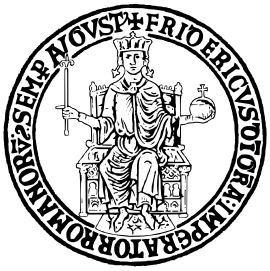
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Implementazione di una procedura di decisione per il frammento guarded in Vampire

Relatori
Prof. Fabio Mogavero
Prof. Massimo Benerecetti

Candidato
Francesco Scarfato
N86/3769

Anno Accademico 2022–2023

Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Implementazione di una procedura di decisione per il frammento guarded in Vampire

Relatori
Prof. Fabio Mogavero
Prof. Massimo Benerecetti

Candidato
Francesco Scarfato
N86/3769

Anno Accademico 2022–2023

Indice

Introduzione

Un *theorem prover* è un importante strumento che permette di verificare formalmente sistemi software e hardware, strettamente collegato ai campi della logica e della matematica. I *theorem prover* sono degli strumenti *general purpose* che puntano a risolvere il maggior numero di problemi nel modo più veloce ed efficiente possibile. Per migliorare questi sistemi, la ricerca si sta muovendo verso la definizione di procedure di decisione per problemi decidibili. Una procedura di decisione è un algoritmo che verifica la soddisfacibilità di un problema e arriva sempre a una terminazione. Nella logica del primo ordine, un sottoinsieme di problemi decidibili è detto frammento. La ricerca di queste procedure di decisione è rilevante poiché, in alcuni casi, questi approcci possono risultare più efficienti di una strategia general purpose. L'ideale sarebbe inglobare queste procedure di decisione nei theorem prover, quando è possibile, e utilizzarle nel caso in cui un problema appartenga a un determinato frammento.

In questa tesi viene presentata un'implementazione di una procedura di decisione per il frammento *guarded* su Vampire, uno degli *automated theorem prover*, per la risoluzione di problemi di logica del primo ordine, più conosciuti ed efficienti in circolazione. Nella prima parte vengono introdotte le caratteristiche base più importanti di Vampire e del frammento *guarded*. Nella seconda parte viene descritta l'implementazione sperimentale della procedura di decisione per questo particolare frammento. Infine, viene presentata un'analisi basata sul confronto tra il software originario e la versione estesa con la procedura di decisione.

1. Vampire theorem prover

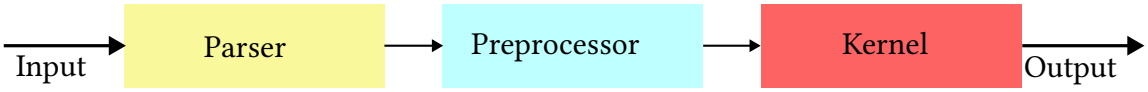
In questo capitolo viene presentato il theorem prover Vampire, la sua architettura e alcune delle caratteristiche più importanti.
Vampire è un sistema che permette di provare la validità di teoremi (matematici e non) di logica del primo ordine. Il linguaggio di programmazione usato è C++ e la prima versione è stata sviluppata principalmente da Andrei Voronkov e Krystof Hoder tra il 1993 e il 1995 all'Università di Manchester. Dopo il 1995, il progetto è stato sospeso e verrà ripreso nel 1998. Dal '98 vengono implementate numerose versioni e, ancora oggi, proseguono gli sviluppi. Vampire ha vinto circa 45 titoli in diverse divisioni della CASC (CADE ATP System Competition) che è il campionato mondiale per gli ATP (Automated Theorem Prover) [1, 2]. Alcune delle principali caratteristiche del sistema sono:

- la velocità
- la portabilità sulle piattaforme più comuni
- la semplicità dell'utilizzo
- è dotato di strategie di ricerca con risorse limitate
- supporta numerose sintassi in input
- la possibilità di parallelizzare, i vari tentativi di prova del problema, su più processori
- può produrre, in base alle opzioni selezionate, output molto dettagliati

1.1 Architettura di Vampire

Vampire ha un'architettura complessa ma, analizzandola ad alto livello, è possibile riconoscere tre moduli principali:

1. *parser*
2. *preprocessor*
3. *kernel*



1.1.1 Parser

Il *parser* è un modulo che permette la lettura di un problema da un file e la conseguente incapsulazione in una classe specifica. Nel caso di un problema di logica del primo ordine con sintassi TPTP, viene diviso in più unità. Ogni unità è una formula o una clausola a cui viene assegnato uno specifico tag in base al tipo (ipotesi, assioma, congettura, teorema, ...).

Nel codice sorgente sono presenti le classi `Problem`, `Unit`, `Formula`, `Clause` e sono in relazione tra loro come mostrato nella figura 1.1. L'attributo *kind* della classe `Unit` permette la distinzione tra `Clause` e `Formula`. Sono presenti altre classi, non rappresentate nella figura, che ereditano `Formula` come: `AtomicFormula`, `JunctionFormula`, `BinaryFormula`, ...

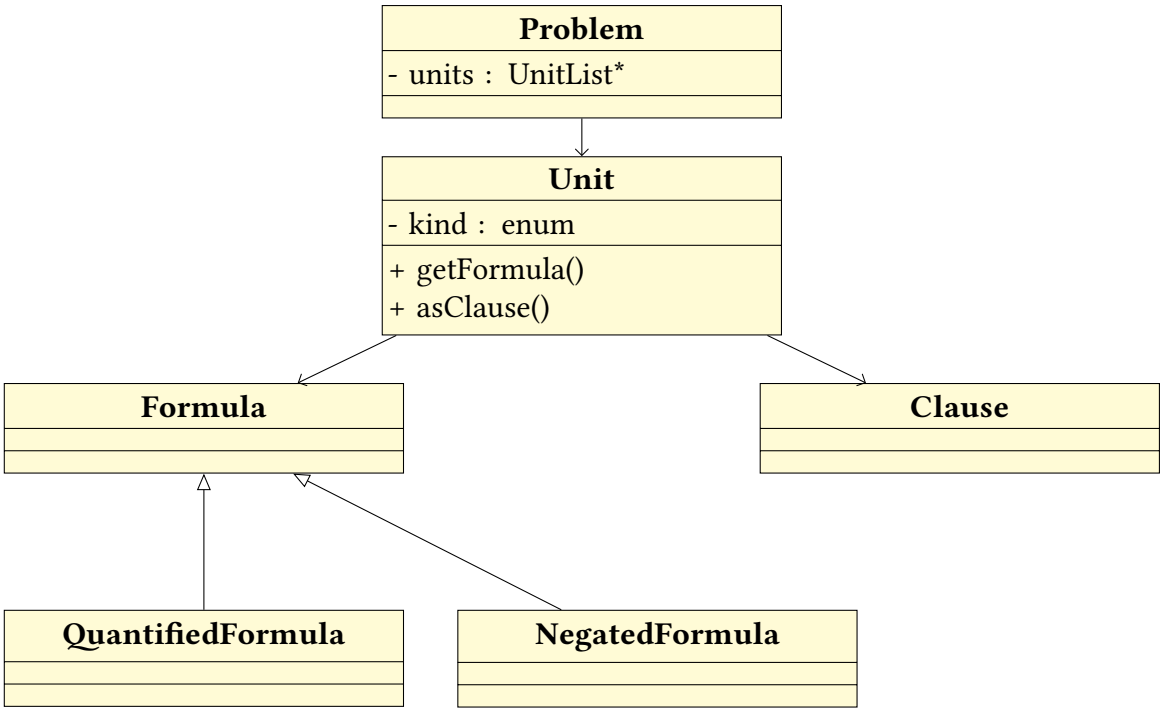


Figura 1.1: Relazioni tra le classi per definire un problema

1.1.2 Preprocessor

Il preprocessor è un modulo che processa il problema in modo che sia trattabile dal *kernel* in fase di risoluzione. Per questo modulo è possibile abilitare numerose opzioni, inoltre

vengono eseguite semplificazioni in modo da rendere il sistema il più veloce possibile. Di seguito sono descritti solo i passaggi fondamentali del *preprocessing* [3]:

I step *Rectify*: Il *preprocessor* verifica se una formula ha variabili libere. Se la formula è aperta, allora genera dei quantificatori per vincolare le variabili libere. Inoltre, verifica che per ogni variabile x ci sia una sola occorrenza di $\exists x$ o $\forall x$.

II step *Simplify*: Il *preprocessor* verifica se una formula contiene $\top$ o $\bot$. Nel caso la formula contenesse uno dei due, allora viene semplificata.

III step *Flatten*: Il *preprocessor* trasforma le formule in modo da renderle uniformi. Questo è solo uno step di formattazione che verrà eseguito più volte nel corso del preprocessing.

IV step *Unused definitions and pure predicate removal*: Il *preprocessor* rimuove i predicati e le definizioni delle funzioni che non vengono usati.

V step *ENNF*: Il *preprocessor* trasforma le formula in *extended negative normal form*.

Definizione 1. Una formula è in *extended negation normal form* se non contiene $\rightarrow$, e tutte le $\neg$ sono spostate, il più possibile, verso l'interno.

VI step *Naming*: Il *preprocessor* definisce un nuovo predicato p che viene usato come nome di una sotto-formula. Questa tecnica viene utilizzata per evitare di generare un numero esponenziale di clausole nei prossimi passaggi di preprocessing.

Definizione 2. Sia $\varphi|_\pi$ sotto-formula di φ in posizione π con variabili libere $\bar{x}$, allora si sostituisce $\varphi|_\pi$ con $p(\bar{x})$ nuovo predicato e viene aggiunta la definizione $\mathrm{def}(\varphi, \pi, p)$ tale che

$$\mathrm{def}(\varphi, \pi, p) = \begin{cases} \forall \bar{x}(p(\bar{x}) \rightarrow \varphi|_\pi) & \text{se } \mathrm{pol}(\varphi, \pi) = 1 \\ \forall \bar{x}(\varphi|_\pi \rightarrow p(\bar{x})) & \text{se } \mathrm{pol}(\varphi, \pi) = -1 \\ \forall \bar{x}(p(\bar{x}) \leftrightarrow \varphi|_\pi) & \text{se } \mathrm{pol}(\varphi, \pi) = 0 \end{cases}$$

in cui $\mathrm{pol}(\varphi, \pi)$ indica la polarità della sotto-formula di φ in posizione π. $\mathrm{pol}(\varphi, \pi) = 0$ si verifica quando il simbolo di livello superiore di $\varphi|_\pi$ è $\leftrightarrow$ o $\otimes$.

Questa tecnica viene adottata solo se il numero di clausole generate supera una certa soglia. Infatti, in Vampire, nella classe `Naming`, è presente un attributo *threshold* che indica proprio questo limite.

VII step *NNF* $\longrightarrow$ Il *preprocessor* trasforma le formule in *negation normal form*.

Definizione 3. Una formula è in *negation normal form* se non contiene $\rightarrow, \leftrightarrow, \otimes$, e tutte le $\neg$ sono spostate, il più possibile, verso l'interno.

VIII step *Skolemization* ⟶ Il *preprocessor* applica questa tecnica per eliminare i ∃ dalle formule.

Definizione 4. Sia $F = \exists x \mid \varphi(y_1, \ldots, y_n, x)$ formula in negation normal form allora la *skolemization* si ottiene tramite la seguente sostituzione:

$$F = \exists x \mid \varphi(y_1, \ldots, y_n, x) \Rightarrow F' = \varphi(y_1, \ldots, y_n, x)\{x \mapsto sk(y_1, \ldots, y_n)\}$$

in cui $\{x \mapsto sk(y_1, \ldots, y_n)\}$ indica che x, la variabile precedentemente vincolata dal quantificatore esistenziale nella formula F, è sostituita da una nuova funzione $sk(y_1, \ldots, y_n)$, detta di Skolem, nella formula F'.

IX step *Clausification* ⟶ Il *preprocessor* trasforma tutte le formule in modo da ottenere un insieme di clausole.

Definizione 5. La *clausification* è il risultato di:

1. $\forall x \mid \varphi(y_1, \ldots, y_n, x) \Rightarrow \varphi(y_1, \ldots, y_n, x)\{x \mapsto X\}$ in cui X è una variabile designata che non occorre in φ
2. Se una formula $F = \varphi' \wedge \varphi''$ allora viene spezzata in due unità diverse $F' = \varphi'$ e $F'' = \varphi''$

Alla fine di questi step, l'insieme di clausole risultanti è pronto per essere passato all'algoritmo di *resolution*.

1.1.3 Kernel

Il *kernel* è il sotto-sistema adibito alla risoluzione del problema. Per raggiungere questo obiettivo, viene implementato un algoritmo di saturazione che permette di trovare una confutazione all'insieme di clausole. É possibile trovare la confutazione all'insieme tramite la sua saturazione con tutte le inferenze presenti nel calcolo. Vampire possiede numerose inferenze ma è possibile sceglierne un sottoinsieme e definire un sistema d'inferenze per la logica del primo ordine.

Formalmente, per definire un sistema d'inferenze bisogna prima definire un *simplification ordering* e una funzione di selezione [1, 2].

Definizione 6. Un ordinamento $\succ$ sui termini è detto *simplification ordering* se rispetta le seguenti condizioni:

1. $\succ$ è ben formato ovvero $\nexists t_0, t_1, \ldots$ sequenza infinita tale che $t_0 \succ t_1 \succ \ldots$
2. $\succ$ è monotona ovvero $l \succ r \rightarrow s(l) \succ s(r)$ per tutti i termini s, l, r
3. $\succ$ è stabile per la sostituzione ovvero $l \succ r \rightarrow l\theta \succ r\theta$
4. $\succ$ ha la *subterm property* ovvero se r sottotermine di l e $l \neq r$ allora $l \succ r$

Questa definizione è estendibile agli atomi, ai letterali e alle clausole.

Definizione 7. Una funzione di selezione sceglie un sottoinsieme non vuoto di letterali in ogni clausola non vuota. In $\underline{L} \vee R$, $\underline{\mathrm{L}}$ indica il letterale selezionato.

Il sistema di inferenze che viene definito di seguito è detto *superposition inference system.*

Definizione 8. Il *superposition inference system* è un sistema di inferenze composto dalle seguenti regole:

- **Resolution**
$$\frac{\underline{A} \vee C_1 \quad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\theta} \tag{1.1}$$
in cui θ è l'unificatore più generale di A e A'.

- **Factoring**
$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta} \tag{1.2}$$
in cui θ è l'unificatore più generale di A e A'.

- **Superposition**
$$\frac{\underline{l = r} \vee C_1 \quad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] = t'} \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] \neq t'} \vee C_2}{(t[r] \neq t' \vee C_1 \vee C_2)\theta} \tag{1.3}$$
in cui θ è l'unificatore più generale di l e s, s non è una variabile, $r\theta \prec l\theta$, solo nella prima regola $L[s]$ non è un *equality literal*[1],

- **Equality resolution**
$$\frac{\underline{s \neq t} \vee C}{C\theta} \tag{1.4}$$
in cui θ è l'unificatore più generale di s e t.

- **Equality factoring**
$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\theta} \tag{1.5}$$
in cui θ è l'unificatore più generale di s e t, $t\theta \prec s\theta$ e $t'\theta \prec t\theta$.

Se la funzione di selezione, seleziona sia letterali negativi sia tutti i letterali massimali, allora è possibile enunciare la seguente proprietà:

[1]Un *equality literal* è una clausola con un singolo letterale in cui è presente un =

Proprietà 1. *Il superposition inference system è sound e completo.*

Una volta definito un sistema di inferenze, è possibile formalizzare il concetto di saturazione espresso precedentemente.

Definizione 9. Un insieme di clausole S è detto saturato rispetto al sistema di inferenze ϕ se, per ogni inferenza in ϕ con le premesse in S, la conclusione dell'inferenza appartiene sempre a S.

Grazie alla proprietà 1, si ottiene quest'altra importante proprietà:

Proprietà 2. *Un insieme di clausole è* ***insoddisfacibile*** *se, e solo se, il più piccolo insieme di clausole contenente S, saturato rispetto al superposition inference system, contiene anche la clausola vuota* $\square$.

Inoltre, un algoritmo di saturazione deve essere *fair*, cioè ogni possibile inferenza deve essere selezionata in un certo momento dell'algoritmo.

Per rendere efficiente l'algoritmo di saturazione, oltre queste regole, vengono introdotte regole di semplificazione per eliminare ridondanze e semplificare le inferenze. Alcune di queste regole sono: *demodulation*, *branch demodulation* e *subsumption resolution*.

In generale, un algoritmo di saturazione è composto dalle seguenti fasi:

1. Viene inizializzato un insieme S in cui vengono memorizzate le clausole
2. Viene selezionata un'inferenza che può essere applicata a delle clausole presenti in S
3. Nel caso fosse generato un risultato, allora viene aggiunto a S
4. Se viene trovata una clausola vuota $\square$, allora il problema è insoddisfacibile

Se un'inferenza genera una clausola, allora è detta generatrice. Le inferenze generatrici sono strettamente collegate alle regole di semplificazioni.

Vampire implementa tre tipi di algoritmi di saturazione: *limited resource strategy*(lrs), *otter* e *discount*. Tutti fanno parte della famiglia dei *given clause algorithm*. L'algoritmo 1 è semplificato ma rappresenta, ad alto livello, come lavorano quelli appartenenti a questa famiglia.

Algoritmo 1 Given clause algorithm

```
var active,passive : sets of clause
var current,new : clause
active = ∅
passive = set of input clauses
while passive ≠ ∅ do
    current = select(passive)
    passive = passive \ {current}
    active = active ∪ {current}
    new = infer(current,active)
    if new is □ then
        return provable
    end if
    passive = passive ∪ {new}
end while
return unprovable
```

Vengono definiti due insiemi di clausole: *passive* e *active*. L'insieme *passive* contiene quelle clausole che aspettano di passare all'insieme *active* e possono solo essere semplificati. L'insieme *active* contiene quelle clausole che sono attive e sono pronte per la generazione delle inferenze. Per passare da passiva ad attiva, una clausola deve essere scelta dalla funzione *select()*, che permette all'algoritmo di acquisire *fairness*. Questa selezione è effettuata dal *kernel* seguendo due parametri: età e peso della clausola. Per implementarla, vengono definite due code di priorità per i rispettivi parametri: nella prima hanno priorità maggiore le clausole più "vecchie", mentre, nella seconda, le clausole più "leggere". Una volta generata tramite l'inferenza sulla clausola selezionata e *active*, la nuova clausola *new* viene aggiunta a *passive* solo se $new \neq \square$.

Otter L'algoritmo *otter*, rispetto all'algoritmo 1, aggiunge alcune semplificazioni in modo da ridurre il numero di clausole.

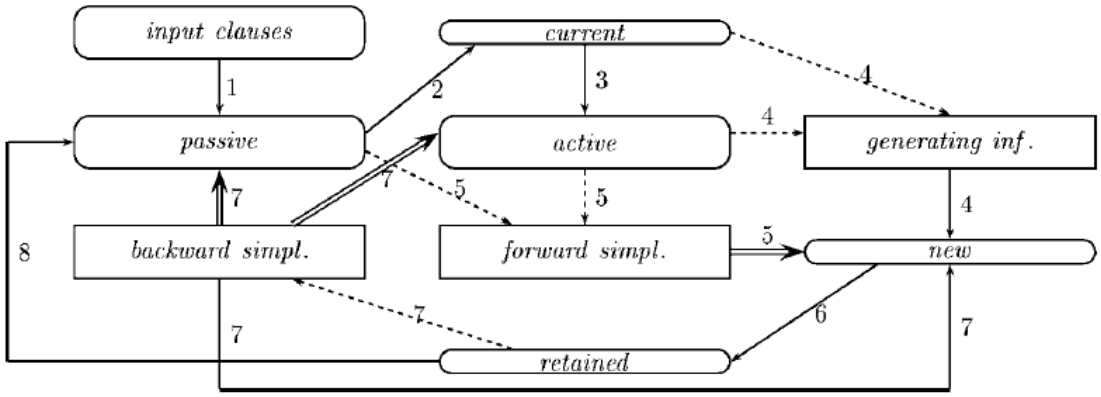


Figura 1.2: Algoritmo otter [2]

In aggiunta all'algoritmo precedente, ci sono:

- *forward simplification*: un'operazione che cerca di semplificare la clausola *new* coinvolgendo anche le clausole negli insiemi *active* e *passive*.
- *retained*: un insieme in cui passa la clausola *new* dopo la *forward simplification*. Su questa clausola viene effettuato un *retention test* e, tramite alcuni criteri (p.e. peso della clausola ovvero grandezza) e l'applicazione di *deletion rules*, viene deciso se la clausola è scartabile o no.
- *backward simplification*: un'operazione eseguita dopo che la clausola ha superato il *retention test*. In questo caso, al contrario della *forward simplification*, si cerca di semplificare gli insiemi *active* e *passive* tramite la clausola generata.

Limited resource strategy L'algoritmo *lrs* è una variante di *otter* poiché, aggiungendo un limite di tempo, cerca di identificare quali clausole nell'insieme *passive* non hanno possibilità di essere selezionate e le scarta.

Discount L'algoritmo *discount* non usa l'insieme *passive* per effettuare semplificazioni. Questa scelta è dettata dalla cardinalità di *passive* in quanto è molto maggiore di *active*, quindi la velocità di inferenza potrebbe rallentare significativamente a causa delle semplificazioni sull'insieme *passive*.

Il *kernel* è costituito da molte parti, le principali sono:

- il *main loop*
- il *generating inference engine*
- il *simplifying inference engine*
- uno *splitter*

Sono definite a corredo numerose strutture dati, come indici e buffer, per gestire le sostituzioni, le unificazioni, ...

Main loop Contiene il vero e proprio algoritmo per la risoluzione del problema.

Generating inference engine É il cuore delle *generating inferences*, in quanto funge da database per queste inferenze ed è responsabile del loro utilizzo.

Simplifying inference engine É simile al precedente ma gestisce le inferenze che permettono le semplificazioni.

Splitter É un componente che interviene sulle clausole generate, che passano il *retention test*, se sono *splittable* ovvero

Definizione 10. Siano $C_1, \dots C_n$ clausole con $n \geq 2$ e tutte le clausole hanno insiemi di variabili disgiunte a coppie, $F = C_1 \vee \dots \vee C_n$ è detta *splittable* poiché è possibile dividerla in n componenti $F_1 = C_1, \dots, F_n = C_n$.

Per rendere più efficiente questa operazione, lo *splitter* collabora con un SAT solver (Minisat o Z3). Questa architettura è definita AVATAR [4].

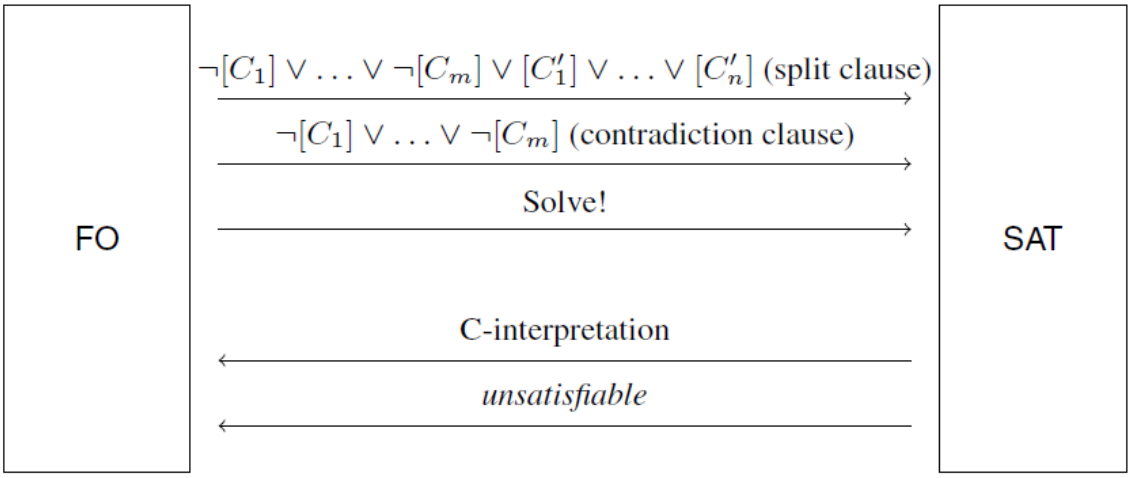


Figura 1.3: Collaborazione tra Vampire (FO) e SAT solver [4]

Il *main loop*, prima di essere avviato, viene configurato tramite la scelta del tipo di algoritmo di saturazione da usare (lrs, otter o discount). Vengono inizializzati un *generating inference engine* e un *simplifying inference engine*, che vengono popolati rispettivamente con delle *generating inferences* e delle *simplifying inferences*. Successivamente questi due *engines* vengono collegati all'algoritmo di saturazione. A questo punto, il *main loop* è configurato correttamente e può essere avviato. Nella figure sottostanti 1.4, 1.5 e 1.6, vengono presentate le relazioni tra le classi principali del kernel e le relazioni tra le classi dell'*inference engine*.

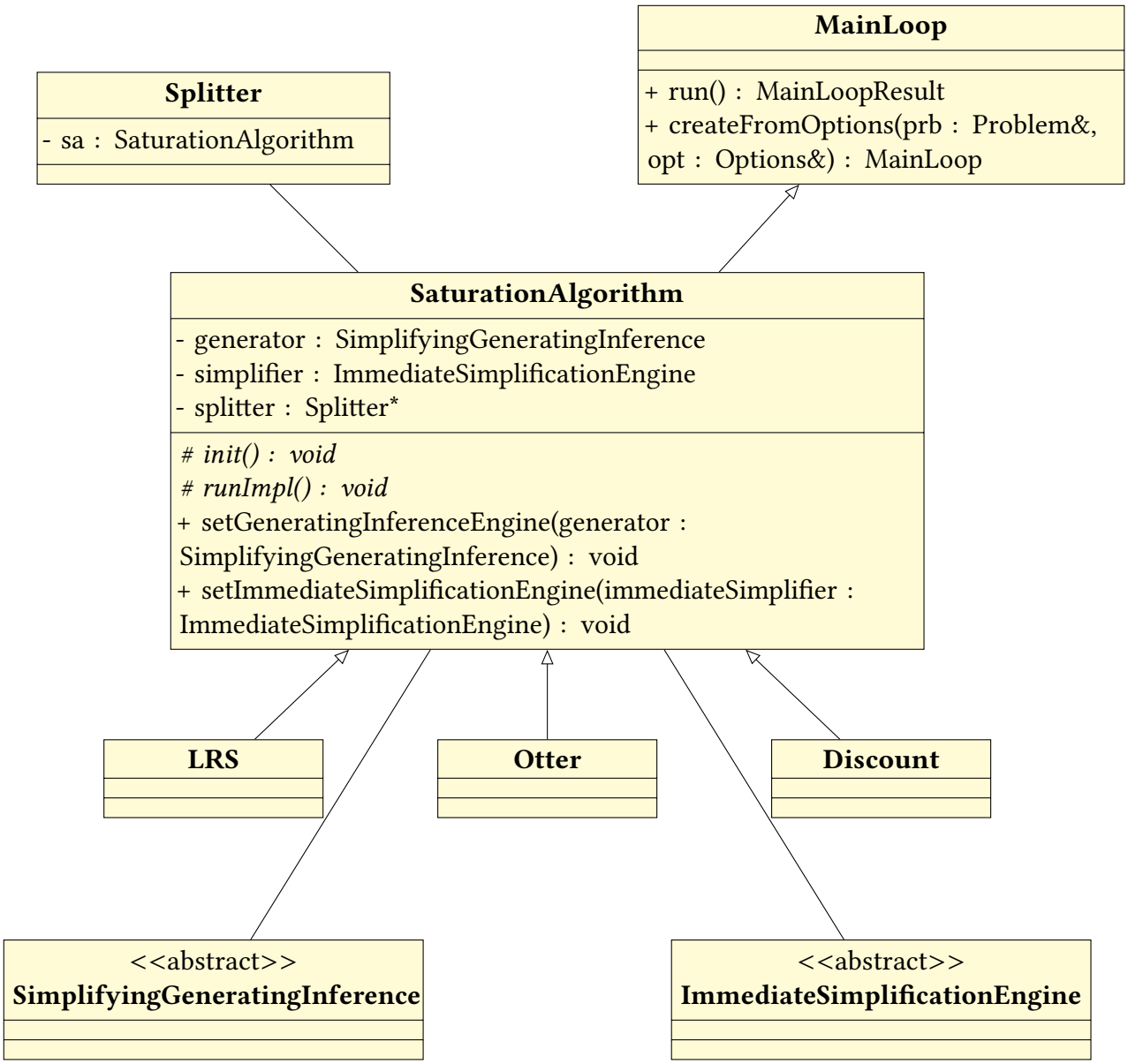


Figura 1.4: Relazioni tra le classi principali del *kernel*

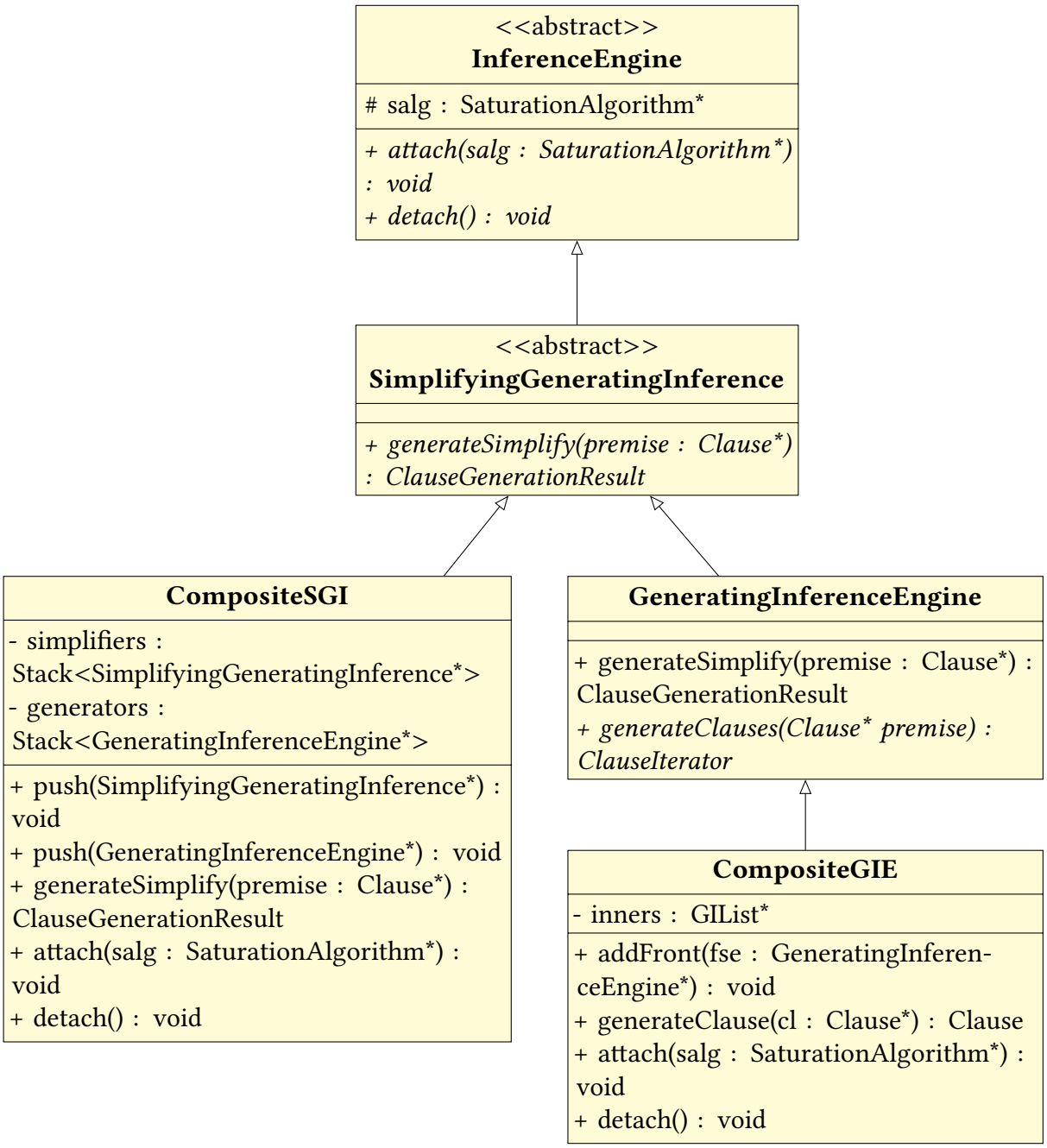


Figura 1.5: Relazioni tra le classi dell'*inference engine* per le *generating inferences*

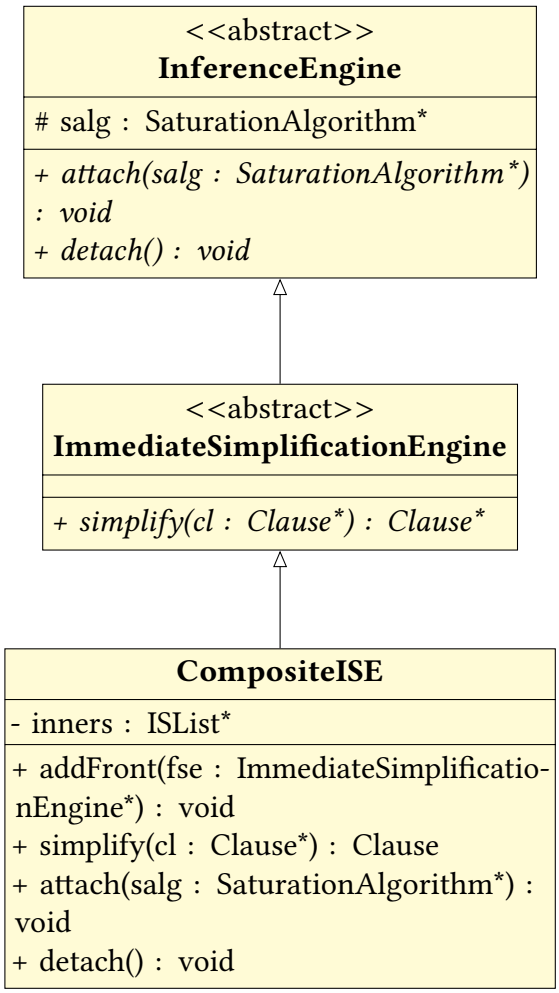


Figura 1.6: Relazioni tra le classi dell'*inference engine* per le *simplifying inferences*

Si noti che tutte le *generating inferences* (resolution, superposition) ereditano la classe `GeneratingInferenceEngine`, così come tutte le *simplifying inferences* ereditano la classe `ImmediateSimplificationEngine`.

Nella fase di configurazione del *main loop*, viene inizializzato un `CompositeGIE` in cui vengono memorizzate tutte le inferenze tramite la funzione `addFront()` nella lista *inners*. Una volta aggiunte tutte le regole di inferenza, viene inizializzato un `CompositeSGI` che memorizza il `CompositeGIE` nello stack *generators* e altre regole di semplificazione nello stack *simplifiers*. Nello stesso modo viene inizializzato un `CompositeISE`.Alla fine, al `SaturationAlgorithm` viene assegnato un generatore (ovvero il `CompositeGIE`) tramite la funzione `setGeneratingInferenceEngine()` e un `ImmediateSimplificationEngine` tramite la funzione apposita.

Esempio 1. Di seguito viene riportato un algoritmo riassuntivo, definendo un algoritmo di saturazione di tipo *otter* con un *superposition inference system* come visto nella definizione 8:

Algoritmo 2 Esempio semplificato per definire un algoritmo di saturazione in VAMPIRE

```
var salg : SaturationAlgorithm
var gie : CompositeGIE
var sgi : CompositeSGI
function createFromOptions
    salg = Otter
    gie.addFront(Resolution)
    gie.addFront(Factoring)
    gie.addFront(Superposition)
    gie.addFront(EqualityResolution)
    gie.addFront(EqualityFactoring)
    sgi.push(gie)
    salg.setGeneratingInferenceEngine(sgi)
    salg.setImmediateSimplificationEngine(createISE())
    return salg
end function
```

La trattazione è stata limitata a una semplificazione di VAMPIRE, che presenta una serie di aspetti più complessi, al fine di fornire una base per la comprensione dei concetti che verranno affrontati nei capitoli successivi.

2. Frammento Guarded

Il frammento *guarded* è un frammento della logica del primo ordine introdotto da Hajnal Andréka, István Nèmeti e Johan van Benthem. L'importanza è dovuta alla possibilità di tradurre molte logiche modali, con significative proprietà computazionali e logiche, in formule del primo ordine appartenenti al frammento *guarded*. É stato provato che questo frammento (con uguaglianza e non) è decidibile in 2-EXPTIME.

Sono state proposte numerose generalizzazioni di questo frammento in quanto alcune logiche modali importanti (come quella temporale) non possono essere tradotte. La generalizzazione piu datata è il frammento *loosely guarded* che ha dei vincoli meno stringenti sulla definizione di guardia. L'implementazione della procedura di decisione in questa tesi riguarda il frammento *guarded* senza uguaglianza [5].

Definizione 11. Il frammento *guarded* è ricorsivamente definito come i seguenti sottoinsiemi di logica del primo ordine senza uguaglianza e simboli di funzione:

1. $\top, \bot \in \mathrm{GF}$.
2. Se a è una formula atomica, allora $a \in \mathrm{GF}$.
3. GF è chiuso rispetto a $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.
4. Sia $A \in \mathrm{GF}$, a formula atomica tale che ogni variabile libera di A occorra almeno una volta negli argomenti di a. Allora
   - $\forall \bar{x}(a \rightarrow A) \in \mathrm{GF}$,
   - $\exists \bar{x}(a \wedge A) \in \mathrm{GF}$,
   - $\forall \bar{x}(a \vee A) \in \mathrm{GF}$ (negazione della precedente)

   con la formula atomica a definita **guardia**.

Dopo aver delineato quando una formula del primo ordine appartiene al frammento *guarded*, di seguito viene definito quando una clausola è *guarded*.

Definizione 12. Una clausola c è *guarded* se soddisfa le seguenti condizioni:

1. Ogni termine funzionale non *ground* in c contiene tutte le variabili di c.

2. Se c è non ground allora c'è un letterale negativo $\neg A$ in c che contiene tutte le variabili di c ma non contiene termini funzionali non *ground*.

   Il letterale negativo $\neg A$ è definito **guardia**.

Se la clausola c è *ground*, allora è *guarded*.

Definizione 13. Una insieme formato da clausole *guarded* è anch'esso *guarded*

2.1 Preprocessing

Come menzionato nel precedente capitolo 1.1.2, il problema viene pre-processato per ottenere un insieme di clausole da sottomettere all'algoritmo di risoluzione. Le operazioni di *preprocessing* sono le seguenti:

1. *Negation normal form*
2. $Struct_\forall$
3. *Skolemization*
4. *Clausification*

L'unica operazione non ancora definita è $Struct_\forall$.

Definizione 14. $Struct_\forall$ è una trasformazione strutturale che è ottenuta da:

1. la sostituzione delle sotto-formule $\forall \bar{x}(a \rightarrow A)$, $\forall \bar{x}(a \vee A)$ che hanno $\bar{y}$ variabili libere, con un nuovo nome $\alpha(\bar{y})$
2. l'aggiunta della formula $\forall \bar{x}\bar{y}(\neg a \vee \neg\alpha \vee A)$

Queste operazioni, oltre a generare un insieme di clausole, preservano l'appartenenza delle formule al frammento *guarded*.

Teorema 1. *Sia $F \in GF$ allora*

1. *$F' = NNF(F) \in GF$,*
2. *$F'' = \text{Struct}_\forall(F') \in GF$,*
3. *applicando skolemization e clausification a F'' si ottiene un insieme di clausole guarded.*

Il teorema è dimostrato nello studio di De Nivelle e De Rijke in [5].

2.2 Resolution

La procedure di decisione per il frammento *guarded* necessita della definizione di una nuova regola di risoluzione. Prima di definire tale regola, è necessario stabilire: un ordinamento e dei parametri secondo cui ordinare.

Definizione 15. La *Vardepth* di un termine/atomo A è definita ricorsivamente come segue:

$$Vardepth(A) = \begin{cases} -1 & \text{se } A \text{ è ground} \\ 0 & \text{se } A \text{ è una variabile} \\ \max\{1 + Vardepth(t_1), \ldots, 1 + Vardepth(t_n)\} & \text{se } A = f(t_1, \ldots, t_n) \end{cases}$$

Definizione 16. La *Vardepth* di un letterale equivale alla *Vardepth* di un atomo.

Definizione 17. La *Vardepth* di una clausola c equivale alla *Vardepth* massimale di un letterale in c.

Definizione 18. Sia A atomo, letterale o clausola, allora $Var(A)$ è definito come l'insieme delle variabili che occorrono in A.

Una volta definiti questi due parametri *Vardepth* e *Var*, è possibile definire l'ordinamento.

Definizione 19. Definiamo l'ordinamento $\prec$ sui letterali A, B come segue:

$$A \prec B \quad \text{se} \quad (Vardepth(A) < Vardepth(B)) \vee (Var(A) \subset Var(B))$$

Ora è possibile introdurre la nuova regola di risoluzione definita *ordered resolution rule*.

Definizione 20. Sia $\prec$ ordinamento sui letterali, $\{A_1\} \cup R_1$ e $\{\neg A_2\} \cup R_2$ clausole allora se:

1. $\{A_1\} \cup R_1$ e $\{\neg A_2\} \cup R_2$ non hanno variabili in comune,
2. A_1 deve essere massimale in R_1 ovvero $\nexists A \in R_1 : (A_1 \prec A)$,
3. A_2 deve essere massimale in R_2 ovvero $\nexists A \in R_2 : (A_2 \prec A)$,
4. A_1 e A_2 hanno un unificatore più generale (*mgu*) θ,

allora $R_1\theta \cup R_2\theta$ è chiamato $\prec$-*ordered resolvent* di $\{A_1\} \cup R_1$ e $\{\neg A_2\} \cup R_2$.

$$\frac{\{\underline{A_1}\} \vee R_1 \quad \{\underline{\neg A_2}\} \vee R_2}{(R_1 \vee R_2)\theta} \tag{2.1}$$

Questa regola è una *resolution rule* (1.1) ristretta da un'ordinamento $\prec$. É possibile definire anche una *factorization rule* (8) ristretta.

Definizione 21. Sia $\prec$ ordinamento sui letterali, $\{A_1, A_2\} \cup R$ clausola allora se:

1. A_1 deve essere massimale in R ovvero $\nexists A \in R : (A_1 \prec A)$,
2. A_1 e A_2 hanno un unificatore più generale (*mgu*) θ,

allora $\{A_1\theta\} \cup R\theta$ è chiamato $\prec$-*ordered factor* di $\{A_1, A_2\} \cup R$.

$$\frac{\{\underline{A_1} \vee \underline{A_2}\} \vee R}{(A_1 \vee R)\theta} \tag{2.2}$$

Nell'algoritmo di risoluzione è necessaria una *resolution rule* ristretta dall'ordinamento $\prec$ poiché preserva la proprietà degli $\prec$-*ordered resolvent* di essere *guarded*. Invece la *factorization rule* **preserva la proprietà delle clausole di essere *guarded* anche senza applicare una restrizione**. Tutto questo viene provato dal seguente teorema per la cui dimostrazione si rimanda a [5]:

Teorema 2.
1. *Se* c_1 *e* c_2 *sono clausole* guarded *allora* c $\prec$-ordered resolvent *di* c_1 *e* c_2 *è* guarded
2. *Se* c_1 *è una clausola* guarded *allora* c factor *di* c_1 *è* guarded

Un algoritmo di risoluzione, per essere definito procedura di decisione, deve terminare ed essere completo.

L'algoritmo con la $\prec$-*ordered resolution rule* e la *factorization rule* termina perché, grazie alla restrizione sulla *resolution rule*, è possibile derivare solo un insieme finito di clausole da un insieme finito di clausole *guarded*.

Lemma 1. *Sia* C *un insieme finito di clausole* guarded. *Se* $\bar{C}$ *è l'insieme generato da* C *tramite la* $\prec$-ordered resolution rule *e la* factorization rule, *allora* $\bar{C}$ *ha dimensione finita.*

La dimostrazione di questo lemma è possibile poiché il numero di variabili e la *Vardepth* sono limitati superiormente [5].

Definizione 22. Un algoritmo di risoluzione è completo per una logica se riesce a risolvere ogni problema appartenente a quella logica.

Nello studio di De Nivelle e De Rijke è presentata la dimostrazione della completezza per la procedura di decisione [5].

3. Implementazione della procedura

Come descritto nel precedente capitolo, la procedura di decisione per problemi appartenenti al frammento *guarded* è formata da una fase di *preprocessing* e una fase di risoluzione con $\prec$-*ordered resolution* e *factorization*.

Per la fase di preprocessing è stato costruito un *preprocessor* per trattare problemi appartenenti al frammento *guarded*. Questa componente è stata implementata seguendo l'architettura del *preprocessor* predefinito.

Algoritmo 3 Preprocessing

var problem : sets of guarded formula
var result : sets of guarded clause
problem = *NNF*(problem)
problem = *flatten*(problem)
problem =$Struct_\forall$(problem)
problem = *skolemize*(problem)
result = *clausify*(problem)
return result

NNF, flatten, skolemize e *clausify* (descritte in 1.1.2) sono le funzioni di Vampire che vengono sfruttate per la trasformazione del problema. Invece la funzione $Struct_\forall$ viene implementata ex-novo in quanto non è presente nel sistema. La funzione è ricorsiva per sfruttare la struttura ad albero delle formule.

Algoritmo 4 $Struct_\forall$

var input, new, α : formula
var $\bar{y}$: sets of guarded clause
$\bar{y}$ = *saveFreeVariable*(input)
α = *createNewLiteral*($\bar{y}$)
new = *generateNewFormula*(*alpha*,input)
addFormulaToUnits(new)
input = α
return input

Si scorre tutto l'albero e nel caso venisse trovata una formula del tipo $\forall \bar{x}(a \vee A)$ si procede come descritto nella definizione 14:

1. si sostituisce la formula con un nuovo predicato $\alpha(\bar{y})$
2. si aggiunge la formula $\forall \bar{x}\bar{y}(\neg a \vee \alpha \vee A)$

Bibliografia

[1] Laura Kovács e Andrei Voronkov. «First-order theorem proving and Vampire». In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 1–35.

[2] Alexandre Riazanov e Andrei Voronkov. «The design and implementation of VAMPIRE». In: *AI communications* 15.2-3 (2002), pp. 91–110.

[3] Giles Reger, Martin Suda e Andrei Voronkov. «New Techniques in Clausal Form Generation.» In: *GCAI* 41 (2016), pp. 11–23.

[4] Andrei Voronkov. «AVATAR: The architecture for first-order theorem provers». In: *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer. 2014, pp. 696–710.

[5] Hans De Nivelle e Maarten De Rijke. «Deciding the guarded fragments by resolution». In: *Journal of Symbolic Computation* 35.1 (2003), pp. 21–58.