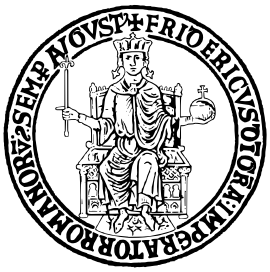
Università degli Studi di Napoli Federico II



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 uno strumento molto importante che permette di verificare formalmente sistemi software e hardware ed è 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 tool, la ricerca si sta muovendo verso la definizione di decisione di procedure per problemi decidibili. Una decisione di procedura è un algoritmo che verifica la soddisfacibilità di un problema e termina sempre. Un sottoinsieme di problemi decidibili nella logica del primo ordine è detto frammento. La ricerca di queste procedure di decisione è essenziale poiché, in alcuni casi, questi approcci possono risultare più efficienti di una strategia general purpose. L'ideale sarebbe inglobare, quando è possibile, queste procedure di decisione nei theorem prover e usarle quando un problema appartiene 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* più famoso ed efficiente in circolazione per la risoluzione di problemi di logica del primo ordine. Il documento è diviso in quattro capitoli:

1. Il primo capitolo fornisce un'introduzione ai concetti base di VAMPIRE propedeutici ai capitoli successivi
2. Il secondo capitolo formalizza tutti i concetti riguardanti il frammento *guarded*
3. Il terzo capitolo descrive l'implementazione della procedura di decisione in VAMPIRE
4. Il quarto capitolo presenta i risultati ottenuti dal confronto tra VAMPIRE e VAMPIRE con la nuova procedura di decisione

1 Panoramica su Vampire

In questo capitolo viene presentato il theorem prover VAMPIRE, la sua architettura e alcune delle caratteristiche più importanti.

1.1 Cos'è VAMPIRE

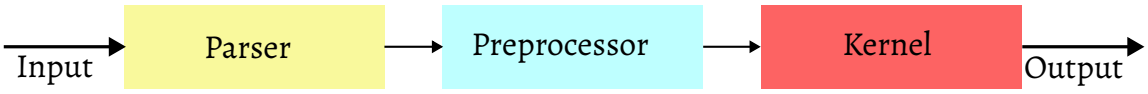
VAMPIRE è un sistema che permette di provare la validità di teoremi (matematici e non) di logica del primo ordine. Il linguaggio di programmazione usato è C++ e la prima versione è stata sviluppata principalmente da Andrei Voronkov e Krystof Hoder tra il 1993 e il 1995 nell'Università di Manchester. Dopo il 1995, il progetto è stato sospeso e viene 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]. Ecco alcune delle principali caratteristiche del sistema:

- è molto veloce
- è portabile sulle piattaforme più comuni
- è semplice da usare
- ha strategie di ricerca con risorse limitate
- supporta numerose sintassi in input
- i vari tentativi per provare il problema possono essere parallelizzati su più processori
- può produrre, in base alle opzioni selezionate, output molto dettagliati

1.2 Architettura di VAMPIRE

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

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



1.2.1 Parser

Il *parser* è un modulo adibito al parsing del problema. Il problema viene letto da un file e viene incapsulato in una classe. Nel caso di un problema di logica del primo ordine con sintassi TPTP, allora 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. Oltre alle rappresentate `NegatedFormula` e `QuantifiedFormula`, sono presenti altre classi che ereditano `Formula` come `AtomicFormula`, `JunctionFormula`, `BinaryFormula`, ...

Nota bene. Un' unità può essere una formula o una clausola ma non entrambe. La distinzione viene effettuata sulla base dell' attributo *kind* che è un' enumerazione.

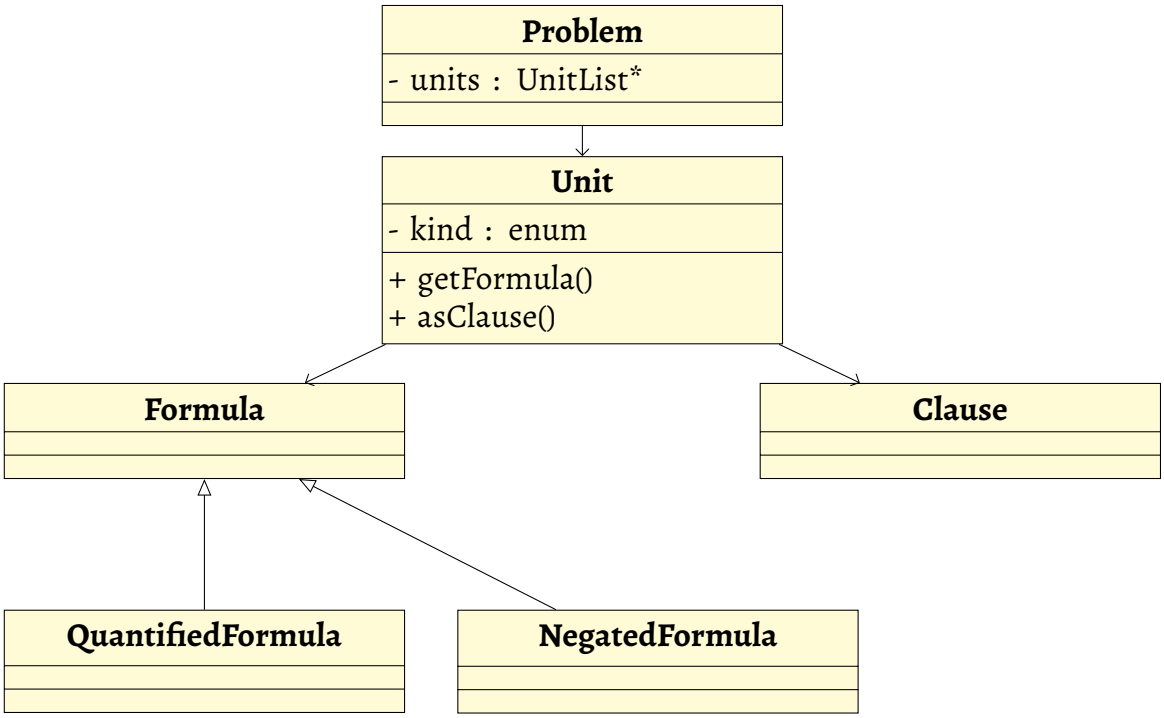


Figura 1.1: Relazioni tra le classi per definire un problema

1.2.2 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 molte semplificazioni in modo da rendere il sistema il più veloce possibile. Di seguito sono descritti solo i passaggi fondamentali (esposti in maniera più ampia in [3]):

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

III step *Flatten* $\longrightarrow$ 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* $\longrightarrow$ Il *preprocessor* rimuove i predicati e le definizioni delle funzioni che non vengono usati.

V step *ENNF* $\longrightarrow$ Il *preprocessor* trasforma le formula in modo da ottenerle 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* $\longrightarrow$ 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 $\otimes$.

Nota bene. 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 formula in modo da ottenerle 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* $\longrightarrow$ Il *preprocessor* applica la tecnica di skolemizzazione per eliminare i $\exists$ dalle formule.

Definizione 4. Sia $F = \exists x \mid \varphi(y_1, \ldots, y_n, x)$ formula in negation normal form allora la skolemizzazione 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* $\longrightarrow$ Il *preprocessor* applica la clausificazione su tutte le formule in modo da ottenere un insieme di clausole.

Definizione 5. La clausificazione 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*.

1.2.3 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. Questo è possibile tramite la saturazione dell'insieme con tutte le possibili 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. Per questo scopo ci si rifà a [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 seleziona un sottoinsieme non vuoto di letterali in ogni clausola non vuota. In $\underline{L} \vee R$ $\underline{\text{L}}$ 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**

$$\frac{\underline{\underline{A}} \vee \underline{A'} \vee C}{(A \vee C)\theta} \tag{1.2}$$

  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**

$$\frac{\underline{\underline{s \neq t}} \vee C}{C\theta} \tag{1.4}$$

  in cui θ è l'unificatore più generale di s e t.

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

- **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:

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 questa 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* □

Inoltre, un algoritmo di saturazione deve essere *fair* quindi 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 □, 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*. Questo algoritmo (1) è molto semplificato ma sintetizza, ad alto livello, come lavorano le soluzioni 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 subire semplificazioni. 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()*. É questa la funzione 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 le clausole più "vecchie" hanno priorità maggiore, 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.

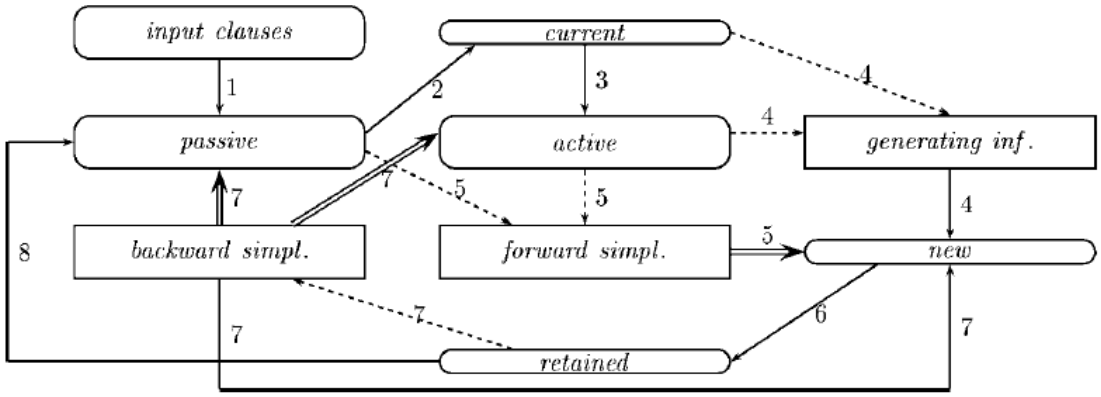


Figura 1.2: Algoritmo otter [2]

In aggiunta all'algoritmo precedente, ci sono:

- *forward simplification* è un'operazione che cerca di semplificare la clausola *new* coinvolgendo anche le clausole negli insiemi *active* e *passive*.
- *retained* è 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* è 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 variazione 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 causata dalla cardinalità di *passive* in quanto è molto maggiore di *active*, quindi la velocità di inferenza potrebbe rallentare molto 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 magazzino 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].

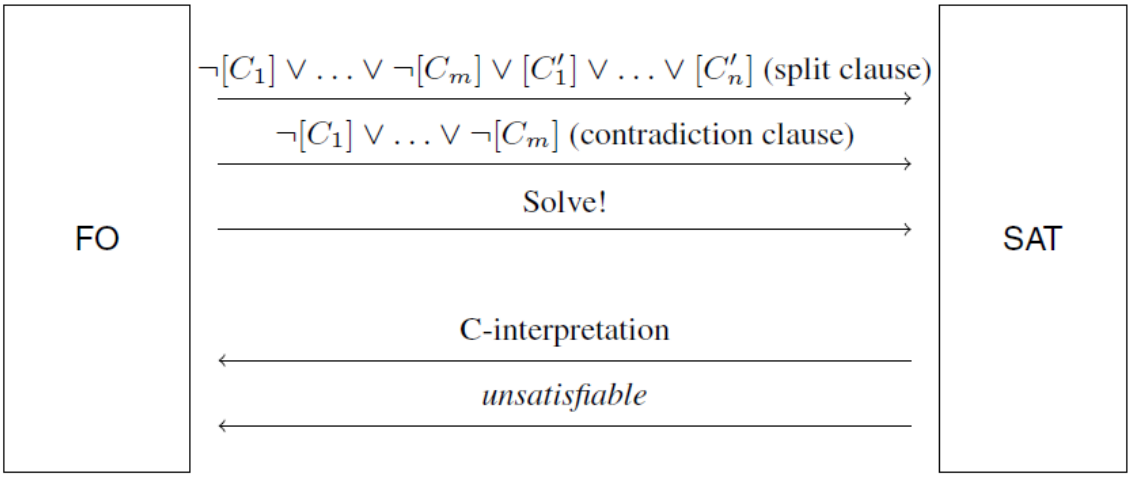


Figura 1.3: Collaborazione tra VAMPIRE (FO) e SAT solver [4]

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*.

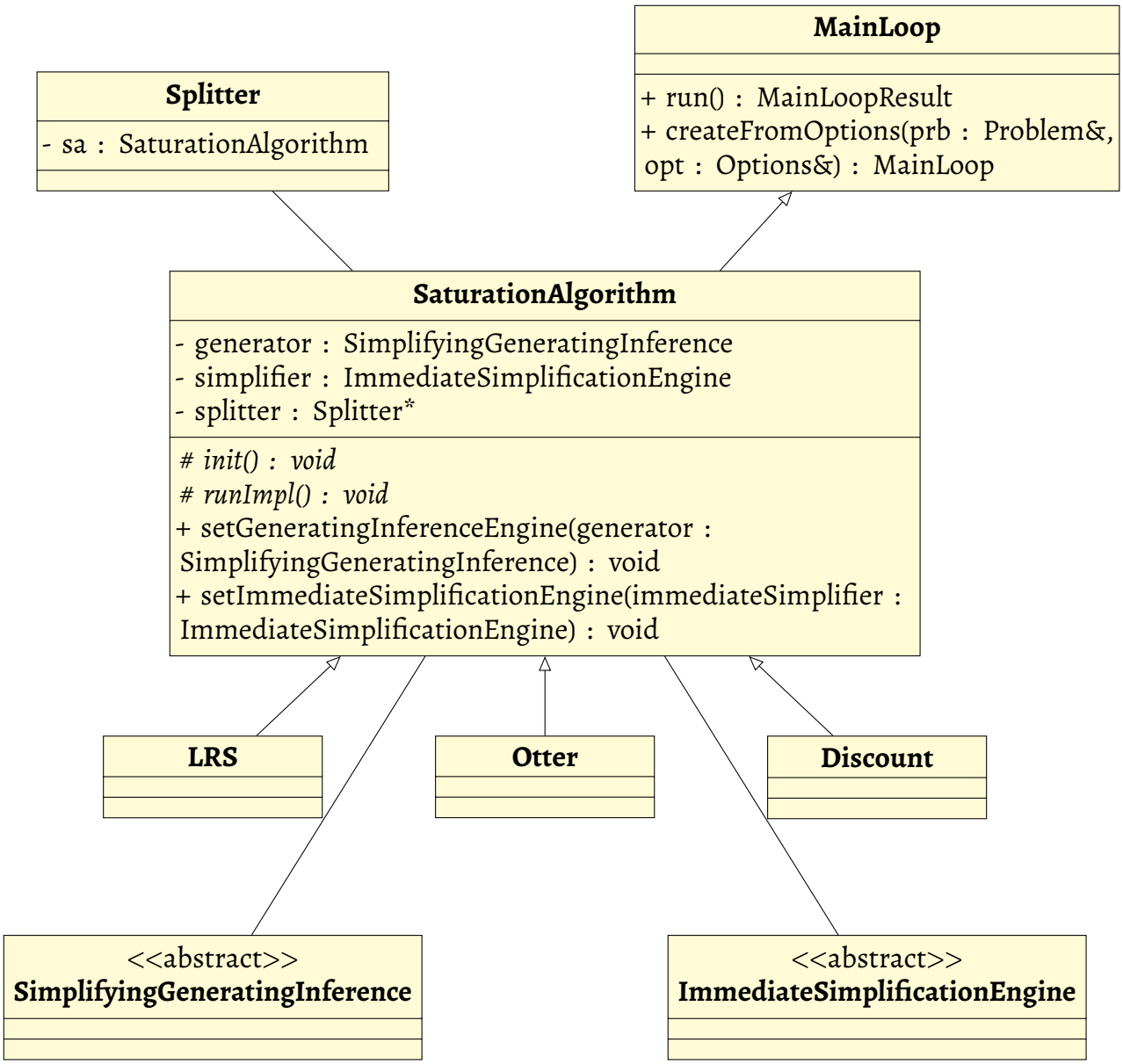


Figura 1.4: Relazioni tra le classi principali del *kernel*

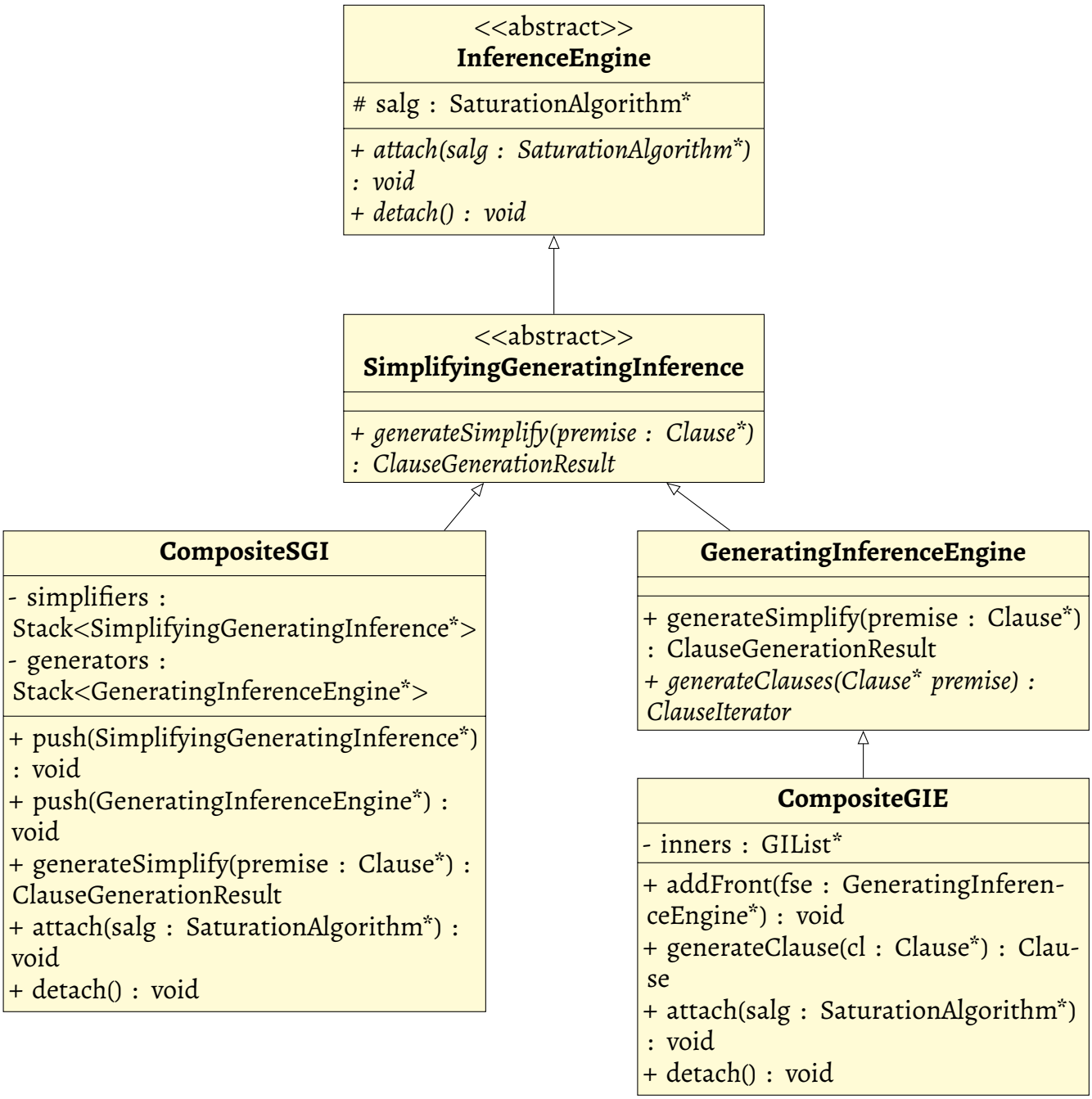


Figura 1.5: Relazioni tra le classi dell'*inference engine* per le *generating inferences*

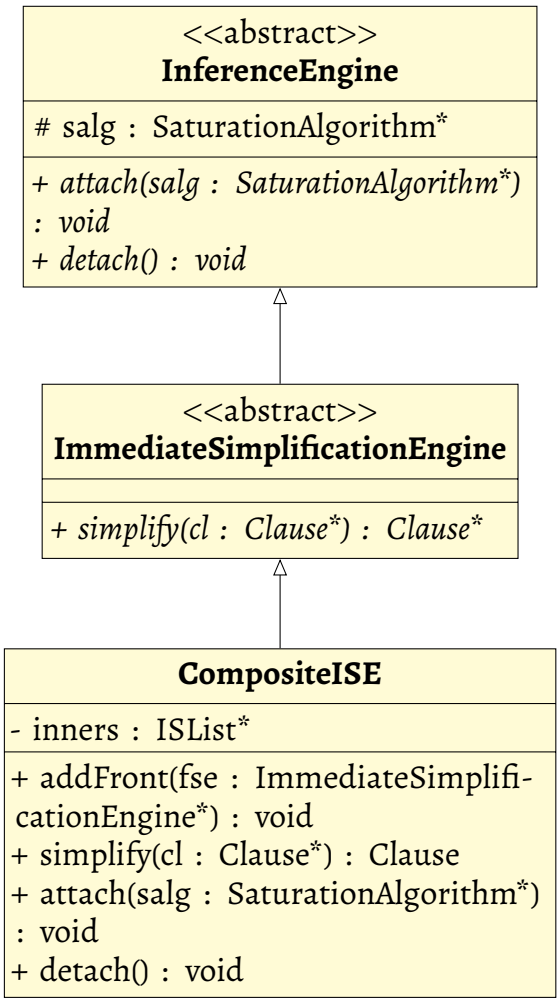


Figura 1.6: Relazioni tra le classi dell'*inference engine* per le *simplifying inferences*

Nota bene. 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` nel 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 finora è stata limitata a una semplificazione di VAMPIRE, che presenta, in realtà, una serie di aspetti più complessi. Questa scelta è stata effettuata per fornire una base per la comprensione dei concetti che verranno affrontati nei capitoli successivi.

2 Frammento Guarded

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.