They showed several mistakes in RACER. We developed several optimizations to both the translation and the model generation approach. Furthermore we created a random generator of test examples.

Chapter 2 is a short introduction to Description Logics. Boolean ABoxes are defined here as well. The two main approaches to check the consistency of Boolean ABoxes, their algorithms and their optimizations are described in Chapter 3. The last chapter contains the description of the implementations of these algorithms, the testing and the description of the random generator of test sets. In Appendix A we include the code of the algorithms and in Appendix B we can find the formulation of some puzzles that were part of the preliminary test of the implementations.

Chapter 2

Description logics and Boolean ABoxes

This chapter provides a brief introduction to Description Logics as a formal language for representing knowledge and reasoning about it. It introduces the syntax and semantics of the language and elements needed to build a knowledge base as well as the standard inference problems. Finally it defines our main object of study, Boolean ABoxes.

2.1 The language

Definition 1 *Let* $\mathsf{C}= \{C_1, C_2, \ldots\}$ *be a countable set of* atomic concepts*,* $\mathsf{R}= \{R_1, R_2, \ldots\}$ *be a countable set of* atomic roles *and* $\mathsf{I} = \{i_1, i_2, \ldots\}$ *be a countable set of* individuals*. For* C,R,I *pairwise disjoint,* $\mathcal{S} = \langle \mathsf{C},\mathsf{R},\mathsf{I}\rangle$ *is a* signature*. An interpretation for a fixed signature* $\mathcal{S}$ *is a tuple* $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}\rangle$*, where*

- $\Delta^{\mathcal{I}}$ *is a nonempty set, called the* domain *of* $\mathcal{I}$*.*
- $\cdot^{\mathcal{I}}$ *is a function assigning an element* $i_j^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ *to each individual* i_j*; a subset* $C_j^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ *to each atomic concept* C_j*; and a relation* $R_j^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ *to each atomic role* R_j

Complex concepts and roles can be built inductively from atomic symbols by means of *constructors*. Table 2.1 and 2.2 show some of them.

A language $\mathcal{L}$ is characterized by the constructors it allows. For a given language $\mathcal{L}$, $\mathsf{C}(\mathcal{L})$ denotes the set of complex concept expressions and $\mathsf{R}(\mathcal{L})$ the set of complex role expressions which can be formed using the constructors of $\mathcal{L}$. We are particularly interested in the language $\mathcal{ALC}$ and its extensions. $\mathcal{ALC}$ allows the universal concept (top), negation, disjunction, conjunction, existential and universal quantification constructors.

To give an example of what can we express in $\mathcal{ALC}$, we suppose that *Accountant* and *Employee* are atomic concepts and *Boss_of* is an atomic role.

Then $Accountant \sqcap \neg Employee$, $\exists Boss\_of.Employee \sqcup \forall Boss\_of.Accountant$ are concepts describing intuitively, the set of individuals that are accountants and not employees, and the set of individuals who are the boss of an employee or the boss of only accountants, respectively.

Adding constructors to $\mathcal{ALC}$ we form more expressive logics. One of the extensions of $\mathcal{ALC}$ is $\mathcal{SHIQ}$, that is $\mathcal{ALC}$ plus qualified number restrictions, role hierarchies, transitive and inverse roles. Following the previous examples, the $\mathcal{SHIQ}$ concept $\leq 1 Boss\_of.Employee$ express the set of individuals which are the boss of at most one employee. The exact meaning of the role hierarchy and the transitive and inverse roles constructors are detailed below. From now on, we will assume the language we are working with is $\mathcal{SHIQ}$.

Definition 2 (Knowledge base) *Fix a description language $\mathcal{L}$. A* knowledge base (KB) *is a pair $K = (T, A)$ where T is a* TBox*, and A is an* ABox*. A* TBox *is a finite, possibly empty set of* axioms *of the forms $C \sqsubseteq D$ or $R \sqsubseteq S$, where C,D are in $\mathsf{C}(\mathcal{L})$ and R,S are in $\mathsf{R}(\mathcal{L})$. They are called* general concept inclusion axioms (GCI) *and* role inclusion axioms *respectively. A* role hierarchy *$\mathcal{R}$ is a set of role inclusion axioms. A* terminology *$\mathcal{T}$ is a set of GCIs. An* ABox *is a finite, possibly empty, set of* assertions *of the forms $i{:}C$ or $R(i, j)$, where $C \in \mathsf{C}(\mathcal{L})$, $R \in \mathsf{R}(\mathcal{L})$, and i, j are individuals.*

The available knowledge of a KB is split into two parts. The information of the TBox, aims to express the definitions of the basic and derived notions and the way they are inter-related. This information is "generic" or "global," being true in every model of the situation and of every individual of the situation. The ABox records "specific" or "local" information, being true for certain particular individuals in the situation.

To illustrate the definitions above, consider the next example. Suppose we are working in the specification of an office work environment, and we have the concepts *Secretary* and *Employee* which represent the set of all secretaries in the modelled domain and the set of employees respectively. We also have the roles $Boss\_of$ and *Colleague* which identify the binary relations in the modelled domain between boss and subordinates and fellow workers respectively.

We can express with a GCI in the TBox that all the secretaries are employees,

$$Secretary \sqsubseteq Employee$$

and with a role inclusion axiom that the relation between a boss and a subordinate is a subset of the fellowship relation.

$$Boss\_of \sqsubseteq Colleague$$

The statements saying that Pedro, a particular individual is an employee,

$$pedro{:}Employee$$

and is related to Pancho as his boss

$$Boss\_of(pedro, pancho)$$

are part of the ABox.

Constructor	Syntax	Semantics
concept name	C	$C^{\mathcal{I}}$
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction ($\mathcal{U}$)	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
negation ($\mathcal{C}$)	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
existential quantif. ($\mathcal{E}$)	$\exists R.C$	$\{i \mid \exists j \in \Delta^{\mathcal{I}}.(\langle i,j\rangle \in R^{\mathcal{I}} \wedge j \in C^{\mathcal{I}})\}$
universal quantif.	$\forall R.C$	$\{i \mid \forall j \in \Delta^{\mathcal{I}}.(\langle i,j\rangle \in R^{\mathcal{I}} \rightarrow j \in C^{\mathcal{I}})\}$
qualified number restriction ($\mathcal{Q}$)	$\geq nR.C$ $\leq nR.C$	$\{i \mid \sharp\{j \mid \langle i,j\rangle \in R^{\mathcal{I}} \wedge j \in C^{\mathcal{I}}\} \geq n\}$ $\{i \mid \sharp\{j \mid \langle i,j\rangle \in R^{\mathcal{I}} \wedge j \in C^{\mathcal{I}}\} \leq n\}$

Table 2.1: Concept constructors.

C, D denote concepts, R, S roles, i, j individuals, n a non-negative integer, $\sharp M$ the cardinality of M and $(R^{\mathcal{I}})^+$ the transitive closure of $R^{\mathcal{I}}$.

Constructor	Syntax	Semantics
inverse role ($\mathcal{I}$)	R^-	$\{\langle i,j\rangle \mid \langle j,i\rangle \in R^{\mathcal{I}}\}$
transitive role	$R \in \mathsf{R}_+$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$

Table 2.2: Role constructors

Definition 3 *Let R_+, $\mathsf{R}_+ \subseteq \mathsf{R}$, be the set of transitive atomic roles, meaning with this, the set of atomic roles whose interpretation will be forced to be a transitive relation. We define the function Inv on roles to avoid considering roles such as R^{--}; this function returns $\mathsf{Inv}(R) = R^-$ if R is an atomic role, and $\mathsf{Inv}(R) = S$ if $R = S^-$. We also define the function Trans which returns true iff R is a transitive role. More precisely, $\mathsf{Trans}(R)$ = true iff $R \in \mathsf{R}_+$ or $\mathsf{Inv}(R) \in \mathsf{R}_+$. We define the relation $\sqsubseteq\!\!\!*$ as the transitive-reflexive closure of $\sqsubseteq$ over $\mathcal{R} \cup \{\mathsf{Inv}(S) \sqsubseteq \mathsf{Inv}(R) \mid S \sqsubseteq R \in \mathcal{R}\}$.*

Continuing our example of KB, we can specify that the relation between fellow workers is transitive, by stating

$$\mathsf{Trans}(Colleague) = True$$

and that the subordinate relation is the inverse of the boss relation as

$$\mathsf{Inv}(Boss\_of) = Subordinate\_of.$$

Definition 4 *Let $\mathcal{I}$ be an interpretation, K a knowledge base, C, D concepts, R, S roles, i, j, k individuals, $\mathcal{T}$ a terminology and $\mathcal{R}$ a role hierarchy.*

- $\mathcal{I} \models C \sqsubseteq D$ *if* $C^\mathcal{I} \subseteq D.^\mathcal{I}$
- $\mathcal{I} \models i{:}C$ *if* $i^\mathcal{I} \in C^\mathcal{I}$.
- $\mathcal{I} \models R(i,j)$ *if* $(i^\mathcal{I}, j^\mathcal{I}) \in R^\mathcal{I}$.
- $\mathcal{I} \models \mathcal{T}$ *if for each* $C \sqsubseteq D \in \mathcal{T}$, $C^\mathcal{I} \subseteq D^\mathcal{I}$.
- $\mathcal{I} \models \mathcal{R}$ *if for each* $R \sqsubseteq S \in \mathcal{R}$, $R^\mathcal{I} \subseteq S^\mathcal{I}$.
- $\mathcal{I} \models K$ *if for every* $\phi \in K$, $\mathcal{I} \models \phi$. *We call* $\mathcal{I}$ *a* **model** *of the knowledge base* K.
- $K \models \phi$, *if for all models* $\mathcal{I}$ *of* K *we have* $\mathcal{I} \models \phi$.

All roles in R_+ *should be assigned to a transitive relation over* $\mathcal{I}$ *for* $\mathcal{I}$ *to be an acceptable model, that is, for all* $R \in \mathsf{R}_+$, $R^\mathcal{I}$ *is transitively closed:*

$$\textit{if } (i,j) \in R^\mathcal{I} \textit{ and } (j,k) \in R^\mathcal{I}, \textit{ then } (i,k) \in R^\mathcal{I}$$

2.2 Boolean ABoxes

The format imposed by ABoxes is something too restrictive. For instance, in planning and diagnosis one often needs to describe the partial knowledge of the world using disjunctions or boolean constraints. In particular consider the logical puzzle PUZ003-1 from TPTP [SS]:

> *There is a barbers' club that obeys the following three conditions:*
>
> *(1) If any member A has shaved any other member B - whether himself or another - then all members have shaved A, though not necessarily at the same time.*
>
> *(2) Four of the members are named Guido, Lorenzo, Petrucio, and Cesare.*
>
> *(3) Guido has shaved Cesare. Prove Petrucio has shaved Lorenzo*

We can express in FOL statement *(1)*:

$$\forall x \forall y \forall z (member(x) \wedge member(y) \wedge shaved(x,y) \wedge member(z) \rightarrow shaved(z,x))$$

If we want to formalize this example in DL we find in statement *(1)* a role assertion conditioned by other assertions. An informal notation to express such conditioning would be:

$$i{:}Member,\ j{:}Member,\ Shaved(i,j),\ k{:}Member \rightarrow Shaved(k,i)$$

for all i, j, k elements of $\{$*Guido, Lorenzo, Petrucio, Cesare*$\}$. Let us make this intuitive notation more precise now.

Definition 5 *An* ABox literal *is an ABox assertion or its negation. An ABox* clause *is a set of ABox literals. Let P_i, Q_j be ABox assertions, $n, m \geq 0$. An* ABox sequent *has the form*

$$P_1, \ldots, P_n \to Q_1, \ldots, Q_m$$

and it is interpreted as the ABox clause

$$\{\neg P_1, \ldots, \neg P_n, Q_1, \ldots, Q_m\}.$$

A Boolean ABox (BABox) *is a set of ABox sequents. A* unit clause *is a singleton.*

Let us define formally the semantics of BABoxes.

Definition 6 *Let $\mathcal{I}$ be an interpretation and l_1 a literal. $\mathcal{I}$ satisfies l_1 if l_1 is positive and $\mathcal{I} \models l_1$, or if $l_1 = \neg l_2$ and $\mathcal{I} \not\models l_2$.*
$\mathcal{I} \models \{l_1, l_2, ..., l_n\}$, if $\mathcal{I} \models l_i$ for some i.
$\mathcal{I}$ satisfies a BABox if it satisfies all its sequents.

Traditional ABoxes are BABoxes with only unit clauses of positive literals. Every further KB will be considered boolean unless a different remark is made.

Suppose we formulate a problem with a traditional TBox and a BABox. Now the issue is how to reason about this Boolean KB. The definition and semantics of a BABox point to a suitable preprocessing of this input to get something that a traditional DL system can deal with. By doing this we do not lose the expressiveness of BABoxes and we are able to execute inferences over the Boolean KB. The following chapter explains the details of the preprocessing step and some optimizations, and in Chapter 4 we give technical details of the implementation and the results of the testing.

Chapter 3

Transforming Boolean ABoxes

Boolean ABoxes provide an extended flexibility to formulate interesting problems. But, we want more than that, we also want to make inferences about the information in the BABox, and we do not want to make these inferences by hand. The expressive power of the language is sufficiently high to make this unfeasible even for very small KBs. The solution is to use an automatic prover. RACER is the first DL system for TBox and ABox reasoning in $\mathcal{ALCQHIR}_{+}(\mathcal{D}^{-})$ ($\mathcal{SHIQ}$ plus a restricted form of concrete domains) [HM01a, HM], giving us the support for the language we are dealing with and a high performance in the reasoning tasks [HM01b]. However, the input format of RACER does not support boolean assertions. What we need is a way to solve the problem of finding whether a BABox is consistent without losing the facility to express and read easily the BABox statements. The most obvious approach comes from the definition of a BABox. A BABox can be seen as a propositional set. If we use a complete propositional model generator, each propositional model represents a list of literals which in principle (with some modifications as we will see in Section 3.2) can be fed into RACER to test if it is modally consistent. But this exhaustive search may be time consuming. Another option could be the translation of the whole BABox into an equisatisfiable KB in a format RACER can work with. Such rewriting has already been proposed in [ABM02]. The details of this translation, the model search method and their implementations will be specified in the first two sections of this chapter. Section 3.3 describes some optimizations for these implementations.

3.1 Translation

We name a KB (T, A) *Boolean*, when T is a traditional TBox and A a BABox. The first approach we describe in this chapter is the translation of a Boolean KB in an equisatisfiable traditional KB. This translation can be thought as a pre-processor for RACER. It is fully described in [ABM02], and in this section only the algorithm and some highlights are shown. Finally our version of the algorithm and some modifications are explained.

3.1.1 The original idea

The algorithm to translate an $\mathcal{ALC}$ Boolean KB into a traditional KB proposed in [ABM02] is:

input: a Boolean KB (T, A) whose BABox A is in clausal form.

- if $R(i, j)$ occurs negatively or in non unit clauses, then uniformly replace $R(i, j)$ in A by $i{:}\exists\, R\,.IamJ$, and add the $(*)$ formulas described below to the KB using a new atomic role *FakeR* and a new atomic concept *IamJ* every time.
  Define the KB $(*)$ as:

  $$\begin{array}{ll} (*1) & R \sqsubseteq \mathit{FakeR} \\ (*2) & \mathit{FakeR}(i, j) \\ (*3) & j{:}\mathit{IamJ} \\ (*4) & i{:} \leq 1 \mathit{FakeR}.\mathit{IamJ}. \end{array}$$

  The $(*i)$ give names to the KB elements. Notice that $(*1)$ is a TBox statement and the other elements belong to the ABox.
  intermediate result: a BABox without negated role assertions or role assertions in non unit clauses.

- push negations inside using the validity $\neg i{:}C \equiv i{:}\neg C$.
  intermediate result: a BABox without negative literals.

- collect disjunctions concerning the same individuals using the validity $i{:}C \vee i{:}D \equiv i{:}C \sqcup D$.
  intermediate result: a BABox in which every clause contains at most one literal $i{:}C$ for each i.

- repeatedly transform disjunctions concerning different individuals. From $\mathrm{KB} \cup \{\{L_1, \ldots, L_n, i{:}C, j{:}D\}\}$ to $\mathrm{KB} \cup \{\{R(i, j)\}, \{L_1, \ldots, L_n, i{:}(C \sqcup \forall\, R\,.D)\}\}$; using a new symbol R every time.

output: a knowledge base (T', A') in RACER format.

The algorithm is proved to preserve satisfiability in [ABM02], but no implementation is presented. This is the motivation for the following section.

3.1.2 The implementation

We provide an implementation of the previous algorithm in Haskell which follows the lines of the pseudocode written below. The code of the implementation can be found in Appendix A.

We assume that we have a function `subst` with parameters `A`,`B`,`C`, where `A` and `B` are literals and `C` a set of clauses, such function returns the result of replacing any occurrence of `A` in `C` by `B`. We also assume the existence of a function `substCl` with parameters `A`,`B`,`C`, where `A` is a list of literals `[l1,l2,...,ln]`, `B` is a literal and `C` is a clause, `substCl` returns the result of replacing the occurrence of all the literals of `A` in `C` by `B`. The function `spreadImp` returns the transformation of a set of sequents into a set of clauses. We also take account of functions to generate new concepts and role names, they are called `newConceptSymbol` and `newRoleSymbol` respectively. For the case of the generation of *FakeR*, the function `FakeRoleOf` returns the specific '*fake*' role name predefined for any role R. This role name is unique for every R. This aspect differs from the original algorithm. Another difference in our implementation is that the input can be more expressive, accepting $\mathcal{SHIQ}$ Boolean KBs. The justifications for these changes are made after the pseudocode.

input: A Boolean KB (T, A), where A is a set of ABox sequents.
output: A traditional KB (T', A')

```
ba2racer(T,A)
{
  B := spreadImp(A)

  \* Step 1 *\
  for every R(i,j) in B negated or in non unit clauses
  {
      FakeR := FakeRoleOf R;
      IamJ := newConceptSymbol;
      B := subst(R(i,j),i:∃ R.IamJ,B);
      T := T ∪ { {R ⊑ FakeR} };
      B := B ∪ { {FakeR(i,j)}, {j:IamJ}, {i:≤ 1FakeR.IamJ} };
  }

  \* Now B is a BABox without problematic
  relational assertions *\

  \* Step 2 *\
  for every clause C in B
  {
    while ¬i:D in C
        C = substCl([¬i:D],i:¬D,C);
  }

  \* Now B has no negative literals *\

  \* Step 3 *\
  for every clause C in B
```

```
  {
      while i:D1,...,i:Dn in C and n > 1
          C = substCl([i:D1,...,i:Dn],i:(D1⊔...⊔Dn),C);
  }

  \* Now every clause in B has one literal i:D for each i *\

  \* Step 4 *\
  for every clause C in B
  {
      while C has two members i:D,j:E;
      {
          R = newRoleSymbol;
          C = substCl([i:D,j:E],i:D ⊔ ∀ R.E,C);
          B = B ∪ {R(i,j)};
      }
  }

  \* Now B has only unit clauses, is an ABox *\

  return(T,B)
}
```

We need to check that this implementation is "equivalent" to the intuitive rewriting process we introduced in the first part of this section. We will provide an explanation for every step marked in the pseudocode.

Step 1 differs from the original algorithm. In [ABM02] a new *FakeR* atom is generated every time a $\neg R(i, j)$ is found. Our approach considers for every role R in a $\mathcal{SHIQ}$ KB only a new fixed unique role atom *FakeR*.

We define $A[\phi/\theta]$ as the result of uniformly replacing ϕ with θ in A and $A[\phi/\theta][\delta/\rho]$ as $(A[\phi/\theta])[\delta/\rho]$.

Let (T, A) be a $\mathcal{SHIQ}$ Boolean KB with the role assertions $R(i_1, j_1), \ldots,$ $R(i_n, j_n)$ negated or in non unit clauses over the same role R. Then, the KB $*^n$ denotes the following:

- $R \sqsubseteq FakeR$ (axiom)
- $FakeR(i_1, j_1)$

  $\vdots$
- $FakeR(i_n, j_n)$
- $j_1 : IamJ_1$

  $\vdots$
- $j_n : IamJ_n$

- i_1: $\leq 1 \mathit{FakeR}.\mathit{IamJ}_1$

  $\vdots$
- i_n: $\leq 1 \mathit{FakeR}.\mathit{IamJ}_n$

where $FakeR$ is a new role atom and $IamJ_i$, $1 \leq i \leq n$, are new atomic concepts.

We want to substitute the problematic role assertions similarly to the way the original algorithm does it. Thus, it is necessary to prove the following statement to support the substitution:
For a $R(i,j) \in \{R(i_1,j_1),\ldots,R(i_n,j_n)\}$

$$(*^n) \models R(i,j) \equiv i{:}\exists\, R\,.\mathit{IamJ}. \tag{3.1}$$

Proof.
(3.1) is most easily proved by a (semi-formal) tableau argument[1].
$\Rightarrow$

1)	$R(i,j)$	
2)	$\neg i{:}\exists\, R\,.\mathit{IamJ}$	
3)	$\neg j{:}\mathit{IamJ}$	($\forall$ rule on 1 and 2)
4)	$j{:}\mathit{IamJ}$	($*^n$ element)
	$\perp$	

$\Leftarrow$

1)	$i{:}\exists\, R\,.\mathit{IamJ}$	
2)	$\neg R(i,j)$	
3)	$R(i,k)$	($\exists$ rule on 1)
4)	$k{:}\mathit{IamJ}$	
5)	$\mathit{FakeR}(i,k)$	(by 3 and axiom)
6)	$\mathit{FakeR}(i,j)$	($*^n$ element)
7)	$j{:}\mathit{IamJ}$	($*^n$ element)
8)	$k=j$	(by 4,5,6,7 and $*^n$ element)
	$\perp$	(by 2,3,8)

QED

Now, we are ready to prove the corrections of our modified version of the role assertion handling.

Claim 1 Let (T,A) a be a Boolean KB with $R(i_1,j_1),\ldots,R(i_n,j_n)$ occurring negatively or in non unit clauses in A. (T,A) is satisfiable iff

$$*^n \cup T \cup A[R(i_1,j_1)/i{:}\exists R.\mathit{IamJ}_1]\ldots[R(i_n,j_n)/i{:}\exists R.\mathit{IamJ}_n]$$

is satisfiable.

[1] The expansion rules can be found in Table 3.1

Proof.

$\Rightarrow$

Let $\mathcal{I} \models (T, A)$. Expand $\mathcal{I}$ to $\mathcal{I}'$ by setting

$$(IamJ_k)^{\mathcal{I}'} = \{j_k^{\mathcal{I}}\} \text{ for all } IamJ_k \text{ such that } 1 \leq k \leq n$$

$$FakeR^{\mathcal{I}'} = R^{\mathcal{I}} \bigcup \{(i_k^{\mathcal{I}}, j_k^{\mathcal{I}})\}_{1 \leq k \leq n}$$

and $\mathcal{I}'$ is equal to $\mathcal{I}$ for all other symbols.

$\mathcal{I}' \models *^n$ and still $\mathcal{I}' \models (T, A)$. Whence, by (3.1),

$\mathcal{I}' \models A[R(i_1, j_1)/i{:}\exists R.IamJ_1] \ldots [R(i_n, j_n)/i{:}\exists R.IamJ_k]$

$\Leftarrow$

Let $\mathcal{I} \models *^n \cup T \cup A[R(i_1, j_1)/i{:}\exists R.IamJ_1] \ldots [R(i_n, j_n)/i{:}\exists R.IamJ_n]$ then $\mathcal{I} \models T$ and by (3.1) $\mathcal{I} \models A$, whence $\mathcal{I} \models (T, A)$ QED

This proof justifies the process of continuously replacing the negated or non unit role assertions by concept assertions, followed by the addition of the corresponding elements of $*^n$ to the TBox. This is achieved in the first step as can be observed in the pseudocode. After finishing, our BABox does not contain negated role assertions or role assertions in non unit clauses, just like in the original algorithm.

The further steps are devoted to the handling of concept assertions in order to transform the BABox obtained of the first step into an ABox.

Step 2 and ***Step 3*** follow from:

$$\mathcal{I} \models \neg i{:}C \iff \mathcal{I} \models i{:}\neg C \tag{3.2}$$

$$\mathcal{I} \models \{i{:}C, i{:}D\} \iff \mathcal{I} \models i{:}C \sqcup D. \tag{3.3}$$

for any model $\mathcal{I}$.

They realize exactly the same labor as in the original algorithm; that is, they introduce the negations in the concept, and join in a single concept all the concept assertions of a clause related to the same individual.

Step 4 At this point where we have either unit clauses with role assertions or clauses with at most one positive concept assertion per individual. The last step reduces the disjunction of concept assertions to the point we only have unit clauses. It is justified by the following claim, explained in [ABM02]:

For any Boolean KB B, $B \cup \{\{L_1, \ldots, L_n, i{:}C, j{:}D\}\}$ is satisfiable if and only if $B \cup \{\{R(i, j)\}, \{L_1, \ldots, L_n, i{:}(C \sqcup \forall\ R\ .D)\}\}$ is satisfiable. (3.4)

As in [ABM02] we use a new role atom R every time.

We can conclude that after a Boolean KB has been transformed by our implementation the output is an equisatisfiable traditional KB and RACER can be fed with this KB to execute inferences. This implementation fullfill the idea of the original translation algorithm of [ABM02] and improves the handling of role assertions. The technical details of the implementation of the pseudocode showed in this section can be consulted in Appendix A.

There is another way to handle a BABox. This second approach is more related to the semantics of BABoxes and is described in the next section.

3.2 Disjunction of knowledge bases

The translation approach focuses on transforming clauses into traditional assertions. There is another way of solving the problem without initially using reductions. The intuition is to regard assertions as propositions and test the modal consistency of every propositional model.

Suppose we have a Boolean KB (T, A). By definition, a BABox is satisfiable if at least one literal of every clause is satisfiable. If we take every assertion as a proposition, we can generate all the propositional models A_i for the clause set. Every A_i is a BABox with only unit clauses and (T, A) is consistent if and only if (T, A_i) is consistent for some i.

In order to prove the consistency of any KB (T, A_i), we want to feed RACER with it and test it. However, we need first to eliminate the negated role assertions. One possibility is to use the translation of the previous section for negated role assertions in a $\mathcal{SHIQ}$ KB. However, under certain conditions, it is possible to erase the negated role assertions and preserve the validity.

We explain in this section these two approaches for the knowledge base consistency checking based on the disjunction of KBs. We use the Davis-Putnam-Logemann-Loveland (DPLL) method [DLL62] to search for the propositional models and then we test them in RACER. The pseudocode for both procedures is shown in the next subsection. We called the two procedures DPLL-RACER procedures.

3.2.1 The implementations

The first DPLL-RACER procedure, `dpllRacerTrans` allows a $\mathcal{SHIQ}$ Boolean KB as input. The second one, `dpllRacer`, is restricted to $\mathcal{SHQ}$ Boolean KBs.

We adapt the standard DPLL procedure to turn it in to an exhaustive model generator. We use two variables; `Sigma` a BABox where all the clausal and propositional handlings of DPLL are done; and `Gamma` a Boolean KB with only unit clauses where the propositional truth assignments of every assertion are saved. Thus, for an input Boolean KB (T, A), the initial call to any DPLL-RACER procedure assigns `Sigma` to the set of ABox sequents A translated into clauses, and `Gamma` to the KB $(T, \{\})$.

`Sigma {l = val}` is the result of applying the unit propagation of the literal `l` when its truth value is `val`. `Gamma + {l}` adds the clause `{l}` to `Gamma`. The output of our DPLL-RACER procedures is `True` if the KB is consistent, and `False` otherwise.

`dpllRacerTrans` applies our translation of negated role assertions explained in Section 3.1 to the propositional model stored in `Gamma` with the function `transformR` Afterwards, `Gamma` is passed to the function `TestRacer` which returns the result of testing the consistency of `Gamma` in RACER.

```
dpllRacerTrans(Sigma,Gamma)
{
   if Sigma == {} then
   {
       Gamma := transformR Gamma;
       if (TestRacer Gamma) then True;
       else  return False;
   }
   if Sigma contains the empty clause then
       return False;
   if Sigma has a unit clause {l} then
       dpllRacerTrans (Sigma {l = true}, Gamma + {l});
   if dpllRacerTrans (Sigma {l = true}, Gamma + {l}) then
       return True;
   else dpllRacerTrans (Sigma {l = false}, Gamma + {~l});
}
```

`dpllRacer` uses a different approach to handle the negated role assertions. First of all, it can only be used with a $\mathcal{SHQ}$ KB. `dpllRacer` uses the function `eraseR` which applies to the input KB the role deletion procedure explained at the end of this section. The output of this function is the modified KB without any $\neg R(i,j)$ or a KB with only the empty clause. We assume we have this function `eraseR`. We also assume the existence of the function `emptyClause` which returns True if a KB contains the empty clause and False otherwise. Notice that `dpllRacer` also uses the function `TestRacer` to check the consistency of `Gamma`.

```
dpllRacer(Sigma,Gamma)
{
   if Sigma == {} then
   {
       Gamma := eraseR Gamma;
       if (emptyClause Gamma) then
           return False;
       else
           if (TestRacer Gamma) then True
           else  return False;
   }
   if Sigma contains the empty clause then
       return False;
```

```
    if Sigma has a unit clause {l} then
        dpllRacer (Sigma {l = true}, Gamma + {l});
    if dpllRacer (Sigma {l = true}, Gamma + {l}) then
        return True;
    else dpllRacer (Sigma {l = false}, Gamma + {~l});
}
```

In both cases the DPLL method ensures the generation of all propositional models of A. As it can be seen, if RACER returns that `Gamma` is consistent, the process is done, otherwise the search continues. If no propositional model is left, the inconsistency of (T, A) will be stated.

We have described two general algorithms to search a model of a Boolean KBs. The difference between these algorithms is the way they handle the negated role assertions when a propositional model is found. Both procedures can be implemented as functions used in the same program, they only need to be called according to whether the KB is $\mathcal{SHQ}$ or $\mathcal{SHIQ}$.

In the next subsection we explain how the function `eraseR` works, and we justify the deletion of the negated role assertions for $\mathcal{SHQ}$. The code of these procedures and the implementation of the disjunction of KBs approach can be found in Appendix A.

3.2.2 Role assertion erasing

The previous section requires a function to erase the negated role assertions of BABoxes with only unit clauses. In this section we describe this function and prove that it preserves satisfiability.

Let (T, A) be a Boolean KB with only unit clauses and no unit clauses $\{P\}$ and $\{Q\}$ such that, $Q = \neg P$. Notice that a Boolean KB produced by the DPLL-RACER procedures before being tested with RACER satisfies this condition.

Definition 7 $(T, A) \diagdown \{\neg R(i, j)\}$ *is defined as*

$$T \cup (A - \{\{\neg R(i, j)\}\}) \cup \{\{i{:}\top\}, \{j{:}\top\}\}$$

and $(T, A) \diagdown \{\{\neg R_1(i_1, j_1)\}, \ldots, \{\neg R_n(i_n, j_n)\}\}$ *as*

$$(\ldots((T, A) \diagdown \{\neg R_1(i_1, j_1)\}) \diagdown \{\neg R_2(i_2, j_2)\}) \ldots \diagdown \{\neg R_n(i_n, j_n)\}).$$

To prove the validity of the deletion we will make use of the tableau algorithm for $\mathcal{SHIQ}$ described in [HST00]. The expansion rules of this tableau can be seen in Table 3.1. We need first to state some definitions.

Definition 8 *Let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a terminology, and $\mathcal{R}$ a role hierarchy. We define*

$\sqcap$-rule	if then	1.- $C_1 \sqcap C_2 \in \mathcal{L}(x)$ is not indirectly blocked, and 2.- $\{C_1, C_2\} \nsubseteq \mathcal{L}(x)$ $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-rule	if then	1.- $C_1 \sqcup C_2 \in \mathcal{L}(x)$ is not indirectly blocked, and 2.- $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$
$\exists$-rule	if then	1.- $\exists S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.- x has no S-neighbor y with $C \in \mathcal{L}(y)$ create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{S\}$ and $\mathcal{L}(y) = \{C\}$
$\forall$-rule	if then	1.- $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.- there is an S-neighbor y of x with $C \notin \mathcal{L}(y)$ $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$
$\forall_+$-rule	if then	1.- $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.- there is some R with $\mathsf{Trans}(R)$ and $R \sqsubseteq^* S$, 3.- there is an R-neighbor y of x with $\forall R.C \notin \mathcal{L}(y)$ $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\forall R.C\}$
choose-rule	if then	1.- $(\bowtie S.C) \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.- there is an S-neighbor y of x with $\{C, \sim C\} \cap \mathcal{L}(y) = \emptyset$ $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{E\}$ for some $E \in \{C, \sim C\}$
$\geq$-rule	if then	1.- $(\geq S.C) \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.- there are not n S-neighbors $y_1, \ldots, y_n$ such that $C \in \mathcal{L}(y_i)$ and $y_i \neq y_j$ for $1 \leq i < j \leq n$ create n new nodes $y_1, \ldots, y_n$ with $\mathcal{L}(\langle x, y_i\rangle) = \{S\}$, $\mathcal{L}(y_i) = \{C\}$ and $y_i \neq y_j$ for $1 \leq i < j \leq n$.
$\leq$-rule	if then	1.- $(\leq S.C) \in \mathcal{L}(x)$, x is not indirectly blocked, and 2.- $\sharp S^{\mathcal{F}}(x, C) > n$ there are S-neighbors y, z of x with not $y \neq z$, y is neither a root nor an ancestor of z, and $C \in \mathcal{L}(y) \cap \mathcal{L}(z)$, 1.-$\mathcal{L}(z) \longrightarrow \mathcal{L}(z) \cup \mathcal{L}(y)$ 2.- if z is an ancestor of x then $\mathcal{L}(\langle z, x\rangle) \longrightarrow \mathcal{L}(\langle z, x\rangle) \cup$ Inv $\mathcal{L}(\langle x, y\rangle)$ else $\mathcal{L}(\langle x, z\rangle) \longrightarrow \mathcal{L}(\langle x, z\rangle) \cup \mathcal{L}(\langle x, y\rangle)$ $\mathcal{L}(\langle x, y\rangle) \longrightarrow \emptyset$ Set $u \neq z$ for all u with $u \neq y$
$\leq_r$-rule	if then	1.- $(\leq S.C) \in \mathcal{L}(x)$, and 2.- $\sharp S^{\mathcal{F}}(x, C) > n$ there are two S-neighbors y, z of x which are both root nodes, $C \in \mathcal{L}(y) \cap \mathcal{L}(z)$, and not $y \neq z$, 1.-$\mathcal{L}(z) \longrightarrow \mathcal{L}(z) \cup \mathcal{L}(y)$ and 2.- For all edges $\langle y, w\rangle$: i. if the edge $\langle z, w\rangle$ does not exists, create it with $\mathcal{L}(\langle z, w\rangle) := \emptyset$ ii. $\mathcal{L}(\langle z, w\rangle) \longrightarrow \mathcal{L}(\langle z, w\rangle) \cup \mathcal{L}(\langle y, w\rangle)$ 3.- For all edges $\langle w, y\rangle$: i. if the edge $\langle w, z\rangle$ does not exists, create it with $\mathcal{L}(\langle w, z\rangle) := \emptyset$ ii. $\mathcal{L}(\langle w, z\rangle) \longrightarrow \mathcal{L}(\langle w, z\rangle) \cup \mathcal{L}(\langle w, y\rangle)$ 4.- Set $\mathcal{L}(y) := \emptyset$ and remove all the edges to/from y. 5.- Set $u \neq z$ for all u with $u \neq y$. 6.- Set $y \doteq z$.

Table 3.1: The complete tableaux expansion rules for $\mathcal{SHIQ}$

$$C_{\mathcal{T}} := \sqcap(\neg C_i \sqcup D_i) \textit{ for } C_i \sqsubseteq D_i \in \mathcal{T}$$

Let U be a transitive role that does not occur in $\mathcal{A}$, $\mathcal{R}$ or $\mathcal{T}$.

$$\mathcal{R}_U := \mathcal{R} \cup \{R \sqsubseteq U, \mathsf{Inv}(R) \sqsubseteq U | R \textit{ occurs in } \mathcal{T}, \mathcal{A} \textit{ or } \mathcal{R}\}$$

Definition 9 *Let B be a Boolean KB, $\mathcal{R}(B)$ be the role hierarchy of B, $\mathsf{R}^{-}(B)$ the set of inverse statements of B, $\mathsf{R}_{+}(B)$ the set of transitive role statements of B, $\mathsf{Rel}(B)$ the set of relational assertions of B.*

In order to preserve the validity during the deletion we will make use of the following condition.

Assumption 1 (Weak) *Let B be a Boolean KB. If $\{\mathcal{R}(B)$, $\mathsf{R}^{-}(B)$, $\mathsf{R}_{+}(B)$, $\mathsf{Rel}(B)\} \models R(i,j)$ then $\{R(i,j)\} \in B$.*

Theorem 2 *Let (T, A) be a $\mathcal{SHQ}$ Boolean KB with only unit clauses, $\{\{P\}$, $\{\neg P\}\} \not\subseteq A$ for any assertion P, and (T, A) satisfies Assumption 1. Let $\{\{\neg R_1(i_1, j_1)\}, \ldots, \{\neg R_n(i_n, j_n)\}\}$ be the set of clauses of A containing a negated role assertion. (T, A) is satisfiable iff $(T, A) \diagdown \{\{\neg R_1(i_1, j_1)\}, \ldots, \{\neg R_n(i_n, j_n)\}\}$ is satisfiable.*

Proof.
$\Rightarrow$
Straightforward.

$\Leftarrow$
Let $B = (T, A) \diagdown \{\{\neg R_1(i_1, j_1)\}, \ldots, \{\neg R_n(i_n, j_n)\}\}$. B is a traditional KB, then if B is consistent we can build a model using a tableau $\mathbf{T}$ expanding a forest $\mathcal{F}$ with the algorithm described in [HST00]. We will prove by induction over the length of the forest, that the expansion of this forest never produces $R \in \mathcal{L}(\langle x_0^i, x_0^j\rangle)$ for any $R(i,j) \notin B$, i, j individuals in B.

First, we apply to B the *internalization* of the GCIs according to [HST00], and we obtain the KB B'. $B' = A' \cup \{a : C_{\mathcal{T}} \sqcap \forall U.C_{\mathcal{T}} \mid a \textit{ occurs in } B\} \cup \mathcal{R}_U$, where A' is the ABox of B.

Then we initialize the forest based on B' as follows:

$$\mathcal{L}(x_0^a) := \{C \mid C(a) \in B'\},$$
$$\mathcal{L}(\langle x_0^a, x_0^b\rangle) := \{S \mid S(a,b) \in B'\},$$
$$x_0^a \neq x_0^b \text{ for all the individuals in } B'.$$

The initialization of $\neq$ was made in that way because we make the unique name assumption as RACER does it.

In this stage no $R \in \mathcal{L}(\langle x_0^i, x_0^j\rangle)$ for any $R(i,j) \notin B$.

Suppose that at level n of the expansion of $\mathcal{F}$, $R \notin \mathcal{L}(\langle x_0^i, x_0^j\rangle)$ for any $R(i,j) \notin B$. We will try to use the rules of expansion of Table 3.1 to produce $R \in \mathcal{L}(\langle x_0^i, x_0^j\rangle)$. The cases we need to check are:

- If $\exists R.C \in \mathcal{L}(x_0^i)$, then if x_0^i is not blocked, and has no R-neighbor y with $C \in \mathcal{L}(y)$; the tableaux creates a new node y with $\mathcal{L}(\langle x_0^i, y\rangle) = \{R\}$ and $\mathcal{L}(y) = \{C\}$. But j is an individual in B', then x_0^j is not y.

- If $\geq nR.C \in \mathcal{L}(x_0^i)$, then if x_0^i is not blocked, and there are not p R-neighbors $y_1, \ldots, y_p$ of x_0^i with $C \in \mathcal{L}(y_m)$ and $y_m \neq y_r$ for $1 \leq m \leq r \leq p$. The algorithm creates p new nodes $y_1, \ldots, y_p$ with $\mathcal{L}(\langle x_0^i, y_m\rangle) = \{R\}$, $\mathcal{L}(y_m) = \{C\}$, and $y_m \neq y_r$ for $1 \leq m \leq r \leq n$. However j is an individual in B', so x_0^j can not be used as a new node y_m for every y_m, $1 \leq y_m \leq p$.

- $\leq -$rule in this presentation of the tableau does not add new edges between root nodes.

- $\leq_r$-rule is designed to be complete when the unique name assumption is not assumed. But RACER works under the unique name assumption and hence the rule will never fire (see initialization).

If the expansion rules yield a complete and clash-free completion forest, the tableau $\mathbf{T}$ based on $\mathcal{F}$ can be used to build a model $\mathcal{I}$, such that $\mathcal{I} \models B$.

In the construction of the model $\mathcal{I}$ we need to verify that $\mathcal{I} \not\models R(i,j)$ if $R(i,j) \notin B$ and i, j individuals in B. The transitivity closure can not be used because Assumption 1 restrain the roles with individuals that can be formed; and if we attempt to use $R(i,k)$,$R(k,j)$ with k not an individual of B (a new node), we know that the forest never produced edges to support $R(k,j)$. In the case of the definition of every $R^{\mathcal{I}}$ according to the role hierarchy we only add roles between instances already connected. Because of the Assumption 1, no new roles connecting individuals of B will be added. The roles added in this construction are between individuals an non individuals (new nodes).

Therefore if the expansion rules yield a complete and clash-free completion forest, the tableau $\mathbf{T}$ based on $\mathcal{F}$ can be used to build a model $\mathcal{I}$, such that $\mathcal{I} \models (T, A) \setminus \{\{\neg R_1(i_1, j_1)\}, \ldots, \{\neg R_n(i_n, j_n)\}\}$ and $\mathcal{I} \not\models R(i,j)$, $R(i,j) \in \{R_i(i_1, j_1), \ldots, R_n(i_n, j_n)\}$; hence $\mathcal{I} \models (T, A)$ QED

We wanted to extend the deletion to $\mathcal{SHIQ}$ KBs, however we found counterexamples. The next two $\mathcal{SHIQ}$ Boolean KBs satisfy Assumption 1, they have only unit clauses and do not contain $\{P\}$ and $\{\neg P\}$ for any assertion P; but the application of the deletion of negated role assertions as in Theorem 2 produces KBs that are not equisatisfiable.

Let B_1 the following unsatisfiable Boolean KB in clausal form:

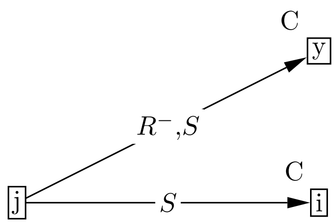
1. $R^- \sqsubseteq S$
2. $\{j{:}\exists R^-.C\}$
3. $\{j{:} \leq 1S.C\}$
4. $\{S(j,i)\}$

5. $\{i{:}C\}$
6. $\{\neg R(i,j)\}$

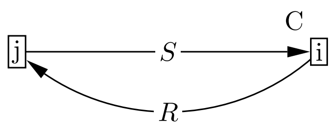
Then $B_1 \diagdown \{\neg R(i,j)\}$ is:

1. $R^- \sqsubseteq S$
2. $\{j{:}\exists R^-.C\}$
3. $\{j{:} \leq 1S.C\}$
4. $\{S(j,i)\}$
5. $\{i{:}C\}$
6. $\{i{:}\top\}$
7. $\{j{:}\top\}$

The construction of a model for $B_1 \diagdown \{\neg R(i,j)\}$ is shown in the following figures. First we use the $\exists-$rule to expand $j{:}\exists R^-.C$ and create a new node y.



Then, with the $\leq -$rule we have:



This counterexample shows that in $\mathcal{SHIQ}$ our deletion does not produce an equisatisfiable KB. If $C = \top$, the counterexample applies also for $\mathcal{SHIN}$. We did not make use of the transitive roles in B_1. The following KB shows a counterexample with transitive roles, and no number restriction.

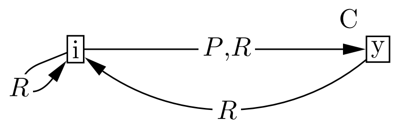
Let B_2 be the next unsatisfiable Boolean KB :

1. $\mathsf{Trans}(R) = True$
2. $P \sqsubseteq R$
3. $P^- \sqsubseteq R$
4. $\{i{:}\exists P.C\}$
5. $\{\neg R(i,i)\}$

$B_2 \setminus \{\neg R(i,i)\}$ is:

1. $\mathsf{Trans}(R) = True$
2. $P \sqsubseteq R$
3. $P \sqsubseteq R^-$
4. $\{i{:}\exists P.C\}$
5. $\{i{:}\top\}$

We expand $i{:}\exists P.C$ to construct a model of $B_2 \setminus \{\neg R(i,i)\}$ represented by:



These two cases show that we can not use the deletion as in Theorem 2 with a $\mathcal{SHIQ}$ Boolean KB. As we saw at the beginning of this section, we use instead the translation to handle the negated role assertions.

From Theorem 2 we can say that our problem in the case of a $\mathcal{SHQ}$ Boolean KB has been reduced to find a way to satisfy Assumption 1. The following operator is designed to achieve this condition.

Definition 10 *Let B be a Boolean KB with only unit clauses and no unit clauses $\{P\}$ and $\{Q\}$ such that, $P = \neg Q$. The operator $*$ over B is defined as:*

Initialize $*B = \{\mathcal{R}(B),\ \mathsf{R}_+(B),\ \mathsf{Rel}(B)\}$, then repeat until stabilization

(a) if $R \sqsubseteq S \in B$ and $\{R(i,j)\} \in *B$, then $*B = *B \cup \{\{S(i,j)\}\}$

(b) if $\mathsf{Trans}(R) = True$ and $\{R(i,j)\} \in *B$ then, for every $\{R(j,k)\}$ appearing in $*B$, $*B = *B \cup \{\{R(i,k)\}\}$

If B has a finite number of elements, then a finite number of iterations are needed to obtain $*B$.

$*B$ has only unit clauses, whence $*B$ is propositionally consistent if $\{P\}$ and $\{\neg P\}$ are not clauses in $*B$. The following claim relates $*B$ to Assumption 1.

Claim 3 Let $\Phi = \{\mathcal{R}(B),\ \mathsf{R}^-(B),\ \mathsf{R}_+(B),\ \mathsf{Rel}(B)\}$. If $\Phi \models R(i,j)$ then $R(i,j) \in *B$.

Proof.

Let $\mathcal{M} \models \Phi$, then $\mathcal{M} \models R(i,j)$ by definition. Assume $R(i,j) \notin *B$. We will construct a model $\mathcal{M}'$ such that $\mathcal{M}' \models \Phi$ and $\mathcal{M}' \not\models R(i,j)$ and we will prove the claim by contraposition. Let $\mathcal{M}' = (\Delta^{\mathcal{M}'}, \cdot^{\mathcal{M}'})$ where $\Delta^{\mathcal{M}'} = \{i, j \mid R(i,j) \in *B\}$ and $(a,b) \in S^{\mathcal{M}'}$ *iff* $S(a,b) \in *B$.

As $R(i,j) \notin *B$ then $\mathcal{M}' \not\models R(i,j)$. We want to prove that $\mathcal{M}' \models \Phi$.

We discard the inverse constructor because we are working with $\mathcal{SHQ}$. Let $S(a,b) \in \mathsf{Rel}(B)$, then $S(a,b) \in *B$ hence $\mathcal{M}' \models S(a,b)$.

To verify $\mathcal{M}' \models \mathsf{R}_+(B)$ let $S \in \mathsf{R}_+(B)$, then $S^{\mathcal{M}'}$ should be transitive. Let $(a,b) \in S^{\mathcal{M}'}$ iff $S(a,b) \in *B$ and $(b,c) \in S^{\mathcal{M}'}$ iff $S(b,c) \in *B$, we want to prove $(a,c) \in S^{\mathcal{M}'}$. By the definition of $*B$ we have $S(a,c) \in *B$ iff $(a,c) \in S^{\mathcal{M}'}$.

In the case of $R \sqsubseteq S \in \mathcal{R}(B)$ we need to prove $R^{\mathcal{M}'} \subseteq S^{\mathcal{M}'}$. We know that $(a,b) \in R^{\mathcal{M}'}$ iff $R(a,b) \in *B$ then by the definition of $*B$ we have $S(a,b) \in *B$ iff $(a,c) \in S^{\mathcal{M}'}$.

We can conclude that if $\mathcal{M}' \models \Phi$ and $\mathcal{M}' \not\models R(i,j)$ then $R(i,j) \notin *B$ whence we prove the claim. QED

Now we are ready to explain how the deletion is made. The function `eraseR` with a Boolean KB B uses the next algorithm:

1 Obtain $*B$, and set $B = *B \cup B$

2 Set $B' = B$. Apply unit propagation to B', if B' has an empty clause return a Boolean KB with only one empty clause, otherwise return

$$B \diagdown \{\{\neg R_1(i_1, j_1)\}, \ldots, \{\neg R_n(i_n, j_n)\}\}$$

where $\{\neg R_k(i_k, j_k)\} \in B$ and $1 \leq k \leq n$.

For Theorem 2 and Claim 3 we know this deletion is correct.

We have proved in this subsection that the deletion executed by the function `eraseR` in `dpllRacer` is valid. We also showed some counterexamples to explain why the deletion procedure is not used with $\mathcal{SHIQ}$ Boolean KBs. This way we justify the division of methods made at the beginning of this section where we explained the handling of Boolean KBs based on the disjunction of KBs. We showed there two methods to transform a propositional model of a Boolean KB in a format RACER can use. One of them uses partially the translation described in Section 3.1, changing the signature of the KB, and works for $\mathcal{SHIQ}$ KBs. The other approach deletes the negated role assertions after some handlings and keep the signature of the KB, but only works with $\mathcal{SHQ}$ KBs.

The next section explains some optimizations we use in the implementation of the two approaches.

3.3 Optimizations

The procedures we described in the previous sections leave almost all the inferences to RACER. They either translate the KB without concerning about the contents of it, or treat it propositionally but without realizing any propositional inference beyond the search method.

Our main aim in this section is to show some propositional reductions that can improve the performance of the search of a model and facilitate the work to RACER.

3.3.1 Propositional reductions

There are several ways to optimize the process of satisfiability checking. The simplest strategies to preprocess the input KB work in the propositional level. They depend of what we want to reduce, number of clauses, number of disjunctions or number of expensive things to translate. What we try to minimize with the procedures of this section is the size of the Boolean KB. A Boolean KB of a smaller size is easier to translate, specially if we reduce the number of elements difficult to translate, like role assertions. In the case of the DPLL-RACER search, if we deal with small Boolean KB it takes less time to find a model, and of course in a small traditional KB RACER will take less time to check consistency.

First, we prepare the KB reducing the negations in the concept assertions, afterwards we apply some propositional optimizations.

The negation of concept assertions

A strategy to improve the propositional reductions, and the DPLL-RACER search is pulling out the negations. This process is better described with the following pseudocode.

```
pullOutNot(C)
{
    if C == Not(i:(Not C1)) then
        pullOutNot(i:C1);
    else if C == i:(Not C1) then
            pullOutNot(Not(i:C1));
         else
            return C;
}
```

`pullOutNot` receives an assertion as parameter, and when this parameter is a concept assertions it tries to eliminate the redundant negations. It does not spread the negation to find the negation normal form of the concept, but a procedure that obtains the NNF is not difficult to build. We did not create it because the BABoxes used in the testing were already in NNF.

We pull out the negations just after we convert the sequents of the Boolean KB into clauses. Afterwards the propositional reductions are made, and if they do not determine the consistency of the input Boolean KB then either the translation or the DPLL search is made.

Three propositional reductions

The following reductions are implemented in both methods of handling Boolean KBs. They work before the translation in one case and before the DPLL-RACER procedure in the other case. The proofs for these reductions are straightforward from the definition of a model for a BABox.

Reduction 1 (Duplicate Deletion) *A Boolean KB* $(T, A) \cup \{\{P, P, L_1, \ldots, L_n\}\}$ *is satisfiable iff* $(T, A) \cup \{\{P, L_1, \ldots, L_n\}\}$ *is satisfiable.*

This reduction minimize the number of assertions in a clause erasing duplicates.

Reduction 2 (Tautology deletion) *A Boolean KB* $(T, A) \cup \{\{P, \neg P, L_1, \ldots, L_n\}\}$ *is satisfiable iff* (T, A) *is satisfiable.*

Reduction 2 points to the minimization of the set of clauses deleting trivially satisfiable clauses that do not add valuable information to the model search, and in the case of the translation, they add unnecessary elements to the KB.

Reduction 3 (Unit propagation) *A Boolean KB* $(T, A) \cup \{\{P\}, \{\neg P, L_1, \ldots, L_n\}\}$ *is satisfiable iff* $(T, A) \cup \{\{P\}, \{L_1, \ldots, L_n\}\}$ *is satisfiable.*

This reduction is a step in de DPLL-RACER procedures, notwithstanding, it is used before the translation and the DPLL search. The main idea of this reduction is to find a propositional inconsistency. If such inconsistency is found there is no need to translate or search for a model, and as it can be seen in Section 4.1, the implementation of both approaches indicate if the inconsistency is propositional and not related to the information in the KB.

As we can see, the reductions shown here help to reduce the size of a Boolean KB deleting the unnecessary information. They are specially important because they can reduce the number of difficult items to translate, specially in the case of the unit propagation, when we have a unit clause with $R(i, j)$, and the same role assertion only appears negated in some non unit clauses. There are more reductions that can be constructed; an example is the transformation in NNF of all the concept assertions. Another idea related to the negated role assertions, is to recognize the cases of $\mathcal{SHIQ}$ where we can apply the role deletion. Such optimizations are left for further investigation.

In this chapter two ways to deal with a Boolean KB were explained. One of them generates a new KB from the original input KB. Such KB has different signature and can be tested in RACER to know the consistency of the input KB. The other way generates many KBs and test them to return the KB consistency; according to the DL used in the input KB, the KBs generated may or may not have the same signature.

The next chapter describes the implementations of the two approaches and the testing over them. Those implementations have already the reductions described here. The code of these reductions can be found in Appendix A.

Chapter 4

Implementation and testing

In the previous chapters we discussed two ways of handling BABoxes and some optimizations for each case. We implemented both approaches in Haskell. The first program, named `ba2racer`, executes our translation described in Section 3.1 over an input Boolean KB. `ba2racer` does not execute the ABox consistency checking by itself. It returns a file with a traditional KB that can be used as an input for RACER to check the consistency of the ABox. The second program, `ba2racerdpll` implements the algorithm based on the disjunction of knowledge bases of Section 3.2. Contrary to the first approach, `ba2racerdpll` calls internally to RACER to check the consistency of the ABox. As output, this program returns whether the input KB is consistent or not. If yes it generates a file with the first consistent traditional KB produced by the algorithm. The code of both programs can be found in Appendix A.

In this chapter we give technical details and preliminary testing of the two programs mentioned. We describe the input format and how the programs operate in the next section. Afterwards, we show the tests with hand crafted cases taken from the TPTP database and comments on some mistakes found in RACER. Finally we show the tests over random generated BABoxes. A description of the random generator and the results of this testing can be found in the last section of this chapter.

4.1 The input format and the programs

We presented in Section 2.2 an intuitive idea and the formal definition of a sequent. Sequents are the core elements of BABoxes. A BABox can be seen as a standard ABox where sequents can be used instead of assertions. Thus, our knowledge bases consist of a BABox and a standard TBox. The use of RACER as our selected DL system provides the input format for all the traditional elements of a KB. Maintaining the RACER-like format except for assertions, we provide a grammar for the input files of `ba2racer` and `ba2racerdpll` in Table 4.1.

input	$\rightarrow$	
	$\mid$	*kbelem input*
kbelem	$\rightarrow$	*tbox_statement*
	$\mid$	*sequent*
tbox_statement	$\rightarrow$	`(impl` *concept concept*`)`
	$\mid$	`(equivalent` *concept concept*`)`
	$\mid$	`(disjoint` *lconcepts*`)`
	$\mid$	`(define-role` *name transitive inverse parents*`)`
transitive	$\rightarrow$	
	$\mid$	`:transitive t`
inverse	$\rightarrow$	
	$\mid$	`:inverse` *name*
parents	$\rightarrow$	
	$\mid$	`:parents (`*lnames*`)`
sequent	$\rightarrow$	*latoms* `->` *latoms* `.`
latoms	$\rightarrow$	
	$\mid$	*atom latoms*2
*latoms*2	$\rightarrow$	
	$\mid$	`,` *atom latoms*2
atom	$\rightarrow$	`(ins` *name concept*`)`
	$\mid$	`(rel` *name name name*`)`
lconcepts	$\rightarrow$	*concept*
	$\mid$	*concept lconcepts*
concept	$\rightarrow$	*name*
	$\mid$	`*top*`
	$\mid$	`*bottom*`
	$\mid$	`(and` *lconcepts*`)`
	$\mid$	`(or` *lconcepts*`)`
	$\mid$	`(not` *concept*`)`
	$\mid$	`(some` *name concept*`)`
	$\mid$	`(all` *name concept*`)`
	$\mid$	`(some` *name concept*`)`
	$\mid$	`(at-most` *num name concept*`)`
	$\mid$	`(at-least` *num name concept*`)`
lnames	$\rightarrow$	
	$\mid$	*name lnames*
name	$\rightarrow$	$[a,..,z,A,..,Z][a,..,z,A,..,Z,0,..,9,\_,-]^*$
num	$\rightarrow$	$[0,..,9]^+$

Table 4.1: The input format for Boolean KBs

The input format allows us to use the DL $\mathcal{SHIQ}$, that is, a logic supported by RACER. Notice that the declaration of a concept assertion is made with the word `ins` instead of the RACER keyword `instance` and we use `rel` instead of `related` for the role assertions. We use these abbreviations to facilitate the writing and reading of the BABox.

The programs `ba2racer` and `ba2racerdpll` can have the same file as input but the output they produce is different. Let us explain the peculiarities of each one.

An example of a typical run for `ba2racer` is:

```
bash-2.04$ ./ba2racer test.txt
```

`test.txt` is a file containing a Boolean KB in the format described in Table 4.1. `ba2racer` transforms the Boolean KB of the file `test.txt` in a traditional KB and places it in the file `test.txt.racer` in RACER format. It adds the line `(abox-consistent?)` at the end of the file, to indicate RACER that the ABox consistency must be checked. Afterwards, we need to feed RACER with `test.txt.racer` to know if the Boolean KB in `test.txt` is ABox consistent. Let us show now the call to RACER for the example.

```
bash-2.04$ racer -f test.txt.racer
;;; RACER Version 1.7
;;; RACER: Reasoner for Aboxes and Concept Expressions Renamed
;;; Supported description logic: ALCQHIr+(D)-
;;; Copyright (C) 1998-2002, Volker Haarslev and Ralf Moeller.
;;; RACER comes with ABSOLUTELY NO WARRANTY; use at your own risk.
;;; Commercial use is prohibited; contact the authors for licensing.
;;; RACER is running on unknown computer as node Unknown

;;; The XML/RDF/RDFS/DAML parser is implemented with Wilbur developed
;;; by Ora Lassila. For more information on Wilbur see
;;; http://wilbur-rdf.sourceforge.net/.

;;; The HTTP interface based on DIG is implemented with CL-HTTP developed
;;; and owned by John C. Mallery. For more information on CL-HTTP see
;;; http://www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html.

(IN-KNOWLEDGE-BASE TEST.TXTABOX TEST.TXTTBOX) -->
(TEST.TXTABOX TEST.TXTTBOX)
(ABOX-CONSISTENT?) --> T
```

The output of RACER `(ABOX-CONSISTENT?) --> T` indicates that the KB of `test.txt.racer` is consistent, and as a consequence of the claims of Section 3.1 the Boolean KB of `test.txt` is consistent too. An output `(ABOX-CONSISTENT ?) --> NIL` will imply that both KBs are inconsistent.

It can occur that the input Boolean KB is propositionally inconsistent. In this case `ba2racer` displays `ABox not consistent`. No further checking with RACER is needed. The output file will contain an empty KB.

For the case of `ba2racerdpll`, an example of a typical run is:

```
bash-2.04$ ./ba2racerdpll test.txt
ABox consistent, file .racer created
```

Similarly to the example of `ba2racer`, the input file `test.txt` has a Boolean KB in our described format. The output of `ba2racerdpll` indicates whether `test.txt` is ABox consistent or not displaying `ABox consistent` or `ABox not consistent` respectively. There is no need to run RACER because `ba2racerdpll` has already run it internally.

Every time `ba2racerdpll` finds a propositional model, it places the generated KB in a file and calls RACER with it as parameter. The file used to store the generated KB is called `<input file>.racer`. The file is rewritten every time a new consistency check is made in that run. Thus, for our example, the KB generated by `ba2racerdpll` containing the model that satisfy the disjunction of KBs in `test.txt` can be found in `test.txt.racer`. The last line of this file is `(abox-consistent?)` like in the case of `ba2racer`. If the response of `ba2racerdpll` is only `ABox not consistent` without the phrase `racer result`, it means that the input Boolean KB was propositionally inconsistent and RACER was not called. The outputted KB will be empty in this case.

After showing how the two programs can be run and their outputs, we are ready to show in the next section examples of formulations of logical puzzles and the description of some mistakes found in RACER.

4.2 Handcrafted cases

In order to get the examples to test `ba2racer` and `ba2racerdpll`, we formalized in DL some puzzles of the TPTP Problem Library [SS]. These puzzles contain situations where a representation of partial knowledge and conditioning of assertions is required. Sequents allow us to express such things. The examples were written using the grammar described in Table 4.1. We present in the following subsection some of these formalized puzzles. After the examples, we describe some problems found in RACER during their formulation.

4.2.1 TPTP puzzles

We outline some interesting formalizations for each problem in this section. The complete examples can be found in Appendix B.

Every modelled TPTP problem can be split into two parts, a description of the puzzle, followed by a conjecture which should be verified. What we first did was to formalize each problem in a KB without the conjecture. Then we checked if every KB was consistent. To prove the conjectures we negated the formalizations that expressed them and added these new formalizations to the consistent KBs. We checked the inconsistency of the ABox in every new KB to show the entailment of the conjecture by the initial KB.

The Dreadbury Mansion

The first example is the formalization of the well known puzzle of the Dreadbury mansion. This problem can be found in the TPTP library under the name PUZ001-1.p

> *Someone who lives in Dreadbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadbury Mansion, and are the only people who live therein. A killer always hates his victim, and is never richer than his victim. Charles hates no one that Aunt Agatha hates. Agatha hates everyone except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone Aunt Agatha hates. No one hates everyone. Agatha is not the butler. Therefore: Agatha killed herself.*

We explain below some of the formulations that we use for this puzzle. We use the symbols `agatha`, `butler` and `charles` to represent Agatha, the butter and Charles respectively. The role assertion `(rel a b killed)` states that `a` killed `b`; `(rel a b richer)` states that `a` is richer than `b`; `(rel a b hates)` states that `a` hates `b`.

In our formalization of the sentence *'Someone who lives in Dreadbury Mansion killed Aunt Agatha'* we use a disjunction of positive role assertions, because we know there are only three possible killers of Aunt Agatha.

```
->(rel charles agatha killed),
(rel agatha agatha killed),
(rel butler agatha killed).
```

To express the phrase *'A killer is never richer than his victim'*, we can say that if an individual killed Agatha then such individual is not richer than her. Sequents only use positive literals, so we express the previous statement with a disjunction of negative role assertions with the role `richer`. In the specific possibility of Charles being the killer we state:

```
(rel charles agatha killed),(rel charles agatha richer)->.
```

To express the full phrase we also state the cases of the butler and aunt Agatha being the killer.

The declaration *'The butler hates everyone Aunt Agatha hates'* can be rewritten as *'If Agatha hates a person then the butler hates this person'*, and it is easily described with a sequent expressing an implications of role assertions.

```
(rel agatha agatha hates) -> (rel butler agatha hates).
(rel agatha charles hates) -> (rel butler charles hates).
(rel agatha butler hates) -> (rel butler butler hates).
```

We can state *'No one hates everyone'* using the `at-most` constructor saying that everyone can hate at most 2 people.

```
->(ins agatha (at-most 2 hates *top* )).
->(ins charles (at-most 2 hates *top* )).
->(ins butler (at-most 2 hates *top* )).
```

The conjecture of the problem is *'Agatha killed herself'*. We represent the negated conjecture with a negated role assertion.

```
(rel agatha agatha killed)->.
```

When we tested the KB with the formulation of this puzzle without the conjecture, `ba2racer` created a file with 53 assertions, and RACER replied the consistency of the translated KB in 0.07[1] seconds. `ba2racerdpll` produced a consistent KB in 0.24 seconds. The number of calls to RACER from `ba2racerdpll` was 4, and the average size of the produced ABoxes was 12.

We added to the previous KB the negated conjecture and the two programs replied that the new KB was inconsistent, hence we proved the fact that aunt Agatha killed herself. `ba2racer` created a file with 49 assertions, and `ba2racerdpll` had an average of ABox size of 13 assertions. The call of RACER after running `ba2racer` last for 0.06 seconds. `ba2racerdpll` answered in 0.12 seconds after 4 calls to RACER.

The Barber Puzzle

The puzzle PUZ003-1.p of the TPTP library was already introduced in Section 2.2 and is shown again below.

> *There is a barbers' club that obeys the following three conditions:*
>
> *(1) If any member A has shaved any other member B - whether himself or another - then all members have shaved A, though not necessarily at the same time.*
>
> *(2) Four of the members are named Guido, Lorenzo, Petrucio, and Cesare.*
>
> *(3) Guido has shaved Cesare.*
>
> *Prove Petrucio has shaved Lorenzo*

We also mentioned in that section an idea for a formalization. Let us be clearer now. `guido`, `lorenzo`, `petrucio` and `cesare` are the individual names representing Guido, Lorenzo, Petrucio, and Cesare respectively. `member` is the concept name indicating the members of the barber's club. `shaved` is the role expressing the relation between barbers and shaved people.

[1]All the time values where obtained in a Linux Readhat 7.0, release 2.4.18-ict4, CPU: Pentium III 800 Mhz Memory 250 Mb.

Statement *(1)* express under what conditions a new role assertion of `shaved` can be stated. We make the formalization of this phrase with the same idea of Section 2.2. For all `A`, `B`, `C` elements of {`guido`, `lorenzo`, `petrucio`, `cesare`}, we add to the KB the following sequent:

```
(ins A member), (ins B member), (rel A B shaved),(ins C member) ->
(rel C A shaved).
```

Notice that when `A`, `B` are the same, it is not necessary to state twice `(ins A member)` in the left side of the sequent.

To express statement *(2)*, we use positive concept assertions of `member`.

```
-> (ins cesare member).
-> (ins lorenzo member).
-> (ins petruchio member).
-> (ins cesare member).
```

Sentence *(3)* is represented as a positive role assertion.

```
-> (rel guido cesare shaved).
```

The conjecture *'Petrucio has shaved Lorenzo'* can be represented with a role assertion. The negated conjecture is shown below.

```
(rel petrucio lorenzo shaved)->.
```

We run the two programs with the file containing the formalization of this problem without the conjecture. We obtained that `ba2racer` generated a RACER file with 61 assertions. When we feed RACER with the output file, it answered that the translated KB was ABox consistent in 0.04 seconds. The file produced by `ba2racerdpll` had 11 assertions and RACER was called only once. `ba2racerdpll` replied in 0.06 seconds.

In the case of the KB with the conjecture `ba2racer` and `ba2racerdpll` detected that it was propositionally inconsistent, so no call to RACER was needed in both cases.

The Mislabelled Boxes

The puzzle of the mislabelled boxes can be found in the TPTP library as PUZ012-1.p. The problem states:

> *There are three boxes a, b, and c on a table. Each box contains apples or bananas or oranges. No two boxes contain the same thing. Each box has a label that says it contains apples or says it contains bananas or says it contains oranges. No box contains what it says on its label. The label on box a says "apples". The label on box b says "oranges". The label on box c says "bananas". You pick up box b and it contains apples. What do the other two boxes contain?*

We use in this formalization the individual names `apples`, `oranges`, `bananas` to represent apples, oranges and bananas respectively. `boxa`, `boxb`, `boxc` are the individual names representing the box a, box b and box c respectively. We use two roles, `contains` and `label` which establish the relation between a box and its contents, and a box and its label.

For the phrase *'Each box contains apples or bananas or oranges'* we have to state a disjunction of the things each box can contain. For example for the box a we have:

```
->(rel boxa apples contains),(rel boxa oranges contains),
(rel boxa bananas contains).
```

We add the same statement for the box b and the box c into the KB to express the whole sentence.

The sentence *'No box contains what it says on its label'* can be rewritten as: if a box has a label l then the box does not contain l. The formalization for the particular case of apples is shown as follows.

```
(rel boxa apples label),(rel boxa apples contains)->.
(rel boxb apples label),(rel boxb apples contains)->.
(rel boxc apples label),(rel boxc apples contains)->.
```

A similar idea is used to express that *'No two boxes contain the same thing.'*.

We want to prove that the box a contains bananas and the box c contains oranges. We express the negated conjecture as follows:

```
(rel boxa bananas contains),(rel boxc oranges contains)->.
```

When we tested the KB containing the formalization without the conjecture, `ba2racer` generates 42 assertions and `ba2racerdpll` 12 assertions for the only call to RACER. When RACER is called after generating the translated file with `ba2racer`, it replied in 0.04 seconds that the KB is ABox consistent. `ba2racerdpll` answers after 0.04 seconds the same thing.

The KB with the conjecture is propositionally inconsistent and is detected in both cases.

As we can see, in the last two cases the conjecture followed simply by propositional reasoning. Examples where modal reasoning is involved would be more interesting. That was an aim for the testing over random BABoxes which are described in Section 4.3. The initial tests over these puzzles were relevant because we discovered some mistakes present in RACER. They are described as follows.

4.2.2 Mistakes in RACER

In the initial faces of the formalization of the puzzles of the TPTP we ran into problems with RACER. In some cases RACER returned unsound answers and in others RACER broke down in an unexpected stack overflow. These things happened when we used the translation approach to check the consistency problem. We illustrate in this section examples of the BABoxes with the problems.

Unsound behavior

The first problem was that RACER answered that a KB was not consistent when in fact there was a model for the KB. The following example represents one of the cases where RACER gave an incorrect answer.

The example was found in the formulation of the problem PUZ001-1[2] in a consistent Boolean KB. It was rewritten using our algorithm of Section 3.1. The translation was tested in RACER Version 1-6r1a, in a Pentium III Linux Readhat 7.0, cpu Mhz 800, Memory 250 Mb. RACER Version 1-6r1a declared the produced KB inconsistent.

To identify where did the mistake exactly lay, we reduced the BABox to the smallest set of consistent sequents whose translation was still reported as inconsistent by RACER. The result was the following BABox whose consistency is trivially verified.

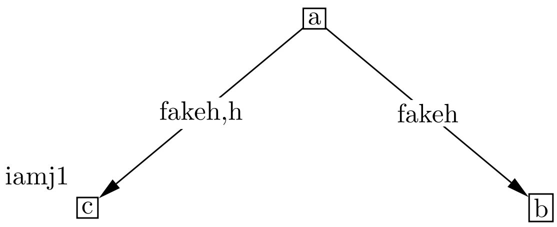
```
(rel a c h), (rel c c h) ->.
(rel a b h), (rel c b h) ->.
-> (rel a c h).
(rel a b h) ->.
```

To restrict even further we applied the same procedure to the translated BABox, obtaining the KB shown below. If any element of the final file is erased, RACER Version 1-6r1a answers that the ABox is consistent.

```
(in-knowledge-base PUZ001-1.pabox PUZ001-1.ptbox )
(signature
 :atomic-concepts(iamj1 iamj3)
 :roles(fakeh)
 :individuals(a b c))
(define-primitive-role h :parents (fakeh))
(related a c fakeh)
(related a b fakeh)
(instance a (some h iamj1))
(instance a (at-most 1 fakeh iamj1))
(instance a (at-most 1 fakeh iamj3))
(instance c iamj1)
```

A model for the previous KB is represented in the following graph:

[2] The complete formulation can be found in Appendix B.



When we tested the previous KB in RACER, it returned

```
;;; RACER Version 1-6r1a
;;; RACER: Reasoner for Aboxes and Concept Expressions Renamed
;;; Supported description logic: ALCQHIr+(D)-
;;; Copyright (C) 1998-2001, Volker Haarslev and Ralf Moeller.
;;; RACER comes with ABSOLUTELY NO WARRANTY; use at your own risk.
;;; Commercial use is prohibited; contact the authors for licensing.

(IN-KNOWLEDGE-BASE PUZ001-1.PABOX PUZ001-1.PTBOX) -->
(PUZ001-1.PABOX PUZ001-1.PTBOX) (ABOX-CONSISTENT?) --> NIL
```

This mistake was found in other similar KBs. RACER is widely used, and it was important to provide this information to the RACER team.

The next problem was also sent to the RACER team. It does not have to do with the soundness of RACER. In this case RACER reaches a stack overflow with a small KB.

Stack overflow

The second problem appeared when RACER did not report the consistency of a small KB. The information of the output pointed to a stack overflow, but as we mentioned before the KBs where the problem was found were too small to produce such stack overflow. The following example of this error is part of the specification in a Boolean KB of puzzle PUZ0019-1[3] of the TPTP. It was tested and reduced under the same conditions of the first one. The resulting Boolean KB was:

```
->(rel b r h),(rel b th h),(rel b pe h),(rel b s h).
```

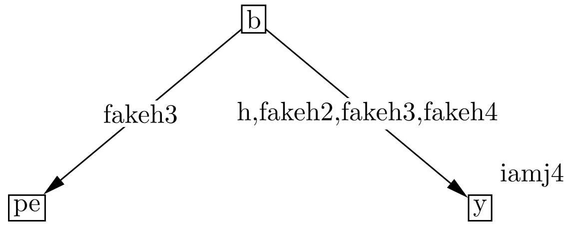
This BABox was rewritten using the translation of [ABM02]. The translation was also reduced to obtain the minimal KB where the problem existed.

```
(in-knowledge-base PUZ019-1.pabox PUZ019-1.ptbox )
(signature
 :atomic-concepts(iamj2 iamj3 iamj4)
 :roles(fake2 fake3 fake4)
```

[3]Not present in Appendix B due to its big size

```
 :individuals(b pe))
(define-primitive-role h :parents (fake2 fake3 fake4))
(instance b (some h iamj4))
(instance b (at-most 1 fake2 iamj2))
(instance b (at-most 1 fake3 iamj3))
(instance b (at-most 1 fake4 iamj4))
(related b pe fake3)
(abox-consistent?)
```

We also show in this case a model of the KB in the following graph.



The output of RACER when we feed it with the KB was:

```
;;; RACER Version 1-6r1a
;;; RACER: Reasoner for Aboxes and Concept Expressions Renamed
;;; Supported description logic: ALCQHIr+(D)-
;;; Copyright (C) 1998-2001, Volker Haarslev and Ralf Moeller.
;;; RACER comes with ABSOLUTELY NO WARRANTY; use at your own risk.
;;; Commercial use is prohibited; contact the authors for licensing.

(IN-KNOWLEDGE-BASE PUZ019-1.PABOX PUZ019-1.PTBOX) -->
(PUZ019-1.PABOX PUZ019-1.PTBOX)
Error: An error of typeCONDITIONS:STACK-OVERFLOW occured,
arguments : COMMON-LISP:NIL
```

The RACER team received the details of the problems we found and they promptly released the version 1.7 of RACER where the previous problems were not present.

There was a last problem discovered during the random generation testing described in the next section. Initially we wanted to test programs with $\mathcal{ALCQ}$ KBs. However there is a mistake in version 1.7 of RACER concerned with the use of the $\mathcal{Q}$ constructors in the BABox. Although the translation uses one of the $\mathcal{Q}$ constructors ($\leq nR.C$), the mistake only arises when there are several concept assertions using the $\mathcal{Q}$ constructors and $n \geq 2$. RACER returns unsound answers in some of these cases. The examples are rather large and complicated to be shown here, however it is important to notice that this problem forced us to use only $\mathcal{ALC}$ in the generation of BABoxes described in the next section. The RACER team has already received a report of this.

4.3 Random generated cases

The problem with handcrafted tests is that they are hardly ‘exhaustive’. For more intensive testing we need automatic generation of BABoxes. As our research is novel there was no random generator of BABoxes previously available, so we implemented in Haskell a random generator of $\mathcal{ALC}$ BABoxes called `genBABox`. We used this generator to test both programs over large sets of BABoxes. Such tests are explained in the second part of this section.

4.3.1 The random generator

In this subsection we explain how the BABoxes generator, `genBABox`, operates. First we show some samples of typical runs of the generator.

The run showed below exemplify how to obtain help about the desired input parameters for the generator. Every time the program finds that the parameters are insufficient or they have an incorrect type this output will be produced.

```
bash-2.04$ ./genBABox
genBABox filename #role-atoms #concept-atoms #instances #role-assertions
#concept-assertions #prob-of-modality(0..100) depth #clauses length-clause
numfiles
```

The following run fulfill the parameters required.

```
bash-2.04$ ./genBABox filename.txt 1 2 3 4 5 50 2 10 3 6
Finalized
```

When `genBABox` finishes we obtain the following files:

```
bash-2.04$ ls filename.txt.*
filename.txt.1  filename.txt.3  filename.txt.5
filename.txt.2  filename.txt.4  filename.txt.6
```

These files contain the generated BABoxes. Notice that the output files have the same name of the value in `filename` but the extension is a number between one and the value in `numfiles`.

We detail now the parameters taken by `genBABox`. The following explanation shows the abbreviations that appeared above in the first run of `genBABox` and the meaning and importance of each one. The values for the numeric parameters are expected to be positive integers. Only in some cases a parameter can take the value of 0.

- `filename`. As we saw before this parameter contains the file name that is going to be taken as base name for the output files generated.

- `#role-atoms`, `#concept-atoms`, `#instances`. These parameters indicate the number of atomic roles, number of atomic concepts and number of individuals that are used in the creation of the BABox. They define the signature, which is the same for every BABox generated in the run. The value of `#instances` can not be 0.

- `#role-assertions` and `#concept-assertions`. With these parameters we set the number of role assertions and number of concept assertions. They establish how many assertions are created every time a new BABox is generated.

- `#prob-of-modality`. This number goes from 0 to 100 and sets the probability that a generated concept has $\forall$ or $\exists$ as its main connective equal to `#prob-of-modality`/100. The probability of modality is discarded when the `depth` is set to 1.

- `depth`. This parameter defines the depth of the produced concepts. The depth cannot be set to 0. If the depth is 1 the concepts produced can be of four types: atomic concepts, negated atomic concepts, disjunction or conjunction of possibly negated atomic concepts. Disjunctions and conjunctions are in RACER format, whence they take a list of arguments. The length of the requested list is bounded by the value of `#concept-atoms`. When the depth is `n`, the concepts that can be generated are: the concepts generated with depth `n-1`, and the concepts produced by the application of the constructors ($\sqcup$, $\sqcap$, $\exists$, $\forall$) to the concepts generated with depth `n-1`. The constructors $\exists$ and $\forall$ are used only when `#prob-of-modality` is bigger than 0.

  The detailed explanation of the creation of a concept, and the use of the probability of modality and the depth is given below under **The creation of assertions**.

- `#clauses`. Initially the generator works with clauses and then it writes them as sequents. The value in the parameter `#clauses` indicates the number of clauses (sequents) that every BABox generated in the run will have.

- `length-clause`. The clauses have always the same length and this is set with this parameter. The length of clause can not be set to 0.

- `numfiles`. The generator can output several BABoxes in a run. This number declares how many BABoxes are going to be generated. This parameter can not be set to 0.

The process of generation of BABoxes runs a generator of ONE BABox `numfiles` times. This generator of one BABox recieves all the parameters of `genBABox` and one extra parameter indicating the number of call, between 1 and `numfiles`. We refer to this number as `call`. The generator for one BABox follows these steps.

1 Create role assertions and concept assertions.

2 Create a propositionally satisfiable set of ABox clauses.

3 Write a file with the BABox.

We explain these steps below. We use the abbreviations to represent the values of the parameters received in the run.

The creation of the assertions

The creation of every assertion is made randomly. First we create `#role-atoms` role atoms, `#concept-atoms` concept atoms, and `#instances` instances. Then we create the assertions using these elements.

The creation of a role assertion is the easiest, it is made by choosing an atomic role and a pair of individuals randomly. This process is repeated `#role-assertions` times.

The case of concept assertions is more difficult. We first explain the creation of a single random concept in the following pseudocode of the function `genConcept`. The parameters expected by this function are two, denoting depth of the concept and probability of modality.

We assume the existence of the following functions: `genRnd` that takes 2 integers i, j as input and returns a random integer that is $\geq i$ and $\leq j$; `getRoleAtom` and `getConceptAtom` which return a randomly chosen atom of the list of role or concept atoms respectively; and `genSimpleConcept` which returns a 'simple' concept, meaning with this a concept that is either a possibly negated concept atom or a disjunction or conjunction of possibly negated concept atoms.

```
genConcept(depth, mod)
{
    if depth == 1 or mod == 0 then
        genSimpleConcept;
    else
    {
        pro := genRnd 1 100;
        if pro =< mod then
        {
            opt := genRnd 1 2;
            case opt of
                1:  genSomeConcept(depth-1,mod);
                2:  genAllConcept(depth-1,mod);
        }
        else
        {
            opt := genRnd 1 4;
            case opt of
                1:  getConceptAtom;
                2:  {c:=getConceptAtom; return (Not c);}
                3:  genAndConcept (depth-1,mod);
                4:  genOrConcept (depth-1,mod);
```

```
        }
    }
}
```

The function `genConcept` does not use the probability of modality `mod` if it is 0 or if the depth `depth` is 1, otherwise it will create a modal concept ($\forall, \exists$) with a probability of `mod` out of 100. The non modal cases include the atomic concept, the negated atomic concept, and the disjunction and conjunction of concepts.

In the case of the modalities, disjunction and conjunction `genConcept` calls functions where `depth` and `mod` are passed as parameter. This is done because these functions require the elaboration of more concepts, meaning with this more recursive calls to `genConcept`. We explain later when these recursive calls to `genConcept` occur.

Notice that the depth was diminished in one unit when `genSomeConcept`, `genAllConcept`, `genAndConcept`, `genOrConcept` were called. This ensure the termination of the concept construction because when the depth is equal to 1, `genConcept` creates only simple concept assertions using `genSimpleConcept` and the recursion ends.

To illustrate the role of the depth in `genConcept` we give some examples of the concepts that can be produced in a call. Suppose `genConcept` has an input depth of 1. In this case `genConcept` discards the probability of modality and calls `genSimpleConcept`. Assume we have three atomic concepts `C1`, `C2`, `C3`. We show below some examples in RACER format of the concepts that `genConcept` can create under these conditions:

```
C3
(not C1)
(and C1 (not C2))
(or C1 (not C2) C3)
```

If the value of the input depth of `genConcept` is 2, the value of the probability of modality is bigger than 0, there is one atomic role `R` and the same atomic concepts; we can expect `genConcept` to produce concept like the previously mentioned concepts and like the following:

```
(not (and C1 C2))
(exists R C1)
(all R (and C1 (not C2)))
```

A bigger depth allows `genConcept` to create longer and more complex concepts. This will be done in recursive calls to `genConcept` in the functions that use the constructors to create concepts. We explain them in the following paragraphs.

The functions `genSomeConcept` and `genAllConcept` are similar, they only change in the operator to construct the concepts, which are `exists` and `all` respectively. These functions obtain randomly an atomic role, and call `genConcept` to get a concept. Then they apply the constructor to create the final modal concept. We only describe `genSomeConcept` in the next pseudocode.

```
genSomeConcept(depth,mod)
{
    role := getRoleAtom;
    conc := genConcept(depth,mod);
    return (exist role conc);
}
```

The functions `genAndConcept` and `genOrConcept` work similarly to the previous functions, the difference is that they call several times to `genConcept` by means of the function `genListConcept`. We assume the existence of this function. `genListConcept` receives as input three numbers and returns a list of concepts. The first number indicates the depth, the second the probability of modality, and the third the length of the list. They are used to create the concepts of the list. The third number sets the length of the list. We also assume the existence of the function `getLength` which returns a random number between 1 and the value in `#concept-atoms`.

Due to the similarity between `genAndConcept` and `genOrConcept` we only reproduce the pseudocode for `genAndConcept`. In the case of `genOrConcept` we use the constructor `or` instead of `and`.

```
genAndConcept(depth,mod)
{
    length := getLength;
    list := genListConcept(depth,mod,length);
    return (and list)
}
```

To create a concept assertion we first generate a concept with the function `genConcept(depth,mod)` where `depth` is the input depth of the program and `mod` is the input probability of modality. Then we instantiate this concept to an individual chosen randomly. We repeat the process `#concept-assertions` times.

Now that we have a set of assertions we are ready to build our set of clauses.

The set of assertional clauses

The generator uses a satisfiable propositional set of clauses to build the set of assertional clauses. It uses `mkcnf` written by Allen van Gelder to produce this propositional set. In the call to `mkcnf` the parameter `force` is set, compelling it to give a satisfiable propositional clause set. We do this because we do not want unsatisfiable sets that will never be tested in RACER by `ba2racerdpll`. The number of propositional variables used in `mkcnf` is set to the total number of assertions.

`mkcnf` returns a set of clauses where the propositions are represented as numbers. The numbers go from 1 to the total number of propositions. A negative number represents the negation of a proposition.

The generator enumerate the assertions and using this enumeration, it substitutes the propositions in the clause set by the assertions. A negated proposition is substituted by the negation of the assertion.

The final output

So far, the generator has produced a set of assertional clauses which must be transformed into sequents. The program applies Definition 6 repeatedly to the set of assertional clauses. This set now can be handled by the programs described in Section 4.1. The final step is to write the set of sequents into a file. The file name is built using `filename` and the value in `call` is used as the extension of the file name.

We now have a tool to create random BABoxes. The wide number of parameters allows us to generate many interesting BABoxes for the tests of the next section.

4.3.2 The testing

In this section we show the tests of the two programs for handling BABoxes `ba2racer` and `ba2racerdpll` over random BABoxes. Given that we only had two weeks (three weekends) for the testing, we organized the test in three parts. First we had an exploratory part, then we made more refined testing and with the results of these we designed the final tests. The major tests for each part were mainly executed during the weekends. The modifications of the programs and some minor tests where made during the week days.

Before we started the tests, we created some scripts to generate several BABoxes and return the analysis of the testing of the two programs. We measured variables like: the median of the time spent to solve the consistency problem and the percentage of consistent BABoxes found. In the case of `ba2racer` we did not measure the translation time, we were interested in the time RACER spent to give an answer about the consistency of the translated BABox.

The main script allow us to test over a range of number of clauses under the same parameters of BABox generation. The exploration of this range is made in a number of points defined by an interval. We can also indicate how many BABoxes will be checked in every point and a timeout for the responses. The rest of the scripts realize smaller tasks and they were mainly used in the exploratory part of the testing.

The exploratory part lasted one weekend and we detected errors in the random generation. The first mistake was the incorrect seed initialization in the calls to `mkcnf`. We also found a mistake related to the random generation inside of `genBABox`. Originally the program `genBABox` generated only one BABox and multiple calls to `genBABox` were used to generate the required number of instances. We used the global random number generator of Haskell in all the random tasks. According to [J^+99] this random number generator is initialized automatically in some system dependant fashion. The problem was that sometimes the generation was too fast and the same number was taken several times as initializer. These two problems made `genBABox` produce several copies of the same BABox. The generated data was not fully random, but we could still use it. We fixed the problems modifying `genBABox` to initialize correctly the seed in the call to `mkcnf` and providing the generation of multiple BABoxes with only one call to `genBABox`.

The results of these tests let us tune and define which part of the test space we could handle in the available time. Even with the excessive redundancy produced by the problems described above we obtained some hints about where shall we direct our tests. We detected some spaces with particular behaviors for each program.

The refined tests gave us hints about where to focus the final part of the testing. We defined the range where we could find a variation from satisfiable to unsatisfiable BABoxes. We found some cases where we needed to know how many times `ba2racerdpll` was calling RACER and the size of the produced ABox. We made small modifications to the programs and the scripts to adjust to these requirements. After the modifications `ba2racer` and `ba2racerdpll` gave us a comment in the output file the size of the ABox and the number of calls to RACER. In the case of `ba2racerdpll`, we were more interested in the average size of all the ABoxes checked than in the particular size of the last ABox emitted. Then, `ba2racerdpll` calculates this average and prints it as the size of the ABox in every generated file. This way we have data even in the case `ba2racerdpll` gets a timeout.

In some of these tests we observed that `ba2racerdpll` was much faster than RACER with the output files of `ba2racer`. `ba2racerdpll` performed better where the BABoxes tended to be merely propositional, and the consistency or inconsistency of them was found after a few tries. `ba2racer` instead, had a good performance when the information in the BABoxes was modally constrained.

We expected that in the final tests `ba2racerdpll` would have a good performance in BABoxes with propositional inconsistencies, a small number of propositional models, or with many modal models. This last case is given when the information of the BABoxes is not modally constrained.

For the case of `ba2racer`, we considered that it would handle better the cases where a modal model was difficult to search, this is, when the information stored in the BABoxes has many modal constraints hypotheses, having many propositional models, but a few modal models or maybe none.

The final tests were designed to check the two hypothesis based on the previous tests. They focused on showing ranges where we could find a variation from satisfiability to unsatisfiability and the particular performances of each program.

The following parameters: 1 atomic role, 3 atomic concepts, 5 individuals, 10 concept assertions, 10 role assertions, a probability of modality of 50, a depth of 3, a length of clause of 3, 80 examples in every check point, timeout of 30 seconds; will be called *the initial configuration* for the tests. In the following tests we keep most of the previous parameters fixed and only vary one or two of them. This initial configuration plays with a small number of role assertions, and the following tests pretend to increment the modal constrains between the individuals increasing the number of concept assertions and roles. We set the depth of 3 to allow concepts like $\forall S.\exists S.C$. We tested seven different configurations.

Every test has figures where we represent the time of the run of RACER over the translated BABoxes produced by `ba2racer` as Ttime and the time for `ba2racerdpll` as Dtime. These points are the median of all the times obtained for each case. The percentage of satisfiable instances is represented as Tsat for the translation approach and Dsat for the other approach. The differences between Tsat and Dsat are due to time outs. In all the cases when both approaches found an answer (sat or unsat) the answers coincided. The average of number of calls to RACER inside `ba2racerdpll` is shown in the variable Dcalls. The average of the size of the BABoxes is represented in Tsize and Dsize.

In the **first** test, shown in Figure 4.1, the range was set from 40 to 150 clauses and we established 23 measuring points with an interval of 5 and. We can observe that the curve representing the proportion of satisfiable BABoxes goes from a high amount to a small amount of consistent BABoxes. It is notable that `ba2racerdpll` worked faster than the translation approach. The size of the ABoxes in `ba2racerdpll` did not variate much along the test, however the ABoxes produced by the translation grew significantly. Notice that RACER over the translated files increased the time spent in the consistency checking in the critic region where the Tsat curve changes from highly satisfiable to highly non satisfiable.

The **second** test shown in Figure 4.2, took the same parameters of the first one except the number of atomic roles. We increased the number of role atoms in one unit to show that the same behavior remains even for a configuration with two atomic roles. This configuration is more likely to occur in a real formalization than one with only one atomic role. We used a range from 10 to 200 with an interval of 10. The curve from satisfiable to unsatisfiable here is smother, after 150 clauses is unlikely to find satisfiable BABoxes but still we find some.

`ba2racerdpll` showed again a good performance. We can see how the number of calls to RACER is reduced as the number of clauses increases showing that it is easy for `ba2racerdpll` to detect a propositionally inconsistent BABox because with more than 100 clauses there is only one propositional model.

So far we have only seen a nice behavior from the side of `ba2racerdpll`. For the **third** and **fourth** tests in Figures 4.3 and 4.4, we will see a radical change. The third test changes the initial configuration adding 10 more concept assertions for a total of 20. The range of clauses is also expanded to 20-150 with

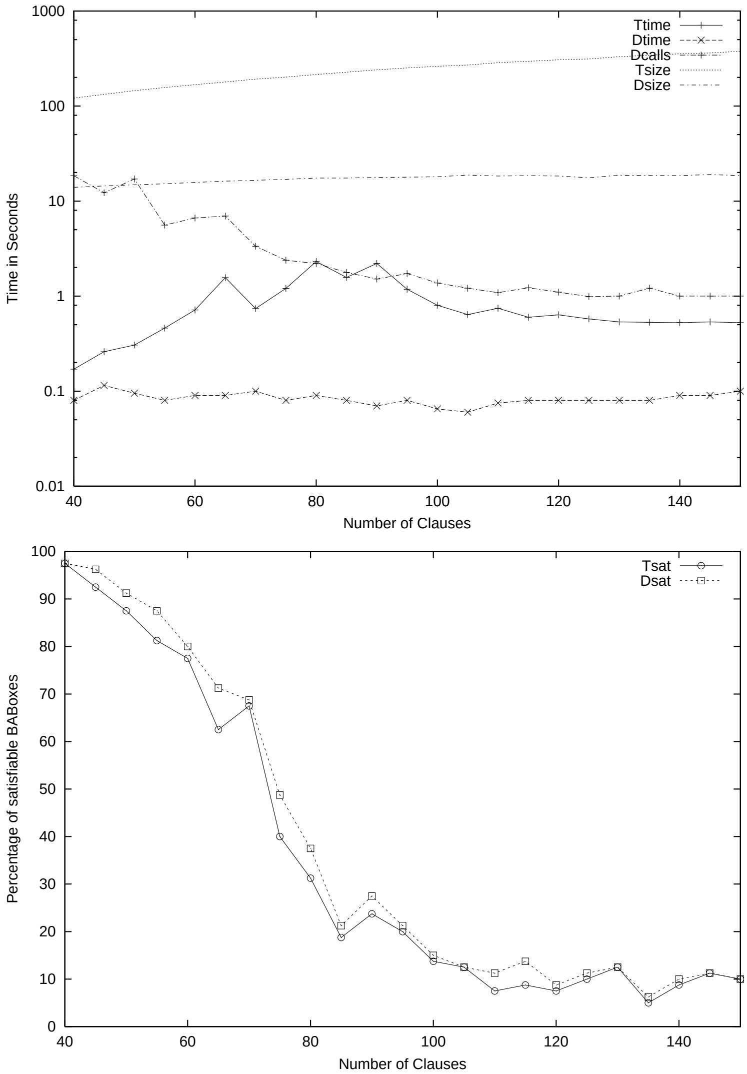


Figure 4.1: Test with 1 atomic role and 10 concept assertions

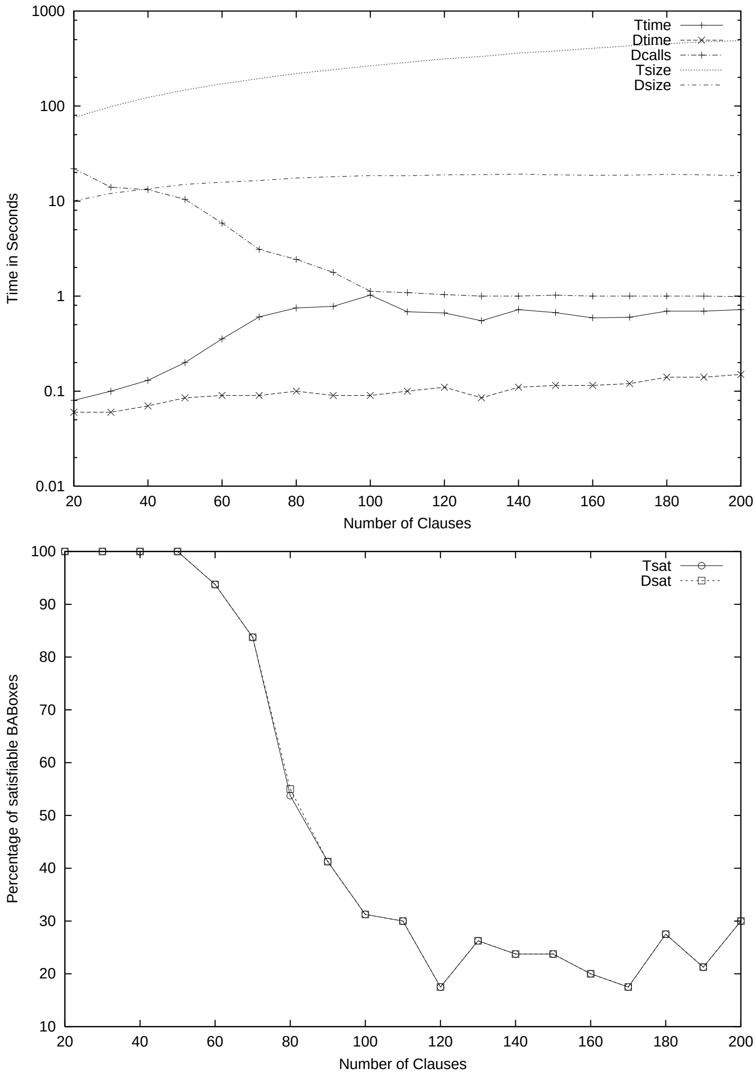


Figure 4.2: Test with 2 atomic roles and 10 concept assertions

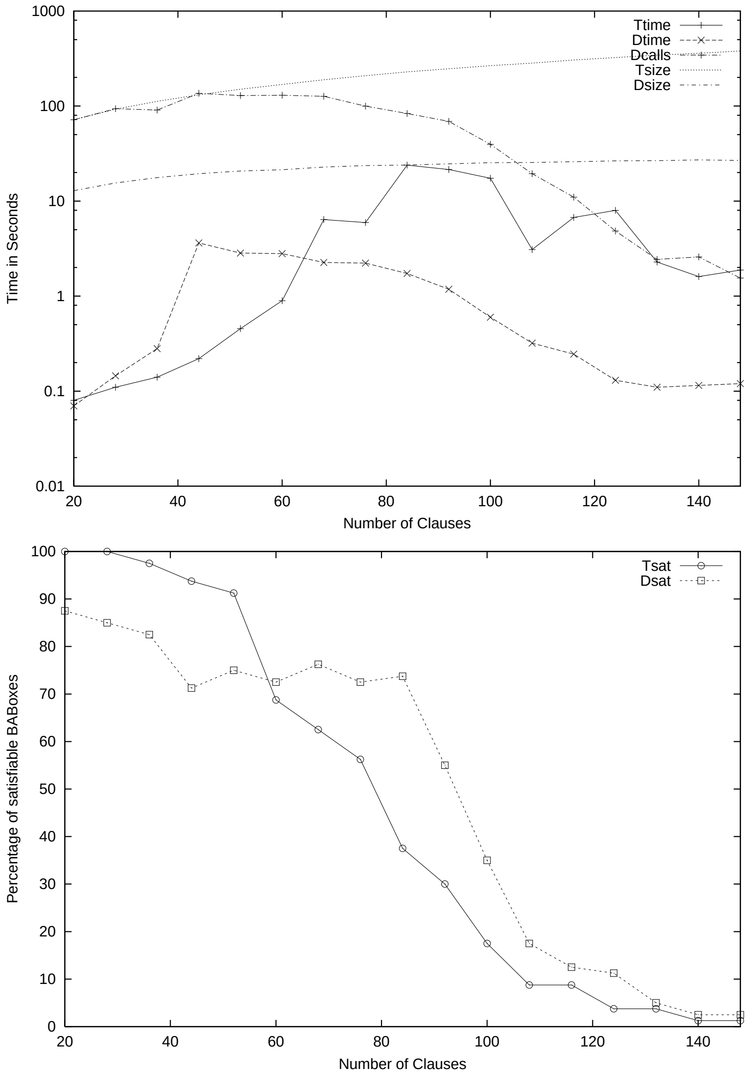


Figure 4.3: Test with 1 atomic role and 20 concept assertions

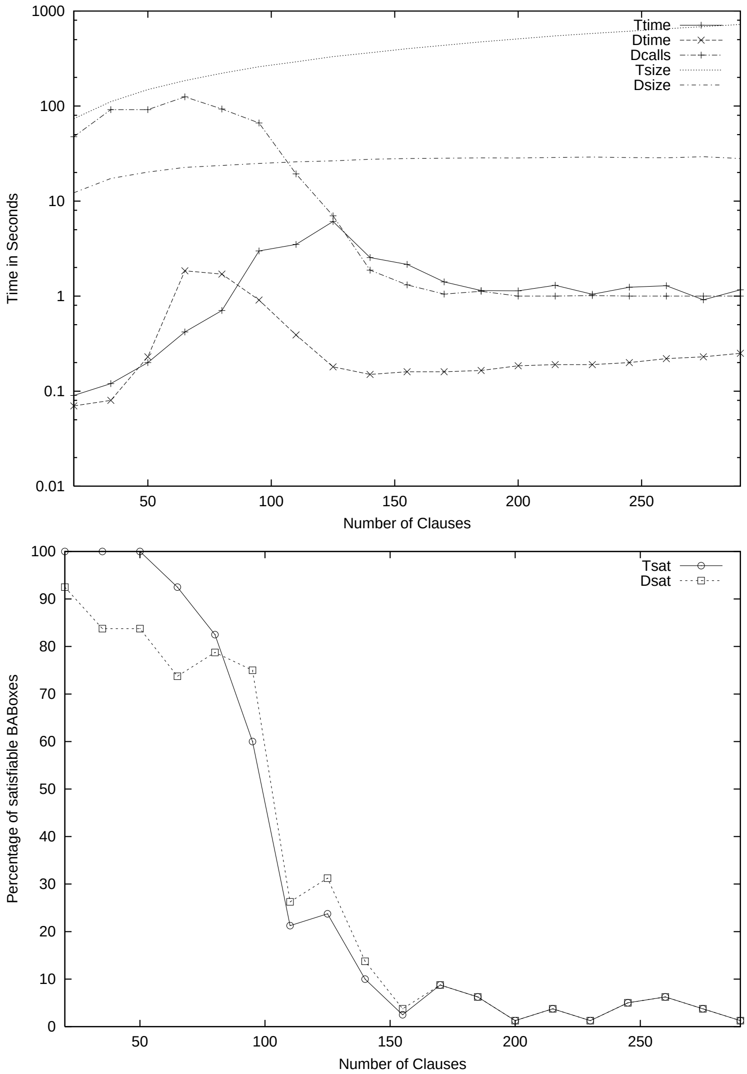


Figure 4.4: Test with 2 atomic roles and 20 concept assertions

an interval of 10. The fourth test keep the configuration of the third test but uses 2 atomic roles and a range of number of clauses of 20-300 with an interval of 20.

In these two tests we can appreciate that with a small number of clauses there are occasions where the translation approach had a better performance than `ba2racerdpll`. There are more timeouts than in the previous tests so we can see a gap between Tsat and Dsat mainly in Figure 4.3.

We increased the number of concept assertions to 30 in the **fifth** test, shown in Figure 4.5. We increased the timeout to 100 seconds in order to minimize the number of timeouts an we had to reduce the number of BABoxes per data point to 50 to make the tests set tractable. The rest of the configuration was like the initial configuration. The range of the number of clauses was from 20 to 150 with an interval of 5.

We can see in the Figure 4.5 the notable increment on the calls to RACER made by `ba2racerdpll`. The translation approach had a good performance until the critic region, then it only reached timeouts. On the contrary, `ba2racerdpll` recovers its good behavior in the unsatisfiable part of the graphic, this was due to the decrease of number of available propositional models and its consequents calls to RACER, as the size of the average KB did not show a big increase. In the case of the translation the size of the average KB kept growing with the size of the BABoxes.

In order to observe the progress of both programs we tested the initial configuration with 70 concept assertions. The critic region was very difficult to handle for both approaches, so we partitioned the **sixth** test in two subtests. We used the range 20-100 and an interval of 10 for the satisfiable part, and 280-450 with an interval of 10 for the unsatisfiable section. Figure 4.6 belongs to the satisfiable part of the test and shows how bad is the performance of `ba2racerdpll` compared to the translation approach. The size of the ABoxes produced by `ba2racerdpll` grew significantly, as well as the number of calls to RACER. `ba2racerdpll` reached a timeout in more than a half of the examples when the number of clauses was bigger than 30. We can see with these data that `ba2racerdpll` is not able to handle very well satisfiable BABoxes which are modally constrained because it has to check too many propositional models. There were many timeouts in `ba2racerdpll`, so there is a big difference between its satisfiability curve and satisfiability curve of the translation approach. The unsatisfiable side of the test is shown in Figure 4.7. Here `ba2racerdpll` had a good performance but not with a big difference of the performance of the translation approach.

The **seventh** test followed the idea of test number six, partitioning the test space in two to avoid the critical region. The Figures 4.8 and 4.9 schematize the tests with the initial configuration but with 2 role atoms and 70 concepts. The ranges of number of clauses where set to 20-150 and 290-450 for the satisfiable and unsatisfiable section respectively. The interval in both cases was 10. The

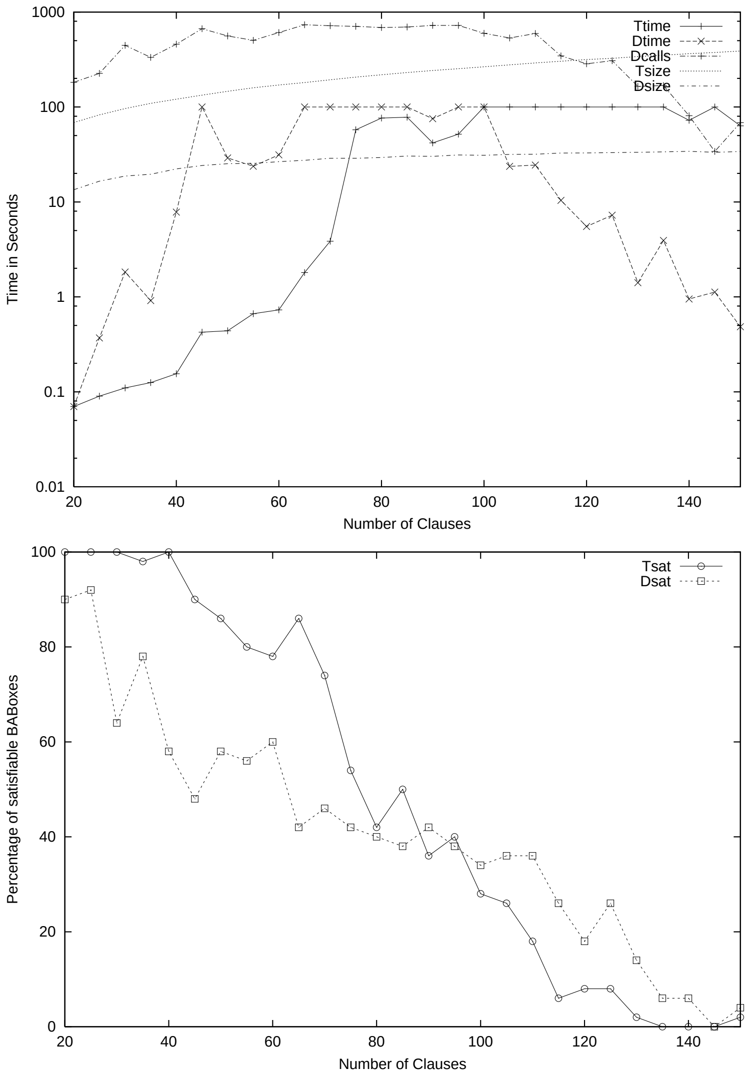


Figure 4.5: Test with 1 atomic role, 30 concept assertions, 50 BABoxes and 100 seconds of timeout

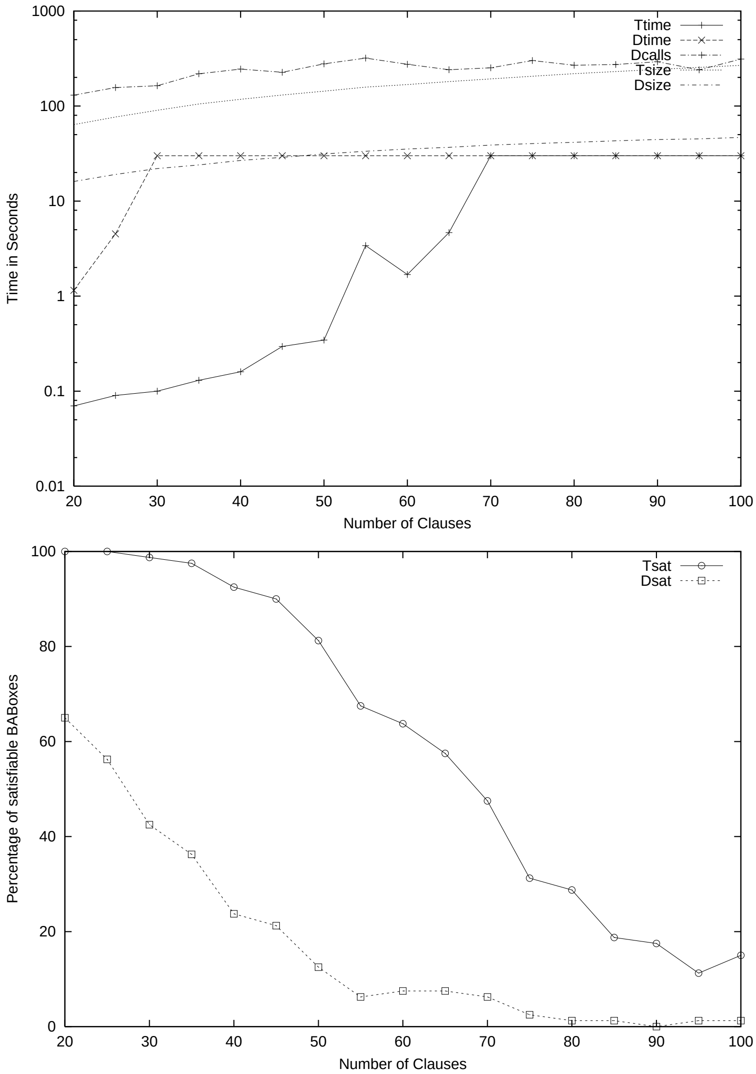


Figure 4.6: Test with 1 atomic role and 70 concept assertions, satisfiable part

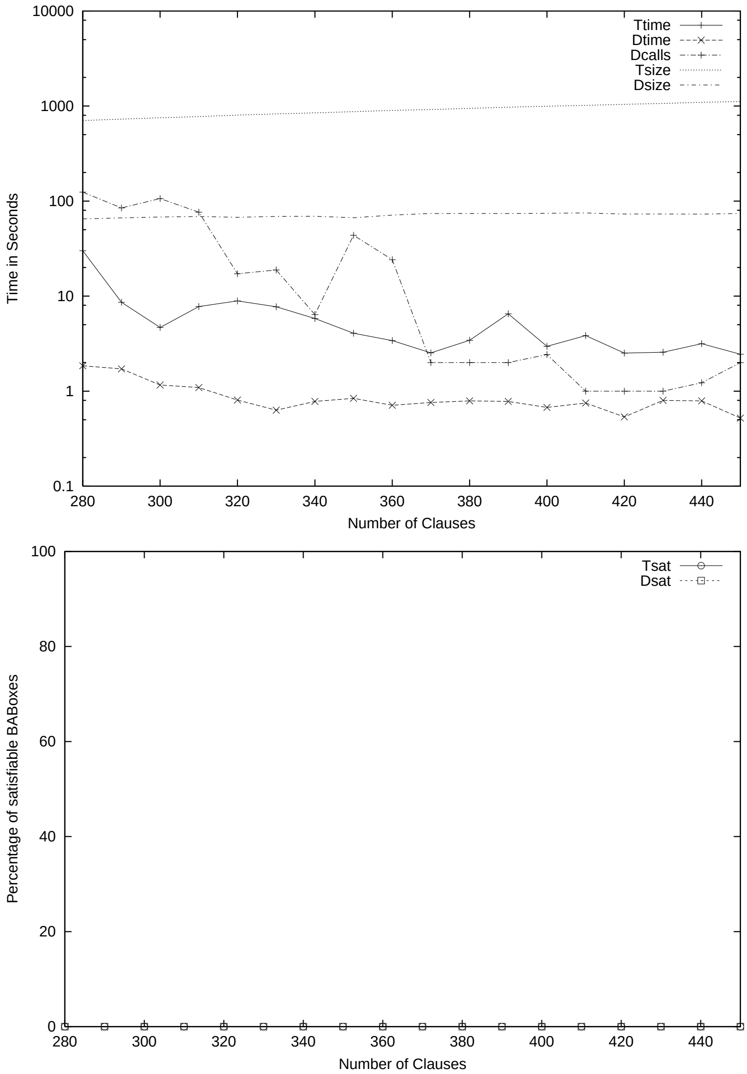


Figure 4.7: Test with 1 atomic role and 70 concept assertions, unsatisfiable part

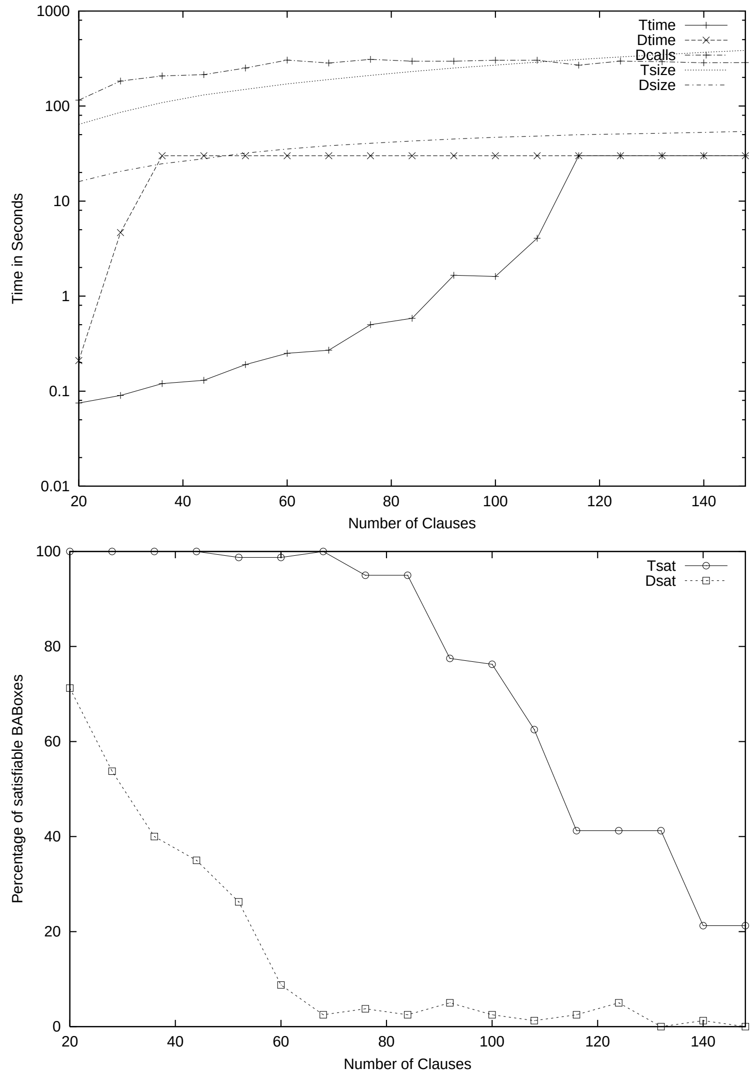


Figure 4.8: Test with 2 atomic roles and 70 concept assertions, satisfiable part

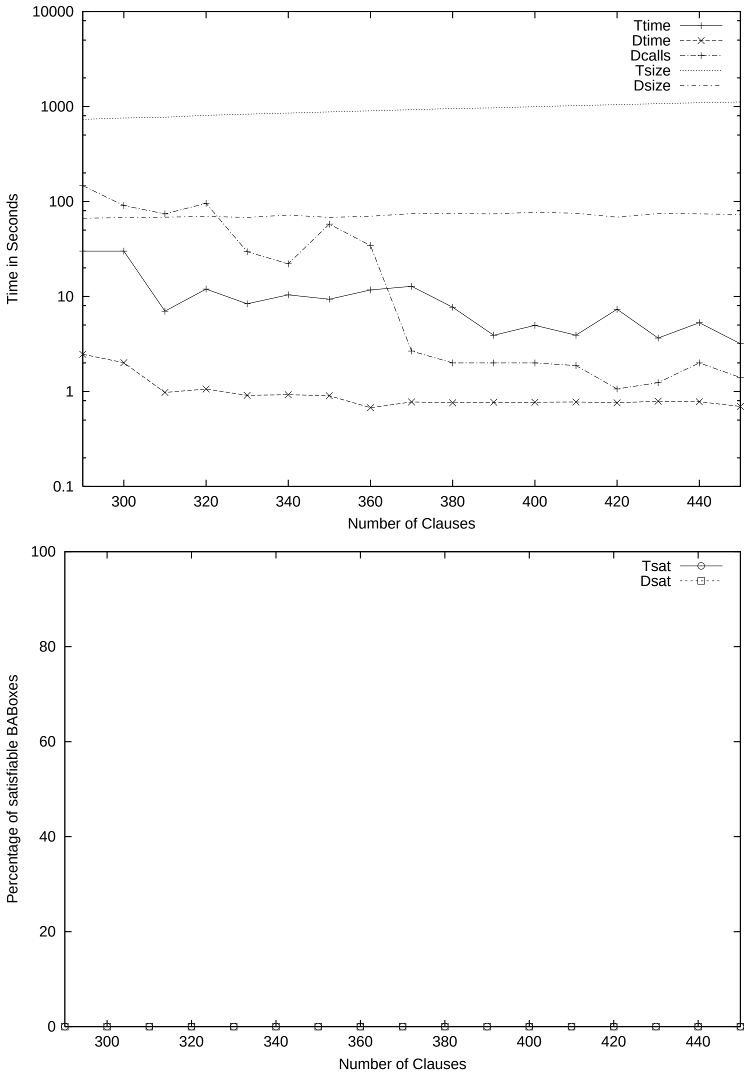


Figure 4.9: Test with 2 atomic roles and 70 concept assertions, unsatisfiable part

behavior of both programs was similar to the presented in the sixth test, and as we saw in other cases where we add one role to the configurations, we had to enlarge the ranges of the number of clauses to obtain the curves.

We have shown in this section that `ba2racerdpll` has a good performance with BABoxes that represent information modally underconstrained. `ba2racerdpll` deals very well with BABoxes that have few propositional models and are easy to declare consistent or inconsistent. On the contrary RACER does not 'see' these cases with the translation but it can handle better the modal constraints of the BABox. In this case `ba2racerdpll` spends time testing all the propositional model it finds, and as we can see in the figures this strategy leads to a timeout.

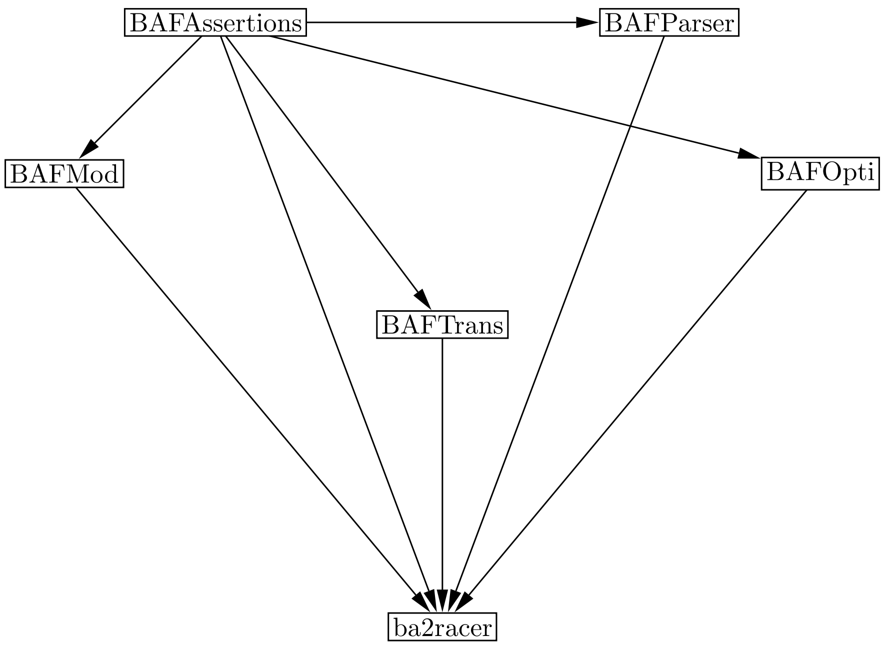
The shape of a 'natural BABox' in a knowledge representation enterprise would be generally a BABox using 2 or more role atoms and more than 5 individuals. With respect of the concept atoms, the number would be higher than 3, and it would be expected a high number of concepts assertions, and among them a numerous set of concepts assertions constructed with the modals $\exists$ and $\forall$. These configuration points to modally constrained BABoxes. We would not expect BABoxes whose consistency is easy to deduce propositionally, otherwise we would not bother to represent such knowledge in a BABox if we can deduce easily the consistency trying a few models. These BABoxes are expected to be consistent, but we do not know for sure if they are. For all this reasons we can locate such BABoxes in the critical region of the sat-unsat curve. As we saw, `ba2racer` together with RACER behaves better than `ba2racerdpll` in modally constrained BABoxes, and specially in the satisfiable part of the critical region. Thus we would prefer to use `ba2racer` in these cases.

We described in this section the different test we effectuate with `ba2racerdpll` and `ba2racer`. The testing were worthwhile, we found some mistakes in RACER and the preliminary random testing indicates in which situations the programs perform well. We also created a generator of BABoxes that was not available before, and that can be useful for further tests

Appendix A

The Code

A.1 ba2racer



BAFAssertions

```
----------------------------------------------------
--                                                --
-- Boolean ABox syntax, data types file           --
--                                                --
-- Berna Martinez September, 2 2002               --
--                                                --
----------------------------------------------------

module BAFAssertions (
    -- types
    Concept(Conn,Top,Bottom,Neg,And,Or,Some,All,Atmost,Atleast),
    Role(Role),
    Sequent(Seq),
    Atom(Ins,Rel,Not),
    Iname(IN),
    RoleInclusion(RoleInclusion),
    KBelem(KBT,KBA),
    KnowledgeBase(KB),
    TboxAxiom(Impl,Equi,Disj,Rdef),
    Transitive(TransitiveNot,Transitive),
    Inverse(InverseNot,Inverse),
    Parents(ParentsNot,Parents),
    Clause,

    -- Show
    show,

    -- Format list
    formatList,
    formatKB,
) where

import List

type Clause = [Atom]

{- Declared Types -}
data Atom
     = Ins Iname Concept
     | Rel Iname Iname Role
     | Not Atom
  deriving (Eq,Ord)

data Concept
     = Conn String
     | Top
     | Bottom
     | Neg Concept
     | And [Concept]
     | Or [Concept]
     | Some Role Concept
     | All Role Concept
     | Atmost Integer Role Concept
     | Atleast Integer Role Concept
  deriving (Eq,Ord)
```

```
data Role
     = Role String
  deriving (Eq,Ord)

data Iname
     = IN String
  deriving (Eq,Ord)

data Sequent
     = Seq [Atom] [Atom]
  deriving (Eq,Ord)

data RoleInclusion
     = RoleInclusion Role [Role]
  deriving (Eq,Ord)

data TboxAxiom
     = Impl Concept Concept
     | Equi Concept Concept
     | Disj [Concept]
     | Rdef Role Transitive Inverse Parents
  deriving (Eq,Ord)

data KBelem
    = KBT TboxAxiom
    | KBA Sequent

data KnowledgeBase
    = KB [TboxAxiom] [Sequent]

data Transitive
    = TransitiveNot
    | Transitive
  deriving (Eq,Ord)

data Inverse
    = InverseNot
    | Inverse Role
  deriving (Eq,Ord)

data Parents
    = ParentsNot
    | Parents [Role]
  deriving (Eq,Ord)

instance Show Atom where
    show (Ins n c) = "(instance " ++ (show n) ++ " " ++ (show c) ++ ")\n"
    show (Rel n i r) = "(related " ++ (show n) ++ " " ++ (show i)
    ++ " " ++ (show r) ++ ")\n"
    show (Not c) = "(not " ++ (show c) ++ ")\n"

instance Show Concept where
    show (Conn c) = c
    show (Top) =  "*top*"
    show (Bottom) = "*bottom*"
    show (And c) = "(and " ++ (formatList (show c) ',' ' ') ++ ")"
    show (Or c) = "(or " ++ (formatList (show c) ',' ' ') ++ ")"
```

```
    show (Neg c) = "(not " ++ (show c) ++ ")"
    show (Some r c) = "(some " ++ (show r) ++ " " ++ (show c) ++ ")"
    show (All r c) = "(all " ++ (show r) ++ " " ++ (show c) ++ ")"
    show (Atmost i r c) = "(at-most " ++ (show i) ++ " " ++ (show r)
    ++ " " ++ (show c) ++ ")"
    show (Atleast i r c) = "(at-most " ++ (show i) ++ " " ++ (show r)
    ++ " " ++ (show c) ++ ")"

instance Show Role where
    show (Role r) = r

instance Show Iname where
    show (IN n) = n

instance Show RoleInclusion where
    show (RoleInclusion a b) = "(" ++ (show a) ++ " :parents (" ++
    (formatList (show b) ',' ' ') ++ "))"

instance Show TboxAxiom where
    show (Impl a b) = "(implies " ++ (show a) ++ " " ++ (show b) ++ ")"
    show (Equi a b) = "(equivalent " ++ (show a) ++ " " ++ (show b) ++ ")"
    show (Disj c) = "(disjoint " ++ (formatList (show c) ',' ' ') ++ ")"
    show (Rdef a b c d) = "(define-primitive-role " ++ (show a) ++ (show b)
    ++ (show c) ++ (show d) ++ ")"

instance Show Transitive where
    show (TransitiveNot) = ""
    show (Transitive) = " :transitive t"

instance Show Inverse where
    show (InverseNot) = ""
    show (Inverse r) = " :inverse " ++ (show r)

instance Show Parents where
    show (ParentsNot) = ""
    show (Parents c) = " :parents (" ++ (formatList (show c) ',' ' ') ++ ")"

{- gives print format -}
formatList :: String -> Char -> Char -> String
formatList s a b = (map (\x -> (if x==a then b else x))(tail(init s)))

{- returns True if it is a Tbox element -}
isKBT (KBT _) = True
isKBT _ = False


{- transforms a set of kbelem in a Tbox,ABox -} formatKB ::
[KBelem] -> KnowledgeBase formatKB kb =
    let
        (tb,ab) = partition isKBT kb
        tb1 = map (\(KBT x) -> x) tb
        ab1 = map (\(KBA x) -> x) ab
    in KB tb1 ab1
```

BAFParser

```
--------------------------------------------------
--                                              --
-- Boolean ABox syntax, parser file             --
--                                              --
-- Berna Martnez August, 20 2002               --
--                                              --
--------------------------------------------------

{
module BAFParse

where

import BAFAssertions
import Prelude
import Char
import IOExts
}

%name parse
%tokentype { Token }

%monad { P } { thenP } { returnP }
%lexer { lexer } { TokenEOF }

%token
    instance        { TokenInstance }
    related         { TokenRelated }
    all             { TokenAll }
    some            { TokenSome }
    top             { TokenTop }
    bottom          { TokenBottom }
    neg             { TokenNeg }
    and             { TokenAnd }
    or              { TokenOr }
    imp             { TokenImp }
    impl            { TokenImpl }
    equi            { TokenEqui }
    disj            { TokenDisj }
    name            { TokenName $$ }
    num             { TokenNum $$ }
    tran            { TokenTransitive }
    inve            { TokenInverse }
    prnt            { TokenParents }
    defrol          { TokenDefRole }
    atmost          { TokenAtmost }
    atleast         { TokenAtleast }
    t               { TokenT }
    '('             { TokenOB }
    ')'             { TokenCB }
    ','             { TokenComma }
    '.'             { TokenDot }
%%

Input:
      {- empty -}                 { [] }
```

```
    | KBelem Input                { $1:$2 }

KBelem:
      TboxAxiom                   { KBT $1 }
    | Sequent                     { KBA $1 }

TboxAxiom:
      '(' impl Concept Concept ')'                      { Impl $3 $4 }
    | '(' equi Concept Concept ')'                      { Equi $3 $4 }
    | '(' disj LConcepts ')'                            { Disj $3 }
    | '(' defrol Role Transitive Inverse Parents ')'    { Rdef $3 $4 $5 $6}

Role :
      name  { Role $1 }

Transitive :
       {- empty -}            { TransitiveNot }
    |  tran t                 { Transitive }

Inverse :
       {- empty -}            { InverseNot }
    |  inve Role              { Inverse $2 }

Parents :
       {- empty -}            { ParentsNot }
    |  prnt '(' LRoles ')'    { Parents $3 }

LRoles :
      {- empty -}       { [] }
    | Role LRoles   { $1:$2 }

Concept :
      name                             { Conn $1 }
    | top                              { Top }
    | bottom                           { Bottom }
    | '(' and LConcepts ')'            { And $3 }
    | '(' or LConcepts ')'             { Or $3 }
    | '(' neg Concept ')'              { Neg $3 }
    | '(' some Role Concept ')'        { Some $3 $4 }
    | '(' all Role Concept ')'         { All $3 $4 }
    | '(' atmost num Role Concept ')'   { Atmost (read $3) $4 $5}
    | '(' atleast num Role Concept ')'  { Atleast (read $3) $4 $5}


LConcepts :
      Concept           { [$1] }
    | Concept LConcepts { $1:$2 }

Iname :
      name  { IN $1 }

Atom :
      '(' instance Iname Concept ')'       { Ins $3 $4 }
    | '(' related Iname Iname Role ')'     { Rel $3 $4 $5 }

LAtoms :
      {- empty -}        { [] }
```

```
    | Atom LAtoms2         { $1:$2 }

LAtoms2 :
      {- empty -}       { [] }
    | ',' Atom LAtoms2  { $2:$3 }

Sequent :
    LAtoms imp LAtoms '.'   { Seq $1 $3 }

---The Monad
{

data ParseResult a
    = ParseOk a
    | ParseFail String

type P a = String -> Int -> ParseResult a

thenP :: P a -> (a -> P b) -> P b
m `thenP` k = \s l ->
    case m s l of
        ParseFail s -> ParseFail s
        ParseOk a -> k a s l

returnP :: a -> P a
returnP a = \s l -> ParseOk a


--the lexer

data Token =
    TokenInstance   | TokenRelated  |
    TokenNeg    | TokenAnd  | TokenOr   | TokenImp  |
    TokenAll    | TokenSome |
    TokenTop    | TokenBottom   |
    TokenOB | TokenCB   |
    TokenComma  | TokenDot  |
    TokenName String | TokenNum String | TokenEOF |
    TokenImpl |
    TokenEqui | TokenDisj |
    TokenT |
    TokenInverse | TokenParents | TokenDefRole | TokenTransitive |
    TokenAtleast | TokenAtmost

lexer :: (Token -> P a) -> P a
lexer cont s = case s of
    [] -> cont TokenEOF []
    ('*':'t':'o':'p':'*':cs) -> cont  TokenTop cs
    ('*':'b':'o':'t':'t':'o':'m':'*':cs) -> cont TokenBottom cs
    (':':'t':'r':'a':'n':'s':'i':'t':'i':'v':'e':cs) -> cont TokenTransitive cs
    (':':'i':'n':'v':'e':'r':'s':'e':cs) -> cont TokenInverse cs
    (':':'p':'a':'r':'e':'n':'t':'s':cs) -> cont TokenParents cs
    ('\n':cs) -> \line -> lexer cont cs (line+1)
    ('\r':cs) -> lexer cont cs
    (' ':cs) -> lexer cont cs
    (',':cs) -> cont TokenComma cs
    ('-':'>':cs) -> cont TokenImp cs
```

```
    ('(':cs) -> cont TokenOB cs
    (')':cs) -> cont TokenCB cs
    ('.':cs) -> cont TokenDot cs
    ('%':cs) -> lexComment cs
    (c:cs)
        | isAlpha c -> lexVarCon (c:cs)
        | isDigit c -> lexNum (c:cs)
        | otherwise -> trace ((show c) ++ "\n") lexer cont cs
 where
    lexComment [] = cont TokenEOF []
    lexComment (c:cs)
        | c == '\n' = \line -> lexer cont cs (line+1)
        | otherwise = lexComment cs
    lexVarCon cs =
        case span isStringPart cs of
            ("ins",rest) -> cont TokenInstance rest
            ("rel",rest) -> cont TokenRelated rest
            ("all",rest) -> cont TokenAll rest
            ("some",rest) -> cont TokenSome rest
            ("or",rest) -> cont TokenOr rest
            ("and",rest) -> cont TokenAnd rest
            ("not",rest) -> cont TokenNeg rest
            ("implies",rest) -> cont TokenImpl rest
            ("equivalent",rest) -> cont TokenEqui rest
            ("disjoint",rest) -> cont TokenDisj rest
            ("at-most",rest) -> cont TokenAtmost rest
            ("at-least",rest) -> cont TokenAtleast rest
            ("t",rest) -> cont TokenT rest
            ("define-role",rest) -> cont TokenDefRole rest
            (name,rest) -> cont (TokenName name) rest
    lexNum cs =
        case span isDigit cs of
            (num,rest) -> cont (TokenNum num) rest

isStringPart c
    | isAlphaNum c = True
    | c ==  '_' = True
    | c == '-' = True
    | otherwise = False

runParser :: String -> [KBelem]
runParser s = case parse s 1 of
    ParseOk e -> e
    ParseFail s -> error s

--this should be defined for all parsers

type Parse = P [KBelem]
parse :: Parse

happyError :: P a
happyError = \s i -> error (
      "Parse error in line " ++ show (i::Int) ++ "\n")
}
```

BAFMod

```
---------------------------------------------------
--                                               --
-- Boolean ABox Module                           --
--                                               --
-- Berna Martinez September, 03 2002             --
--                                               --
---------------------------------------------------

module BAFMod(
--functions
    qsort,
    treatParams,
    spreadImp,
    onlyOneDefRole,
    unifyTbox,
    signature,
)
where

import List

import BAFAssertions

{- qsort: sorting algorithm with elimination of duplicates -}
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_gr_x
                 where
                   elts_lt_x   = [y | y <- xs, y < x]
                   elts_gr_x = [y | y <- xs, y > x]

{- return the parameters -}
treatParams :: [String] -> (Bool, String)
treatParams []     = (False, "")
treatParams (h:l)  = (True, h)


{- spreadImp: spread de implication rule p -> q == ~p \/ q -}
spreadImp :: [Sequent] -> [Clause]
spreadImp cs = (map (\(Seq a b) -> (map Not a) ++ b) cs)

{- elem speial for Role names -}
elemRole :: Role -> [TboxAxiom] -> Bool
elemRole _ [] = False
elemRole a ((Rdef b _ _ _):cs)
    | a == b = True
    | otherwise = (elemRole a cs)

{- checks if there are two definitions for the same role -}
onlyOneDefRole :: [TboxAxiom] -> [TboxAxiom] -> (Bool,Role)
onlyOneDefRole [] _ = (False,(Role "null"))
onlyOneDefRole ((Rdef a b InverseNot c):cs) tb =
    let
        tb1 = filter ((/=) (Rdef a b InverseNot c)) tb
    in  if (elemRole a tb1)
            then (True,a)
```

```
            else  (onlyOneDefRole cs tb)
onlyOneDefRole ((Rdef a b (Inverse c) d):cs) tb =
    let
        tb1 = filter ((/=) (Rdef a b (Inverse c) d)) tb
    in  if (elemRole a tb1)
            then (True,a)
            else  if (elemRole c tb1) then (True,c) else (onlyOneDefRole cs tb)
onlyOneDefRole (_:cs) tb = (onlyOneDefRole cs tb)


{-adds relations as parenst-}
addParent :: Parents -> [Role] -> Parents
addParent (ParentsNot) k = (Parents k)
addParent (Parents i) k = (Parents (qsort(union i k)))

{-unifies the axioms created with the axioms read-}
unifyTboxS :: [TboxAxiom] -> RoleInclusion ->[TboxAxiom]
unifyTboxS [] (RoleInclusion a i) = [(Rdef a (TransitiveNot)
 (InverseNot) (Parents i))]
unifyTboxS ((Rdef a b InverseNot d):cs) (RoleInclusion j k)
    | a == j =
        let newd = addParent d k
        in ((Rdef a b InverseNot newd):cs)
    | otherwise =
        let newcs = unifyTboxS cs (RoleInclusion j k)
        in ((Rdef a b InverseNot d):newcs)
unifyTboxS ((Rdef a b (Inverse c) d):cs) (RoleInclusion j [k])
    | a == j =
        let newd = addParent d [k]
        in ((Rdef a b (Inverse c) newd):cs)
    | c == j =
        let
            kinv = (Role ((show k) ++ "inv"))
            newax = (Rdef k (TransitiveNot) (Inverse kinv) (ParentsNot))
            newd = addParent d [kinv]
        in ([(Rdef a b (Inverse c) newd),newax] `union` cs)
    | otherwise =
        let newcs = unifyTboxS cs (RoleInclusion j [k])
        in ([(Rdef a b (Inverse c) d)] `union` newcs)
unifyTboxS (c:cs) s = c:(unifyTboxS cs s)

unifyTbox :: [TboxAxiom] -> [RoleInclusion] ->[TboxAxiom]
unifyTbox tb [] = tb
unifyTbox tb ((RoleInclusion i j):cs)=
    let
        newtb = unifyTboxS tb (RoleInclusion i j)
    in unifyTbox newtb cs

{- Creates the signature of the KB -}
signature :: [TboxAxiom] -> Clause -> String
signature tb ab =
    let
        rol1 = (qsort ((concatMap getRoleTB tb) ++ (concatMap getRoleAB ab)))
        \\ (concatMap getRoleDfTB tb)
        con1 = qsort ((concatMap getConnTB tb) ++ (concatMap  getConnAB ab))
        ind1 = qsort (concatMap getInamAB ab)
        rol2 = formatList (show rol1) ',' ' '
```

```
        con2 = formatList (show con1) ',' ' '
        ind2 = formatList (show ind1) ',' ' '
    in "(signature \n :atomic-concepts(" ++ con2 ++ ")\n :roles(" ++ rol2 ++
    ")\n :individuals(" ++ ind2 ++"))\n"

getRoleDfTB :: TboxAxiom -> [Role]
getRoleDfTB (Rdef a _ (Inverse c) _) = [a,c]
getRoleDfTB (Rdef a _ _ _) = [a]
getRoleDfTB _ = []

getRoleTB :: TboxAxiom -> [Role]
getRoleTB (Impl a b) = (getRoleC a) ++ (getRoleC b)
getRoleTB (Disj a) = concatMap getRoleC a
getRoleTB (Equi a b) = (getRoleC a) ++ (getRoleC b)
getRoleTB _ = []

getRoleAB :: Atom -> [Role]
getRoleAB (Ins _ b) = (getRoleC b)
getRoleAB (Rel _ _ c) = [c]

getConnTB :: TboxAxiom -> [Concept]
getConnTB (Impl a b) = (getConnC a) ++ (getConnC b)
getConnTB (Disj a) = concatMap getConnC a
getConnTB (Equi a b) = (getConnC a) ++ (getConnC b)
getConnTB _ = []

getConnAB :: Atom -> [Concept]
getConnAB (Ins a b) = (getConnC b)
getConnAB _ = []

getInamAB :: Atom -> [Iname]
getInamAB (Ins a b) = [a]
getInamAB (Rel a b c) = [a,b]

getRoleC :: Concept -> [Role]
getRoleC (Neg c) = getRoleC c
getRoleC (And c) = concatMap getRoleC c
getRoleC (Or c) =  concatMap getRoleC c
getRoleC (Some r c) = r:(getRoleC c)
getRoleC (All r c) = r:(getRoleC c)
getRoleC (Atmost _ r c) = r:(getRoleC c)
getRoleC (Atleast _ r c) = r:(getRoleC c)
getRoleC _ = []

getConnC :: Concept -> [Concept]
getConnC (Conn n) = [(Conn n)]
getConnC (Neg c) = getConnC c
getConnC (And c) = concatMap getConnC c
getConnC (Or c) = concatMap getConnC c
getConnC (Some _ c) = getConnC c
getConnC (All _ c) = getConnC c
getConnC (Atmost _ _ c) = getConnC c
getConnC (Atleast _ _ c) = getConnC c
getConnC _ = []

formatList1 :: Clause -> Clause
formatList1 [(Not (Rel a b c))] =  [(Ins a Top),(Ins b Top)]
```

```
formatList1 [(Not (Ins a b))] = [(Ins a (Neg b))]
formatList1 c = c
```

BAFOpti

```
----------------------------------------------------
--                                                --
-- Boolean ABox Optimizations                     --
--                                                --
-- Berna Martinez September 9, 2002               --
--                                                --
----------------------------------------------------


module BAFOpti (
    -- Resolution step
    resStep,
    pulloutNot,
    unitProp1,
    compCl,
    denial,
    simplify,
) where

import BAFAssertions
import List


{- pulloutNot: gets the negations out of the concept in an assertion -}
pulloutNot :: [Clause] -> [Clause]
pulloutNot cs = (map (map pulloutNotA) cs)

pulloutNotA :: Atom -> Atom
pulloutNotA (Not (Ins a (Neg b))) = pulloutNotA (Ins a b)
pulloutNotA (Ins a (Neg b)) = pulloutNotA (Not(Ins a b))
pulloutNotA c = c

{- returns if the empty clause is present in a set of clauses -}
empty :: [Clause] -> Bool
empty c = elem [] c

{- returns the negation of an assertion -}
denial :: Atom -> Atom
denial (Not c) = c
denial c = (Not c)

{- makes the propagation of one truth value -}
simplify :: Atom -> [Clause] -> [Clause]
simplify a c = map (filter ((/=) (denial a))) (filter (notElem a) c)

{- returns if a clause is not a tautology -}
isNotTaut :: Clause -> Bool
isNotTaut [] = True
isNotTaut (c:cs)
        | elem (denial c) cs = False
        | otherwise = isNotTaut cs
```

```
{- unit propagation (step) optimization -}
{- returns only an empty clause if a contradiction is found -}
unitProp1 :: [Clause] -> [Clause] -> [Clause]
            --propagate until no unit clauses
unitProp1 [] lc
    |empty lc = [[]]
    |otherwise = lc
unitProp1 (([c]):cs) lc
    |empty lc = [[]]
    |otherwise =
        let
            newlc = simplify c lc
            newlc1 = unitProp1 newlc newlc
        in
            if (newlc1==[[]]) then [[]]
            else (([c]):newlc1)

unitProp1 (([]):cs) _ = [[]]
unitProp1 (_:cs) lc = unitProp1 cs lc

{- executes the resolution step -}
resStep :: [Clause] -> [Clause]
resStep lc =
    let
                lc1 = nubBy compCl (filter isNotTaut (map nub lc))
                    --erase all the tautologies
    in unitProp1 lc1 lc1    --applies unit propagation

{- if the two clauses are the same returns ture -}
compCl :: Clause -> Clause -> Bool
compCl a b = and (map (\x -> elem x b) a)
```

BAFTrans

```
----------------------------------------------------
--                                                --
-- Boolean ABox to Racer, translator module       --
--                                                --
-- Berna Martinez September, 03 2002              --
--                                                --
----------------------------------------------------
module BAFTrans(
    --functions
    spreadRoles,
    spreadDiff,
    spreadSame,
    spreadNot,
)
where

import List

import BAFAssertions

{- spreadNot: spread the negation rule in the instances of concepts -}
```

```
spreadNot :: Atom -> Atom

spreadNot (Not(Ins a b)) = (Ins a (Neg b))
spreadNot c = c

{- spreadSame: joins all the concepts of the same instance in one assertion-}
spreadSame :: Clause -> Clause
spreadSame []          = []
spreadSame ((Ins a b):cs) =
    let
        (alike,rest) = partition (\(Ins c d) -> if a == c
                                                then True else False) cs
    in
        if (alike==[])
        then (Ins a b):(spreadSame rest)
        else
            (Ins a (Or (b:(map (\(Ins _ d) -> d) alike)))):(spreadSame rest)
spreadSame cs = cs

{- spreadDiff joins the instantiations related to different instances -}
spreadDiff :: [Clause] -> Integer -> [Clause]
spreadDiff [] _ = []
spreadDiff ([(Rel a b c)]:cs) i = ([(Rel a b c)]:(spreadDiff cs i))
spreadDiff (c:cs) i = let (newcl,newi) = (spreadDiffCl c i)
                      in newcl ++ (spreadDiff cs newi)

{-returns the new role assertions and clause-}
spreadDiffCl :: Clause -> Integer -> ([Clause],Integer)
spreadDiffCl [] i = ([],i)
spreadDiffCl [a] i = ([[a]],i)
spreadDiffCl ((Ins a b):(Ins c d):cs) i =
    let
        ----NewRoleSymbol
        newr = Role ("role" ++ (show i))
        ----NewRoleSymbol
        newunit = [(Rel a c newr)]
        newatom = (Ins a (Or [b,(All newr d)]))
        (finalcls,finali) = (spreadDiffCl (newatom:cs) (i+1))
    in (newunit:finalcls,finali)

{- spreadRoles: substitute the negated or not atomic roles -}
spreadRoles :: [Clause] -> [Clause] -> Int-> ([Clause],[RoleInclusion],[Clause])
spreadRoles prev [] _= (prev,[],[])
spreadRoles prev ([(Rel i j r)]:cs) count =
    spreadRoles (prev ++ [[(Rel i j r)]]) cs count
spreadRoles prev (c:cs) count =
    let
        c1 = isThereRelation c
    in if  c1 == []
        then spreadRoles (prev ++ [c]) cs count
        else
             let
            (newprev,newnext,axiom,unitcl) = spreadRolesAux prev (c:cs) c1 count
            (newcls,axioms,unitcls) = spreadRoles newprev newnext (count+1)
             in (newcls,axiom:axioms,unitcl++unitcls)

spreadRolesAux :: [Clause] -> [Clause] -> Clause -> Int ->
```

```
([Clause],[Clause],RoleInclusion,[Clause])
spreadRolesAux prev1 next1 [(Rel i j r)] count=
    let
         ----FakeRoleOf
        faker = Role ("fake" ++ (show r))
        ----FakeRoleOf
        axiomrfr = (RoleInclusion r [faker])
        ijrelfaker = [(Rel i j faker)]
        ----NewConceptSymbol
        iamj = Conn ("iamj" ++ (show count))
        ----NewConceptSymbol
        jiniamj = [(Ins j iamj)]
        atmo1 = [(Ins i (Atmost 1 faker iamj))]
        substit = (Ins i (Some r iamj))
        newprev = subst (Rel i j r) substit prev1
        newnext = subst (Rel i j r) substit next1
    in (newprev,newnext,axiomrfr,[ijrelfaker,jiniamj,atmo1])

{-returns the role negated or in a non unit clause -}
isThereRelation :: Clause -> Clause
isThereRelation [] = []
isThereRelation ((Rel i j r):_) = [(Rel i j r)]
isThereRelation ((Not(Rel i j r)):_) = [(Rel i j r)]
isThereRelation (_:cs) = (isThereRelation cs)

{- Sustitution function for Relations (Atomic Level) -}
subst :: Atom -> Atom -> [Clause] -> [Clause]
subst a b c = (map (map (substCl a b)) c)

substCl :: Atom -> Atom -> Atom -> Atom
substCl a b (Not c)
            | a == c    = (Not b)
            | otherwise = (Not c)
substCl a b c
            | a == c    = b
            | otherwise = c
```

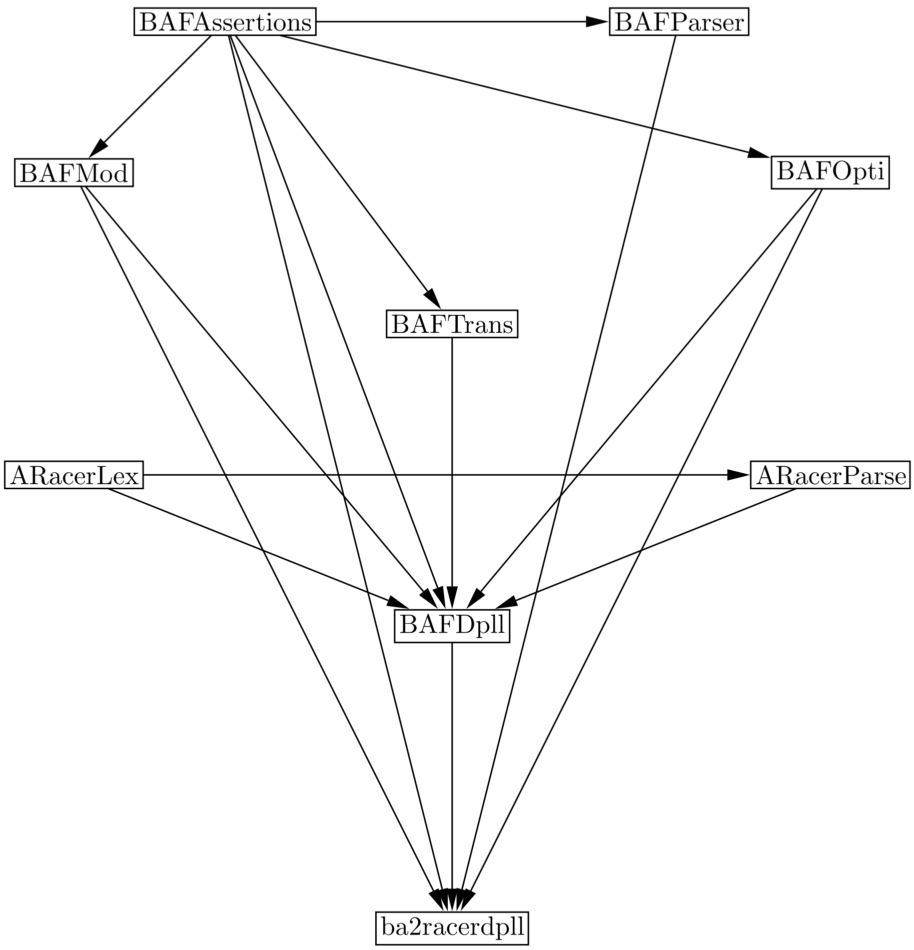
ba2racer

```
----------------------------------------------------
--                                                --
-- Boolean ABox to Racer, main file               --
--                                                --
-- Berna Martinez September, 03 2002              --
--                                                --
----------------------------------------------------

module Main (main)

where

import System
import Char
import List

import BAFAssertions
```

```
import BAFParse
import BAFMod
import BAFOpti
import BAFTrans

{----------------------------------}
{-  -}
{----------------------------------}

ba2racer :: [TboxAxiom] -> [Clause] -> String

ba2racer tb ab =
    let
        (abox1,tbox1,unitcl) = spreadRoles [] ab 1
        tbox2 = unifyTbox tb tbox1
        abox2 = nub(concat ((spreadDiff (map (spreadSame . (map spreadNot)) abox1) 1)
             ++ unitcl))
        call = ";calls:1\n"
        size = ";size:" ++ (show (length abox2)) ++ "\n"
        tbox = ((formatList (show tbox2) ',' '\n') ++ "\n") ++ "\n"
        abox = (concatMap show abox2) ++ "\n"
        sign = (signature tbox2 abox2) ++ "\n"
        outp = call ++ size ++ sign ++ tbox ++ abox
    in outp

{----------------------}
{- Main ----------------}
{----------------------}

main :: IO ()

main =
   do {
       args <- getArgs;
       let {
            (noerror, fname) = treatParams args;
           } in
       if noerror
       then do {
        fstr <- readFile fname;
        let
          (KB tb ab) = formatKB(runParser (map toLower fstr));
        extension = ".racer";
          inknldg = "(in-knowledge-base " ++ fname ++ "abox " ++ fname ++ "tbox )\n"
          aboxcons = "(abox-consistent?)"
          (rep,rn) = onlyOneDefRole tb tb
        in if rep
           then putStr ("Double definition of " ++ (show rn) ++ "\n")
           else
        case (resStep (pulloutNot(spreadImp ab))) of
           [] -> putStr "ABox consistent\n" >>
                 writeFile (fname ++ extension) (";calls:0\n;size:0\n" ++ inknldg
                 ++ "(instance a *top*)\n(abox-consistent?)\n")
           [[]] -> putStr "ABox not consistent\n" >>
                 writeFile (fname ++ extension) (";calls:0\n;size:0\n" ++ inknldg
                 ++ "(instance a *bottom*)\n(abox-consistent?)\n")
           pf1 -> writeFile (fname ++ extension)
```

```
            (inknldg ++ (ba2racer tb pf1) ++ aboxcons)
  }
 else putStr "ba2racer inputfile\n";
}
```

A.2 ba2racerdpll



BAFDpll

```
----------------------------------------------------
--                                                --
-- Boolean ABox, Dpll Method                      --
--                                                --
-- Berna Martinez September 7, 2002               --
--                                                --
----------------------------------------------------


module BAFDpll (
    giveAtom,
    unitProp2,
    isSHIQ,
    testRacer,
    eraseR,
    transformR,
) where

import List
import System

import ARacerLex
import ARacerParse

import BAFAssertions
import BAFOpti
import BAFTrans
import BAFMod

{- Format an assertion for de role deletion approach -}
formatList1 :: Clause -> [Clause]
formatList1 [(Not (Rel a b c))] =  [[(Ins a Top)],[(Ins b Top)]]
formatList1 [(Not (Ins a b))] = [[(Ins a (Neg b))]]
formatList1 c = [c]


{- is a SHIQ knowledge base? -}
isSHIQ :: TboxAxiom -> Bool
isSHIQ (Rdef _ _ (Inverse _) _ ) = True
isSHIQ _ = False

{- Unit propagation for  -}
unitProp2 :: [Clause] -> ([Clause],[Clause])
unitProp2 c =
    case  (unitProp1 c c) of
        [[]] -> ([],[[]])
        c1 ->  partition (((==) 1).length) c1

{- Give atom for Dpll -}
giveAtom :: [Clause] -> Atom
giveAtom c =
    let
        min = (minimum (map length c))
        c1 = filter (((==) min).length) c
    in moms c1
```

```
{- moms heuristic -}
moms :: [Clause] -> Atom
moms c = maxi (genSameCount (concat c))

genSameCount :: [Atom] -> [(Int,Atom)]
genSameCount [] = []
genSameCount (c:cs) =
    let
        (a,b) = partition ((==) c) cs
        c1 = (length a + 1,c)
    in (c1:(genSameCount b))

maxi :: [(Int,Atom)] -> Atom
maxi c =
    let
        (_,atom) = maximum c
    in atom

{- Generation of *B -}
gimmeFixPoint :: [TboxAxiom] -> [Clause] -> [Clause]
gimmeFixPoint tb ab =
    let
        ab1 = filter isRoleAssertion ab
        ab2 = fixPoint tb ab1
    in ab `union` ab2


fixPoint :: [TboxAxiom] -> [Clause] -> [Clause]
fixPoint [] c = c
fixPoint _ [] = []
fixPoint t a =
    let
        a1 = (concatMap (setFixPoint t a) a)
        a2 = nubBy compCl ( a ++ a1 )
    in
        if (length a) == (length a2) then a
        else fixPoint t a2

setFixPoint :: [TboxAxiom] -> [Clause] -> Clause -> [Clause]
setFixPoint t rn a = concatMap (fixPointRule a rn) t

fixPointRule :: Clause -> [Clause] -> TboxAxiom -> [Clause]
fixPointRule [(Rel a b c)] cl (Rdef d e f g)
    | (c == d) =
        let
            tra = filter (((>) 0).length) (map (fixPointTra (Rel a b c) e) cl )
            par = fixPointPar (Rel a b c) g
        in  tra ++ par
    | otherwise = []


fixPointTra :: Atom -> Transitive -> Clause -> Clause
fixPointTra _  TransitiveNot _= []
fixPointTra (Rel a b c) Transitive [(Rel d e f)]
    | ((c == f) && (b == d)) = [(Rel a e c)]
    | otherwise = []
```

```
fixPointPar :: Atom -> Parents -> [Clause]
fixPointPar a (Parents b) = map (fixPointPar1 a) b
fixPointPar _ ParentsNot = []

fixPointPar1 :: Atom -> Role -> Clause
fixPointPar1 (Rel a b c) d =
    [(Rel a b d)]


{- returns true if in the unit clause there is a role assertion -}
isRoleAssertion :: Clause -> Bool
isRoleAssertion [(Rel _ _ _)] = True
isRoleAssertion _ = False

{- TestRacer -}
testRacer :: [TboxAxiom] -> [Clause] -> String -> Int -> Int -> IO (Int, Int, Bool)
testRacer tb ab name count size=
    let
        countn = count + 1
        abtm = nub (concat (concatMap formatList1 ab))

        tbox = (formatList (show tb) ',' '\n') ++ "\n"
        abox = concatMap show abtm
        sign = (signature tb abtm) ++ "\n"

        sizen = size + (length abtm)
        pcount = ";calls:" ++ (show countn) ++ "\n"
        psize = ";size:" ++ (show (sizen `div` countn)) ++ "\n"
        inknow = "(in-knowledge-base rabox rtbox )\n"
        aboxcons = "(abox-consistent?)"
    in
        do{
            err <- writeFile name (pcount ++ psize ++ inknow ++ sign ++ tbox ++ abox ++ aboxcons);
            err1 <- system ("racer -f " ++ name ++" > tmp.tmp");
            file <- readFile "tmp.tmp";
            return (countn,sizen,(racParser(lexera file)));
          }

{- eraseR -}
eraseR :: [TboxAxiom] -> [Clause] -> [Clause]
eraseR tb ab =
    let
        ab1 = gimmeFixPoint tb ab
        (at1,ab2) = unitProp2 ab1
    in case ab2 of
        [[]] -> [[]]
        [] -> at1

{- transformR -}
transformR :: [TboxAxiom] -> [Clause] -> ([TboxAxiom],[Clause])
transformR tb ab =
    let
        (abr,tbr,clr) = spreadRoles [] ab 1
        tbox1 = unifyTbox tb tbr
        abox1 = abr ++ clr
    in (tbox1,abox1)
```

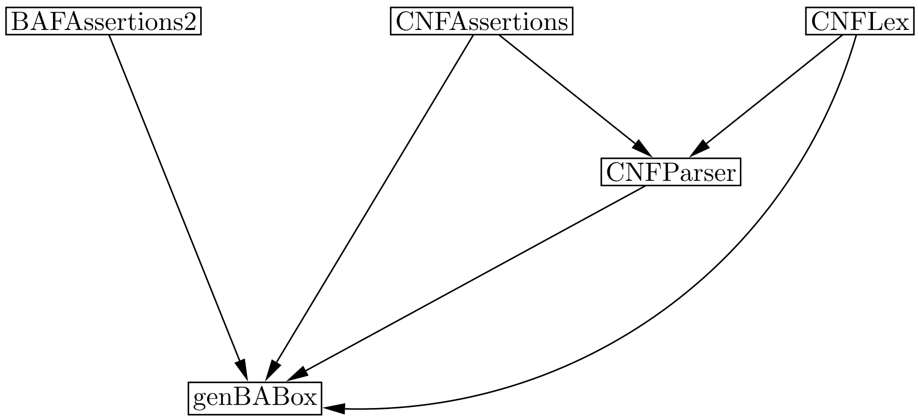
ba2racerdpll

```
----------------------------------------------------
--                                                --
-- Boolean ABox to Racer, main file               --
--                                                --
-- Berna Martinez September, 09 2002              --
--                                                --
----------------------------------------------------

module Main (main)

where

import System
import Char
import List

import BAFParse
import BAFAssertions
import BAFOpti
import BAFDpll
import BAFMod


{------------------------------------}
{-  -}
{------------------------------------}

{-- call dpllRacer tb ab [] name --}

dpllRacer :: [TboxAxiom] -> [Clause] -> [Clause] ->
String -> Int -> Int -> IO (Int, Int, Bool)
dpllRacer tb ab at name count size =
    let
        (at1,ab1) = unitProp2 ab
        at2 = at ++ at1
    in case ab1 of
        [] ->
            let
                res = eraseR tb at2
            in
                case res of
                    [[]] -> let var = (count,size,False) in do return var
                    abres -> testRacer tb abres name count size
        [[]] -> let var = (count,size,False) in do return var
        ab2 ->
            let
                a = giveAtom ab2
                ab3 = simplify a ab2
            in do {
                (countn,sizen,res) <- dpllRacer tb ab3 ([a]:at2) name count size;
                if res then return (countn,sizen,True)
                else
                    let
                        a1 = denial a
                        ab4 = simplify a1 ab2
                    in dpllRacer tb ab4 ([a1]:at2) name countn sizen
```

```
                }

{-- call dpllRacerTrans tb ab [] name --}

dpllRacerTrans :: [TboxAxiom] -> [Clause] -> [Clause] ->
String -> Int -> Int -> IO (Int, Int, Bool)
dpllRacerTrans tb ab at name count size =
    let
        (at1,ab1) = unitProp2 ab
        at2 = at ++ at1
    in case ab1 of
        [] ->
            let
                (newtb,newab) = transformR tb at2
            in testRacer newtb newab name count size
        [[]] -> let var = (count,size,False) in do return var
        ab2 ->
            let
                a = giveAtom ab2
                ab3 = simplify a ab2
            in do {
                (countn,sizen,res) <-
                    dpllRacerTrans tb ab3 ([a]:at2) name count size;
                if res then return (countn,sizen,True)
                else
                    let
                        a1 = denial a
                        ab4 = simplify a1 ab2
                    in dpllRacerTrans tb ab4 ([a1]:at2) name countn sizen
                }


{-----------------------}
{- Main ----------------}
{-----------------------}

main :: IO ()

main =
   do {
       args <- getArgs;
       let {
            (noerror, fname) = treatParams args;
           } in
       if noerror
       then do {
                fstr <- readFile fname;
                let
                     (KB tb ab) = formatKB(runParser (map toLower fstr));
                 extension = ".racer";
                     inknldg = "(in-knowledge-base " ++ fname ++ "abox " ++ fname ++ "tbox )\n"
                     aboxcons = "(abox-consistent?)"
                     (rep,rn) = onlyOneDefRole tb tb
                in if rep
                   then putStr ("Double definition of " ++ (show rn) ++ "\n")
                   else
                case (resStep(pulloutNot(spreadImp ab))) of
```

```
              [] -> putStr "ABox consistent\n"  >>
              writeFile (fname ++ extension) ";calls:0\n;size:0\n"
              [[]] -> putStr "ABox not consistent\n"  >>
              writeFile (fname ++ extension) ";calls:0\n;size:0\n"
              pf1 ->
                  do{
                      if (or (map isSHIQ tb))
                      then
                          do {
                              (tot,size,res) <-
                                  dpllRacerTrans tb pf1 [] (fname ++ extension) 0 0 ;
                              if res
                              then putStr "ABox consistent, file .racer created\n"
                              else putStr "ABox not consistent, racer result\n"
                              }
                      else
                          do {
                              (tot,size,res) <- dpllRacer tb pf1 [] (fname ++ extension) 0 0 ;
                              if res
                              then putStr "ABox consistent, file .racer created\n"
                              else putStr "ABox not consistent, racer result\n"
                              }
                  }
         }
 else putStr "ba2racer inputfile\n";
}
```

A.3 genBABox


BAFAssertions2
CNFAssertions
CNFLex
CNFParser
genBABox

BAFAssertions2

```
----------------------------------------------------
--                                                --
-- Boolean ABox syntax, data types file           --
--                                                --
-- Berna Martinez August, 20 2002                 --
--                                                --
----------------------------------------------------

module BAFAssertions2 (
    -- types
    Concept(Conn,Top,Bottom,Neg,And,Or,Some,All,Atmost,Atleast),
    Role(Role),
    Sequent(Seq),
    Atom(Ins,Rel,Not),
    Iname(IN),
    Clause,

    -- Show
    show,

    -- Format list
    formatList,
) where

import List
type Clause = [Atom]

{- Declared Types -}
data Atom
     = Ins Iname Concept
     | Rel Iname Iname Role
     | Not Atom
  deriving (Eq,Ord)

data Concept
     = Conn String
     | Top
     | Bottom
     | Neg Concept
     | And [Concept]
     | Or [Concept]
     | Some Role Concept
     | All Role Concept
     | Atmost Int Role Concept
     | Atleast Int Role Concept
  deriving (Eq,Ord)

data Role
     = Role String
  deriving (Eq,Ord)

data Iname
     = IN String
  deriving (Eq,Ord)

data Sequent
```

```
     = Seq [Atom] [Atom]
  deriving (Eq,Ord)

instance Show Atom where
    show (Ins n c) = "(ins " ++ (show n) ++ " " ++ (show c) ++ ") "
    show (Rel n i r) = "(rel " ++ (show n) ++ " " ++ (show i) ++ " " ++
        (show r) ++ ") "
    show (Not c) = "(not " ++ (show c) ++ ") "

instance Show Concept where
    show (Conn c) = c
    show (Top) =  "*top*"
    show (Bottom) = "*bottom*"
    show (And c) = "(and " ++ (formatList (show c) ',' ' ') ++ ")"
    show (Or c) = "(or " ++ (formatList (show c) ',' ' ') ++ ")"
    show (Neg c) = "(not " ++ (show c) ++ ")"
    show (Some r c) = "(some " ++ (show r) ++ " " ++ (show c) ++ ")"
    show (All r c) = "(all " ++ (show r) ++ " " ++ (show c) ++ ")"
    show (Atmost i r c) = "(at-most " ++ (show i) ++ " " ++ (show r) ++
        " " ++ (show c) ++ ")"
    show (Atleast i r c) = "(at-least " ++ (show i) ++ " " ++ (show r) ++
        " " ++ (show c) ++ ")"

instance Show Role where
    show (Role r) = r

instance Show Iname where
    show (IN n) = n

instance Show Sequent where
    show (Seq a b) = (tail(init(show a))) ++ "->" ++
        (tail(init(show b))) ++ ".\n"

formatList :: String -> Char -> Char -> String
formatList s a b = (map (\x -> (if x==a then b else x))(tail(init s)))
```

CNFAssertions

```
----------------------------------------------------
--                                                --
-- MkCnf Syntax, data types file                  --
--                                                --
-- Berna Martinez September, 18 2002              --
--                                                --
----------------------------------------------------

module CNFAssertions (
    -- types
    Var(Var),

    -- Show
    show,
) where

{- Declared Types -}
data Var
```

```
     = Var Int
  deriving (Eq,Ord)

instance Show Var where
    show (Var n) = (show n)
```

CNFLex

```
----------------------------------------------------
--                                                --
-- MkCnf Syntax, lexer file                       --
--                                                --
-- Berna Martinez September, 18 2002              --
--                                                --
----------------------------------------------------

module CNFLex

where

import Prelude
import Char
import IOExts

data Token =
    TokenVar Int | TokenEOL

lexer :: String -> [Token]
lexer [] = []
lexer ('-':cs) = lexVarCon ('-':cs)
lexer ('p':cs) = lexComment cs
lexer ('c':cs) = lexComment cs
lexer ('\n':cs) = lexer cs
lexer ('\r':cs) = lexer cs
lexer (' ':cs) = lexer cs
lexer ('0':cs) = TokenEOL:lexer cs
lexer (c:cs)
    | isDigit c = lexVarCon (c:cs)
    | otherwise  =
        trace ((show c) ++ "\n") lexer cs

lexVarCon cs =
    case span isDigitPart cs of
        (num,rest) -> (TokenVar (read num)) : lexer rest

isDigitPart c
    | isDigit c = True
    | c ==  '-' = True
    | otherwise = False

lexComment :: [Char] -> [Token]
lexComment [] = []
lexComment (c:cs)
    | c == '\n' = lexer cs
    | otherwise = lexComment cs
```

CNFParser

```
----------------------------------------------------
--                                                --
-- MkCnf Syntax , parser file                     --
--                                                --
-- Berna Martinez September, 30 2002              --
--                                                --
----------------------------------------------------

{
module CNFParse

where

import CNFLex
import CNFAssertions
}

%name parse
%tokentype { Token }

%token
    eol             { TokenEOL }
    num             { TokenVar $$ }

%%

Input :
  Clause                { [$1] }
| Clause  Input         { $1:$2 }

Clause :
  eol                   {[]}
| num Clause            { (Var $1):$2 }

{
happyError :: [Token] -> a
happyError _ = error "Parse error"
}
```

genBABox

```
----------------------------------------------------
--                                                --
-- GenerateBoolean ABoxes, main file              --
--                                                --
-- Berna Martinez September, 18 2002              --
--                                                --
----------------------------------------------------

module Main (main)

where

import System
```

```
import Char
import List
import Random
import BAFAssertions2
import CNFLex
import CNFParse
import CNFAssertions

{-----------------------------------}
{- -}
{-----------------------------------}

{- namefile #rolenames #conceptnames #instancenames
#roleass #conceptass #probmod depth #clauses lengtcl numfil-}
treatParams :: [String] -> (Bool,String,Int,Int,Int,Int,Int,Int,Int,Int,Int,Int)

treatParams (n:rn:cn:ins:ra:ca:p:d:c:l:nf:cs)  =
    let
        instn = read ins
        length = read l
        numfil = read nf
    in
        if (instn == 0 || length == 0 || numfil == 0)
        then (False,"",0,0,0,0,0,0,0,0,0,0,0)
        else (True,n,(read rn),(read cn),instn,(read ra),(read ca),
            (read p),(read d),(read c),length,numfil)
treatParams c     = (False,"",0,0,0,0,0,0,0,0,0,0,0)

genRnd :: Int -> Int -> IO Int
genRnd a b = getStdRandom (randomR (a,b))

{- modified version of show -}
show1 :: [Sequent] -> String
show1 [] = ""
show1 (c:cs) = (show c) ++ (show1 cs)

{- sets negative a propositional variable -}
mulNot :: Var -> Var
mulNot (Var k) = (Var (-1*k))

{- returns true if k is a negative propositional variable-}
divNeg :: Var -> Bool
divNeg (Var k) = k < 0

{- creates sequents of propositional variable clauses -}
{- list_of_assertions propositional_clauses -}
substCNF :: Clause -> [[Var]] -> [Sequent]
substCNF _ [] = []
substCNF lst (c:cs) =
    let
        (a,b) = partition divNeg c
        a1 = substCNF1 lst (map mulNot a)
        b1 = substCNF1 lst b
        s1 = (Seq a1 b1)
        cs1 = substCNF lst cs
    in (s1:cs1)
```

```
{- converts a propositional clause in an assertional clause -}
{-list_of_assertions propositional_clause-}
substCNF1 :: Clause -> [Var] -> Clause
substCNF1 _ [] = []
substCNF1 l ((Var n):cs) = (elemn n l):(substCNF1 l cs)

{- returns the element n of the list -}
elemn :: Int -> Clause -> Atom
elemn 1 (c:cs) = c
elemn n (_:cs) = elemn (n-1) cs

{- generates a list of concepts -}
{- size_of_the_list #conceptnames #rolenames depth-1
#instancenames #probmod -}
genListConcept 0 i j k l p =
    do {
        return []
       }
genListConcept mem i j k l p =
    do {
        head <- genConceptN i j k l p;
        tail <- genListConcept (mem-1) i j k l p;
        let
            list = (head:tail)
        in return(list)
       }

{- #instancenames -}
selectInstance :: Int -> IO Iname
selectInstance i  =
    do {
        inam <- genRnd 1 i;
        return (IN ("inam" ++ (show inam)));
       }

{- #conceptnames #rolenames depth #instancenames #probmod -}
genNegConcept :: Int -> Int -> Int -> Int -> Int -> IO Concept
genNegConcept i j k l p =
    do {
        conn <- genConceptN i j k l p;
        return (Neg conn);
       }

{- #conceptnames #rolenames depth #instancenames #probmod -}
genAndConcept :: Int -> Int -> Int -> Int -> Int -> IO Concept
genAndConcept i j k l p =
    do {
        mem <- genRnd 2 i;
        conlist <- genListConcept mem i j k l p;
        return (And conlist);
       }


{- #conceptnames #rolenames depth #instancenames #probmod -}
genOrConcept :: Int -> Int -> Int -> Int -> Int -> IO Concept
genOrConcept i j k l p =
    do {
```

```
        mem <- genRnd 2 i;
        conlist <- genListConcept mem i j k l p;
        return (Or conlist);
       }

{- #conceptnames #rolenames depth #instancenames #probmod -}
genSomeConcept :: Int -> Int -> Int -> Int -> Int -> IO Concept
genSomeConcept i j k l p =
    do {
        roln <- genRnd 1 j;
        conn <- genConceptN i j k l p;
        return (Some (Role ("roln" ++ (show roln))) conn);
       }

{- #conceptnames #rolenames depth #instancenames #probmod -}
genAllConcept :: Int -> Int -> Int -> Int -> Int -> IO Concept
genAllConcept i j k l p =
    do {
        roln <- genRnd 1 j;
        conn <- genConceptN i j k l p;
        return (All (Role ("roln" ++ (show roln))) conn);
       }

{- generates a list of concepts of depth -}
{- #conceptnames size_of_the_list -}
genConcept1L _ 0 = return []
genConcept1L i rest =
    do {
        conc <- genRnd 1 i;
        opti <- genRnd 1 2;
        tail <- genConcept1L i (rest-1);
        case opti of
            1 -> (return ((Conn ("conn" ++ (show conc))):tail))
            2 -> (return ((Neg (Conn ("conn" ++ (show conc)))):tail))
       }

{- create concepts for a depth of one -}
{- #conceptnames -}
genConcept1 ::  Int -> IO Concept
genConcept1 i  =
    do {
        conc <- genRnd 1 i;
        opti <- genRnd 1 4;
        len <- genRnd 2 i;
        case opti of
            1 -> (return (Conn ("conn" ++ (show conc))))
            2 -> (return (Neg (Conn ("conn" ++ (show conc)))))
            3 ->
                do {
                    lst <- genConcept1L i len;
                    return (And lst);
                   }
            4 ->
                do {
                    lst <- genConcept1L i len;
                    return (Or lst);
                   }
```

```
       }

{- generate CONCEPTS -}
{- #conceptnames #rolenames depth #instancenames #probmod -}
genConceptN :: Int -> Int -> Int -> Int -> Int -> IO Concept
genConceptN i _ 1 _ _  = genConcept1 i
genConceptN i _ _ 0 _  = genConcept1 i
genConceptN i j k l p =
    do {
        conc <- genRnd 1 i;
        opti <- genRnd 1 100;
        if (opti <= p) then
            do{
                modu <- genRnd 1 2;
                case modu of
                    1 -> genSomeConcept i j (k-1) l p
                    2 -> genAllConcept i j (k-1) l p
              }
        else
            do{
                pro <- genRnd 1 4;
                case pro of
                    1 -> return (Conn ("conn" ++ (show conc)))
                    2 -> return (Neg (Conn ("conn" ++ (show conc))))
                    3 -> genAndConcept i j (k-1) l p
                    4 -> genOrConcept i j (k-1) l p
              }
       }

{- Generates a lists of concept assertions -}
{- #conceptass #conceptnames #rolenames depth #instancenames #probmod -}
genListConceptAss 0 _ _ _ _ _= return []
genListConceptAss m i j k l p=
    do {
        i1 <- selectInstance l;
        c1 <- genConceptN i j k l p;
        tail <- genListConceptAss (m-1) i j k l p;
        let
            head = (Ins i1 c1)
            list = (head:tail)
        in return(list)
       }

{- #rolenames -}
{- Generates a role name -}
genRoleName :: Int -> IO Role
genRoleName i  =
    do {
        reln <- genRnd 1 i;
        return (Role ("roln" ++ (show reln)));
       }

{- Generates a lists of role assertions -}
{- #roleass #rolenames #instancenames -}
genListRoleAss 0 _ _ = return []
genListRoleAss m i j =
    do {
```

```
        i1 <- selectInstance j;
        i2 <- selectInstance j;
        r1 <- genRoleName i;
        tail <- genListRoleAss (m-1) i j;
        let
            head = (Rel i1 i2 r1)
            list = (head:tail)
        in return(list)
       }

{- Generates a list of #roleass + #conceptass assertions -}
{- #roleass #conceptass #probmod #rolenames #conceptnames
 #instancenames depth -}
genListAssertions i j p k l m n =
    do {
        rolea <- genListRoleAss i k m;
        conca <- genListConceptAss j l k n m p;
        let
            list = rolea ++ conca
        in return(list)
       }

{- Generates ONE BABox and puts it in namefile -}
{- namefile #rolenames #conceptnames #instancenames #roleass
#conceptass #probmod depth #clauses lengtcl -}
genBABoxFile na rn cn ins ra ca p d c l =
    let
        tot = ra + ca
    in
        do {
            seed <- genRnd 1 10000;
            lst <- genListAssertions ra ca p rn cn ins d;
            err1 <- system ("./mkcnf " ++ (show tot) ++
                    " " ++ (show c) ++ " " ++
                    (show l) ++ " " ++ (show seed) ++
                    " -f> tmp.tmp");
            fstr <- readFile "tmp.tmp";
            let
                pf = parse (lexer fstr)
                seq = substCNF lst pf
            in do {
                writeFile na (show1 seq);
                return True;
                }
            }

{- Generates several BABOxes and puts them in files -}
{- #filename #rolenames #conceptnames #instancenames
#roleass #conceptass #probmod depth #clauses lengthcl numfiles -}
genBABoxFileS name rn cn ins ra ca p d c l 1 =
    genBABoxFile (name ++ ".1") rn cn ins ra ca p d c l
genBABoxFileS name rn cn ins ra ca p d c l nf =
    do{
        l1 <- genBABoxFile (name ++ "." ++ (show nf)) rn cn ins ra ca p d c l;
        l2 <- genBABoxFileS name rn cn ins ra ca p d c l (nf-1);
        return (l1 && l2)
     }
```

```
{-----------------------}
{- Main ----------------}
{-----------------------}

main :: IO ()

main =
   do {
       args <- getArgs;
       let {
            (noerror, fname) = treatParams args;
           } in
       if noerror
       then do {
        fstr <- readFile fname;
        let
         (KB tb ab) = formatKB(runParser (map toLower fstr));
         extension = ".racer";
         inknldg = "(in-knowledge-base " ++ fname ++ "abox "
            ++ fname ++ "tbox )\n"
         aboxcons = "(abox-consistent?)"
         (rep,rn) = onlyOneDefRole tb tb
        in if rep
           then putStr ("Double definition of " ++ (show rn) ++ "\n")
           else
        case (resStep(pulloutNot(spreadImp ab))) of
         [] -> putStr "ABox consistent\n"  >>
            writeFile (fname ++ extension) ";calls:0\n;size:0\n"
         [[]] -> putStr "ABox not consistent\n"  >>
            writeFile (fname ++ extension) ";calls:0\n;size:0\n"
         pf1 ->
           do{
            if (or (map isSHIQ tb))
            then
             do {
                (tot,size,res) <-
                    dpllRacerTrans tb pf1 [] (fname ++ extension) 0 0 ;
                if res
                then putStr "ABox consistent, file .racer created\n"
                else putStr "ABox not consistent, racer result\n"
                }
            else
             do {
                (tot,size,res) <-
                    dpllRacer tb pf1 [] (fname ++ extension) 0 0 ;
                if res
                then putStr "ABox consistent, file .racer created\n"
                else putStr "ABox not consistent, racer result\n"
                 }
            }
               }
       else putStr "ba2racer inputfile\n";
      }
```

Appendix B

Examples

1 PUZ001-1.p

```
%-----------------------------------------------------------
% File     : PUZ001-1 : TPTP v2.5.0. Released v1.0.0.
% Domain   : Puzzles
% Problem  : Dreadbury Mansion
% Version  : Especial.
%            Theorem formulation : Made unsatisfiable.
%English:
% Someone who lives in Dreadbury Mansion killed Aunt Agatha.
% Agatha, the butler, and Charles live in Dreadbury Mansion,
% and are the only people who live therein. A killer always
% hates his victim, and is never richer than his victim.
% Charles hates no one that Aunt Agatha hates. Agatha hates
% everyone except the butler. The butler hates everyone not
% richer than Aunt Agatha. The butler hates everyone Aunt
% Agatha hates. No one hates everyone. Agatha is not the
% butler. Therefore : Agatha killed herself.

%Someone who lives in Dreadbury Mansion killed Aunt Agatha.
%agatha, the butler, and charles live in Dreadbury Mansion,
->(rel charles agatha killed),(rel agatha agatha killed),
(rel butler agatha killed).

%and is never richer than his victim.
(rel charles agatha killed),(rel charles agatha richer)->.
(rel agatha agatha killed),(rel agatha agatha richer)->.
(rel butler agatha killed),(rel butler agatha richer)->.

%charles hates no one that Aunt agatha hates
(rel agatha agatha hates), (rel charles agatha hates) ->.
(rel agatha charles hates), (rel charles charles hates) ->.
(rel agatha butler hates), (rel charles butler hates) ->.
```

```
%No one hates everyone
->(ins agatha (at-most 2 hates *top* )).
->(ins charles (at-most 2 hates *top* )).
->(ins butler (at-most 2 hates *top* )).

%agatha hates everyone except the butler
-> (rel agatha agatha hates).
-> (rel agatha charles hates).

%a killer always hates his victim
(define-role killed :parents (hates))

%The butler hates everyone Aunt agatha hates.
(rel agatha agatha hates) -> (rel butler agatha hates).
(rel agatha charles hates) -> (rel butler charles hates).
(rel agatha butler hates) -> (rel butler butler hates).

%The butler hates everyone not  richer than Aunt agatha.
->(rel agatha agatha richer),(rel butler agatha hates).
->(rel butler agatha richer),(rel butler butler hates).
->(rel charles agatha richer),(rel butler charles hates).

%agatha is not the butler.
%default, unique name assumption included in RACER
```

Conjecture:

```
%agatha killed her self
(rel agatha agatha killed) -> .
```

2 PUZ003-1.p

```
%--------------------------------------------------------------
% File     : PUZ003-1 : TPTP v2.5.0. Released v1.0.0.
% Domain   : Puzzles
% Problem  : The Barber Puzzle
% Version  : Especial.
% English  :
% There is a barbers' club that obeys the following three
% conditions:
% (1) If any member A has shaved any other member B - whether
% himself or another - then all members have shaved A,
% though not necessarily at the same time.
% (2) Four of the members are named cesare, Lorenzo, Petrucio,
% and Cesare.
% (3) cesare has shaved Cesare.
% Prove Petrucio has shaved Lorenzo
%The Barber Puzzle
%One shaved then all shaved
(ins guido member),(rel guido guido shaved),
```

```
(ins petruchio member)->(rel petruchio guido shaved).
(ins guido member),(rel guido guido shaved),
(ins lorenzo member)->(rel lorenzo guido shaved).
(ins guido member),(rel guido guido shaved),
(ins cesare member)->(rel guido cesare shaved).

(ins guido member),(ins petruchio member),
(rel guido petruchio shaved),
(ins lorenzo member)->(rel lorenzo guido shaved).
(ins guido member),(ins petruchio member),
(rel guido petruchio shaved),
(ins cesare member)->(rel cesare guido shaved).

(ins guido member),(ins lorenzo member),
(rel guido lorenzo shaved),
(ins petruchio member)->(rel petruchio guido shaved).
(ins guido member),(ins lorenzo member),
(rel guido lorenzo shaved),
(ins cesare member)->(rel cesare guido shaved).

(ins guido member),(ins cesare member),
(rel guido cesare shaved),
(ins petruchio member)->(rel petruchio guido shaved).
(ins guido member),(ins cesare member),
(rel guido cesare shaved),
(ins lorenzo member)->(rel lorenzo guido shaved).


(ins petruchio member),(rel petruchio petruchio shaved),
(ins guido member)->(rel guido petruchio shaved).
(ins petruchio member),(rel petruchio petruchio shaved),
(ins lorenzo member)->(rel lorenzo petruchio shaved).
(ins petruchio member),(rel petruchio petruchio shaved),
(ins cesare member)->(rel cesare petruchio shaved).

(ins petruchio member),(ins guido member),
(rel petruchio guido shaved),
(ins lorenzo member)->(rel lorenzo petruchio shaved).
(ins petruchio member),(ins guido member),
(rel petruchio guido shaved),
(ins cesare member)->(rel cesare petruchio shaved).

(ins petruchio member),(ins lorenzo member),
(rel petruchio lorenzo shaved),
(ins guido member)->(rel guido petruchio shaved).
(ins petruchio member),(ins lorenzo member),
(rel petruchio lorenzo shaved),
(ins cesare member)->(rel cesare petruchio shaved).

(ins petruchio member),(ins cesare member),
```

```
(rel petruchio cesare shaved),
(ins guido member)->(rel guido petruchio shaved).
(ins petruchio member),(ins cesare member),
(rel petruchio cesare shaved),
(ins lorenzo member)->(rel lorenzo petruchio shaved).


(ins lorenzo member),(rel lorenzo lorenzo shaved),
(ins petruchio member)->(rel petruchio lorenzo shaved).
(ins lorenzo member),(rel lorenzo lorenzo shaved),
(ins guido member)->(rel guido lorenzo shaved).
(ins lorenzo member),(rel lorenzo lorenzo shaved),
(ins cesare member)->(rel cesare lorenzo shaved).

(ins lorenzo member),(ins petruchio member),
(rel lorenzo petruchio shaved),
(ins guido member)->(rel guido lorenzo shaved).
(ins lorenzo member),(ins petruchio member),
(rel lorenzo petruchio shaved),
(ins cesare member)->(rel cesare lorenzo shaved).

(ins lorenzo member),(ins guido member),
(rel lorenzo guido shaved),
(ins petruchio member)->(rel petruchio lorenzo shaved).
(ins lorenzo member),(ins guido member),
(rel lorenzo guido shaved),
(ins cesare member)->(rel cesare lorenzo shaved).

(ins lorenzo member),(ins cesare member),
(rel lorenzo cesare shaved),
(ins petruchio member)->(rel petruchio lorenzo shaved).
(ins lorenzo member),(ins cesare member),
(rel lorenzo cesare shaved),
(ins guido member)->(rel guido lorenzo shaved).


(ins cesare member),(rel cesare cesare shaved),
(ins petruchio member)->(rel petruchio cesare shaved).
(ins cesare member),(rel cesare cesare shaved),
(ins lorenzo member)->(rel lorenzo cesare shaved).
(ins cesare member),(rel cesare cesare shaved),
(ins guido member)->(rel guido cesare shaved).

(ins cesare member),(ins petruchio member),
(rel cesare petruchio shaved),
(ins lorenzo member)->(rel lorenzo cesare shaved).
(ins cesare member),(ins petruchio member),
(rel cesare petruchio shaved),
(ins guido member)->(rel guido cesare shaved).
```

```
(ins cesare member),(ins lorenzo member),
(rel cesare lorenzo shaved),
(ins petruchio member)->(rel petruchio cesare shaved).
(ins cesare member),(ins lorenzo member),
(rel cesare lorenzo shaved),
(ins guido member)->(rel guido cesare shaved).

(ins cesare member),(ins guido member),
(rel cesare guido shaved),
(ins petruchio member)->(rel petruchio cesare shaved).
(ins cesare member),(ins guido member),
(rel cesare guido shaved),
(ins lorenzo member)->(rel lorenzo cesare shaved).

%Four of the members are named
-> (ins cesare member).
-> (ins lorenzo member).
-> (ins petruchio member).
-> (ins cesare member).

%cesare has shaved Cesare
-> (rel cesare cesare shaved).
```

Conjecture:

```
%Prove Petruchio has shaved Lorenzo (conjecture)
(rel petruchio lorenzo shaved)->.
```

3 PUZ012-1.p

```
%------------------------------------------------------------------
% File     : PUZ012-1 : TPTP v2.5.0. Bugfixed v1.2.1.
% Domain   : Puzzles
% Problem  : The Mislabeled Boxes
% Version  : Especial.
%English  :
% There are three boxes a, b, and c on a table. Each box contains
% apples or bananas or oranges. No two boxes contain the same
% thing. Each box has a label that says it contains apples or says
% it contains bananas or says it contains oranges. No box contains
% what it says on its label. The label on box a says "apples".
% The label on box b says "oranges". The label on box c says
% "bananas". You pick up box b and it contains apples. What do
% the other two boxes contain?


%Each box contains something
->(rel boxa apples contains),(rel boxa oranges contains),
(rel boxa bananas contains).
->(rel boxb apples contains),(rel boxb oranges contains),
```

```
(rel boxb bananas contains).
->(rel boxc apples contains),(rel boxc oranges contains),
(rel boxc bananas contains).


%No two boxes contain the same thing
(rel boxa apples contains),(rel boxa oranges contains)->.
(rel boxa apples contains),(rel boxa bananas contains)->.
(rel boxa bananas contains),(rel boxa oranges contains)->.

(rel boxb apples contains),(rel boxb oranges contains)->.
(rel boxb apples contains),(rel boxb bananas contains)->.
(rel boxb bananas contains),(rel boxb oranges contains)->.

(rel boxc apples contains),(rel boxc oranges contains)->.
(rel boxc apples contains),(rel boxc bananas contains)->.
(rel boxc bananas contains),(rel boxc oranges contains)->.


(rel boxa apples contains),(rel boxb apples contains)->.
(rel boxa apples contains),(rel boxc apples contains)->.
(rel boxb apples contains),(rel boxc apples contains)->.

(rel boxa oranges contains),(rel boxb oranges contains)->.
(rel boxa oranges contains),(rel boxc oranges contains)->.
(rel boxb oranges contains),(rel boxc oranges contains)->.

(rel boxa bananas contains),(rel boxb bananas contains)->.
(rel boxa bananas contains),(rel boxc bananas contains)->.
(rel boxb bananas contains),(rel boxc bananas contains)->.

%Each box has a label
->(rel boxa apples label),(rel boxa oranges label),
(rel boxa bananas label).
->(rel boxb apples label),(rel boxb oranges label),
(rel boxb bananas label).
->(rel boxc apples label),(rel boxc oranges label),
(rel boxc bananas label).

%Label is wrong
(rel boxa apples label),(rel boxa apples contains)->.
(rel boxb apples label),(rel boxb apples contains)->.
(rel boxc apples label),(rel boxc apples contains)->.
(rel boxa oranges label),(rel boxa oranges contains)->.
(rel boxb oranges label),(rel boxb oranges contains)->.
(rel boxc oranges label),(rel boxc oranges contains)->.
(rel boxa bananas label),(rel boxa bananas contains)->.
(rel boxb bananas label),(rel boxb bananas contains)->.
(rel boxc bananas label),(rel boxc bananas contains)->.
```

```
%Box A labelled apples
->(rel boxa apples label).
%Box B labelled oranges
->(rel boxb oranges label).
%Box C lablelled bananas
->(rel boxc bananas label).

%Box B contains apples).
->(rel boxb apples contains).
```

Conjecture

```
%boxa contains bananas and boxc oranges
(rel boxa bananas contains),(rel boxc oranges contains)->.
```

Bibliography

[ABM02] C. Areces, P. Blackburn, and M. Marx. Description logic with boolean aboxes. 2002.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *ACM*, 5:394– 397, 1962.

[HM] V. Haarslev and R. Möller. RACER: Renamed abox and concept expression reasoner. `http://kogs-www.informatik.uni-hamburg.de/~race/`.

[HM01a] V. Haarslev and R. Möller. Description of the RACER system and its applications. In *Proc. International Workshop on Description Logics (DL-2001)*, pages 1–3, Standford, USA, August 2001.

[HM01b] V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 4–10, Seattle, Washington, USA, August 2001.

[HST00] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In David MacAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, number 1831 in Lecture Notes in Computer Science, Germany, 2000. Springer-Verlag.

[J$^+$99] S. Peyton Jones et al. *Standard Libraries for the Haskell 98*, January 1999.

[SS] G. Sutcliffe and C. Suttner. TPTP problem library. `http://www.math.miami.edu/~tptp/`.