Implementation-Design for the CoBa Belief Change Logic *Revision* : 1.0

Sven Thiele <sthiele@rz.uni-potsdam.de>

— *Preliminary Draft* —

`design.tex` — `May 30, 2005` at `13:48h`	
Read ◯	*Modified* ◯

1 Introduction

In this document I describe a Beliefchange-logic for CoBa its architectural design the central classes with its interfaces, the semantics and some implementation details.

2 General Considerations

The following design of the CoBa -beliefchange-logic should provide a framework of interfaces, open to further development, that gives the ability to dynamicly change the implementing classes by reusing most of the older ones. The Interfaces evolves around the following basic requirements and ideas.

1. We want to provide an easy way to compute beliefchanges accordingly to the CoBa -approach. The *BeliefChangeScenario* class provides an minimal interface to set up an *BeliefChangeScenario* and compute the resulting knowledgebases by hiding the implementation and the other used classes from the user.

2. We need a representation of the logical sentences that allows us to do some basic transformation operations and gives essential informations about the logical sentence. The *ISent* objects are our represenation of the logical sentences.

3. We have a stringbased repesentation of the logical sentences that need to parsed and from that our *ISent* representation has to be created. Therefor the *SentenceParser* is needed to create *ISent* objects from Strings.

4. We want to use a satisfiability library like sat4j to check our logical sentences for consistency. The *Checker* class wraps the access the sat-library and provides an interface for easy and fast checking the consistency of logical sentences.

5. Where the beliefchange algorithm is based on finding maximal consistent equivalence sets, we need an easy way to create and access these equivalences. The *EquivalenceSet* class has an interface that allows creating equivalences and to provides needed informations about equivalence sets.

3 Proposed Interfaces

The design presented here specifies interfaces, it does not, or at least tries not to, dictate an implementation. There are infinitely many possible differing ways to implement CoBA ensuring the qualities mentioned above. The following sections present interfaces for the concepts CoBA deals with: logical sentences, parsers, the set of equivalences and the sat checker. The interfaces are presented in IDL (Interface Definition Language). There are no constructors in the interfaces as these depend on a concrete implementation. However, in the paragraphs accompanying an interface possible initializers are described.

3.1 Belief Change Scenario

The core of the CoBA -logic is the *BeliefChangeScenario* class it has to implement the actual CoBA -algorithm provide the setter methods for the components of the concrete *BeliefChangeScenario*, additional setter methods to calibrate optional computation steps like the used search-algorithm, the used way of merging and to set choice or sceptical change and last but not least a method to start the CoBA -algorithm and return the resulting knowledgebase.

Listing 1: BeliefChangeScenario.idl

```
// Indicates that an logical sentence
// passed to the BeliefChangeScenario
// has syntactical Errors
exception SyntaxErrException {};

// Indicates that there are mutually inconsistent
// constraints
exception InconsistencyException{};

// BeliefChangeScenario represents a set of logical sentences that
// should be merged revised or contracted and it implements the
// change-algorithm that computes the resulting knowledgebase
interface BeliefChangeScenario
{

  // Sets the knowledge bases of the BeliefChangeScenario to be the
  // same as the Strings in the Vector K. If K is null or empty,
  // then the knowledge base is set to T by default.
  // K is the Vector of knowledge bases represented as Strings.
  // throws SyntaxErrException if one of the knowledge bases has
  // syntactical errors.
  void setKB(Vector K) throws SyntaxErrException

  // Sets the revision-sentence in the BeliefChangeScenario to R.
  // If R is null or empty, then the revision sentence is set to T
  // by default.
  // R is a string representation of the revision sentences.
  // throws SyntaxErrException if the revision sentence has
  // syntactical errors.
  void setRevisor(String R) throws SyntaxErrException

  // Sets the contraction sentences to be the negations of those in
  // the Vector C.
  // C is the Vector of contraction sentences, each represented as
  // an String.
  // throws SyntaxErrException if an contraction sentence is not
  // syntactical correct.
```

```
    void setContractor(Vector C) throws SyntaxErrException

    // Sets the entailment-based integrity constraints to ICe.
    // ICe  is a string representation of the entailment-based
    // integrity constraints.
    // throws SyntaxErrException if the EBIC is not
    // syntactical correct.
    void setEBIC( String ICe ) throws SyntaxErrException

    // Sets the consistency-based integrity constraints to those in
    // the Vector ICc.
    // ICc the Vector of consistency-based integrity constraints
    // represented as Strings.
    // throws SyntaxErrException if one of the integrity constraints
    // has syntactical errors.
    void setCBIC( Vector ICc ) throws SyntaxErrException

    // Sets the equivalence set search algorithm to algo .
    // algo is a String value indicating which searchalgorithm is to
    // be used to find the maximal equivalencesets.
    boolean setAlgo( String algo )

    // Sets the change type to choice change if isChoiceOn is true,
    // or to skeptical change otherwise.
    // isChoiceOn is a boolean value indicating whether choice change
    // is to be applied or not.
    void setChoiceType( boolean isChoiceOn )

    // Sets the merge type to projection merge if isProjOn is true,
    // or to default merge otherwise.
    // isProjOn is a boolean value indicating whether
    // projection-merge is to be applied or not.
    void setProjMerge( boolean isProjOn )

    // Checks if there are inconsistencies among the parameters of
         this
    // BeliefChangeScenario, executes this BeliefChangeScenario using
    // the selected change type, search algorithm and merge type,
    // and finally returns  the resultant knowledge base.
    // throws an InconsistencyException if there are inconsistencies
    // among the parameters of this BeliefChangeScenario.
    // returns a Vector of the resultant knowledge bases.
    Vector change() throws InconsistencyException

};
```

The *BeliefChangeScenario* implements the actual CoBa -algorithm and provides a minimal interface by hiding the other used classes of the beliefchange-logic. To make clear how the *BeliefChangeScenario* works and how it uses the other classes of the beliefchange-logic I will give an abstract description of its lifecircle showing how this class should be used and how the implemented algorithm works.

1. Create a *BeliefChangeScenario* object via a public constructor `BeliefChangeScenario ()`

2. Setup the components of the *BeliefChangeScenario* using the provided setter methods

   - `BCS.setKB(Vector K);`

- `BCS.setRevisor(String R);`
- `BCS.setContractor(Vector C);`
- `BCS.setCBIC(Vector ICc);`
- `BCS.setEBIC(String ICe);`

Within these methods of the *BeliefChangeScenario* a *SentenceParser* is used to create *ISent* objects from the *String* representation of the logical sentences, these *ISent* objects are the internal represenation of logical sentences on which we work. It may happen that synatctical maleformed sentences are passed to these methods, in this case a *SyntaxErrException* is thrown.

3. Calibrate the optional computation steps of the CoBa -algorithm

   - `BCS.setAlgo(String algo);`
   - `BCS.setChoiceType(boolean ChoiceIsOn);`
   - `BCS.setProjectionType(boolean ProjIsOn);`

4. Start the CoBa -algorithm using the method `change();`

5. Check for inconsistent sentences of the *BeliefChangeScenario.* Within the `change()` method at first a instance of the class *Checker* is used to test if the components of the *BeliefChangeScenario* are mutually consistent if not a *InconsistencyException* is thrown.

6. Next step is to create new renamed logical sentences based on the CoBa -algorithm, again a *SentenceParser* is used to create the new sentences as *ISent* objects.

7. Creating the equivalences. The CoBa -algorithm is based on finding maximal sets of consistent equivalences. To create and manage these equivalences an instance of the *EquivalenceSet* class is used. In this step the following methods of the *EquivalenceSet* are used to create and store the equivalences.

   - `EQSet.addEquivalences_NumberedtoPrimed(Collection c,int n);`
   - `EQSet.addEquivalences_PrimedtoUnprimed(Collection c);`

8. The *EquivalenceSet* also provides a logical sentences that allows to activate a equivalence by making a single atom true. To get this sentence the method `getEQPart()` is called after all needed equivalences have been created.

9. Initiate the *Checker.* For each logical sentence (There may exist more than one because of mutually inconsistent contractors and consistency-based integrity constraints.) that needs to be checked against the maximal sets of equivalences a *Checker* ist created and initialized with the logical sentence.

   - `C.init(ISent s);`

10. Search For maximal consistent sets of equivalences. The search-algorithm implemented in the *BeliefChangeScenario* are walking systematical through the searchspace, using *BitSets* to represent an equivalence set. The search-algorithm is using the *EquivalenceSet* method `getEQSetasVecInt(BitSet eqset )` to get a *VecInt* representation of a equivalence set. The *VecInt* is numerical encoding of the equivalence set where each equivalence atom that should be true is encoded as integer number. The *VecInt* is passed into the `check(VecInt v)` method of the *Checker* to test the consistens.

11. After the maximal sets of equivalences have been found the resulting knowledgebases can be created and returned.

3.2 ISent

ISent objects are the representation of the logical sentences. An *ISent* class has to provide methods to get information about the type of the object if it represents a atomic sentence, a disjunction, conjunction, implication, a tautology or a contradiction, it must provide information about how deep the logical sentence is nested and which atoms are used in this sentence. It also needs to provide some basic transformation operations and a *String* representation of the sentence.

Listing 2: ISent.idl

```
// ISent represents a logical sentence.
interface ISent
{
  // Returns the type of this sentence.
  // The method returns an int indicating the sentence type.
  int getType();

  // Returns the depth (levels of nesting) of this sentence.
  // The method returns an int indicating the depth of this
       sentence.
  int getDepth();

  // Returns a set consisting of the atoms of this sentence.
  // The method returns a HashSet consisting of the atoms of this
       sentence.
  HashSet getAtoms();

  // Returns the string representation of this sentence.
  String toString()

  // Returns an equivalent ISent sentence, in which, all the
       implications have been
  //  replaced with disjunctions, and all the biconditionals with a
        conjunction of two disjunctions.
  ISent implicationsOut();

  // Returns an equivalent ISent sentence, in which the negation
       signs appear only on the atoms.
  ISent negationsIn();

  // Returns an equivalent ISent sentence, in which the outer
       disjunction signs are moved inside.
  ISent disjunctionsIn();

  // Returns an equivalent ISent sentence, in which the outer
       conjunction signs are moved inside.
  ISent conjunctionsIn();

};
```

3.3 SentenceParser

The *SentenceParser* class is used to create *ISent* objects from *String* representations of logical sentences, it also has to check the syntax of the parsed sentences.

Listing 3: SentenceParser.idl

```
// The SentenceParser is used to create ISent objects from string
     representations.
interface SentenceParser
{
  // Parses the string s and returns the coresonponding ISent
       sentence.
  // s is the string to be parsed as an ISent sentence.
  // The method throws an SyntaxErrException if s is syntactically
       malformed.
  // The method returns the corresponding ISent sentence for s.
  ISent parseString(String s) throws SyntaxErrException

};
```

3.4 Checker

The *Checker* is a wrapper for the actually used satisfiability library. A *Checker* should be initialised with a logical sentence provide a method to check if this sentence is consistent with a clause.

Listing 4: Checker.idl

```
// The Checker wraps the access the sat-library and provides an
     interface
// for easy and fast checking the consistency of logical sentences.
interface Checker
{
  // Initializes the Checker with the logical sentence s. The
       sentence s is transformed into a cnf
  // and the clauses are fed into the Solver.
  // The parameter s represents the logical sentence that should be
        checked against clauses.
  void init(ISent s)

  // With this method you can check if the logical sentence used to
        initialize the checker is consistent with
  // the clause represented by the VecInt v.
  // The parameter v VecInt is a Datastructure from the SAT4J
       Library and represents a logical clause a simple conjunction
        of literals represented as Integer value.
  // If a Atom should be true its integer value is in the VecInt
       and if it should be false it negative IntegerValue is in the
        VecInt.
  // The returns true if the clause AND the initialization sentence
        are satisfiable else false.
  boolean check(VecInt v)

};
```

3.5 EquivalenceSet

The *EquivalenceSet* class has an interface that allows creating equivalences and to provides needed informations about the equivalence set. The class has to provide the number of equivalences stored in the *EquivalenceSet* and a mapping between the *BitSet* representation of a equivalence set and Vector of equivalences.

Listing 5: EquivalenceSet.idl

```

interface EquivalenceSet
{
  // Creates a set of equivalences from the strings in c with the
       superscript n. For example,
  // given a c containing {p, q} and an int i, it creates a vector
       of ISents {p^i=p',q^i=q'}.
  // The parameter c represents the Collection of strings from
       which to generate equivalences.
  // The parameter n is an int indicating the superscript for the
       generated equivalences.
  void addEquivalences_NumberedtoPrimed(Collection c,int n)

  // Creates a set of equivalences from the strings in c. For
       example, given a c containing
  // {p, q}, it creates a vector of ISents {p=p', q=q'}.
  // The parameter c represents the Collection of strings from
       which to generate equivalences.
  void addEquivalences_PrimedtoUnprimed(Collection c)

  // Returns the corresponding set of atoms for the BitSet bs.  For
        example, if a bit k in bs
  // is set, then the returned set of atoms will contain the first
       atoms of the equivalence indexed k.
  // The parameter bs is the BitSet whose corresponding atoms are
       to be returned.
  //  The method returns a Vector containing the corresponding
       atoms for bs.
  Vector getAtomsFromEQSet(BitSet bs )

  // We want to identify a equivalence stored in this class by  a
       single atom. With this method a
  // new logical sentence is created
  // for each equivalence stored in this class there is a sentence
       (EQ1 > p=p') the conjuction of all these sentence is what i
       call EQPart of the
  // logical formula that we like to check against EQSets and what
       this method returns.
  // The method returns the ISent representation of the conjunction
        of all implications (newEQAtom > p=p')
  // stored in this EquivalenceSet
  ISent getEQPart()

  // Returns the corresponding set of equivalences for the BitSet
       bs.  For example, if a bit k in
  // bs is set, then the returned set of equivalences will contain
       the biconditional indexed k.
  // The parameter bs is the BitSet whose corresponding
        equivalences are to be returned.
  // The method returns a Vector containing the corresponding
        equivalences for bs.
  Vector getEQSet(BitSet bs)

  // Returns the corresponding VecInt representation for the BitSet
         bs.  For example, if a bit k in bs
  // is set, then the returned VecInt will contain the id of the
        equivalence indexed k.
  // The parameter bs is the BitSet whose corresponding VecInt is
        to be returned.
  // The method returns a VecInt containing the corresponding atom
        ids for bs.
  VecInt getEQSetasVecInt(BitSet bs)

  // Returns the number of equivalences of this EquivalenceSet.
  int getNumberOfEquivalences()

};
```

4 Implementation considerations

In this section the implementation of the above mentioned interfaces should be discussed, with respect to the current state, further development and reuse of of actually implementing classes.

With respect to maybe many implementations care must be taken to ensure an implementation of the interfaces works with all of them. This means that all implementations should work using only the defined interfaces and they should reuse as much as possible the already defined classes. It is recommended to use only a *SentenceParser* to create *ISent* objects and to use a single *EquivalenceSet* to create and store all the equivalences. The implementations shall be defined in the package *beliefchangelogic*.

An implementation for ISent should provide the methods `equals()` and `clone()` and the `toString()` method should be used as standard for printing in all classes.

4.1 BeliefChangeScenario

The BeliefChangeScenario is the central class it can be devided into 3 main tasks, that are

1. to setup the logical formulas that in fact have to be checked against the set of equivalences,
2. find the maximal sets of equivalences and
3. construct the resulting knowledgebase.

How the setup of the formulas that are checked against the set of equivalences is done depends on whether there is projection merge or symetric merge switched on. Depending on which setup process has been choosen the following things may differ, the formulas that are checked against the equivalence sets and the set of equivalences it self. In the actually implementation those results of the setup process are stored in attributes of the *BeliefChangeScenario* the instance of the class *EquivalenceSet* and a *Vector* of *Checker* instances initialised with the formulas which have to be checked against the equivalences. It seems that the setup process does not have any further influence to the beliefchange process. This gives thought to a further decomposition of the beliefchange process and maybe should lead to an encapsulation of the setup process and defining a interface that returns the above mentioned parameters. Actually there exist two methods `setupsymetic()` and `setupprojection()` which need as input the set of knowledgebases to merge a revision sentence and a set of contraction sentences.

setupsymetric

- At first all the atoms are appear in more that one knowledgebase that should be merged are stored in `m_KBCommonAtoms`.
- Then all atoms that appear in a knowledgebase to merge and also in the revision or contraction sentence are stored in `m_CommonAtoms`.

- Then the equivalences are created. For each knowledgebase that has to be merged and for all the atoms that appear in this knowledgebase as well as in `m_KBCommonAtoms`, the method `addEquivalences_NumberedtoPrimed()` is called to create the equivalence `(a'=a^n)` where `n` is the number of the knowledgebase.

- For all atoms in `m_CommonAtoms` the method `addEquivalences_PrimedtoUnprimed()` is called to create the equivalences `(a=a')`

- Then we get the a logical sentence from the *EquivalenceSet* that allows to make a equivalences true by choosing a single atom true. This sentences is called EQ-part and is a conjuction of implications where so called EQ-atom implies a equivalence like `(EQ1 > (a'=a2))`.

  ```
  getEQPart();
  ```

- The next step is to create new renamed sentences from the knowledgebases that have to be merged. For each knowledgebase a new renamed is created where the atoms in `m_CommonKBAtoms` are replaced by atoms numbered with index of the knowledgebase. For example knowledgebase `1` is represented by `((a&b)+c))` and `m_CommonKBAtoms` contains `c` the new sentence is `((a&b)+c^1)`. This is done using the method `primeString(String s , Collection<String> c, int n)` which breaks the logical sentences represented by the *String* `s` into parts down to the atoms and if the atom is contained in the *Collection* `c` adds the superscript `n`.

  ```
  atom = atom+"^"+n;
  ```

- The remaining Atoms that are contained in `m_CommonAtoms` are replaced by primed atoms following the same procedure but calling `primeString()` with `0` as the integer value which causes that only a `"'"` is added. For example the input sentence is the result of the former step `((a&b)+c^1)` and `m_CommonAtoms` contains `c` and `a` the result is `((a'&b)+c^1)`, because `c` could not be primed it has already been replaced by `c^1` and only `a` remains to be replaced by `a'`. The resulting sentences are stored in `m_KBNumbered`.

- The setup for symetric merge also produces sentences that have only been primed by calling `primeString()` with the original sentence and `m_CommonAtoms` and `0`. The resulting sentences are stored in `m_KBPrimed`.

- Then the sentences that need to be checked against the set of equivalences are created. For symetricmerge these are for each contraction sentence, the disjunction of the sentences in `m_KBPrimed` conjoined with the sentences in `m_KBNumbered` conjoined with the revision sentence the EQ-Part and the contraction sentence. For each contradiction sentence a *Checker* is created and initialised with the appropriate sentence and the *Checker* objects are stored in `m_KBURevUCon`.

- As the final result of the setup process we got the *EquivalenceSet* with all equivalence we have to test and a set of *Checker* objects initialised with the formulas we want to check against the set of equivalences.

The next step the search for a set of maximal sets of equivalences consistent with the formulas can also be encapsulated as there already are implemented some different search strategies. We have to distinguish two kinds of search algorithms here

1. Search algorithms that are based on a maximality check and find one first maximal set of equivalences and can then compute more on demand until all sets are found.

2. Search algorithms that have to search the whole searchspace to check for maximality and then have found all maximal sets.

To define a common interface for both kind of search-algorithms this interface would need iterator methods to access the maximal sets like `getFirst()` and `getNext()` as well as a method to get the set of all maximal sets like `getAll()`. The current implementation has only search algorithms that have to search all solutions and return the whole set, but in the further development of coba there is a high probablity that some more will be implemented.

AllDetEq1 This search algorithm does an depthfirst search and spans an binary searchtree, it performs maximality checks to determine the maximal consistent sets and collects them.

Algorithm 1: AllDetEq1

```
Global: A set of knowledgebases $K$.
Input : Three sets of atoms $C$,$D$ and $E$.
begin
    if $C = \emptyset$ then
        isMaximal:= false;
        foreach $k \in K$ do
            if not Sat( $k \cup E \cup \bigvee D$) then
                isMaximal:= true;
                break;
        if isMaximal then
            Add2MaxConsistent($E$);
        return;
    choose $q \in C$;
    sat:= true;
    foreach $k \in K$ do
        if not Sat($k \cup \bigcup E \cup \{q\}$) then
            sat:= false;
            break;
    if sat then
        AllDetEq1( $C \setminus \{q\}$, $D$, $E \cup \{q\}$);
        AllDetEq1( $C \setminus \{q\}$, $D \cup \{q\}$, $E$);
    else
        AllDetEq1( $C \setminus \{q\}$,$D$, $E$);
end
```

AllDetEq2_2 This search algorithm does an depthfirst search and spans an asymetric searchtree it collects maximal consistent sets to avoid satchecks.

Algorithm 2: ALLDETEQ2_2

```
Global: A set of knowledgebases $K$.
Input : Two sets of atoms $ActualEqs$, and $NottestetEqs$, and the last
        Atom added to $ActualEqs$ $lastadded$.
begin
    if $ActualEqs \subseteq F$ for some $F \in MaxConsistent$ then
        /* ActualEqs is known to be consistent                    */
        while $NottestedEqs \neq \emptyset$ do
            choose $p \in NottestedEqs$;
            $NottestedEqs := NottestedEqs \setminus \{p\}$;
            ALLDETEQ2_2($ActualEqs \cup \{p\}$,$NottestedEqs$, $p$);
        return;
    /* ActualEqs have to be checked                               */
    sat:= true;
    foreach $k \in Ks$ do
        if not SAT($k \cup \bigcup ActualEqs$) then
            sat:= false;
            break;
    if sat then
        if $NottestedEqs = \emptyset$ then
            ADD2MAXCONSISTENT($ActualEqs$);
        while $NottestedEqs \neq \emptyset$ do
            choose $p \in NottestedEqs$;
            $NottestedEqs := NottestedEqs \setminus \{p\}$;
            ALLDETEQ2_2( $ActualEqs \cup \{p\}$,$NottestedEqs$, $p$);
    else
        if $NottestedEqs \neq \emptyset$ then
            if not $ActualEqs \setminus \{lastadded\} \subseteq F$ for some
            $F \in MaxConsistent$ then
                /* remove last added Atom from ActualEqs and add
                   it the to maximal consistent                   */
                ADD2MAXCONSISTENT($ActualEqs \setminus \{lastadded\}$)
end
```

AllDetEq3 This search algorithm does an depthfirst search and spans an asymetric searchtree it collects maximal consistent sets.

Algorithm 3: AllDetEq3

```
Global: A set of knowledgebases $K$.
Input : Two sets of atoms $C$, and $E$
begin
    existbigger:= false;
    while $E \neq \emptyset$ do
        choose $q \in E$;
        $E := E \setminus \{q\}$;
        sat:= true;
        foreach $k \in K$ do
            if not Sat($k \cup \bigcup C \cup \{q\}$) then
                sat:= false;
                break;
        if sat then
            existbigger:= true;
            AllDetEq3( $C \cup \{q\}$, $E$);
        else
            increase heuristic value for q;
        if checkForConsistentSuperSets($C \cup E$) then return;
    if not existbigger then
        Add2MaxConsistent($C$);
end
```

AllDetEQ4

Heuristic Some of the current implemented searchalgorithms like AlldetEQ2_2 and AllDetEQ3 make use of a heuristic to improve search. The heuristic used is as simple as effictive, for each atom a heuristic value is stored, this value is initialy 0, if the atom is a definte cause for a conflict (adding this atom causes a inconsitent set) the heuristic value of this atom is increased. The heuristic choose function always returns a atom with the highest heuristic value. By applying this heuristic the searchalgorithm should find faster the minimal inconsitent sets to cut the searchspace.

The third step the construction of the resulting sentence can start after the search process has finished and a set of solutions is provided. There are two kind of construction methods implemented, the optimistic which chooses one solution and creates a new sentence by renaming and the sceptical which uses all solutions and returns a disjunction of the new renamed sentences.

conjoin_UnPrimedAndRevisor

- The input for the renaming is the conjuction of the numbered and primed sentences in `m_KBNumbered` which is first unnumbered which means that in the sentence each atom numbered with `n` for which an equivalence `(a'=a^n)` in the selected solution existd is replaced by `a'`. If there is no such equivalence for the numbered atom it is replaced by the negation of its primed version `~a'`. After this step no numbered atoms remain in the sentence, this has basically removed the conflicts between the knowledge-bases to merge.

- The next renaming step is to unprime the remaining primed atoms. If the unprimed version dont belong to `m_CommonAtoms` they can be simply replaced by it's unprimed version, because they are only a results of the unnumbering step and not cause to a conflict with the revision sentence. If the unprimed version belongs to `m_CommonAtoms` and an equivalence `(a=a')` exists in the solution they are replaced by the unprimed version `a`. If the unprimed version belongs to `m_CommonAtoms` but no equivalence `(a=a')` exists in the solution it has to replaced by the negated unprimed version `~a`. After this step no more primed atoms exist in the sentence and the conflict to the revision sentence is solved.

- The next step is to removed the information from the contraction sentence which means each atom `a` that belong to the `m_CommonAtoms_Con_KB` and for this no equivalence `(a=a')` exist in the solution, has to be removed. This is done by replacing these atoms with the tautology or the contradiction. For example we have the logical sentence `((a&b&c)+d)` and the set of atoms that have to be contracted is `{a,d}` we create for every possible interpretation of this set of atoms a new sentence and disjoin them `((F&b&c)+F)+((T&b&c)+F)+((F&b&c)+T)+((T&b&c)+T)`. In the resulting sentence all the knowledge conflicting with the contractors are removed.

The BeliefChangeScenario returns as result a *Vector* of the resulting knowledgebases.

4.2 ISent

ISent objects are a internal representation of the logical sentences in the coba-framework. We want create all kind of logical sentences as *ISent* objects to deal with them over a common set of methods. To provide this common interface within the current implementation *ISent* is a abstract base class for different sub classes implementing the different types of logical sentences, so *ISent* can be seen as a collection of classes implementing all types of logical sentences. This collection implementing *ISent* needs at first to provide methods to create all kind of sentences as their are the constructor of an atomic sentence `Atom(String s)` the methods to create negation `makeNeg(ISent s)`, disjunction `makeDis(Collection<ISent> c)`, conjuction `makeConj(Collection<ISent> c)`, implication `makeImp(ISent s1, ISent s2)` and equivalences `makeBiImp(ISent s1, ISent s2)`. There also exist implementation of the contradiction and the tautology for those we dont need construction methods because they exist as single static objects. In Fact there exist six diffent classes implementing *ISent*

Atom is implementing the atomic sentences providing the `Atom(String s)` constructor to create a atoms labeled with the *String* s. The clas *Atom* also has register new created *Atoms* in the *AtomRegistry* and unregsiter them if the object is destroyed.

Negation is implementing the negation of sentences and provides the construction method `makeNeg(ISent s)`.

BinaryCompound is implementing implications and equivalences and provides the construction methods `makeImp(ISent s1, ISent s2)` and `makeBiImp(ISent s1, ISent s2)`.

MultipleCompound is implementing conjuctions and disjuctions of sentences and provides the construction methods `makeConj(Collection<ISent> c)` and `makeDis(Collection<ISent> c)`.

Contradiction is implementing the aways false sentence and provides the static attribute `bottom` representing the one existing contradiction.

Tautology is implementing the aways true sentence and provides the static attribute `top` representing the one existing tautology.

All *ISent* objects should only be created via the *SentenceParser* which uses the construction method of *ISent* and makes sure that the created sentences are syntactical correct.

4.3 SentenceParser

The *SentenceParser* is used everywhere in the CoBa -framework to create *ISent* objects from stringrepresentations in *EquivalenceSet* to create the equivalences in *BeliefchangeScenario* to create the renamed sentences in the *Checker* to create new sentences in CNF. The *SentenceParser* keeps track that no forbidden symbols are used in the sentences and that the parenthesis are set correctly.

4.4 Checker

The *Checker* class is a wrapper for the satchecking engine we use, actually the sat4j library. The *Checker* has to be initialised with an *ISent* object that should be checked against several sets of atoms. Within this initialisation at first a CNF-formula is created and fed into the solver provided by the sat library. It may happen that this formula itself is a contradiction, what means that no matter which sentences are added to this formula it always stays false. If the formula is contradiction an attribute indicating this is set to true. The check method uses the initialised solver to test it against a conjuction of atoms representing a partial interpretation of the formula and returns `true` if it is the subset of a model and `false` otherwise. If the attributes of *Checker* are indicating that the formula is a contradiction it the sat check is skipped and `false` is returned.

4.5 EquivalenceSet

The *EquivalenceSet* is the class that is used to create and manage the equivalences between the different knowledgebases. Two different kind of equivalences can be created, equivalences between normal atoms and primed atoms like $(a \equiv a')$ and equivalences between numbered atoms and primed atoms like $(a^2 \equiv a')$. These equivalences are created with the methods `addEquivalences_PrimedtoUnprimed (Collection<String> c)` and `addEquivalences_NumberedtoPrimed(Collection<String> c ,int n)` which work basically this way.

At first the new Atoms are created with the *SentenceParser* and stored in the *EquivalenceSet*.

`primedAtom = parseString(a+"'"); //a'` the primed version of an atom.

`numberedAtom = parseString(a+"^"+n); //a^2` the numbered version of an atom.

`EQAtom = parseString("EQ"+k); //EQ1` a new atom (EQ-Atom).

Then the equivalences are created and stored.

`parseString("("+a+"="+a+"')"); //(a=a')` a equivalence to primed version of an atom.

`parseString("("+a+"'="+a+"^"+n+")"); //(a'=a^2)` a equivalence between the primed version of an atom and a numbered version of an atom.

At next a implication is created and stored that allows making an equivalence true by choosing a single EQ-Atom true.

`parseString("(EQ"+k+">("+a+"'="+a+"^"+n+"))"); //(EQ1 > (a'=a^2))`

To use the feature of using EQ-Atoms to make an equivalence true in a formula it is necessary to conjoin the formula with the sentence provided by method `getEQPart()` of the *EquivalenceSet* which simply returns a conjunction of all these stored implications.

This article was processed using the comments style on May 30, 2005.
There are 0 comments to be processed.