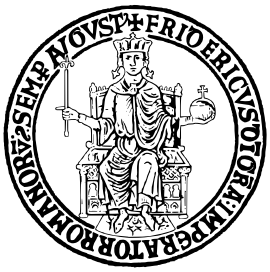
Università degli Studi di Napoli Federico II



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

Relatore
Prof. Fabio Mogavero

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

Relatore
Prof. Fabio Mogavero

Candidato
Francesco Scarfato
N86/3769

Anno Accademico 2022–2023

Indice

–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

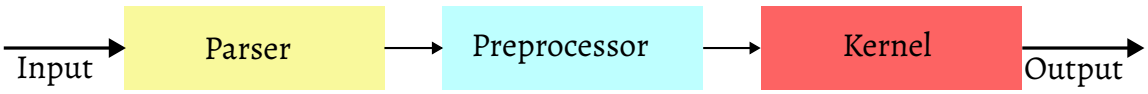
VAMPIRE è un sistema che permette di provare la validità di teoremi (matematici e non) di logica del primo ordine. Il linguaggio di programmazione usato è C++ e la prima versione è stata sviluppata principalmente da Andrei Voronkov e Krystof Hoder tra il 1993 e il 1995 nell'Università di Manchester. 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). 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.

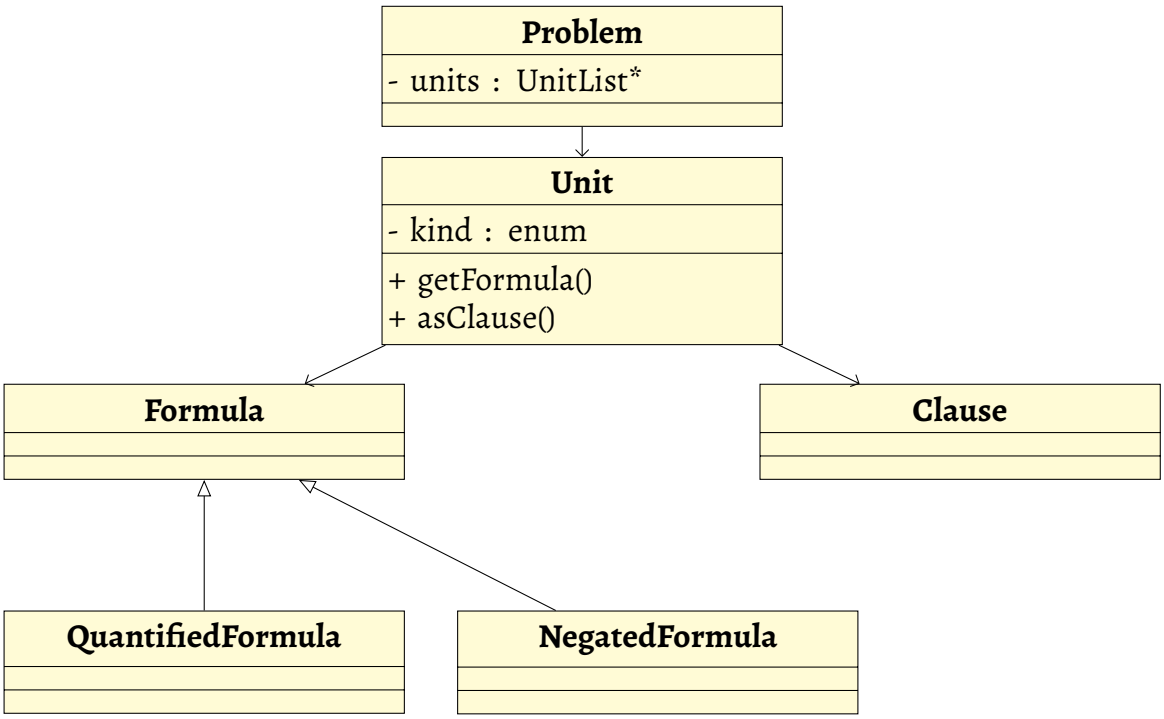


Figura 1.1: Relazioni tra le classi

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:

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 $\text{def}(\varphi, \pi, p)$ tale che

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

in cui $\text{pol}(\varphi, \pi)$ indica la polarità della sotto-formula di φ in posizione π. $\text{pol}(\varphi, \pi) = 0$ 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 $F = \exists x \mid \varphi(y_1, \dots, y_n, x)$ formula in negation normal form allora la skolemizzazione si ottiene tramite la seguente sostituzione:

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

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

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 $\underline{L} \vee R$ $\underline{\mathrm{L}}$ 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**

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

  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.

- **Equality factoring**

$$\frac{\underline{\underline{s = t}} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\theta} \tag{1.5}$$

  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* □

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

Per rendere efficiente l'algoritmo di saturazione, oltre a 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*. [1, 2, 3]

Algoritmo 1 Given clause algorithm

```
1:  var active,passive : sets of clause
2:  var current,new : clause
3:  active = ∅
4:  passive = set of input clauses
5:  while passive ≠ ∅ do
6:      current = select(passive)
7:      passive = passive ∖ {current}
8:      active = active ∪ {current}
9:      new = infer(current,active)
10:     if new is □ then
11:         return provable
12:     end if
13:     passive = passive ∪ {new}
14: end while
15: return unprovable
```

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.