Capitolo 1

Il Theorem prover Vampire

Vampire [?] [?] [?] è un dimostratore di teoremi automatico per la logica del primo ordine basato sulle regole di *Resolution* e *Paramodulation*. 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

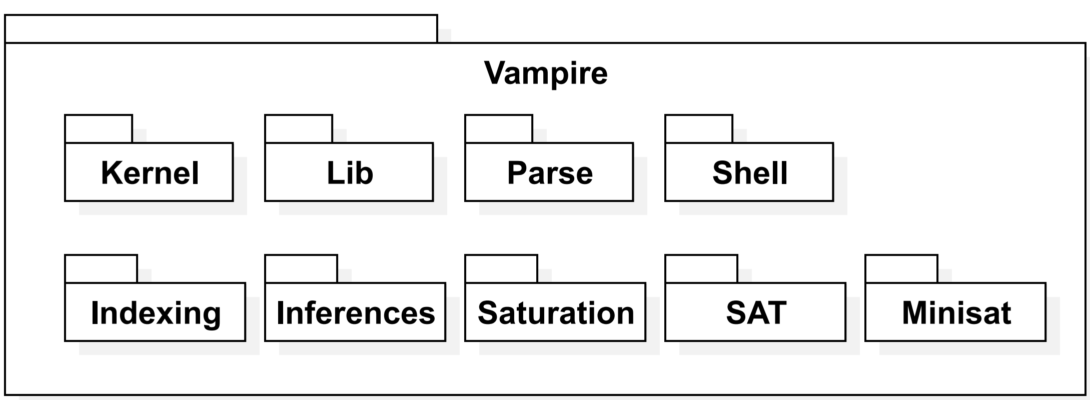

Figura 1.1: Struttura delle cartelle di Vampire.

generale 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 TPTP. 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 1.4. Nella cartella *Indexing* sono presenti i componenti per l'indicizzazione dei termini. Le particolari strutture per l'unificazione verranno trattate nella sezione 1.3. 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 1.5 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.

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 a 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.

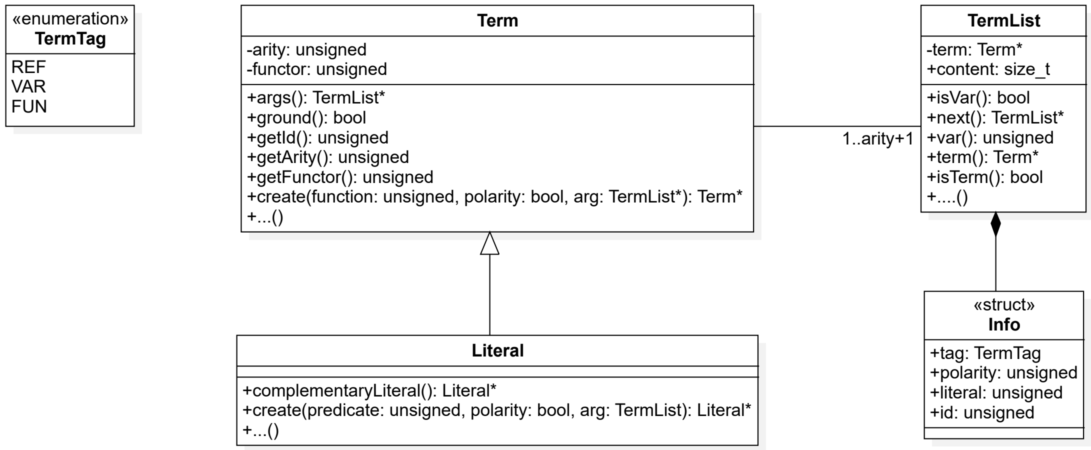

Figura 1.2: Struttura dei termini

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 1.3. Literal in 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:

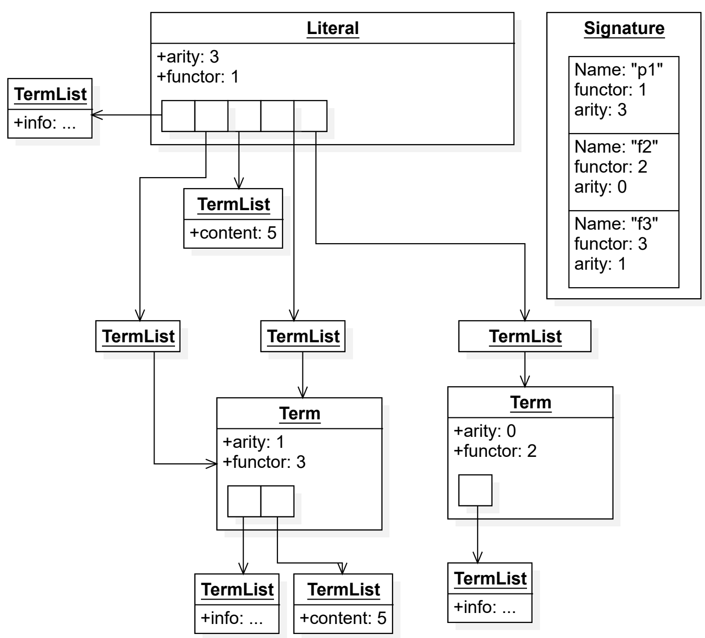

Figura 1.3: Esempio di rappresentazione di un termine

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.

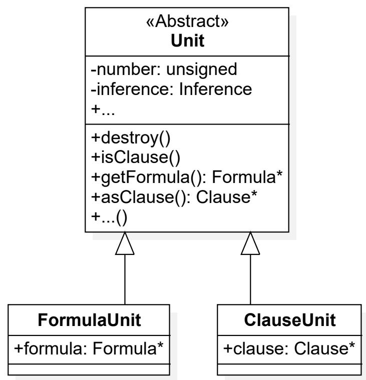

Figura 1.4: Struttura delle unità

Le unità come mostrato in figura 1.4 sono rappresentate dalla classe astratta *Unit*, che si specializza nelle classi *FormulaUnit* e *ClauseUnit* che contengono rispettivamente un puntatore ad un oggetto di tipo *Formula* e *Clause*. 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 $\vee$.

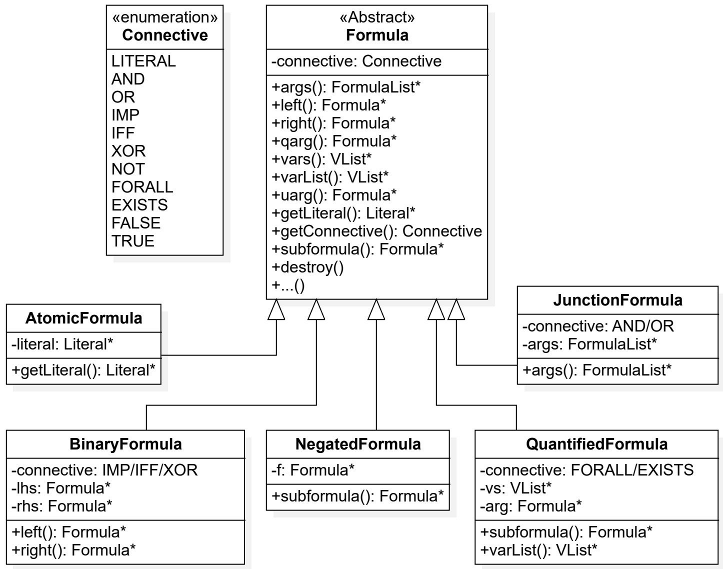

Figura 1.5: Struttura delle formule

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 **??**.

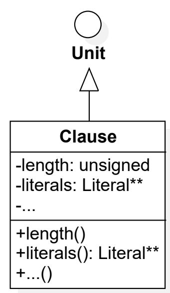

Figura 1.6: Struttura delle Clausole

1.3 Unificazione e Substitution Trees

1.4 Preprocessing

(vampire utilizza solo clausole per la dimostrazione)

1.5 Algoritmo di Saturazione

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*. Uno dei componenti Built-in di Vampire è la classe *Sat2FO* 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.

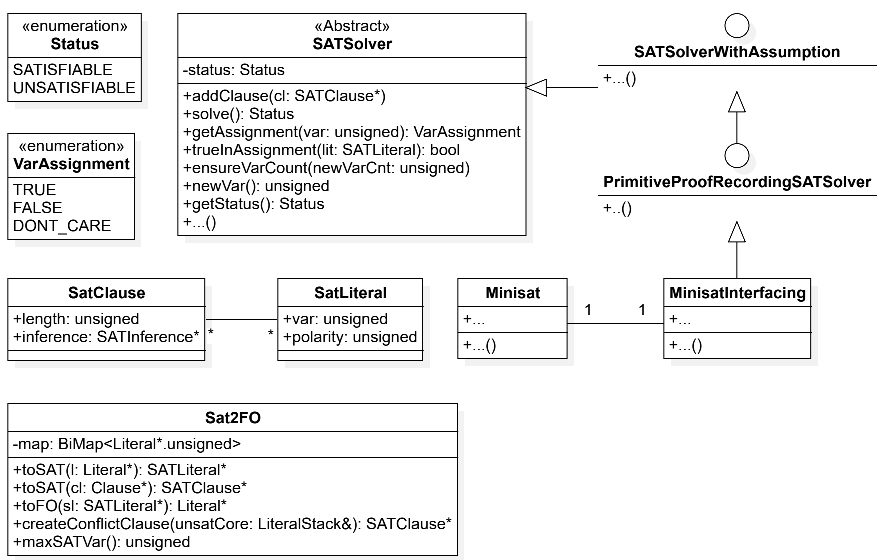

Figura 1.7: Classi e interfacce per il SAT-Solver

Ad esempio si pensi di voler determinare la soddisfacibilità della formula CNF FO ground $\varphi := (p_1(f_1) \lor p_2 \lor \neg p_3) \land (\neg p_2 \lor \neg p_3) \land (\neg p_1(f_1))$. 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

A questo punto è possibile 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