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 Candidato

Prof. Fabio Mogavero

Prof. Massimo Benerecetti

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

In	trodu	ızione	1				
1	Van	pire theorem prover	3				
	1.1	Architettura di VAMPIRE	3				
		1.1.1 Parser	4				
		1.1.2 Preprocessor	4				
		1.1.3 Kernel	6				
2	Fran	nmento Guarded	16				
	2.1	Preprocessing	17				
	2.2	Resolution	18				
3	Imp	plementazione della procedura 20					
	3.1	Preprocessing	21				
	3.2	Resolution	26				
	3.3	Classificazione	27				
4	Rist	ıltati sperimentali	32				
	4.1	Problemi FOF	34				
	4.2	Problemi CNF	38				
Bi	bliog	rafia	42				

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, ...

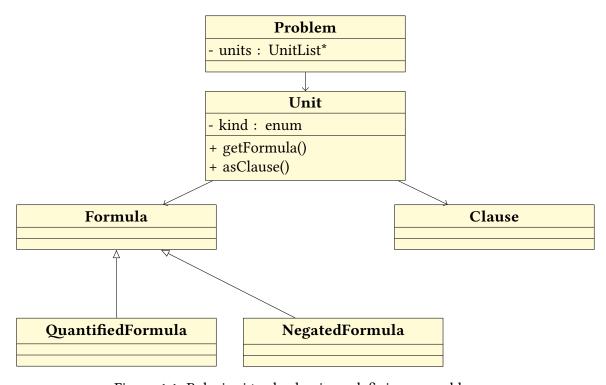


Figura 1.1: Relazioni tra le classi per definire un problema

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 $def(\varphi, \pi, p)$ tale che

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

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

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 $NNF \longrightarrow II$ preprocessor transforma 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* \longrightarrow Il *preprocessor* applica questa tecnica 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 *skolemization* 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 $\{x \mapsto sk(y_1, ..., y_n)\}$ indica che x, la variabile precedentemente vincolata dal quantificatore esistenziale nella formula F, è sostituita da una nuova funzione $sk(y_1, ..., y_n)$, 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. $\forall x \mid \varphi(y_1, ..., y_n, x) \Rightarrow \varphi(y_1, ..., y_n, x)\{x \mapsto X\}$ 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 > sui termini è detto *simplification ordering* se rispetta le seguenti condizioni:

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

$$\frac{\underline{l=r} \vee C_1}{(L[r] \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l=r} \vee C_1}{(t[r] = t' \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l=r} \vee C_1}{(t[r] = t' \vee C_1 \vee C_2)\theta} \quad \frac{\underline{l=r} \vee C_1}{(t[r] \neq t' \vee C_1 \vee C_2)\theta} \tag{1.3}$$

in cui θ è l'unificatore più generale di l e s, s non è una variabile, $r\theta < l\theta$, solo nella prima regola L[s] non è un equality literal¹,

Equality resolution

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

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

Equality factoring

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

in cui θ è l'unificatore più generale di s e t, $t\theta < s\theta$ e $t'\theta < t\theta$.

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

 $^{^1 \}mathrm{Un}$ 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 \Box .

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 □, 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.

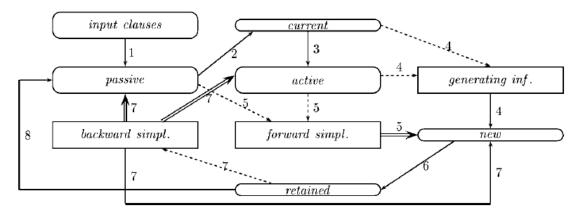


Figura 1.2: Algoritmo otter [2]

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 $C_1, \ldots C_n$ clausole con $n \ge 2$ e tutte le clausole hanno insiemi di variabili disgiunte a coppie, $F = C_1 \lor \cdots \lor C_n$ è detta *splittable* poiché è possibile dividerla in n componenti $F_1 = C_1, \ldots, F_n = C_n$.

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

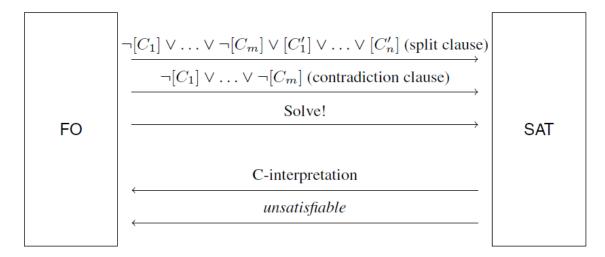


Figura 1.3: Collaborazione tra VAMPIRE (FO) e SAT solver [4]

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.

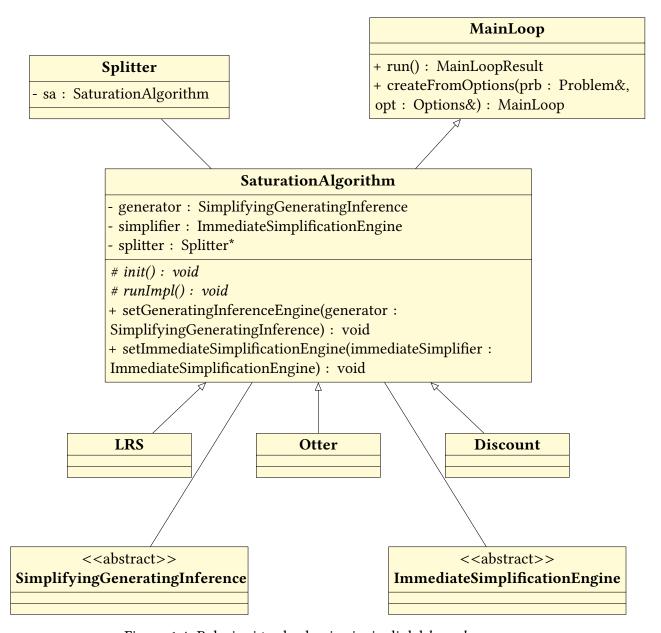


Figura 1.4: Relazioni tra le classi principali del kernel

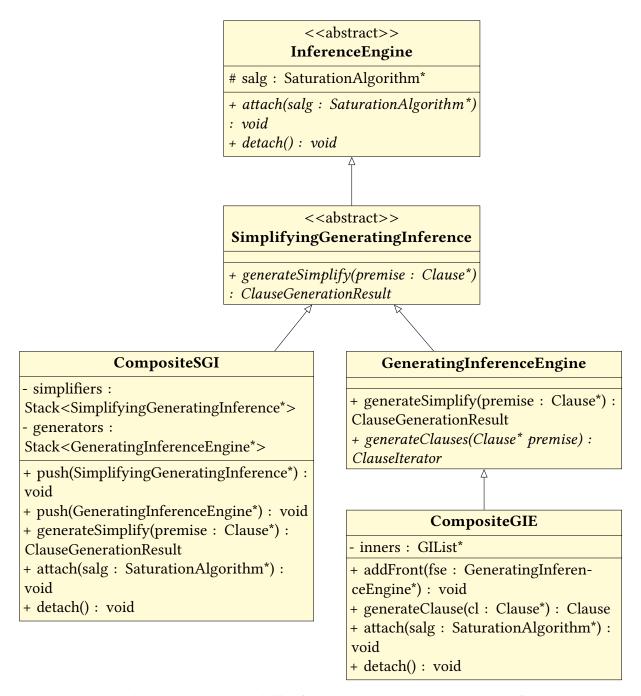


Figura 1.5: Relazioni tra le classi dell'inference engine per le generating inferences

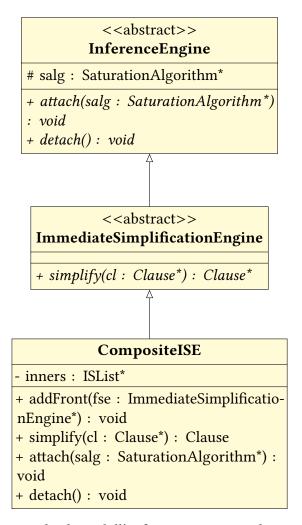


Figura 1.6: Relazioni tra le classi dell'inference engine per le simplifying inferences

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 , $\bot \in GF$.
- 2. Se a è una formula atomica, allora $a \in GF$.
- 3. GF è chiuso rispetto a \neg , \land , \lor , \rightarrow , \leftrightarrow .
- 4. Sia $A \in GF$, a formula atomica tale che ogni variabile libera di A occorra almeno una volta negli argomenti di a. Allora
 - $\forall \bar{x}(a \to A) \in GF$,
 - $\exists \bar{x}(a \land A) \in GF$,
 - $\forall \bar{x}(a \lor A) \in GF$ (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 *∀*
- 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 $\forall \bar{x}(a \to A), \, \forall \bar{x}(a \lor A)$ che hanno \bar{y} variabili libere, con un nuovo nome $\alpha(\bar{y})$
- 2. l'aggiunta della formula $\forall \bar{x}\bar{y}(\neg a \lor \neg \alpha \lor A)$

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:

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

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 < sui letterali *A*, *B* come segue:

$$A < B$$
 se $(Vardepth(A) < Vardepth(B)) \lor (Var(A) \subset Var(B))$

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 $\not\equiv A \in R_1 : (A_1 < A)$,
- 3. A_2 deve essere massimale in R_2 ovvero $\not\equiv A \in R_2 : (A_2 \prec A)$,
- 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 $\not\equiv A \in R : (A_1 < A)$,
- 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 < poiché preserva la proprietà degli <-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 <-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 *<-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 tramite *<-ordered resolution* e *factorization*.

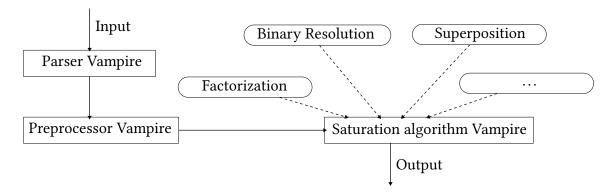


Figura 3.1: Struttura di VAMPIRE prima delle modifiche

L'idea per l'implementazione della procedura è:

- 1. costruire un nuovo *preprocessor* che trasformi le formule *guarded* in clausole e sostituirlo nel sistema
- 2. implementare una nuova inferenza (<-ordered resolution) e "iniettarla" nell'algoritmo di saturazione
- 3. eliminare tutte le inferenze non necessarie dall'algoritmo di saturazione

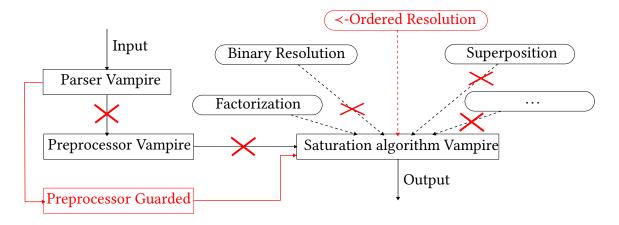


Figura 3.2: Struttura di VAMPIRE dopo le modifiche

3.1 Preprocessing

A proposito della fase di *preprocessing*, è stato sviluppato un *preprocessor* per trattare problemi appartenenti al frammento *guarded*. Questa componente è stata implementata seguendo l'architettura del *preprocessor* predefinito di VAMPIRE.

Algoritmo 3 Preprocessing

```
var problem : sets of guarded formula
var result : sets of guarded clause
problem = NNF(problem)
problem = flatten(problem)
problem = Structy(problem)
problem = skolemize(problem)
result = clausify(problem)
computeVarDepth(result)
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 le funzioni $Struct_\forall$ e computeVarDepth vengono implementate ex-novo in quanto non presenti nel sistema.

Structy è una funzione ricorsiva per sfruttare la struttura ad albero delle formule.

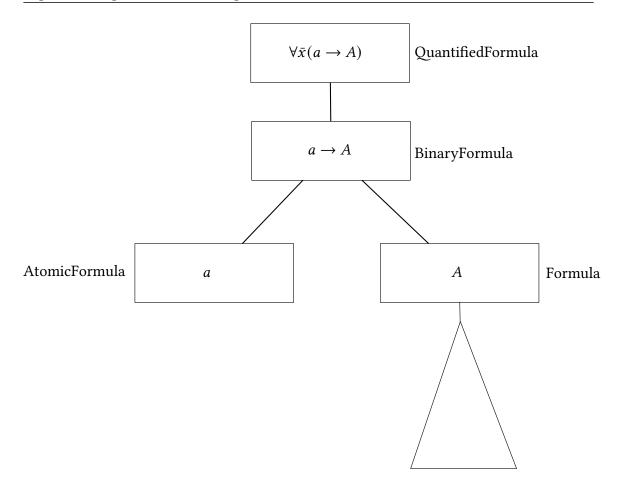


Figura 3.3: Struttura ad albero della formula

L'intero albero della formula viene visitato tramite una visita post-order. Durante la visita l'algoritmo controlla se nell'intero albero sono presenti formule del tipo $\forall \bar{x} (a \lor A)$ con almeno una variabile libera. Nel caso venga trovata una formula in questa forma, viene applicata la trasformazione strutturale $Struct_{\forall}$.

Algoritmo 4 Visita post-order

```
function PostOrderVisit(input : formula)
   case connective of input
       switch \top, \bot, Literal do
           return input
       switch ¬,∃ do
           return PostOrderVisit(subformula of input)
       switch \rightarrow, \leftrightarrow do
           var left, right : formula
           left = PostOrderVisit(left subformula of input)
           right = PostOrderVisit(right subformula of input)
           return input
       switch \land, \lor do
           for all subformula in input do
               subformula = PostOrderVisit(subformula)
           end for
           return input
       switch ∀ do
           var subformula: formula
           subformula = PostOrderVisit(subformula of input)
           if (connective of subformula is \vee) \wedge (hasFreeVar(input)) then
               input=Struct_{\forall}(input)
           end if
           return input
end function
```

Per l'implementazione della trasformazione strutturale, si segue la definizione 14.

Algoritmo 5 Struct∀

```
function STRUCT_{\forall} (input : formula)

var new, \alpha : formula

var \bar{y} : sets of free variables

\bar{y} = saveFreeVariable (input)

\alpha = createNewLiteral(\bar{y})

new = generateNewFormula(alpha,input)

addFormulaToUnits (new)

input = \alpha

return input

end function
```

In primis viene creato un nuovo letterale/formula atomica α con le variabili libere \bar{y} tramite la funzione *createNewLiteral*() e, subito dopo, la formula in input viene sostituita da α . Successivamente viene generata una nuova formula del tipo $\forall \bar{x}\bar{y}(\neg a \lor \neg \alpha \lor A)$ tramite la funzione *generateNewFormula*() e viene aggiunta al problema con la funzione *addFormulaToUnits*().

La funzione *computeVarDepth*() calcola il parametro *VarDepth* (definizione 15) di ogni letterale presente nelle clausole risultanti. Per salvare il risultato è stata modificata la classe Literal ed è stato aggiunto un attributo privato *varDepth*. La visita dei termini per il calcolo della *VarDepth* è stato semplificato grazie a un iteratore già implementato in VAMPIRE. *computeVarDepth*() è diviso in due componenti:

- la prima funzione scorre i termini più esterni del letterale (termini rossi nella figura 3.4) tramite un SubtermIterator
- 2. la seconda funzione va in profondità e scorre i sottotermini (termini verdi nella figura 3.4) tramite un altro SubtermIterator

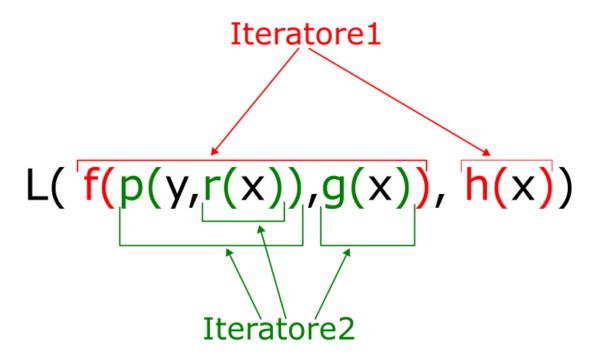


Figura 3.4: Uso degli iteratori per il calcolo della VarDepth

In questo caso la prima funzione è iterativa, mentre la seconda è ricorsiva per sfruttare la struttura del letterale.

Algoritmo 6 Prima funzione che scorre i termini esterni del letterale

```
function ComputeVarDepth1(lit: literal)
   if literal is ground then
       setLiteralVarDepth(-1, lit)
       return
   end if
   var max, varDepth : int
   max = 0
   for all term in lit do
                                             ▶ Iteratore1 qui scorre i termini esterni
       varDepth = 0
       varDepth = 1 + ComputeVarDepth2(term)
       if max < varDepth then
          max = varDepth
       end if
   end for
   setLiteralVarDepth(max, lit)
end function
```

Se il letterale è $ground^1$, la VarDepth di lit viene impostata a -1 tramite la funzione setLiteralVarDepth(). Nel caso non sia ground, allora viene calcolata la massima profondità di ciascun termine esterno tramite la seconda funzione ComputeVarDepth2(). Alla fine il risultato sarà la massima profondità calcolata tra tutti i termini.

Algoritmo 7 Seconda funzione che scorre i sottotermini in profondità

¹è un letterale privo di variabili

3.2 Resolution

La nuova inferenza implementata, <-ordered resolution, eredita la classe GeneratingInferenceEngine (figura 1.5), in modo da avere tutti i metodi necessari per essere inserita nell'algoritmo di saturazione e generare nuove clausole. La funzione principale di questa classe è generateClause() che sfrutta le varie strutture dati messe a disposizione da VAMPIRE.

Il metodo viene eseguito dopo la selezione di una clausola e la conseguente attivazione. La struttura dati fondamentale è l'indice che tiene traccia di tutte le clausole e permette di trovarne una da unificare con quella selezionata. Sia $\{A_1\} \cup R_1$ clausola selezionata, allora il risultato della query sull'indice viene salvato in un altra struttura dati che comprende:

- una clausola $\{\neg A_2\} \cup R_2$ candidata a unificare con la clausola selezionata
- un letterale $\{\neg A_2\}$ candidato all'unificazione con uno dei letterali della clausola selezionata (in questo caso $\{A_1\}$)

Algoritmo 8 Funzione generateClause()

```
var selectedLiteral, candidateLiteral : literal
var selectedClause, candidateClause, result : clause
if IsMaximal(selectedLiteral, selectedClause) \wedge IsMaximal(candidateLiteral) then
    result = unify(selectedClause, candidateClause)
    return result
else
    return NULL
end if
```

Sia < ordinamento definito come nella definizione 19, la funzione generateClause() verifica se il letterale $\{A_1\}$ in $\{A_1\} \cup R_1$ e il letterale $\{\neg A_2\}$ in $\{\neg A_2\} \cup R_2$ sono massimali (secondo l'ordinamento <) tramite la funzione IsMaximal() . Nel caso siano massimali, allora viene applicata la resolution e l'unificazione tramite la funzione unify().

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

In caso contrario non viene generata alcuna clausola.

Algoritmo 9 Funzione che controlla se un letterale è il massimale in una clausola

```
function IsMaximal(maxLit : literal, cl : clause)
  for all lit in cl do
    varDepth = getVarDepth(lit)
    if (max<varDepth) ∧ (Var(lit)⊈Var(maxLit)) then
        return ⊥
    end if
  end for
    return ⊤
end function</pre>
```

Seguendo la definizione 19, la funzione IsMaximal() controlla se un letterale *maxLit* è massimale nella clausola ovvero:

- se la *VarDepth* di *maxLit* è maggiore o uguale di tutte le *VarDepth* degli altri letterali nella clausola, o
- se l'insieme delle variabili di *maxLit* contiene l'insieme delle variabili degli altri letterali nella clausola

La funzione restituisce \bot se entrambe le condizioni non sono rispettate.

3.3 Classificazione

A fini sperimentali, viene implementato un classificatore per riconoscere, tra un insieme di problemi casuali, quali appartengono al frammento *guarded*. Il classificatore viene richiamato dopo il *parser* di VAMPIRE ed è composto da due funzioni principali: una per analizzare le formule generiche della logica del primo ordine e un'altra per analizzare clausole.

La prima funzione sfrutta, come il *preprocessor*, la struttura ad albero della formula ed è dunque ricorsiva.

Algoritmo 10 Visita dell'albero della formula parte 1

```
1: function Visit(input : formula)
 2:
        case connective of input
 3:
            switch \top, \bot, Literal do
 4:
                return ⊤
            switch ¬ do
 5:
                return Visit(subformula of input)
 6:
 7:
            switch \rightarrow, \leftrightarrow do
                var left, right: formula
 8:
                left = Visit(left subformula of input)
 9:
                if left = \top then
10:
                    return Visit(right subformula of input)
11:
                end if
12:
13:
                return \( \perp \)
            switch \land, \lor do
14:
                for all subformula in input do
15:
                    subformula = Visit(subformula)
16:
                   if subformula \neq \top then
17:
                        return 1
18:
                    end if
19:
                end for
20:
                return ⊤
21:
            switch ∃ do
22:
                var subformula: formula
23:
                var vars : sets of bounded variables
24:
                var lit: literal
25:
                var guardFound : bool
26:
                subformula = subformula of input
27:
                vars = getBoundedVariables(input)
28:
                if connective of subformula is \land then
29:
30:
                    guardFound = \bot
                   while sub-subformula in subformula \land guardFound = \bot do
31:
                        if sub-subformula is Literal then
32:
                            lit = literal of sub-subformula
33:
                            if isPositive(lit) \land isGuard(lit, vars) then
34:
                                guardFound = T
35:
                            end if
36:
                        end if
37:
                   end while
38:
                   if guardFound == \top then
39:
                        return Visit(subformula)
40:
                    else
41:
                        return 1
42:
                    end if
43:
                else
44:
45:
                    return ⊥
                end if
46:
```

Algoritmo 11 Visita dell'albero della formula parte 2

```
switch ∀ do
47:
               var subformula: formula
48:
               var vars: sets of bounded variables
49:
               var lit: literal
50:
               var guardFound : bool
51:
               subformula = getSubformula(input)
52:
               vars = getBoundedVariables(input)
53:
               left = getLeft(subformula)
54:
               if (connective of subformula is \rightarrow) \land (left is Literal) then
55:
                   lit = literal of left
56:
                   if isPositive(lit) ∧ isGuard(lit, vars) then
57:
                       return Visit(right subformula of subformula)
58:
                   else
59:
60:
                       return ⊥
                   end if
61:
               else if connective of subformula is ∨ then
62:
                   guardFound = \perp
63:
                   while subformula in input \land guardFound = \bot do
64:
                       if subformula is Literal then
65:
                           lit = literal of subformula
66:
                           if isNegative(lit) ∧ isGuard(lit, vars) then
67:
                               guardFound = \top
68:
                           end if
69:
                       end if
70:
                   end while
71:
                   if guardFound = \top then
72:
                       return Visit(subformula)
73:
                   else
74:
                       return ⊥
75:
                   end if
76:
               end if
77:
78:
               return \perp
79: end function
```

Durante la visita l'algoritmo controlla se nell'intero albero sono presenti formule del tipo:

- $\forall \bar{x}(a \to A)$ (riga 55),
- $\forall \bar{x} (a \lor A)$ (riga 62),
- $\exists \bar{x}(a \land A) \text{ (riga 29)}$

Nel primo caso verifica che *a* sia un letterale positivo e una guardia tramite le funzioni *isPositive*() e *isGuard*() (riga 57). Nel secondo e nell'ultimo caso si scorrono tutte le sottoformule per trovare, rispettivamente, una guardia negativa (riga 67) e una guardia positiva (riga 34).

Algoritmo 12 Funzione che verifica se un letterale è una guardia

```
function IsGuard(lit: literal, vars: sets of variables)
   var literalVars : sets of variables
   literalVars = getVariablesOf(lit)
   for all variable in vars do
       var foundVariable = \bot
       while litVar in literalVars \land foundVariable == \bot do
           if litVar == variable then
               foundVariable = ⊤
           end if
       end while
       if foundVariable == \bot then
           return ⊥
       end if
   end for
   return ⊤
end function
```

La funzione IsGuard() verifica se un letterale è una guardia, ovvero se contiene almeno una volta le variabili libere *vars* delle altre sottoformule.

Algoritmo 13 Funzione che verifica se una clausola è guarded

```
var cl : clause
var vars : clause variables set
var checkGuard: bool
checkGuard = ⊥
for all lit in cl do
   if isNegative(lit) \land foundVariable == \bot then
       checkGuard = IsGuard(lit, vars)
       if checkGuard == \top then
          continue
       end if
   end if
   for all term in lit do
       if ¬containsAllVariablesOfClause(term,vars) then
       end if
   end for
end for
return checkGuard
```

La seconda funzione che analizza le clausole verifica se esiste una guardia negativa (riga 6), tramite la funzione IsGuard(), e se i termini funzionali non ground contengono tutte le variabili della clausola (righe 12, 13).

4. Risultati sperimentali

In questo capitolo vengono presentati i risultati sperimentali del confronto tra VAMPIRE originale e VAMPIRE con la nuova procedura di decisione.

Per gli esperimenti sono stati utilizzati i problemi della libreria TPTP versione 8.2.0 [6]. I problemi presi in esame sono privi di uguaglianza, poiché la procedura di decisione tratta solo questo tipo di problemi. É possibile dividere i problemi in due insiemi: uno comprende quelli composti da formule generiche di logica del primo ordine (FOF), mentre l'altro comprende i problemi composti da clausole (CNF).

Tabella 4.1: Numero di problemi

	FOF	CNF
Senza equality	1969	2274
Guarded	52	62

Entrambi gli insiemi sono stati dati in input al classificatore (capitolo 3.3) e il risultato è quello riportato in tabella 4.1: 52 problemi di tipo FOF su 1969 e 62 problemi di tipo CNF su 2274 sono *guarded*. 7 di questi 52 problemi di tipo FOF non vengono presi in considerazione nei grafici seguenti, in quanto non possono essere risolti in un tempo limite stabilito. Di seguito viene fornita una numerazione dei problemi FOF e CNF in modo da rendere più chiara la lettura dei grafici.

Tabella 4.2: Numerazione dei problemi FOF

Problema	N.	Problema	N.	Problema	N.
LCL181+1	1	SWB002+2	16	SYN390+1	31
LCL230+1	2	SWV011+1	17	SYN391+1	32
LCL637+1.001	3	SYN001+1	18	SYN392+1	33
LCL637+1.005	4	SYN007+1.014	19	SYN393+1.003	34
MED011+1	5	SYN040+1	20	SYN406+1	35
NLP263+1	6	SYN041+1	21	SYN407+1	36
PHI014+1	7	SYN044+1	22	SYN416+1	37
PUZ005+1	8	SYN045+1	23	SYN915+1	38
PUZ068+2	9	SYN046+1	24	SYN916+1	39
PUZ069+2	10	SYN047+1	25	SYN942+1	40
PUZ079+2	11	SYN068+1	26	SYN973+1	41
PUZ080+2	12	SYN356+1	27	SYN977+1	42
PUZ128+1	13	SYN387+1	28	SYN978+1	43
PUZ138+2	14	SYN388+1	29	SYO525+1.015	44
SWB001+2	15	SYN389+1	30	SYO525+1.018	45

Tabella 4.3: Numerazione dei problemi CNF

							Problema	N.
Problema	N	Problema	N.		Problema	N.	SYN085-1.010	46
GRA001-1	1	PUZ013-1	16		SYN010-1.005.005	31	SYN086-1.003	47
HWV003-3	2	PUZ014-1	17		SYN011-1	32	SYN087-1.003	48
KRS004-1	3	PUZ015-2.006	18		SYN028-1	33	SYN089-1.002	49
LAT260-2	4	PUZ016-2.004	19		SYN029-1	34	SYN090-1.008	50
LAT261-2	5	PUZ016-2.005	20		SYN030-1	35	SYN091-1.003	51
LAT264-2	6	PUZ028-3	21		SYN032-1	36	SYN092-1.003	52
LAT265-2	7	PUZ028-4	22		SYN040-1	37	SYN093-1.002	53
LAT267-2	8	PUZ030-2	23		SYN041-1	38	SYN094-1.005	54
LCL181-2	9	PUZ033-1	24		SYN044-1	39	SYN095-1.002	55
LCL230-2	10	PUZ036-1.005	25		SYN045-1	40	SYN096-1.008	56
MSC007-1.008	11	SET856-2	26		SYN046-1	41	SYN097-1.002	57
NUM285-1	12	SYN001-1.005	27		SYN047-1	42	SYN098-1.002	58
PUZ004-1	13	SYN003-1.006	28		SYN054-1	43	SYN302-1.003	59
PUZ008-2	14	SYN004-1.007	29		SYN062-1	44	SYN724-1	60
PUZ009-1	15	SYN008-1	30		SYN063-2	45	SYN915-1	61
\				•			SYN916-1	62

Nei test, Vampire con la nuova procedura di decisione è indicato con Guarded. I primi test sono stati eseguiti con le seguenti opzioni di Vampire:

--sa otter -t 10m -m 8000

L'algoritmo di saturazione scelto è *otter*, il tempo massimo di timeout è 10 minuti e il limite massimo di memoria occupabile è 8000 Mb.

4.1 Problemi FOF

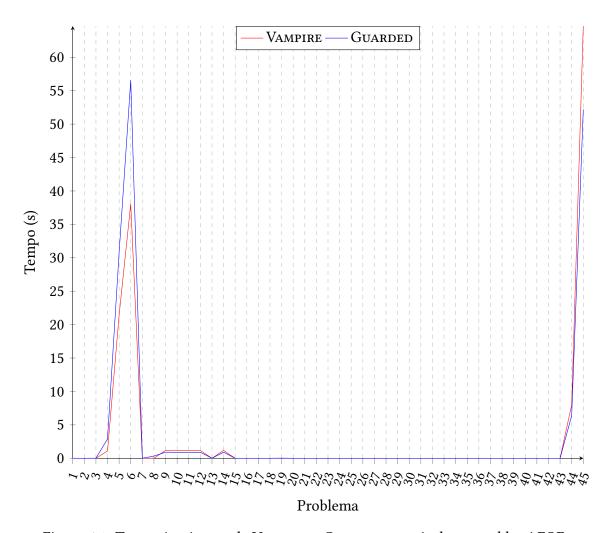


Figura 4.1: Tempo impiegato da VAMPIRE e GUARDED per risolvere problemi FOF

Dai primi test sui problemi FOF emerge che, nella maggior parte dei problemi, GUARDED è efficiente tanto quanto VAMPIRE in termini di tempo. La differenza tra i due, essendo nell'ordine dei millisecondi, non è significativa. Ci sono alcune eccezioni: i problemi MED011+1 (n.5) e NLP263+1 (n.6) sono a favore di VAMPIRE rispettivamente di circa 10 e 18 secondi, mentre il problema SYO525+1.018 (n.45) è a favore di GUARDED di circa 12 secondi.

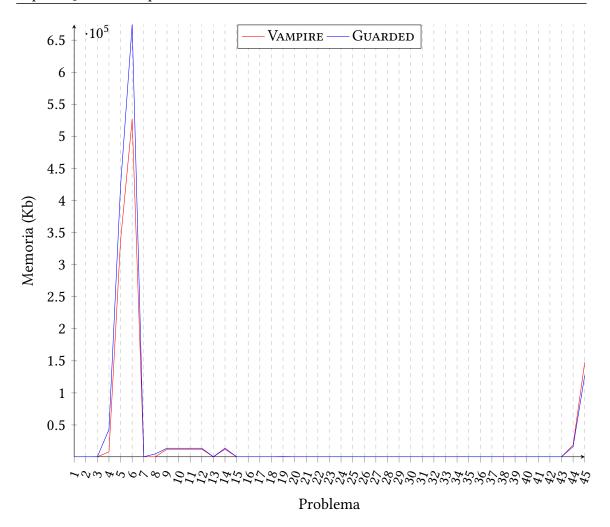


Figura 4.2: Memoria occupata da VAMPIRE e GUARDED per risolvere problemi FOF

Vampire e Guarded sono molto simili anche in termini di memoria occupata per la risoluzione di un problema. L' unica eccezione rilevante è il problema NLP263+1 (n.6) in cui Guarded occupa 6.7×10^5 Kb di memoria, mentre Vampire solo 5.2×10^5 Kb.

Alla fine di questa batteria di test, l'analisi ha posto particolare attenzione alle eccezioni.

Memoria: La tendenza di Guarded ad occupare più memoria in alcuni problemi proseguirà anche nei prossimi test poiché, in problemi complessi e di grandi dimensioni, la trasformazione strutturale $Struct_\forall$ aggiunge nuove formule del tipo $\forall \bar{x}\bar{y}(\neg a \lor \neg \alpha \lor A)$ quindi occupa molto più spazio rispetto a Vampire.

Tempo: Nelle eccezioni in cui VAMPIRE impiega meno tempo, è utilizzata una tecnica di *preprocessing* detta *unused predicate definition removal.* Una definizione di un predicato è una formula del tipo

$$\forall (X_1,\ldots,X_n): p(X_1,\ldots,X_n) \leftrightarrow F$$

in cui p predicato non risulta in F. Con questa tecnica VAMPIRE si può comportare in 3 modi:

- 1. se p è presente nel resto del problema solo in forma positiva, allora \leftrightarrow della definizione viene sostituito da \rightarrow ;
- 2. se *p* non è presente nel resto del problema, allora la definizione può essere eliminata;
- 3. altrimenti la definizione non viene modificata

In tal modo VAMPIRE riesce a risolvere i problemi FOF in modo più rapido.

Nei prossimi test, la tecnica di *unused predicate definition removal* (updr) verrà disattivata da VAMPIRE infatti le opzioni saranno:

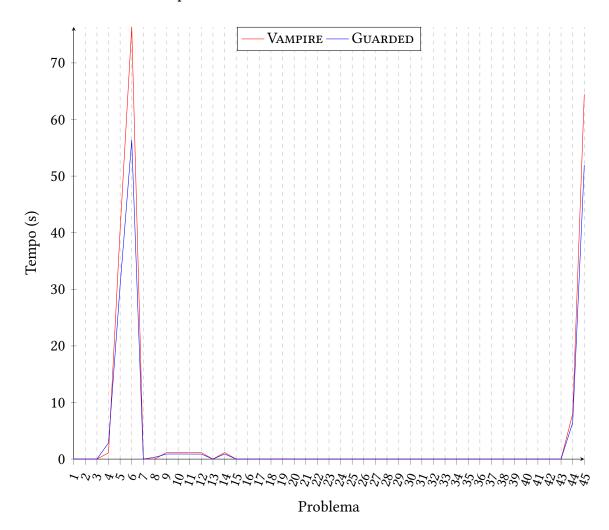


Figura 4.3: Tempo impiegato da Vampire e Guarded per risolvere problemi FOF con updr disattivata

Con la disattivazione della *updr*, le prestazioni di Vampire peggiorano significativamente. Infatti, in questo modo Vampire impiega più tempo di Guarded, con una differenza di circa 10 secondi nel problema MED011+1 (n.5) e di circa 20 secondi in NLP263+1 (n.6). In questi test Vampire impiega circa 20 secondi in più nel problema MED011+1 (n.5) e circa 38 secondi in più nel problema NLP263+1 (n.6) rispetto alla batteria di test precedente.

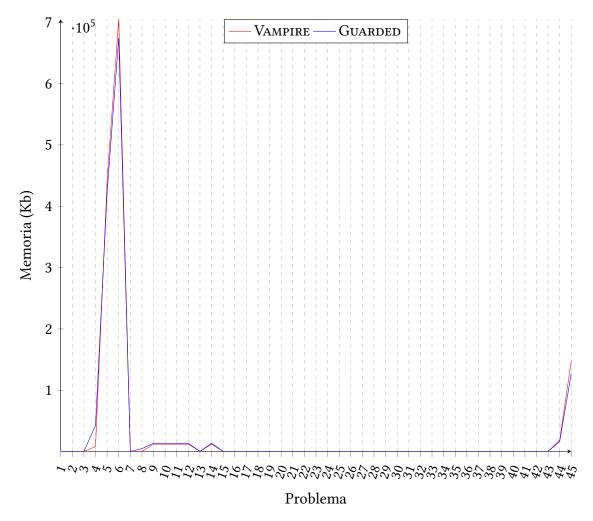


Figura 4.4: Memoria occupata da Vampire e Guarded per risolvere problemi FOF con updr disattivata

Disattivando l'*updr*, la quantità di memoria occupata da VAMPIRE per il problema NLP263+1 aumenta al punto da superare quella utilizzata da GUARDED. Nonostante ciò, la differenza non è da considerare significativa.

4.2 Problemi CNF

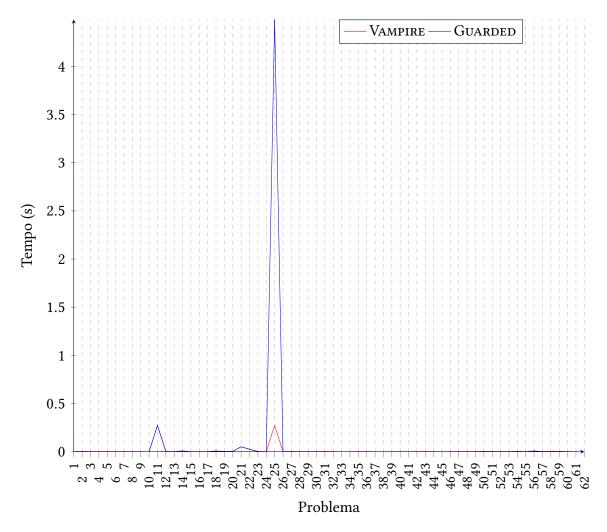


Figura 4.5: Tempo impiegato da VAMPIRE e GUARDED per risolvere problemi CNF

Dai test sui problemi CNF emergono gli stessi risultati dei precedenti test sui problemi FOF: nella maggior parte dei problemi, Guarded è efficiente quanto Vampire, sia in termini di tempo che di memoria. In questo caso, la differenza è ancora minore rispetto ai problemi FOF. L'unica eccezione è il problema PUZ036-1.005 (n.25) in cui Vampire impiega circa 0,2 secondi e occupa 1100 Kb, mentre Guarded circa 4 secondi e 15000 Kb.

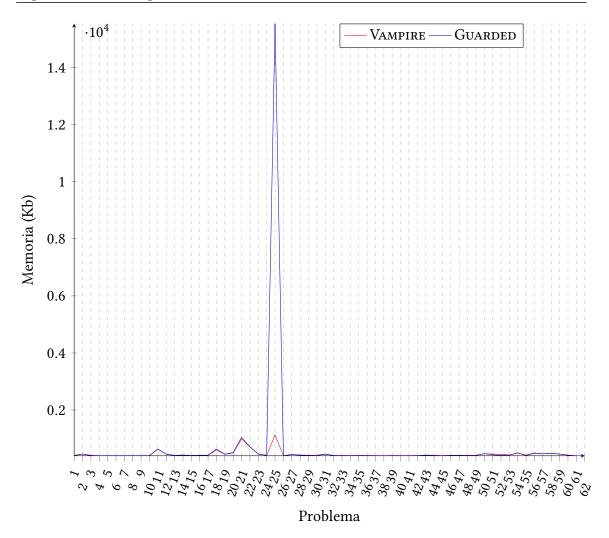


Figura 4.6: Memoria occupata da VAMPIRE e GUARDED per risolvere problemi CNF

Alla fine di questi primi test sui problemi CNF, è stata effettuata l'analisi sul problema PUZ036-1.005 (n.25). Vampire, nell'algoritmo di saturazione, utilizza la *forward subsumption*, un'inferenza che fa parte delle *forward simplification* (capitolo 1.1.3) utile per eliminare le ridondanze. Questa tecnica permette a Vampire di essere più veloce ed efficiente.

Nei prossima batteria di test, la tecnica di *forward subsumption* (fs) verrà disattivata da VAMPIRE infatti le opzioni saranno:

Si noti che i test sono stati effettuati anche con updr disattivata, ma non stati riportati i risultati poiché non c'era nessun cambiamento significativo. Lo stesso vale anche per l'opzione fs nei problemi FOF. Infatti, su questi problemi, sono stati eseguiti i test anche con fs disattivata ma i risultati non hanno portato a nessun cambiamento rilevante.

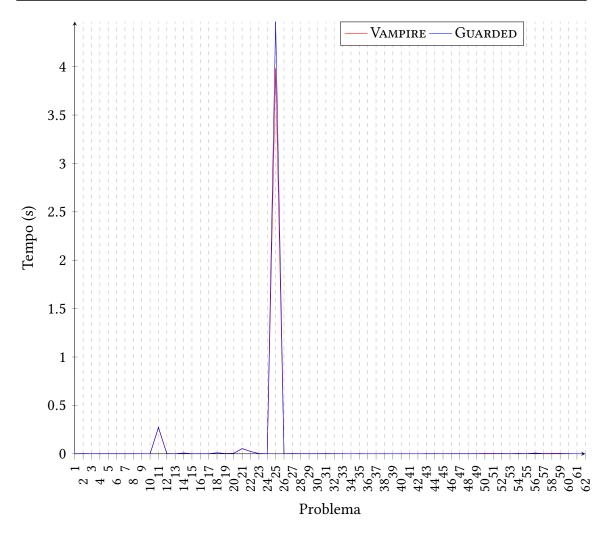


Figura 4.7: Tempo impiegato da VAMPIRE e GUARDED per risolvere problemi CNF con *forward subsumption* disattivata per entrambi

Con la disattivazione della *forward subsumption*, Vampire peggiora sia in termini di tempo che di memoria occupata nel problema PUZ036-1.005 (n.25). La differenza di tempo tra Vampire e Guarded si assottiglia molto in questo problema, infatti è di circa mezzo secondo.

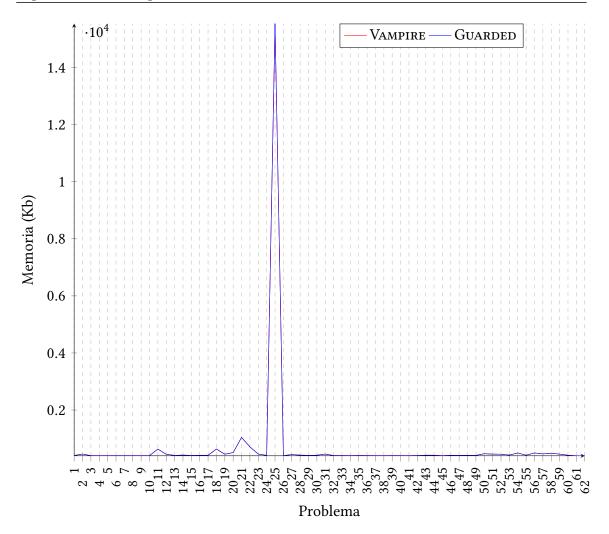


Figura 4.8: Memoria occupata da VAMPIRE e GUARDED per risolvere problemi CNF con *forward subsumption* disattivata per entrambi

Con la disattivazione della *forward subsumption*, VAMPIRE e GUARDED occupano circa la stessa quantità di memoria.

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 VAM-PIRE». 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.
- [6] G. Sutcliffe. «The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0». In: *Journal of Automated Reasoning* 59.4 (2017), pp. 483–502.