Capitolo 1

Implementazione di procedure di decisione per frammenti Binding in Vampire

(descrizione dell'approccio, minime modifiche al kernel, utilizzo di funzionalità esistenti,)

1.1 Algoritmo di Classificazione

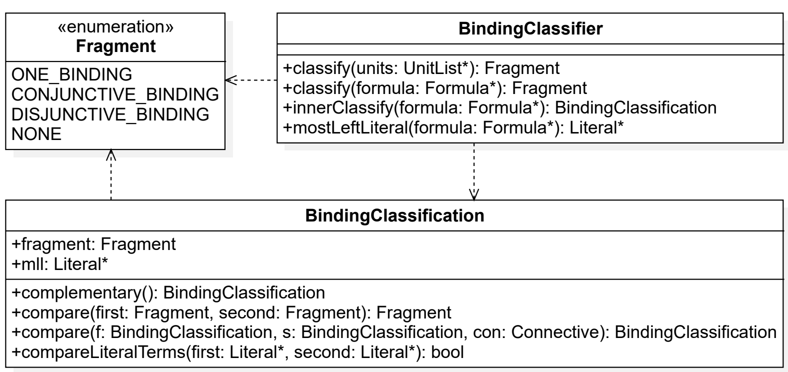

Figura 1.1: Classificatore

La precondizione più importante per la correttezza dell'algoritmo di decisione è che la formula faccia parte del frammento $\forall 1B$. Per questo motivo è stato creato un classificatore che prende in input una formula rettificata e restituisce l'elemento del frammento a cui appartiene. Una formula è rettificata se non contiene $\top$ o $\bot$. L'algoritmo non fa altro che verificare la forma sintattica della formula e capisce a quale grammatica della sezione **??** appartiene. Per questo scopo sono state create due funzioni delle Classificatore Esterno (Algoritmo 1) e Classificatore Interno (Algoritmo 3). La prima verifica la parte della formula senza quantificatori mentre la seconda verifica la parte interna ai quantificatori e confronta i termini dei letterali. Entrambi gli algoritmi hanno una struttura di visita dell'albero sintattico in postOrder e hanno una complessità lineare rispetto alla dimensione della formula.

Algorithm 1: Classificatore esterno

Firma: classify(φ) **Input:** φ Una formula rettificata
Output: Un elemento dell'enumerazione Fragment

```
switch φ do
    case Literal do
        return ONE_BINDING;
    end
    case A[∧, ∨]B do
        return compare(classify(A), classify(B));
    end
    case ¬A do
        return classify(A).complementary();
    end
    case [∀, ∃]A do
        sub := φ;
        connective := connective of φ;
        repeat
            sub := subformula of sub;
            connective := connective of sub;
        until connective ∉ {∀, ∃};
        (fragment, _) := innerClassify(sub);
        return fragment;
    end
    case A ⇔ B do
        return compare(classify(A ⇒ B), classify(B ⇒ A));
    end
    case A ⊕ B do
        return classify(A ⇔ B).complementary();
    end
    case A ⇒ B do
        return compare(classify(¬A), classify(B));
    end
end
```

Algorithm 2: Compare esterno

Firma: compare(A, B) **Input:** A, B due elementi dell'enumerazione Fragment
Output: Un elemento dell'enumerazione Fragment

```
if A = B then
    return A;
end
if One_Binding ∉ {A, B} then
    return None;
end
return max(A, B);
```

Il classificatore esterno si appoggia ad una funzione ausiliaria chiamata *compare* che prende in input due elementi dell'enumerazione Fragment e restituisce il frammento risultante dalla combinazione booleana ($\wedge, \vee$) dei due frammenti. La combinazione di due *1B* è sempre un *1B* mentre la combinazione di un *1B* con un *CB* o *DB* è sempre un *CB* o *DB* . Infine la combinazione di un *CB* con un *DB* fa parte del frammento Boolean Binding che però in questa sezione verrà chiamato *None*. Per la comparazione è stato creato un ordinamento dei frammenti che segue una struttura a rombo:

$$\begin{tikzcd}[column sep=small] & \textit{None} \arrow[dl, dash] \arrow[dr, dash] & \\ \textit{CB} \arrow[dr, dash] & & \textit{DB} \arrow[dl, dash] \\ & \textit{1B} & \end{tikzcd}$$

Dove *1B* è il minimo e *None* è il massimo. Il risultato è un reticolo e la funzione compare restituisce l'estremo superiore dei due frammenti. La funzione complementary restituisce il frammento della negazione di una formula di un determinato frammento. In particolare il complementare di un *1B* è *1B* mentre il complementare di un *CB* è *DB* e viceversa.

Algorithm 3: Classificatore interno

Firma: innerClassify(φ) **Input:** φ Una formula rettificata
Output: Una coppia (Fragment, Literal)
switch φ **do**
  case *Literal l* **do**
    return $(ONE\_BINDING, l)$;
  end
  case $A[\wedge, \vee]B$ **do**
    return $innerCompare(innerClassify(A), innerClassify(B),$ connective of $\varphi)$;
  end
  case $\neg A$ **do**
    return $innerClassify(A).complementary()$;
  end
  case $A[\Rightarrow, \Leftrightarrow, \oplus]B$ **do**
    return $innerCompare(innerClassify(A), innerClassify(B),$ connective of $\varphi)$;
  end
  else
    return $(None, null)$;
  end
end

La struttura del classificatore interno è molto simile a quella del classificatore esterno, mentre il comparatore interno è leggermente più complesso. Il caso base è quando la formula è un singolo letterale che è sempre un *1B* . La visita in postOrder restituisce una coppia (Fragment, Literal) che rappresenta il frammento a cui appartiene la formula e un letterale di rappresentanza della formula in questo caso il letterale più a sinistra. Il letterale serve a mantenere una reference alla lista di termini delle formule del frammento *1B* .

Algorithm 4: Compare interno

Firma: innerCompare(A, B, con) **Input:** A, B due coppie (Fragment, Literal), *con* un connettivo
Output: Una coppia (Fragment, Literal)

```
switch A.first, B.first, con do
    case One_Binding, One_Binding, _ do
        if A.second has same terms of B.second then
            return A;
        end
        else if conn = ∧ then
            return (Conjunctive_Binding, null);
        end
        else if conn = ∨ then
            return (Disjunctive_Binding, null);
        end
    end
    case [One_Binding, Conjunctive_Binding | Conjunctive_Binding, One_Binding], ∧ do
        return (Conjunctive_Binding, null);
    end
    case [One_Binding, Disjunctive_Binding | Disjunctive_Binding, One_Binding], ∨ do
        return (Disjunctive_Binding, null);
    end
    case Conjunctive_Binding, Conjunctive_Binding, ∧ do
        return (Conjunctive_Binding, null);
    end
    case Disjunctive_Binding, Disjunctive_Binding, ∨ do
        return (Disjunctive_Binding, null);
    end
end
return (None, null);
```

La combinazione booleana di due frammenti *1B* (all'interno di un quantificatore) può portare a tre diversi risultati. Se i termini dei letterali di rappresentanza sono uguali allora la combinazione è ancora un *1B* altrimenti la combinazione è un *CB* se il connettivo è $\wedge$ e un *DB* se il connettivo è $\vee$. Il termine *null* viene usato come sostituto del letterale di rappresentanza in formule del frammento *CB* e *DB* in quanto sono una combinazione di più *1B* e non hanno un letterale di rappresentanza. Due frammenti *CB* rimangono *CB* solo se il loro connettivo è $\wedge$. La combinazione di un *1B* con un *CB* è un *CB* se il connettivo è $\wedge$. Stesso discorso per i *DB* e il connettivo $\vee$. In tutti gli altri casi la combinazione è *None*. Nell'algoritmo 4 sono stati omessi i casi con connettivi $\Rightarrow, \Leftrightarrow, \oplus$ in quanto non sono riconducibili a formule composte da $\wedge$ e $\vee$ come è stato fatto ad esempio nell'algoritmo 1. Con la funzione complementary applicata ad una coppia: (Fragment, Literal).complementary() si intende la coppia (Fragment.complementary(), Literal).

1.2 Preprocessing

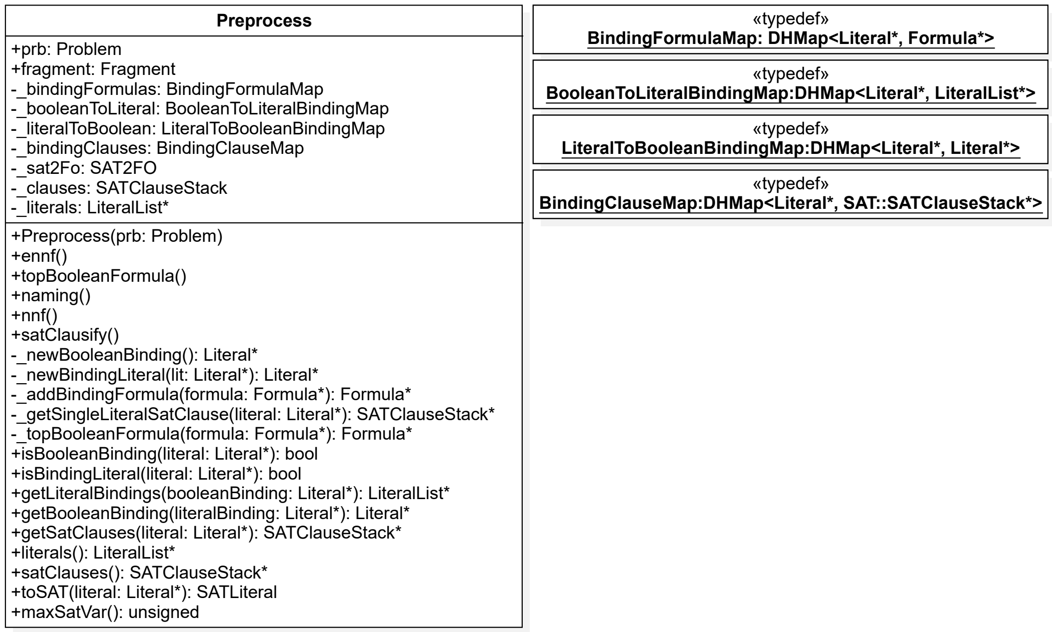

Figura 1.2: Struttura del Preprocessing

In questa sezione verrà descritto l'algoritmo di preprocessing utilizzato per trasformare una formula in input del frammento *1B* o *CB* in una struttura trattabile dall'algoritmo di decisione. Per utilizzare il SatSolver di Vampire per la ricerca degli implicanti è necessario clausificare la formula. Inoltre per evitare un esplosione esponenziale di formule causate dalle forme NNF e CNF è necessario utilizzare tecniche di naming. Qui sorgono i primi problemi visto che ne la clausificazione ne il naming sono processi conservativi rispetto ai frammenti. Ad esempio la semplice formula del frammento *1B* $\forall x_1(p_1(x_1)) \vee p_2$ clausificata diventa $\{\{p_1(x_1), p_2\}\}$ che fa parte del frammento *DB* . L'approccio utilizzato è stato quello di creare una nuova formula ground che rappresenta la struttura booleana esterna della formula originale, applicare le funzioni standard di preprocessing e mantenere una serie di strutture per risalire ai componenti originali. Per questo scopo viene introdotto un nuovo insieme di simboli di predicato $\Sigma_b = \{b_1, b_2, ...\}$. I predicati di Σ_b con arità 0 saranno chiamati *booleanBinding* e saranno associati ad una formula del frammento *1B* o *CB* . I predicati di Σ_b con arità $n > 0$ saranno chiamati *literalBindign* e fungeranno da rappresentati dei τ-Binding delle formule *1B* . Il preprocessing seguirà pressochè questa struttura:

1. Rettificazione
2. Trasformazione in ENNF
3. Creazione della formula booleana esterna (FBE) e associazione dei booleanBinding
4. Naming della FBE
5. Trasformazione in NNF della FBE
6. Creazione dei literalBinding e Sat-Clausificazione delle formule mappate dai booleanBinding
7. Creazione delle Sat-Clausole della FBE

La rettificazione e la trasformazione in ENNF sono processi conservativi rispetto ai frammenti e quindi verranno applicate direttamente le funzioni standard di Vampire. La creazione della FBE e l'associazione dei booleanBinding avviene tramite l'algoritmo 5.

Algorithm 5: Top Boolean Formula

Firma: topBooleanFormula(φ)
Input: φ una formula rettificata
Output: Una formula ground
GlobalData: bindingFormulas una mappa da booleanBinding a formula **switch** φ **do**

```
    case Literal l do
        return new AtomicFormula(l);
    end
    case A[∧, ∨]B do
        return new JunctionFormula(topBooleanFormula(A), connective of φ,
         topBooleanFormula(B));
    end
    case ¬A do
        return new NegatedFormula(topBooleanFormula(A));
    end
    case [∀, ∃]A do
        b = newBooleanBinding();
        bindingFormulas[b] := φ;
        return new AtomicFormula(b);
    end
    case A[⇔, ⇒, ⊕]B do
        return new BinaryFormula(A, connective of φ, B);
    end
end
```

L'algoritmo prende in input una formula rettificata e restituisce una formula ground sostituendo le sottoformule quantificate con un nuovo booleanBinding aggiungendo la sottoformula originale alla mappa bindingFormulas. Da adesso in poi qualunque modifica fatta alla FBE preserverà l'appartenenza al frammento originale. Gli step successivi sono quindi applicare le funzioni standard di Vampire per il naming e la trasformazione in NNF. La trasformazione in NNF potrebbe portare alla negazione di qualche booleanBinding e va quindi aggiunta alla mappa bindingFormulas la formula negata associata.

```
foreach l ∈ literals(φ) do
    if ¬l.polarity() then
        continue

    end

    positiveFormula := bindingFormulas[positiveLiteral(l)]
    bindingFormulas[l] := newNegatedFormula(positiveFormula)

end
```

A questo punto inizia il processo di SatClausificazione delle formule interne (quelle associate ai booleanBinding). Ogni letterale ground che non è un booleanBinding viene trasformato in una SatClausola di lunghezza 1 composta dal solo satLetterale associato al letterale.

foreach $l \in literals(\varphi)$ **do**
    if *l is not a booleanBinding* **then**
        $bindingClauses[l] := newSatClause\{toSat(l)\}$
    end
end

Per essere clausificate le formule della mappa bindingFormulas vanno trasformate in NNF, Skolemizzate. Anche in questo caso vengono utilizzate le funzioni standard di Vampire. Ogni booleanBinding è associato ad una formula del frammento ConjunctiveBinding, per questo dopo la skolemizzazione il quantificatore universale viene distribuito sull'and per ottenere le sottoformule del frammento OneBinding. Per ogni sottoformula OneBinding viene creato un nuovo LiteralBinding in rappresentaza della sottoformula. Il nuovo letterale avrà gli stessi termini del letterale più a sinistra della sottoformula (che sono gli stessi di tutti i letterali della sottoformula). Successivamente la formula viene SatClausificata. Si aggiunge alla mappa satClauses la coppia composta dal nuovo LiteralBinding e le satClausole della sottoformula. Alla mappa literalToBooleanBindings viene aggiunta la coppia composta dal nuovo LiteralBinding e il booleanBinding associato mentre alla mappa booleanBindingToLiteral viene aggiunta la coppia composta dal booleanBinding e la lista dei LiteralBinding che rappresentano le sottoformule della formula originale.

while $bindingFormulas \neq \emptyset$ **do**
    $(booleanBinding, formula) := bindingFormulas.pop()$
    $formula := nnf(formula)$
    $formula := skolemize(formula)$
    $toDo := \emptyset$

    if *formula is ConjunctiveBinding* **then**
        $formula := distributeForAll(formula)$
        "Add each subformula to the todo list"
    end
    else
        toDo.add(formula)
    end
    $literalBindings := \emptyset$ **while** $todo \neq \emptyset$ **do**
        $subformula := todo.pop()$
        $literalBinding := newLiteralBinding(subformula.mostLeftLiteral())$
        $clauses := SatClausifyBindingFormula(subFormula)$

        $satClauses[literalBinding] := clauses$
        $literalToBooleanBindings[literalBinding] := booleanBinding$
        $literalBindings.add(literalBinding)$
    end
    $booleanBindingToLiteral[booleanBinding] := literalBindings$
end

La funzione SatClausifyBindingFormula è una funzione che prende in input una formula la clausifica e converte tutte le clausole in SatClausole in modo che ogni satLetterale ha lo stesso indice del funtore del predicato associato. Questo è differente da quello che viene fatto dalla classe Sat2Fo che associa ogni puntatore a letterale ad un nuovo SatLetterale con un nuovo indice arbitrario. L'ultimo step è la SatClausificazione della FBE che avviene tramite le funzioni standard di Vampire della classe Sat2Fo. È importante ricordare che i satLetterali delle formule interne sono diversi dai satLetterali della FBE nonostante possano avere lo stesso indice.

Si prenda ad esempio la formula del frammento CB :

$$(\forall x_1, x_2((p_1(x_1) \vee p_2(x_1)) \wedge p_2(f_1(x_2))) \wedge \forall x_1(p_3(x_1) \Rightarrow p1(x_1))) \vee (\forall x_1(p_2(x_1)) \Rightarrow p_4)$$

Il primo passo di preprocessing prevede la rettificazione e la trasformazione in ENNF. La formula è già rettificata mentre la trasformazione in ENNF porta all'eliminazione del $\Rightarrow$:

$$(\forall x_1, x_2((p_1(x_1) \vee p_2(x_1)) \wedge p_2(f_1(x_2))) \wedge \forall x_1(\neg p_3(x_1) \vee p1(x_1))) \vee (\forall x_1(p_2(x_1)) \Leftrightarrow p_4)$$

La creazione della FBE porta alla generazione di un booleanBinding per ogni sottoformula quantificata:

$$(b_1 \wedge b_2) \vee (b_3 \Leftrightarrow p_4)$$

La mappa bindingFormulas contiene le seguenti coppie:

$b_1 \ \rightarrow \forall x_1, x_2((p_1(x_1) \vee p_2(x_1)) \wedge p_2(f_1(x_2)))$

$b_2 \ \rightarrow \forall x_1(\neg p_3(x_1) \vee p1(x_1))$

$b_3 \ \rightarrow \forall x_1(p_2(x_1))$

La formula ottenuta è troppo piccola per poter applicare il namig quindi si procede direttamente con la trasformazione in NNF:

$$(b_1 \wedge b_2) \vee ((\neg b_3 \vee p_4) \wedge (b_3 \vee \neg p_4))$$

Durante il processo di NNF il booleanBinding b_3 è stato negato e quindi va aggiunto alla mappa bindingFormulas:

$\neg b_3 \ \rightarrow \exists x_1(\neg p_2(x_1))$

A questo punto vengono trasformate in NNF e Skolemizzate le formule associate ai booleanBinding, vengono poi creati i literalBindings e le SatClausole delle formule interne. Il booleanBinding b_1 è associato ad una formula CB quindi viene distribuito il quantificatore universale sull'and e creati due literalBindings. La skolemizzazione della formula associata a $\neg b_3$ porta alla formula::

$\neg b_3 \ \rightarrow \neg p_2(sk_1)$

Vengono create così le mappe booleanBindingToLiteral e la sua inversa literalToBooleanBindings:

booleanBindingToLiteral	**literalToBooleanBindings**
$b_1 \rightarrow \{b_4(x_1), b_5(f_1(x_1))\}$	$b_4(x_1) \rightarrow b_1$
$b_2 \rightarrow \{b_6(x_1)\}$	$b_5(f_1(x_1)) \rightarrow b_1$
$b_3 \rightarrow \{b_7(x_1)\}$	$b_6(x_1) \rightarrow b_2$
$\neg b_3 \rightarrow \{b_8(sk_1)\}$	$b_7(x_1) \rightarrow b_3$
	$b_8(sk_1) \rightarrow \neg b_3$

Le formule associate ai literalBindings vengono clausificate:

- $\forall x_1, x_2((p_1(x_1) \vee p_2(x_1))) \rightarrow \{\{(p_1(x_1), p_2(x_1))\}\}$
- $\forall x_1, x_2(p_2(f_1(x_2))) \rightarrow \{\{p_2(f_1(x_2))\}\}$
- $\forall x_1(\neg p_3(x_1) \vee p1(x_1)) \rightarrow \{\{\neg p_3(x_1), p1(x_1)\}\}$
- $\forall x_1(p_2(x_1)) \rightarrow \{\{p_2(x_1)\}\}$
- $\neg p_2(sk_1) \rightarrow \{\{\neg p_2(sk_1)\}\}$

E successivamente SatClausificate e associate ai literalBindings:

- $b_4(x_1) \rightarrow \{\{s_1, s_2\}\}$
- $b_5(f_1(x_1)) \rightarrow \{\{s_2\}\}$
- $b_6(x_1) \rightarrow \{\{\neg s_3, s_1\}\}$
- $b_7(x_1) \rightarrow \{\{s_2\}\}$
- $b_8(sk_1) \rightarrow \{\{\neg s_2\}\}$

Gli ultimi due step sono la clausificazione della FBE:

$$\{\{b_1, \neg b_3, p_4\}, \{b_2, \neg b_3, p_4\}, \{b_1, b_3, \neg p_4\}, \{b_2, b_3, \neg p_4\}\}$$

E la SatClausificazione tramite sat2Fo:

$$\{\{s_1, \neg s_2, s_3\}, \{s_4, \neg s_2, s_3\}, \{s_1, s_2, \neg s_3\}, \{s_4, s_2, \neg s_3\}\}$$

Che crea internamente una bi-mappa che associa ogni satletterale ad un letterale:

- $s_1 \leftrightarrow b_1$
- $s_2 \leftrightarrow \neg b_3$
- $s_3 \leftrightarrow p_4$
- $s_4 \leftrightarrow b_2$

1.3 Procedura di Decisione

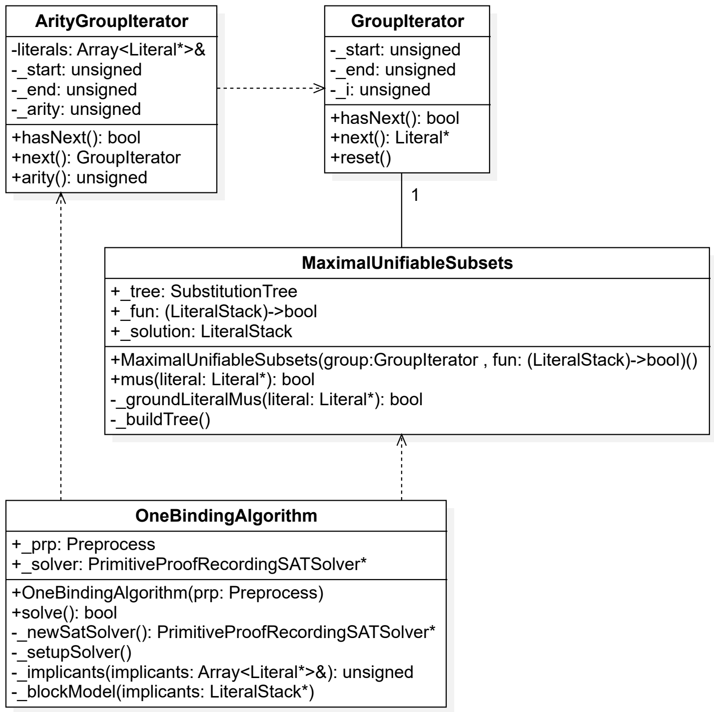

Figura 1.3: Struttura dell'algoritmo di decisione

In questa sezione verrà descritta l'implementazione dell'algoritmo di decisione **??** per frammenti Binding descritto nel capitolo **??**. L'algoritmo è composto da tre parti principali: la ricerca degli implicanti, la ricerca di tutti i sottoinsiemi unificabili e la ricerca di un assegnamento proposizionale che soddisfi la congiunzione della conversione booleana dei sottoinsiemi unificabili. Da questo momento si assuma di avere una formula preprocessata con tutte le strutture ausiliarie come descritto nella sezione precedente.

La ricerca degli implicanti è la parte più facile da implementare (sat esterna). Data la FBE SatClausificata è sufficiente utilizzare il satSolver integrato in Vampire ed estrapolarne una assegnazione. Dopo aver ottenuto l'insieme degli implicanti proposizionali se la sua relativa formula del primo ordine è insoddisfacibile allora è sufficiente creare una clausola bloccante e cercare un nuovo assegnamento. Se non sono disponibili nuovi assegnamenti allora la formula originale è insoddisfacibile.

La ricerca di tutti i sottoinsiemi unificabili è senza dubbio la parte più complessa dell'algoritmo. L'approccio utilizzato nell'algoritmo **??** è troppo astratto e non utilizzabile nella pratica. Anche il solo problema di iterare su tutti i sottoinsiemi di un insieme è un problema non trivale. Vanno quindi necessariamente fatti dei tagli nello spazio di ricerca. La prima osservazione che si può fare è che se un insieme di letterali è unificabile allora i letterali hanno tutti la stessa arità. È quindi possibile ordinare l'insieme di implicanti in base all'arità e ricercare per ogni 'Gruppo di Arità' tutti i sottoinsiemi unificabili. Già in questo modo si riduce notevolmente lo spazio di ricerca eliminando tutti quei sottoinsiemi composti da letterali di arità diversa. La seconda osservazione è che dati due sottoinsiemi $U' \subseteq U$ se la congiunzione della conversione booleana dei letterali di U è soddisfacibile allora lo sarà anche quella di U'. Questo riduce ulteriormente lo spazio di ricerca ai soli sottoinsiemi massimali unificabili. Sfortunatamente la ricerca di tutti i sottoinsiemi massimali unificabili (Maximal Unifiable Subsets / MUS) è un problema NP-Completo così come il suo problema complementare cioè il problema di ricercare tutti i sottoinsiemi minimali non unificabili (minimal non unifiable subsets / mnus). Per questo motivo è stato creato un algoritmo euristico meno restrittivo che itera almeno su tutti i sottoinsiemi massimali non escludendo però la possibilità di trovare anche qualche sottoinsieme non massimale.

Dopo aver ottenuto un insieme di τ-Binding unificabili l'algoritmo procede con la ricerca di un assegnamento proposizionale che soddisfi la congiunzione della conversione booleana dei sottoinsiemi unificabili (sat interna). Anche in questo caso il problema è molto semplice. Ogni τ-Binding è rappresentato da un bindingLiteral creato nella fase di preprocessing e ogni bindingLiteral è associato ad un insieme di satClausole che rappresentano la conversione booleana citata sopra. Grazie a questa indicizzazione è possibile utilizzare il satSolver integrato per verificare la soddisfacibilità.

Maximal Unifiable Subsets

Algorithm 6: Maximal Unifiable Subsets

Firma: mus(*literal*)
Input: *literal* un puntatore ad un letterale
Output: $\top$ o $\bot$
GlobalData: S una mappa da letterali a bool
if $S[literal]$ **then**
  return $\top$;
end
if *literal is ground* **then**
  return $groundLiteralMus(literal)$;
end
$S[literal] = \top$;
$res := mus(literal, \emptyset)$;
$S[literal] = \bot$;
return res;

Algorithm 7: Maximal Unifiable Subsets

Firma: mus($literal$, $FtoFree$)
Input: $literal$ un puntatore ad un letterale, $FtoFree$ un puntatore ad una lista di letterali
Output: $\top$ o $\bot$
GlobalData: **S** una mappa da letterali a interi, **fun** una funzione da lista di letterali a bool, **tree** un SubstitutionTree

```
isMax := ⊤;
uIt = tree.getUnifications(query : literal, retrieveSubstitutions : true);
toFree := ∅;
while uIt.hasNext() do
    (u, σ) := uIt.next();
    if S[u] = 0 then
        S[u] = 1;
        l := literal^σ;
        if l = literal then
            u' := u^σ;
            if u' = u then
                FtoFree := FtoFree ∪ {u};
            end
            else
                toFree := toFree ∪ {u};
            end
        end
        else
            isMax = ⊥;
            if ¬mus(l, toFree) then
                return ⊥;
            end
            S[u] = −1;
            toFree := toFree ∪ {u};
        end
    end
end
if isMax then
    if ¬fun({x | S[x] = 1}) then
        return ⊥;
    end
end
while toFree ≠ ∅ do
    S[toFree.pop()] = 0;
end
return ⊤;
```

Algorithm 8: Maximal Unifiable Subsets Ground

Firma: groundMus($literal$)
Input: $literal$ un puntatore ad un letterale ground
Output: $\top$ o $\perp$
GlobalData: **S** una mappa da letterali a interi, **fun** una funzione da lista di letterali a bool, **tree** un SubstitutionTree

```
if S[literal] ≠ 0 then
    return ⊤;
end
uIt = tree.getUnifications(query : literal, retrieveSubstitutions : true);
solution := ∅;
while uIt.hasNext() do
    (u, σ) := uIt.next();
    if S[u] = 0 then
        if u is ground then
            S[u] = −1;
        end
        solution := solution ∪ {u};
    end
end
return fun(solution);
```

Procedura di decisione

Algorithm 9: Algoritmo di decisione

```
Firma: solve(prp)
Input: prp il problema pre-processato
Output: ⊤ o ⊥
satSolver := newSatSolver();
satSolver.addClauses(prp.clauses);
while satSolver.solve() = SATISFIABLE do
    res := ⊤;
    implicants := getImplicants(satSolver, prp);
    implicants := sortImplicants(implicants);
    if implicants contains only ground Literals then
        return ⊤;
    end
    agIt := ArityGroupIterator(implicants);
    while res And agIt.hasNext() do
        maximalUnifiableSubsets := SetupMus(group, internalSat);
        foreach lit ∈ group do
            if ¬maximalUnifiableSubsets.mus(lit) then
                res := ⊥;
                blockModel(maximalUnifiableSubsets.getSolution());
                Break;
            end
        end
        if res = ⊤ then
            return ⊤;
        end
    end
end
return ⊥;
```

Dato il problema preprocessato l'algoritmo comincia impostando il satSolver con le satClausole ottenute nella fase di preprocessing. Se la formula è soddisfacibile allora si recupera l'insieme degli implicanti tramite la funzione getImplicants 10.

Algorithm 10: getImplicants

Firma: getImplicants(solver, prp)
Input: *solver* un sat solver, *prp* il problema pre-processato
Output: Una lista letterali

```
implicants := ∅;
foreach l ∈ prp.literals() do
    satL := prp.toSat(l);
    if solver.trueInAssignment(satL) then
        if prp.isBooleanBinding(l) then
            implicants := implicants ∪ prp.getLiteralBindings(l);
        end
        else
            implicants := implicants ∪ {l};
        end
    end
end
return implicants;
```

Per ogni letterale del problema viene recuperato il corrispondente satLetterale. Se il satLetterale è soddisfatto dall'assegnamento allora se il letterale originale non è un booleanBinding viene aggiunto all'insieme di implicanti, altrimenti vengono aggiunti tutti i literalBinding associati al booleanBinding.

Dopo aver ottenuto l'insieme di implicanti, l'insieme viene ordinato in base all'arità dei letterali. La funzione sortImplicants può essere estesa aggiungendo varie euristiche. La prima euristica che è stata implementata è quella di spingere i letterali ground all'inizio di ogni gruppo di arità in modo da utilizzare l'algoritmo 8 per ridurre il numero di chiamate ricorsive che verrebbero fatte dall'algoritmo 7. La seconda euristica è quella di ordinare i letterali in base ai sottotermini in modo da avere vicino sequenze di letterali che hanno stessi termini per sfruttare l'ottimizzazione vista nell'algoritmo 7.

Se l'insieme di implicanti contiene solo letterali ground (che non sono literalBindings) allora la formula è soddisfacibile perchè l'assegnamento per la formula esterna è valido anche per le formule interne e l'algoritmo termina restituendo $\top$.

Altrimenti per ogni gruppo di arità si imposta l'ambiente per la ricerca dei mus e viene chiamata la funzione mus 6 per ogni letterale del gruppo. Se una di queste chiamate restituisce $\perp$ allora la formula FO corrispondente all'assegnamento booleano trovato è insoddisfacibile e si procede con la ricerca di un nuovo assegnamento. Se tutte le chiamate restituiscono $\top$ allora la formula è soddisfacibile e l'algoritmo termina restituendo $\top$. Se non sono disponibili nuovi assegnamenti allora la formula è insoddisfacibile e l'algoritmo termina restituendo $\perp$.

Algorithm 11: Sat interna

Firma: internalSat(literals)
Input: *literals* una lista di letterali
Output: $\top$ o $\perp$

```
if literals.length = 1 And getSatClauses(literals.top()).length = 1 then
    return ⊤;
end
satSolver := newSatSolver();
foreach l ∈ literals do
    satSolver.addClause(getSatClauses(l));
end
return satSolver.solve() = SATISFIABLE;
```

La funzione internalSat viene chiamata ogni volta che la funzione 7 trova un nuovo mus. Se il mus è composto da un solo letterale e la lista di satClausole associata è composta da una sola clausola allora la formula non può entrare in contraddizione, di conseguenza è soddisfacibile e la funzione restituisce $\top$, evitando di impostare il satSolver. In caso contrario viene impostato il satSolver con le clausole associate ai literalBindings dalla mappa satClauses e viene chiamato il metodo solve. La funzione restituisce $\top$ se il satSolver restituisce SATISFIABLE altrimenti $\bot$.