Capitolo 1

Il Theorem prover Vampire

Vampire [**?**] [**?**] [**?**] è un dimostratore di teoremi automatico per la logica del primo ordine basato su *Resolution*. Nasce nel 1998 come progetto di ricerca degli autori Andrei Voronkov e Alexandre Riazanov, adesso è correntemente mantenuto e sviluppato da un team più ampio presso il dipartimento di Computer Science dell'Università di Manchester. Il software è open-source, sviluppato in C++ e al momento della scrittura di questa tesi è giunto alla versione 4.8 con licenza BSD-3. Vampire incorpora un complesso sistema strutture dati, algoritmi per la manipolazione di formule e termini e un vasto sistema di inferenze. Uno dei suoi punti di forza è l'efficienza, Il team di sviluppo infatti partecipa annualmente al *CASC* (The CADE ATP System Competition), una competizione tra sistemi ATP, e fino ad ora ha sempre vinto almeno in una categoria ogni anno. Questa ambizione per l'efficienza ha influenzato molto la struttura di Vampire e la sua implementazione. Questo è sia un lato positivo che negativo, infatti se da un lato ci si ritrova con funzioni efficienti e ben ottimizzate, dall'altro lato ci si ritrova spesso con un codice complesso e difficile da comprendere che predilige la velocità alla pulizia. Ogni suo componente è riconducibile ad un articolo che ne spiega il funzionamento ad alto livello ma spesso, alcune scelte implementative sono poco o per nulla documentate. Spesso lo stesso nome di una funzione o di una classe fa intuire il suo scopo e funzionamento ma non sempre è così e altrettanto spesso si è costretti a fare 'Reverse Engineering' del codice sorgente per capire come è stato utilizzato in altri contesti. Questo è un problema di cui il team di sviluppo è consapevole e negli ultimi anni sta cercando di migliorare. In questo capitolo si cercherà di dare una panoramica generale di Vampire,

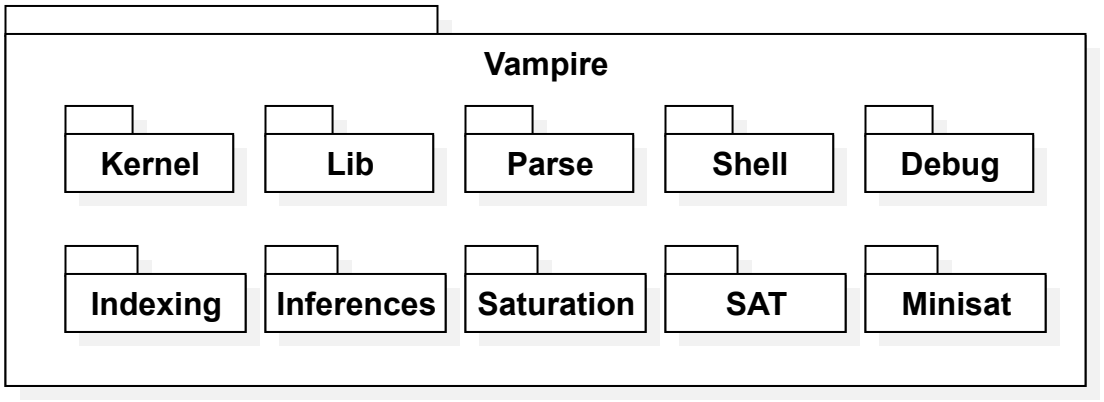

Figura 1.1: Struttura delle cartelle di Vampire.

spiegando le sue componenti principali e come queste interagiscono tra di loro, con un focus particolare

su quelle che sono state utilizzate per la realizzazione della procedura di decisione per frammenti Binding. Nella figura 1.1 è mostrata la disposizione delle cartelle di Vampire. La struttura è molto piatta ma assolutamente organizzata. Nella cartella *Kernel* sono presenti le componenti principali del sistema come ad esempio le strutture per le formule e i termini, e il 'Main Loop' del programma che si occupa di gestire il processso di dimostrazione. La struttura delle formule verrà trattata nella sezione 1.1. Nella cartella *Lib* sono presenti le strutture dati e le funzioni di utilità come Array, Mappe, Liste, Stack, ecc. Nella cartella *Parse* sono presenti le classi che decodificano i file in formato TPTP o SMT. Nella cartella *Shell* sono presenti le classi per la gestione dell'input/output da riga di comando e tutte le funzioni necessarie per il Preprocessing. Gli step del preprocessing verranno approfonditi nella sezione 1.3. Nella cartella *Indexing* sono presenti i componenti per l'indicizzazione dei termini. Le particolari strutture per l'unificazione verranno trattate nella sezione 1.5. Nelle cartelle *Inferences* e *Saturation* sono presenti le classi che contengono le regole di inferenza e gli algoritmi di saturazione. Questi verranno trattati nelle sezioni 1.4 e 1.6. Nelle cartelle *SAT* e *Minisat* sono presenti le interfacce per utilizzare i SAT-Solver e il codice di Minisat, un SAT-Solver open-source. Il funzionamento dei sat solver verrà discusso nella sezione 1.6. Nella cartella *Debug* sono presenti le classi e le macro per la misurazione dei tempi e le statistiche di esecuzione. Alcuni esempi verranno mostrati nella sezione 1.7.

1.1 I Termini

I termini, insieme a clausole e formule, sono la struttura dati più importante in un dimostratore di teoremi ed è quindi fondamentale che siano rappresentati nel modo più efficiente possibile. Nella figura 1.2 è mostrata una rappresentazione ad alto livello e molto semplificata della struttura dei termini implementata in Vampire. Un termine come inteso nella sezione **??** è rappresentata dalla classe *TermList*. TermList è composto da tre elementi principali: *term*, *content* e *info*. I tre componenti sono definiti all'interno di una **union** per risparmiare memoria.

- *term* è un puntatore ad un oggetto della classe *Term*
- content è un intero di 64 bit
- info è una struttura BitField di esattamente 64 bit

Essendo definiti all'interno di una union, ogni TermList dovrebbe occupare esattamente 64 bit di memoria. In vampire ogni variabile è rappresentata da un numero intero senza segno mentre i termini complessi composti da funzioni sono rappresentati dalla classe Term. Se TermList rappresenta una variabile allora content shiftato di 2 bit verso destra rappresenta l'indice di quella variabile ($content/4$), nel caso rappresenti una funzione allora term punta ad un oggetto di tipo Term che contiene l'effettiva struttura della funzione. Nella classe Term il nome della funzione è rappresentata da un intero senza segno globalmente univoco definito nella classe *Signature*. La classe Signature contiene le informazioni relative all'indice, arità e nome di funzioni e predicati. Term inoltre contiene un Array di TermList di lunghezza pari ad $arity + 1$ che rappresenta gli argomenti della funzione listati da destra verso sinistra. L'elemento in posizione 0 contiene un Termlist fittizio che contiene le info dello stesso termine.

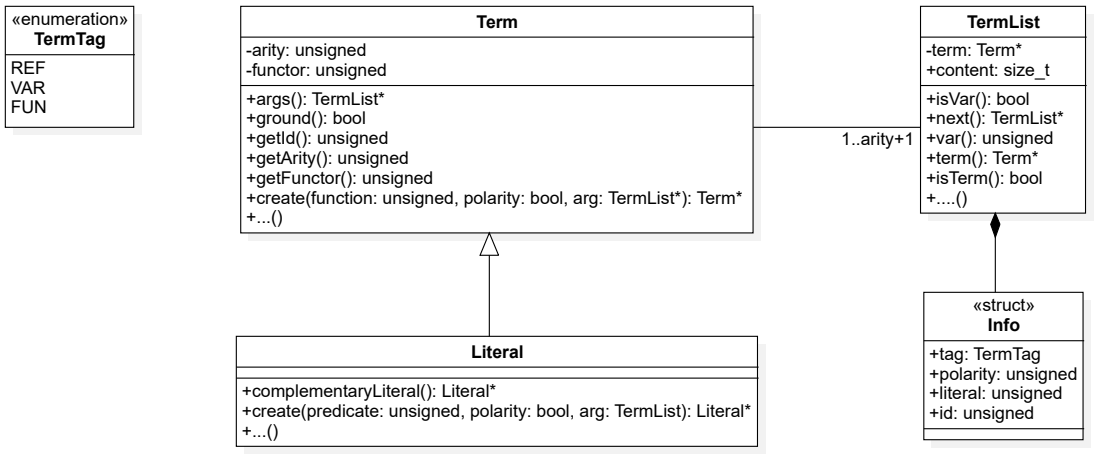

Figura 1.2: Struttura dei termini

Tutti i termini di default sono rappresentati da una struttura Perfectly Shared (come descritto in **??**) per risparmiare memoria e velocizzare le operazioni di confronto. I letterali sono rappresentati dalla classe *Literal* che è una specializzazione della classe *Term*. Nell'implementazione Vampire non fa nessuna distinzione tra nomi di funzione funzioni o predicati essi sono infatti rappresentati entrambi nella Signature come funzioni. Termini e Letterali sono salvati nella Signature in strutture di indicizzazione (SubstitutionTree) per permettere un accesso veloce. Un accenno a queste strutture verrà fatto nella sezione 1.5. Literal contiene inoltre funzioni specifiche per la manipolazione dei letterali, come *complementaryLiteral* che restituisce lo stesso letterale negato (dalla struttura di indicizzazione se presente altrimenti ne crea uno nuovo). Le funzioni Term::create e Literal::create sono le funzioni utilizzate per creare nuovi termini e letterali e inserirli nella struttura di indicizzazione.

Ad esempio, il predicato $\neg p_1(f_2, f_3(x_5), x_5, f_3(x_5))$ viene rappresentato in memoria con una struttura del genere:

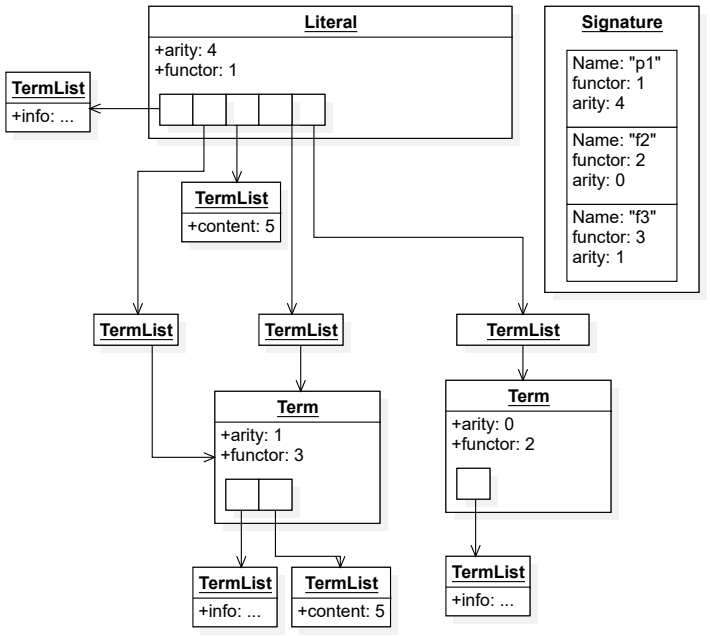

Figura 1.3: Esempio di rappresentazione di un termine

1.2 Unità, Formule e Clausole

Vampire prende in input formule del tipo $A_1 \wedge A_2 \wedge ... \wedge A_n \wedge \neg C$, dove $A_1, A_2, ..., A_n$ sono assiomi e C è la congettura, e cerca di dimostrarne l'insoddisfacibilità. Per fare ciò il problema principale viene scomposto in una lista di elementi chiamati *Unità*. Un unità è una formula o una clausola affiancata da una regola di inferenza che lo ha generata. In sostanza vi sono due tipi di inferenze, quelle che rappresentano unità date in input come *Axiom* per indicare che l'unità è un assioma in input o *Negated Conjecture* per indicare che l'unità è la negazione della congetture e quelle che rappresentano altre formule/clausole generate all'interno del processo dimostrativo. Le inferenze di questo tipo includono anche una reference alle formule che hanno generato la nuova unità, rendendo quindi possibile risalire alla dimostrazione al termine dell'esecuzione.

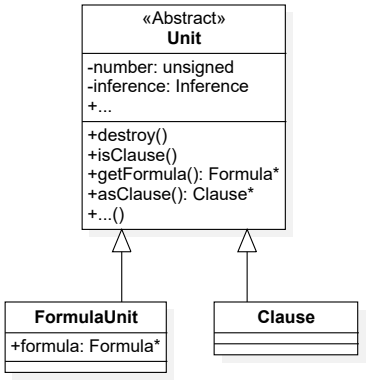

Figura 1.4: Struttura delle unità

Le unità come mostrato in figura 1.4 sono rappresentate dalla classe astratta *Unit*, che si specializza

nelle classi *FormulaUnit* che contiene un puntatore ad un oggetto di tipo *Formula* e *Clause* che verrà trattata in seguito. Le formule sono rappresentate da una struttura ad albero esattamente come quelle vista in **??** e **??**. La classe *Formula* 1.5 è una classe astratta che si specializza nelle classi:

- *AtomicFormula* che rappresenta una formula composta da un solo letterale.
- *BinaryFormula* rappresenta le formule binarie $A \Rightarrow B$, $A \Leftrightarrow B$ e $A \oplus B$.
- *NegatedFormula* rappresenta le formule negate del tipo $\neg A$.
- *QuantifiedFormula* rappresenta le formule quantificate del tipo $\forall / \exists x1, x2, ..., xn : A$.
- *JunctionFormula* rappresenta le formule composte dalla concatenazione di $\wedge$ e $\vee$.

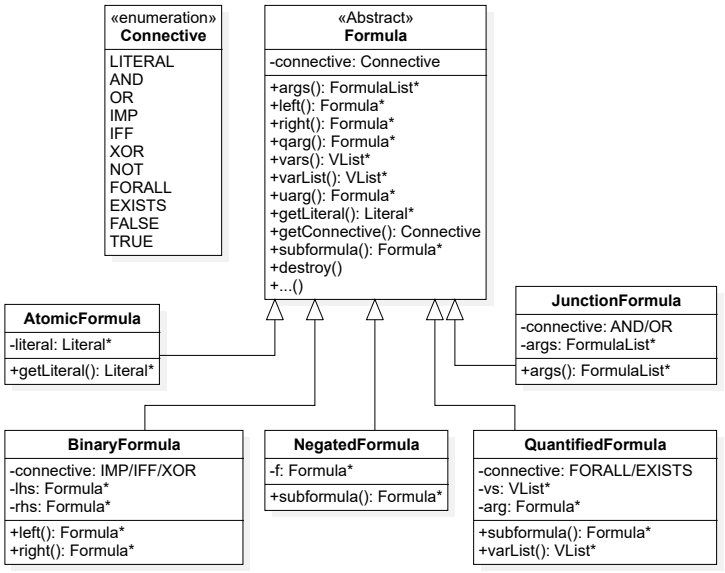

Figura 1.5: Struttura delle formule

Le clausole sono rappresentate dalla classe *Clause* 1.6 che è una specializza della classe *Unit*. Ogni clausola contiene un Array di letterali e sono quindi rappresentate in maniera molto simile alla notazione insiemistica vista in **??**.

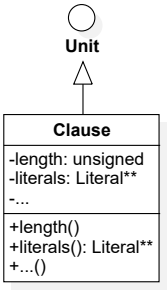

Figura 1.6: Struttura delle Clausole

1.3 Preprocessing

Vampire è in grado di elaborare formule del primo ordine in qualsiasi forma, tuttavia, poiché il suo algoritmo di saturazione opera esclusivamente su clausole, è necessario eseguire delle operazioni di trasformazioni sull'input. L'insieme di queste operazioni è chiamato *Preprocessing*. Il preprocessing non ha solo lo scopo di convertire la formula in input in una equisoddisfacibile in forma clausale ma ha anche l'obbiettivo di rendere il problema più semplice (se possibile). In generale tutte le operazioni non superano la complessità di $O(n \cdot log(n))$ e le operazioni non essenziali sono attivabili/disattivabili tramite opzioni da riga di comando. Di seguito sono riportate le principali operazioni di preprocessing eseguite da Vampire:

1. ***Rectify***: Verifica se la formula contiene variabili libere e in caso affermativo le quantifica e verifica che la stessa variabile non sia quantificata più volte nello stesso ramo dell'albero sintattico.
2. ***Simplify***: Semplifica e Rimuove le occorrenze di $\bot$ e $\top$ quanto possibile.
3. ***Flatten***: Unisce sequenze di $\wedge/\vee$ in un'unica congiunzione/disgiunzione o sequenze di $\forall/\exists$ in un'unica quantificazione.
4. ***Unused definitions and pure predicate removal*** (opzionale): Elimina i predicati costanti che non possono creare contraddizioni.
5. ***ENNF***: Trasforma la formula in forma ENNF come visto in **??**.
6. ***Naming*** (opzionale): Applica una tecnica di naming simile a come visto in **??** ma estesa alla logica del primo ordine.
7. ***NNF***: Trasforma la formula in forma NNF come visto in **??**.
8. ***Skolemization***: Elimina i quantificatori esistenziali come visto in **??** con l'unica differenza che si evita di convertire la formula in PNF.
9. ***Clausification***: Clausifica la formula in modo simile a come accennato in **??**.

1.4 Algoritmo di Saturazione

Vampire fa parte di una famiglia di ATP basati su saturazione che implementa la *Given Clause Architecture* (*GCA*). Data una formula in formato CNF la GCA prevede due insiemi di clausole dette *Attive* e *Passive*. Inizialmente l'insieme delle clausole attive è vuoto e l'insieme delle clausole passive contiene tutte le clausole della formula. Dopo questo setup iniziale comincia quello che viene chiamato *Main Loop*. Il Main Loop è un ciclo che termina quando l'insieme delle clausole passive è vuoto o quando viene trovata una clausola vuota. Ad ogni iterazione il Main Loop seleziona una clausola dall'insieme delle clausole passive. Questa clausola viene chiamata *Given Clause* (*GC*). Lo step successivo consiste nell'applicare tutte le inferenze possibili tra la GC e le clausole attive. Le nuove clausole generate vengono aggiunte all'insieme delle clausole passive mentre la GC viene spostata nell'insieme delle clausole attive. Se una delle nuove clausole generate è la clausola vuota all'ora il Main Loop termina e la formula è insoddisfacibile. Se invece l'insieme delle clausole passive viene totalmente svuotato allora il sistema è Saturo e la formula è soddisfacibile. Nel caso la formula non sia insoddisfacibile l'insieme delle clausole passive potrebbe non svuotarsi mai, in questo caso Il Main Loop termina quando sono terminate le risorse disponibili.

Algorithm 1: Architettura Given Clause

Firma: Saturation(φ)
Input: φ Una formula in formato CNF
Output: $\top$ se φ è soddisfacibile, $\bot$ altrimenti
$active = \emptyset$
$passive = \varphi$
while $passive \neq \emptyset$ **do**
    $current := select(passive)$;
    $passive.remove(current)$;
    $active.add(current)$;
    $newClauses := infer(current, active)$;
    if $\square \in newClauses$ **then**
        return $\bot$;
    end
    $passive.add(newClauses)$;
end
return $\top$;

Come già accennato in **??** Otter è stato uno dei primi ATP basati su saturazione e su GCA. In particolare Otter aggiunge a GCA due nuovi step chiamati di semplificazione. Il sistema di inferenze viene diviso in due classi, le inferenze generative e di semplificazione. Le inferenze generative prendono una o più clausole e generano nuove clausole. Le inferenze di semplificazione prendono una o più clausole e inferiscono una nuova clausola, generalmente più corta, che rende le premesse ridondanti in modo da poterle sostituire con la clausola generata. Nel primo step di semplificazone chiamato *Forward semplification* dopo aver selezionato una clausola dall'insieme delle clausole passive, si tenta di applicare le inferenze di semplificazione alla clausola selezionata. Il secondo step di semplificazione chiamato *Backward semplification* consiste nell'applicare le inferenza di semplificazione alle clausole attive e passive. Gli algoritmi che implementano questa struttura vengono detti *Otter* o che implementano la *Otter's Architecture*. Un esempio di Otter's Architecture è mostrato in 2.

Algorithm 2: Architettura Otter

Firma: Saturation(φ)
Input: φ Una formula in formato CNF
Output: $\top$ se φ è soddisfacibile, $\bot$ altrimenti

```
active = ∅
passive = φ
while passive ≠ ∅ do
    current := select(passive);
    passive.remove(current);
    if retained(current) then
        current := forwardSimplify(current, active, passive);
        if current = □ then
            return ⊥;
        end
        if retained(current) then
            (active, passive) := backwardSimplify(current, active, passive);
            active.add(current);
            newClauses := infer(current, active);
            if □ ∈ newClauses then
                return ⊥;
            end
            passive.add(newClauses);
        end
    end
end
return ⊤;
```

Con la funzione *retained* si intende una funzione che restituisce *true* se la clausola è utile per la dimostrazione e *false* altrimenti. Ad esempio se la clausola è una tautologia viene scartata. Questa fase è detta *Retention Test*. Vampire implementa tre algoritmi di saturazione, *Otter*, *LRS Discount*. Otter è una versione leggermente modificata della Otter's Architecture. Al posto di avere solo due insiemi di clausole attive e passive, ha un terzo insieme chiamato *unprocessed*. L'algoritmo LRS (Low resource strategy) è una versione modificata di Otter che guida l'algoritmo in base ai limiti di tempo e memoria dati in input e il tempo/memoria restanti. In particolare quando si effettua il retention test LSR valuta se la clausola è troppo grande da processare in base al tempo e alla memoria rimanente. Se la clausola è troppo grande viene scartata anche a costo della perdita di completezza. Nel caso specifico in cui la formula è soddisfacibile e LSR scarta una clausola, l'algoritmo termina con un errore. Nel caso in cui la formula è insoddisfacibile LSR in alcuni casi performa meglio quando si restringono i limiti di tempo e memoria. L'algoritmo Discount è simile ad Otter solo che gli step di semplificazione vengono eseguiti solo sull'insieme delle clausole attive. Questo è dovuto al fatto che l'insieme delle clausole passive è significativamente più ampio rispetto a quello delle clausole attive. Le clausole attive sono spesso solo l'1% di quelle passive, il che comporta tempi prolungati durante gli step di semplificazione poiché è necessario dedicare molto tempo alla semplificazione delle clausole passive.

Vampire implementa un vasto sistema di inferenze ma il sottoinsieme minimo necessario per la completezza (per la soddisfacibilità di formule senza uguaglianza) è composto dalle inferenze di *Resolution* e *Factoring*.

Resolution:

$$\frac{\{L(\tau_1, ..., \tau_n)\} \cup C_1, \{\neg L(\omega_1, ..., \omega_n)\} \cup C_1}{(C_1 \cup C_2)^\sigma}$$

Factoring:

$$\frac{\{L(\tau_1, ..., \tau_n)\} \cup C \cup \{L(\omega_1, ..., \omega_n)\}}{(\{L(\tau_1, ..., \tau_n)\} \cup C)^\sigma}$$

Con L un letterale, C_1, C_2 clausole, $\tau_1, ..., \tau_n, \omega_1, ..., \omega_n$ termini e σ un unificatore.

1.5 Unificazione e Substitution Trees

Come visto nella sezione precedente l'unificazione è un passo fondamentale per l'applicazione delle regole di inferenza di *Resolution* e *Factoring*. La GCA inoltre prevede che dopo aver selezionato la GC si applichino tutte le possibili inferenze tra la GC e le clausole attive. Senza un'apposita struttura di indicizzazione la ricerca delle inferenze applicabili sarebbe spaventosamente lento. Si pensi di voler verificare se è applicabile la regola di resolution tra due clausole C_1 e C_2. Un algoritmo naive potrebbe essere ad esempio quello di scorrere tutti i letterali di C_1 e C_2 e verificare se la polarità di un letterale è opposta a quella di un altro letterale e se sono unificabili. Questo algoritmo avrebbe un costo nel caso peggiore di $\mid C_1 \mid \cdot \mid C_2 \mid$ per il costo di $unifiable$. È chiaro che una strategia del genere non è sostenibile soprattutto se l'obbiettivo è quello di essere il più rapidi possibile. Vampire utilizza una struttura di indicizzazione chiamata *Substitution Tree* (ST) per velocizzare le operazioni di unificazione. Un SubstitutionTree è una struttura ad albero in cui ogni nodo rappresenta una sostituzione. Si aggiunge oltre all'insieme standard di variabili Σ_x un altro insieme di variabili $\Sigma_x^* = \{*_1, *_2, ...\}$, dette speciali, che servono per la costruzione delle sostituzioni. I termini speciali sono termini che possono contenere anche variabili speciali oltre a quelle standard.

Per sostituzione in un ST si intende una mappa da variabili speciali a termini speciali e vengono scritte in notazione: $\{*_i = \tau,\ *_j = \tau', ...\}$. Per $im(\sigma)$ si indica l'insieme dei termini speciali che compaiono a destra del simbolo dell'uguaglianza in σ. Per $dom(\sigma)$ si indica l'insieme delle variabili speciali che compaiono a sinistra del simbolo dell'uguaglianza in σ. Per composizione di sostituzioni $\sigma_1 \circ \sigma_2$ si intende la sostituzione che si ottiene applicando σ_2 ai termini mappati da σ_1. Ad esempio se $\sigma_1 = \{*_0 = f(*_1, *_2)\}$ e $\sigma_2 = \{*_1 = a, *_2 = g(*_3)\}$ allora $(\sigma_1 \circ \sigma_2) = \{*_0 = f(a, g(*_3))\}$.

Un nodo di un ST è rappresentato da una coppia (σ, T) dove σ è una sostituzione e T è un insieme di ST. Vi sono tre tipi di nodi:

- **Nodo Radice**: Se il nodo è la radice dell'albero allora σ è la sostituzione vuota e T è un insieme non vuoto
- **Nodo Foglia**: Se il nodo è una foglia allora T è l'insieme vuoto. Ad ogni nodo foglia viene associato una struttura chiamata *LeafData* che può contenere dati arbitrari.
- **Nodo Interno**: Se il nodo è interno ne σ ne T sono vuoti.

Ogni ST rispetta i seguenti vincoli:

1. Per ogni percorso $(\sigma_1, T_1), ..., (\sigma_n, T_n)$ dalla radice ad una foglia, l'immagine della composizione delle sostituzioni non contiene variabili speciali (è un termine standard): cioè vi sono solo termini standard in $im(\sigma_1 \circ ... \circ \sigma_n)$.
2. Per ogni percorso $(\sigma_1, T_1), ..., (\sigma_n, T_n)$ dalla radice ad una foglia, ogni variabile speciale è mappata al massimo una volta per percorso: cioè per ogni $i \neq j$, $dom(\sigma_i) \cap dom(\sigma_j) = \emptyset$.

In Vampire è implementata una variante di ST chiamati *Downward Linear Substitution Tree* che facilitano le operazioni di inserimento e ricerca. Una trattazione più approfondita può essere trovata in [**?**]. Un ST accetta termini e letterali. Il processo di inserimento costruisce un percorso di sostituzioni all'interno dell'albero in modo tale che applicando tutte le sostituzioni del percorso alla variabile $*_0$ si ottiene il termine inserito inizialmente. Prima dell'inserimento le variabili dei termini vengono normalizzate in modo da avere gli indici più bassi possibili. I termini originali vengono poi salvati nelle

LeafData per essere recuperati. Per la normalizzazione delle variabili vengono utilizzati i *Variable Banks*, che sono essenzialmente un secondo indice delle variabili, che servono a tenere sempre disgiunte le variabili dei termini "query" (i termini passati come argomento in una funzione di ricerca) da quelle dei termini nel ST. Così facendo è possibile utilizzare, quando necessario, algoritmi di unificazione senza occurrence check tra letterali query e letterali nel ST. Ad esempio se si vuole inserire il termine $f(x, y)$ viene salvato nel ST come $f(x_{1/1}, x_{2/1})$ dove /1 indica il Variable Bank.

Vampire mette a disposizione numerose classi per gestire varie tipologie di indicizzazione tramite ST. Ad esempio la classe *LiteralSubstitutionTree* viene utilizzata per salvare i letterali in un ST. Nei LeafData vengono salvati il letterale e la clausola a cui appartiene. La funzione $getUnifications(Literal* query, bool\ complementary, bool\ retrieveSubstitutions)$ restituisce un iteratore di letterali che unificano con il letterale query. Se complementary è true allora cerca solo i letterali con polarità opposta. Se retrieveSubstitutions è true allora restituisce anche la sostituzione. Tornando all'esempio dell'inizio della sezione, per trovare le clausole su cui si può applicare la regola di resolution con la GC un algoritmo più semplice e molto più efficiente rispetto a quello proposto inizialmente consiste nell'inserire tutte le clausole attive nel ST e chiamare la funzione $getUnifications$ su ogni letterale della GC:

$newClauses := \emptyset$
foreach $l \in GC$ **do**
    $iter := getUnifications(l, true, true)$
    while $iter.hasNext()$ **do**
        $res := iter.next()$
        $(l' : Literal*, C : Clause*) := res.leafData()$
        $\sigma := res.substitution()$
        $newClauses.add(resolution(l, l', GC, C, \sigma))$
    end
end

LiteralSubstitutionTree crea una mappa da simboli di predicato a ST in modo tale che ogni letterale con lo stesso predicato venga inserito nello stesso ST. Per inserire letterali con simboli di predicato diversi si può utilizzare la classe *SubstitutionTree* a patto che i predicarti abbiano stessa arità. Con la classe *SubstitutionTree::UnificationsIterator* e la funzione *SubstitutionTree::iterator* è possibile ottenere gli stessi risultati dell'esempio precedente:

1.6 Il SAT-Solver

Vampire non implementa un SAT-Solver ma ha un vasto sistema di interfacce per utilizzare al meglio SAT-Solver esterni. Al momento gli unici SAT-Solver supportati sono MiniSat e Z3, anche se l'inclusione di Z3 è ancora in fase sperimentale. Per utilizzare un SAT-Solver è necessario creare Clausole e letterali appositi per la rappresentazione delle costanti proposizionali. Nella figura 1.7 è mostrata la struttura delle classi e delle interfacce per il SAT-Solver. La classe *SATLiteral* rappresenta una costante proposizionale ed è costituita da una coppia intero-booleano che rappresenta l'indice della costante e la sua polarità. La classe *SATClause* come la classe *Clause* è costituita da un Array, in questo caso di *SATLiteral*. La classe *Sat2FO* è uno dei componenti Built-in di Vampire che si occupa di convertire letterali e clausole del primo ordine (FO) in oggetti di tipo *SATLiteral* e *SATClause* (SAT) e viceversa. La chiamata della funzione *Sat2FO::toSat(Literal*)* aggiunge ad una bi-mappa il puntatore al letterale e lo associa ad un nuovo numero unsigned ≥ 1 se non è già presente altrimenti restituisce il numero già associato. La funzione *Sat2FO::toFO(SATLiteral*)* restituisce il puntatore al letterale associato al numero passato come argomento se presente altrimenti *nullptr*.

La classe astratta *SATSolver* rappresenta un generico SAT-Solver e contiene le funzioni virtuali comuni a tutti i SAT-Solver come *addClause(SATClause*)*, *solve* e *trueInAssignment(SATLiteral)* per

aggiungere clausole, risolvere il problema e ottenere l'assegnamento delle variabili proposizionali (se soddisfacibile). Ogni variabile prima di essere utilizzata ha bisogno di essere 'registrata' tramite la funzione *newVar* che restituisce l'indice incrementale della nuova variabile registrata. Un altro metodo è quello di utilizzare la funzione *ensureVarCount(count)* che assicura che il numero di variabili registrate sia almeno pari a count. Se si è utilizzata la classe *Sat2FO* è possibile utilizzare la combinazione *SATSolver::ensureVarCount(Sat2FO::maxSATVar())* per assicurarsi che il numero di variabili registrate sia almeno pari al numero di costanti proposizionali utilizzate.

SATSolverWithAssumption estende *SATSolver* e permettere di aggiungere delle assunzioni (Dei letterali che devono essere veri nella soluzione). *PrimitiveProofRecordingSATSolver* estende *SATSolverWithAssumption* e definisce le funzioni per risalire alla dimostrazione di insoddisfacibilità. *MinisatInterfacing* è una classe che si occupa di interfacciare Vampire con la classe *Minisat* che contiene l'effettivo codice di Minisat.

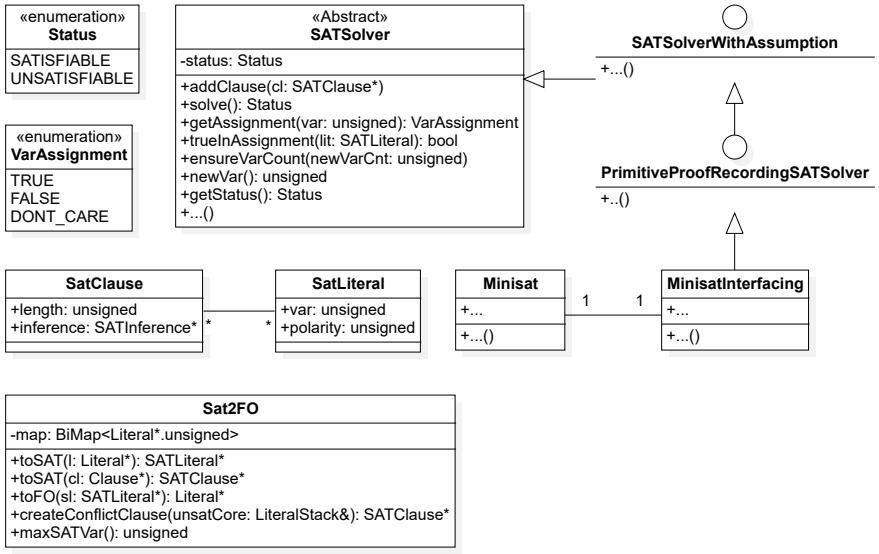

Figura 1.7: Classi e interfacce per il SAT-Solver

Ad esempio si pensi di voler determinare la soddisfacibilità della formula CNF FO ground $\varphi := (p_1(f_1) \vee p_2 \vee \neg p_3) \wedge (\neg p_2 \vee \neg p_3) \wedge (\neg p_1(f_1))$. In primo luogo le clausole vengono divise in unità e rappresentate come Array di letterali:

$$unitList := [[p_1(f_1), p_2, \neg p_3], [\neg p_2, \neg p_3], [\neg p_1(f_1)]]$$

Applicando la funzione *Sat2FO::toSat(Clause*)* ad ogni clausola si ottiene una lista di SATClausole:

$$satUnitList := [[1, 2, -3], [-2, -3], [-1]]$$

A questo punto vanno registrate le variabili nel SAT-solver e aggiunte le clausole:

$satSolver = newSatSolver()$
$satSolver.ensureVarCount(sat2Fo.maxSATVar())$
for $c \in satUnitList$ **do**
  $satSolver.addClause(c)$

end

È possibile quindi chiamare la funzione *solve* del SAT-Solver per ottenere la soddisfacibilità della formula. In questo caso la formula è soddisfacibile un possibile assegnamento è $[1 \rightarrow false, 2 \rightarrow false, 3 \rightarrow false]$.

SatSolver non ha una funzione per ottenere l'assegnamento direttamente ma è possibile ottenere l'assegnamento di ogni singola variabile tramite la funzione *trueInAssignment(SATLiteral)*.

$\alpha := EmptyMap < Literal*, bool > ()$
foreach $c \in unitList$ **do**
  foreach $l \in c$ **do**
    if $l.polarity()$ **then**
      $\alpha[l] := solver.trueInAssignment(sat2Fo.toSat(l))$
    end
  end
end

Per chiedere al SAT-Solver di cercare un altro assegnamento è possibile aggiungere una nuova clausola che rende l'assegnamento trovato incompatibile. Una clausola del genere è detta clausola bloccante (Blocking Clause) o clausola di conflitto (Conflict Clause). In questo caso una possibile clausola bloccante è $[1, 2, 3]$. Con l'aggiunta di questa clausola la formula diventa insoddisfacibile e il SAT-Solver restituirà *UNSATISFIABLE* alla chiamata di *solve*. Un modo per creare una clausola bloccante è quello di utilizzare la funzione Built-in di *Sat2FO::createConflictClause(LiteralStack)* che prende in input una lista di letterali e restituisce una SatClausola con i letterali negati.

1.7 Misurazione dei Tempi

Quando le performance sono un fattore critico è necessario avere un insieme di strumenti per misurare i tempi di esecuzione. Vampire mette a disposizione vari modi per misurare i tempi ed eseguire statistiche. In questa sezione ne verranno trattati essenzialmente tre. Il primo metodo più classico consiste semplicemente nel rilevare due tempi e calcolare la differenza. Questo può essere fatto utilizzando la funzione *elapsedMilliseconds* della classe *Timer* che restituisce il tempo in millisecondi trascorso dall'inizio dell'esecuzione del programma. Un timer globale è disponibile nell'oggetto globale *env* della classe *Lib/Environment*.

$t_{start} := env.timer \rightarrow elapsedMilliseconds()$
...
$t_{end} := env.timer \rightarrow elapsedMilliseconds()$
$\Delta t := t_2 - t_1$

Il secondo metodo consiste nell'utilizzare la macro *TIME_TRACE(name)* che misura il tempo trascorso tra l'invocazione e la fine del blocco di codice. 'name' è una stringa che di norma dovrebbe essere definita nella classe *Debug/TimeProfiling* con tipo *static constexpr const char* const*. È possibile chiamare più volte *TIME_TRACE* (con 'name' diversi) in più blocchi annidati e alla fine dell'esecuzione, con l'opzione *-tstat* attiva, Vampire stamperà un report con un albero delle chiamate e i tempi trascorsi, il numero di chiamate e il tempo medio per chiamata. Un esempio di report è mostrato in figura 1.8.

Il terzo metodo non è serve a misurare il tempo di esecuzione ma conta il numero di invocazioni. La macro *RSTAT_CTR_INC(name)* definita in *Debug/RuntimeStatistics* definisce un contatore associato ad ogni 'name' e lo incrementa di 1 ad ogni invocazione. Anche in questo caso Vampire stamperà un report alla fine dell'esecuzione con il formato *'name': 'count'*. Vampire utilizza questa macro ad esempio per contare il numero di clausole create e il numero di clausole eliminate. Un esempio di report è mostrato in figura 1.9.

```
===== start of time trace =====
[root] (total: 6772 µs, avg: 6772 µs, cnt: 1)
  ├──[61%] main loop (total: 4169 µs, avg: 4169 µs, cnt: 1)
  │      ├──[99%] run (total: 4149 µs, avg: 4149 µs, cnt: 1)
  │      │      ├──[58%] forward simplification (total: 2431 µs, avg: 29 µs, cnt: 83)
  │      │      │      ├──[94%] forward subsumption (total: 2295 µs, avg: 27 µs, cnt: 83)
  │      │      │      │      └──[45%] forward subsumption resolution (total: 1053 µs, avg: 15 µs, cnt: 70)
  │      │      │      ├──[ 0%] splitting component index usage (total: 6569 ns, avg: 96 ns, cnt: 68)
  │      │      │      ├──[ 0%] term sharing (total: 5989 ns, avg: 2994 ns, cnt: 2)
  │      │      │      └──[ 0%] splitting component index maintenance (total: 1265 ns, avg: 316 ns, cnt: 4)
  │      │      ├──[19%] activation (total: 823 µs, avg: 24 µs, cnt: 33)
  │      │      │      ├──[76%] clause generation (total: 628 µs, avg: 3344 ns, cnt: 188)
  │      │      │      │      ├──[66%] resolution (total: 419 µs, avg: 1559 ns, cnt: 269)
  │      │      │      │      │      └──[21%] term sharing (total: 91 µs, avg: 569 ns, cnt: 161)
  │      │      │      │      └──[ 0%] term sharing (total: 3489 ns, avg: 872 ns, cnt: 4)
  │      │      │      ├──[ 8%] add clause (total: 67 µs, avg: 2054 ns, cnt: 33)
  │      │      │      │      └──[85%] binary resolution index maintenance (total: 57 µs, avg: 1749 ns, cnt: 33)
  │      │      │      │             └──[ 8%] term sharing (total: 5064 ns, avg: 389 ns, cnt: 13)
  │      │      │      ├──[ 6%] clause selection (total: 51 µs, avg: 1563 ns, cnt: 33)
  │      │      │      │      └──[88%] literal selection (total: 45 µs, avg: 1381 ns, cnt: 33)
  │      │      │      ├──[ 0%] splitting (total: 1689 ns, avg: 51 ns, cnt: 33)
  │      │      │      └──[ 0%] redundancy check (total: 1669 ns, avg: 50 ns, cnt: 33)
  │      │      ├──[ 5%] passive container maintenance (total: 222 µs, avg: 2249 ns, cnt: 99)
  │      │      │      ├──[63%] forward subsumption index maintenance (total: 141 µs, avg: 2521 ns, cnt: 56)
  │      │      │      │      └──[ 9%] term sharing (total: 12 µs, avg: 359 ns, cnt: 36)
  │      │      │      └──[ 8%] unit clause index maintenance (total: 19 µs, avg: 1903 ns, cnt: 10)
  │      │      ├──[ 0%] immediate simplification (total: 32 µs, avg: 373 ns, cnt: 87)
  │      │      ├──[ 0%] SAT solver (total: 11 µs, avg: 5903 ns, cnt: 2)
  │      │      ├──[ 0%] backward simplification (total: 3697 ns, avg: 56 ns, cnt: 66)
  │      │      ├──[ 0%] minimizing solver time (total: 2094 ns, avg: 2094 ns, cnt: 1)
  │      │      └──[ 0%] splitting model update (total: 721 ns, avg: 721 ns, cnt: 1)
  │      └──[ 0%] init (total: 17 µs, avg: 17 µs, cnt: 1)
  ├──[23%] parsing (total: 1615 µs, avg: 1615 µs, cnt: 1)
  │      └──[ 1%] term sharing (total: 25 µs, avg: 387 ns, cnt: 67)
  ├──[ 7%] preprocessing (total: 527 µs, avg: 527 µs, cnt: 1)
  │      ├──[45%] property evaluation (total: 241 µs, avg: 120 µs, cnt: 2)
  │      ├──[ 5%] term sharing (total: 26 µs, avg: 714 ns, cnt: 37)
  │      └──[ 1%] naming (total: 9235 ns, avg: 577 ns, cnt: 16)
  └──[ 0%] sat proof minimization (total: 20 µs, avg: 20 µs, cnt: 1)
===== end of time trace =====
```

Figura 1.8: Esempio di report di Time Trace

```
----  Runtime statistics ----
clauses created: 93
clauses deleted: 19
ssat_new_components: 2
ssat_non_splittable_sat_clauses: 1
ssat_sat_clauses: 3
total_frozen: 1
------------------------------
```

Figura 1.9: Esempio di report di Runtime Statistics