Indice

INTRODUZIONE

tuProlog è un interprete Prolog scritto in Java, attualmente giunto alla versione 2.6.
Alcuni anni fa era stato progettato, per la versione 2.4 disponibile all'epoca, un supporto per il multithreading esplicito, il cui prototipo però non era stato incluso nelle distribuzioni ufficiali a causa del parallelo sviluppo di altre funzionalità più prioritarie e della successiva riorganizzazione del motore.
La presente tesi si prefigge l'obiettivo di attualizzare tale lavoro, farne un'analisi critica e valutarne l'integrazione con l'attuale architettura di tuProlog fino a sviluppare un nuovo prototipo.
In particolare, tramite uno studio preliminare di fattibilità, occorrerà stabilire se sia possibile e opportuno riutilizzare, in tutto o in parte, l'approccio alla base del prototipo precedente o se invece occorra adottare un approccio diverso.

La tesi è quindi organizzata come segue. Il capitolo primo presenta l'attuale motore tuProlog, spiegando quali sono le caratteristiche principali dell'interprete e facendo un'analisi di alto livello del suo funzionamento. Inoltre sono introdotte alcune nozioni di base, tra cui la differenza tra parallelismo implicito ed esplicito. Nel capitolo secondo si analizza il precedente prototipo e vengono discusse le scelte progettuali che stanno dietro all'architettura. Il capitolo terzo presenta dapprima alcune possibili direzioni per l'elaborazione dell'architettura del nuovo prototipo alla luce dell'analisi critica svolta riguardo il modello precedente, fino a giungere alla definizione di una nuova architettura, che viene quindi discussa in dettaglio. Il successivo capitolo discute le problematiche di sincronizzazione legate all'adozione di un modello concorrente, con particolare riferimento all'ambiente di esecuzione e alla sincronizzazione delle esecuzioni parallele dei processi. Infine, Il capitolo quinto presenta la libreria risultante con l'insieme delle rispettive primitive e i relativi esempi d'uso.

CAPITOLO 1

INTERPRETE PROLOG

Nella programmazione logica di Prolog un problema è un quesito da risolvere sulla base di conoscenza espressa come sequenza di clausole che compongono una *teoria*. Le *clausole* si esprimono secondo le due categorie di *fatti* e *regole*: si dice che un programma Prolog è costituito da un insieme di fatti che dichiarano un certo stato di cose, da un insieme di regole che definiscono relazioni fra stati di cose e da *query* (o *goal*) a cui associare una o più soluzioni alternative deducibili logicamente dal programma.

L'interprete Prolog si occupa di dimostrare i teoremi, cercando di derivare la conclusione dalle premesse attraverso le regole di inferenza. L'ordine con cui esaminare le clausole del programma viene specificato con una strategia di ricerca depth-first con backtracking. Si tratta di una ricerca in profondità, combinata con la selezione della chiamata procedurale più a sinistra. Questa ricerca può essere eseguita con l'utilizzo di un unico stack che rappresenta il ramo in esplorazione e contiene riferimenti ai rami da esplorare in caso di fallimento. Il modello computazionale di Prolog prevede di costruire l'albero di ricerca (o albero AND-OR) considerando un'alternativa alla volta: ogni clausola viene decomposta in sottogoal ed ognuno di essi rappresenta un ramo o una foglia dell'albero. L'interprete opera su una congiunzione di goal in modo sequenziale, da sinistra a destra e passa ad un goal successivo solo quando quello precedente è stato completamente soddisfatto.

In caso di successo la risposta conterrà il binding delle variabili presenti nel goal. Ad esempio un goal viene presentato come un predicato del tipo:

```
genitore(Bob,X).
```

su una base di conoscenza `genitore(Bob,Anna).` che ha il seguente significato:

'*esiste un qualche elemento (un X) per cui il predicato genitore(Bob, X) è vero'*.

Il goal in questo caso banale ha successo e nella soluzione la variabile X sarà legata al valore Anna (`X/Anna`).

In caso di fallimento si effettua un backtracking, ossia si esplorano, uno alla volta, rami

alternativi dell'albero (se ne esistono e se non sono stati ancora esplorati) alla ricerca di soluzioni.

Durante il backtracking, i legami già effettuati delle variabili vengono annullati in ordine inverso, da destra a sinistra. Vengono quindi ricercate altre corrispondenze con clausole differenti da quelle utilizzate in precedenza, cercando di soddisfare il goal.

1.1 Parallelismo implicito ed esplicito

La programmazione logica ha avuto origine dall'interpretazione procedurale delle *clausole di Horn*[1]. Secondo questa interpretazione, una clausola di Horn della forma

$$A \leftarrow B_1, \dots, B_n$$

è vista come la definizione di una procedura, dove A è il nome della procedura e $B_1, \dots, B_n$ è l'insieme delle chiamate di procedura che ne costituiscono il corpo. Da questo tipo di interpretazione sono nati i linguaggi logici, dei quali Prolog è il più importante e diffuso.

La tecnica di backtracking viene utilizzata per la gestione del *nondeterminismo* (ovvero come pilotare in maniera adeguata le scelte del sistema davanti a situazioni non definibili a priori), il quale deriva dal fatto che le clausole con uguale testa rappresentano modi alternativi per la soluzione di uno stesso goal. Come spiegato all'inizio del capitolo, se l'unificazione con la clausola scelta fallisce, il Prolog passa alla clausola successiva con la stessa testa e tenta l'unificazione con essa. Questa non è l'unica strategia di prova per le clausole di Horn. In particolare, è possibile definire paradigmi di concorrenza basati sul concetto di *parallelismo implicito*.

Il parallelismo implicito in Prolog è conseguenza di una nuova interpretazione delle clausole di Horn: l'interpretazione a processi. Secondo quest'ultima, un sottogoal B_i in un goal $\leftarrow B_1, \dots, B_n$ è visto come un processo. L'obiettivo dei linguaggi logici concorrenti è lo sfruttamento di alcune forme di parallelismo, dette *parallelismo AND* e *parallelismo OR*, all'interno dei programmi logici.

Per parallelismo OR si intende la risoluzione in parallelo di tutte le clausole la cui testa unifica con il sottogoal corrente. Ad esempio, dato il goal $? - p(X)$, in presenza delle seguenti clausole:

[1] Console Luca *et al.*, *Programmazione logica e Prolog*, UTET, Milano 1997.

$$p(X) :- q(X).$$
$$p(X) :- r(X).$$

il parallelismo OR risolve in parallelo le due clausole ed ottiene i valori di X di entrambe.

Per parallelismo AND si intende invece la risoluzione in parallelo di tutti i sottogoal che compongono il goal da risolvere. Ad esempio, nel caso in cui si debba risolvere il goal $? - p(X), q(Y)$, vengono risolti in parallelo i due sottogoal $p(X)$ e $q(Y)$. Si noti come in Prolog in entrambi i casi presentati la risoluzione avviene in modo sequenziale. In particolare, nell'ultimo esempio la risoluzione di $q(Y)$ avviene dopo che $p(X)$ è stata risolta.

In generale, nel parallelismo di tipo implicito vengono espresse, all'interno dell'interprete, le varie forme di parallelismo presenti nei programmi logici; dall'analisi del programma vengono ricavate le attività che possono essere eseguite in parallelo, senza che vi sia alcun tipo di intervento da parte del programmatore.

In alternativa, l'esplicitazione del parallelismo viene realizzata attraverso l'arricchimento dei costrutti del linguaggio, con l'introduzione di primitive per la gestione della concorrenza. Molte di queste primitive comprendono meccanismi di sincronizzazione, comunicazione e distribuzione dei compiti tra i processi.

Il vantaggio della programmazione parallela esplicita riguarda il controllo assoluto da parte del programmatore di ogni esecuzione parallela. Un programmatore esperto può trarre vantaggio dal parallelismo esplicito per produrre codice molto efficiente. Tuttavia, programmare sfruttando questo tipo di parallelismo è spesso complesso, specialmente per programmatori non specializzati, a causa del lavoro di pianificazione necessario per estrarre la concorrenza intrinseca della computazione e sincronizzare le esecuzioni parallele.

1.2 L'attuale motore tuProlog

La risoluzione di un programma Prolog richiede diverse fasi di sviluppo, che nell'interprete tuProlog vengono gestite separando le responsabilità.

Ogni responsabilità all'interno del sistema viene associata ad un singolo manager, il quale controlla parte del motore inferenziale tuProlog.

La classe `Prolog` rappresenta il motore inferenziale e coordina tutta l'attività dei manager. Sono presenti sei diversi manager, ciascuno dei quali è responsabile della gestione di un preciso insieme di funzionalità:

1. `LibraryManager`
   Carica e rimuove le librerie di predicati a runtime; all'avvio vengono caricate alcune librerie di default, mentre tutte le altre, comprese quelle definite dall'utente, possono essere caricate a runtime.

2. `TheoryManager`
   Carica la teoria nel sistema e crea il database delle clausole; gestisce i predicati che modificano la teoria in modo dinamico.

3. `OperatorManager`
   Inizializza e gestisce una struttura destinata a contenere la lista degli operatori definiti, ordinata secondo la loro priorità.

4. `FlagManager`
   Definisce e gestisce i flag, ovvero termini che possono assumere solo un insieme predefinito di valori.

5. `PrimitiveManager`
   Si occupa dell'effettiva esecuzione dei predicati, dei funtori e delle direttive; identifica i predicati presenti nella teoria, in base al fatto che siano definiti nelle librerie o dall'utente. A seconda del tipo di riconoscimento, i predicati vengono trattati diversamente durante il processo di risoluzione della query: nel primo caso vengono eseguiti come metodi Java, mentre nel secondo vengono gestiti dalla macchina a stati del motore.

6. `EngineManager`
   Incapsula l'unità di elaborazione capace di risolvere una query.

Tutti i manager, a parte PrimitiveManager e OperatorManager, comunicano con il motore Prolog in seguito ad eccezioni o per notificare messaggi di *warning* o *spy*. Durante lo svolgimento di particolari compiti, tuttavia, può accadere che i manager si richiamino l'un l'altro direttamente.

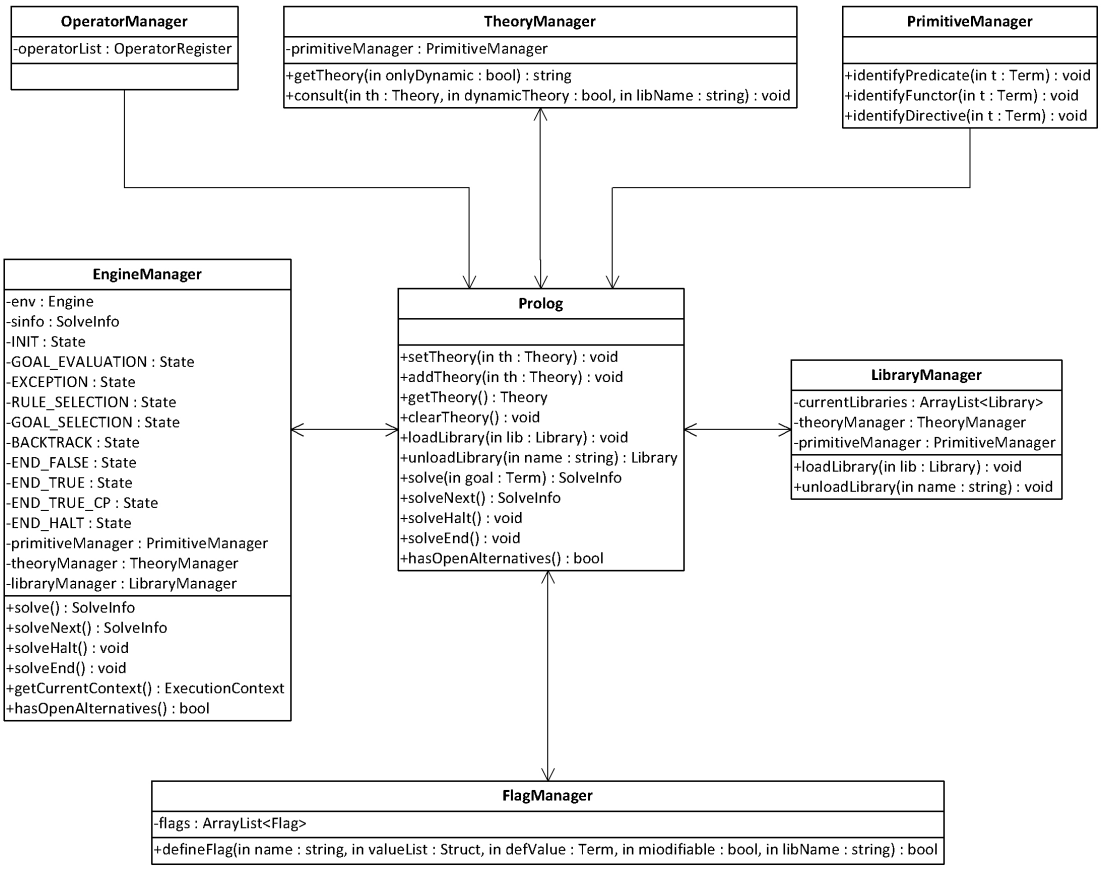

Figura 1.2–1 Le relazioni che intercorrono tra i diversi manager e il motore Prolog e i principali servizi che ogni entità espone verso l'esterno.

Di seguito viene presa in esame l'architettura interna dell'interprete tuProlog, facendo particolare attenzione alla parte che riguarda la risoluzione delle query.

Il sistema è organizzato in maniera molto strutturata. Prolog è il nucleo del sistema, si occupa di gestire le chiamate da parte dell'utente e di indirizzarle ai vari manager in base alle loro funzionalità. Quando viene richiesta la risoluzione di una query, Prolog delega l'esecuzione a EngineManager.

In una prima fase, EngineManager chiama i rispettivi manager per inizializzare le librerie e identificare i predicati della query. Successivamente, il flusso di esecuzione,

che inizia con la ricerca della prima soluzione ed eventualmente di tutte le altre, consiste in una o più chiamate alla classe *Engine* che implementa l'automa a stati finiti.
Questa macchina, nella quale ogni stato rappresenta un passo nella risoluzione della teoria, è di fondamentale importanza in quanto realizza il motore di risoluzione Prolog.
Engine mantiene sia lo stato corrente che lo stato successivo in cui la macchina deve transitare. In particolare, il motore comprende uno stato iniziale (*State Init*), quattro stati che rappresentano il processo di risoluzione di una query (*State Goal Selection, State Goal Evaluation, State Rule Selection, State Backtrack*), uno stato di errore che riguarda le eccezioni che possono essere sollevate durante la valutazione del goal (*State Exception*) e quattro stati finali che rappresentano i diversi modi in cui la risoluzione può terminare (*State Halt, State True, State True_CP, State False*) e che dipendono dal successo o fallimento della ricerca.

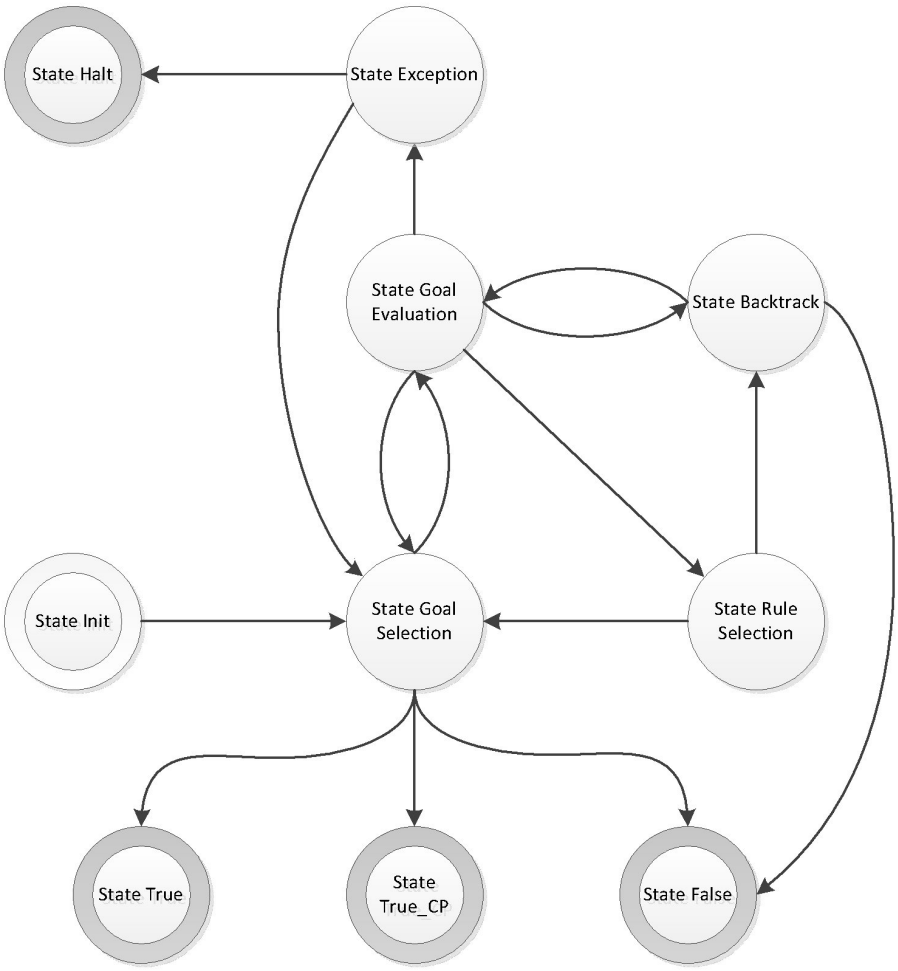

Figura 1.2–2 Diagramma che rappresenta la macchina a stati dell'interprete tuProlog.

EngineManager inizializza tutti gli stati e fa partire la macchina dallo stato Init che si occupa di creare il contesto in cui la query viene valutata; dopodiché si passa allo stato Goal Selection che seleziona il prossimo sottogoal dalla query corrente. Se il sottogoal non viene trovato e il processo di esecuzione termina, si presentano due differenti possibilità: se è presente un punto di scelta, lo stato finale è True_CP, altrimenti True. Nel caso in cui invece il sottogoal esiste ma non è valido, l'esecuzione fallisce con stato False, mentre se il sottogoal risulta essere valido, esso viene valutato nello stato Goal Evaluation. Se il predicato è primitivo la macchina transita nello stato Goal Selection o Backtrack, a seconda se la valutazione ha successo o fallisce, altrimenti transita nello stato Rule Selection. In questo stato viene scelta una clausola compatibile con la risoluzione della teoria (si cerca la testa della clausola che possa essere unificata con il predicato). Una volta trovata la clausola si ritorna allo stato di Goal Selection, altrimenti lo stato successivo è quello di Backtrack. In qualsiasi momento venga raggiunta una condizione di errore durante la valutazione del goal (che può essere di tipo *PrologError* o *JavaException*), la macchina transita nello stato di Exception. Si risale all'indietro l'albero alla ricerca di un sottogoal che unifichi con il predicato dell'eccezione lanciata. A questo punto si presentano due diverse strade: si è arrivati alla radice dell'albero e l'errore non risulta gestibile, quindi l'esecuzione termina e lo stato finale è quello di Halt, oppure si è trovato un predicato compatibile e la macchina può continuare la sua esecuzione dallo stato di GoalSelection. Lo stato di Backtrack si occupa di trovare una clausola che unifichi con il sottogoal a partire dal contesto creato nell'ultimo punto di scelta aperto; se non viene trovata una clausola alternativa, la macchina termina nello stato False.

Una volta che l'automa ha concluso il flusso di esecuzione, EngineManager restituisce il risultato a Prolog. Se sono presenti soluzioni alternative, il manager in base ai comandi ricevuti dall'utente, fa ripartire l'automa dallo stato di Backtrack per dare inizio ad una nuova computazione. Il carattere ricorsivo del procedimento di unificazione, dovuto ai termini composti in una teoria, comporta un elevato numero di punti di scelta.

Direttamente coinvolti nel processo di risoluzione di una query, risultano essere anche il LibraryManager, il TheoryManager e il PrimitiveManager, che si occupano di eseguire operazioni propedeutiche alla risoluzione vera e propria.

CAPITOLO 2

IL PROTOTIPO PRECEDENTE

Dalla versione 2.4 alla 2.6 sono state effettuate diverse modifiche all'interprete tuProlog che hanno portato a un disallineamento tra la versione del prototipo precedente e quella dell'attuale interprete.

Il modello di threading del prototipo si rifà a SWI-Prolog (*Sociaal-Wetenschappelijke Informatica*, dal nome del gruppo dell'università di Amsterdam che ha inizialmente sviluppato il progetto). Si tratta di un interprete Prolog open source, scritto in C, basato sul paradigma del parallelismo esplicito e, pertanto, dotato di molte librerie e predicati built-in. In quel modello sono presenti strumenti di sincronizzazione e comunicazione potenti, ma con alcuni limiti: uno di questi consiste nel fatto che quando un thread risolve una query è solo possibile risalire al suo stato di terminazione e non alla soluzione vera e propria; un altro grande limite è l'assenza di predicati backtrackabili, visto che in SWI-Prolog l'utente non ha la possibilità di effettuare richieste di computazioni successive alla prima.

Il precedente prototipo fa riferimento all'implementazione dell'interprete SWI-Prolog, cercando però di superarne i limiti.

2.1 Ambiente e contesto di esecuzione

Una distinzione fondamentale è quella tra *ambiente di esecuzione* e *contesto di esecuzione.*

In generale, con ambiente di esecuzione si intende l'insieme di associazioni che vengono create nella parte più esterna del programma e che normalmente sono visibili all'interno di ogni sottoprogramma o sottoblocco. Per contesto di esecuzione si intende invece l'insieme delle informazioni che permettono di riconoscere e descrivere il singolo sottoprogramma.

In ambito concorrente è possibile effettuare una trasposizione delle due definizioni:

- L'ambiente di esecuzione ha una visione dinamica dell'*intero sistema*, che include le informazioni che devono essere condivise tra i processi, come i predicati, i fatti e i dati globali (primitive, flag, operatori definiti, etc…).

- Il contesto di esecuzione invece fa riferimento al *contesto privato* del processo, cioè a tutte quelle informazioni che vengono generate durante la risoluzione della query, come i punti di scelta e le unificazioni.

All'interno del sistema è quindi presente un unico ambiente di esecuzione, con numerosi contesti, ciascuno dei quali viene associato alla risoluzione di una singola query.

A livello astratto, ogni thread ha il compito di risolvere uno specifico goal: il contesto di esecuzione è rappresentato da un EngineManager che si occupa dell'elaborazione della query (incapsula la macchina a stati finiti), mentre l'ambiente di esecuzione è mappato nei restanti cinque manager poiché condivisi da tutti i processi.

In realtà nel modello del prototipo non è possibile effettuare un'associazione biunivoca tra una query (intesa come l'insieme delle soluzioni alternative) e un processo, quanto piuttosto tra un processo e una singola soluzione, perchè viene generato un nuovo thread per ogni risultato da produrre. Tra una computazione e l'altra relativa alla stessa query, il thread che ha trovato una soluzione termina e trasferisce il suo contesto di esecuzione al thread successivo che viene creato per cominciare una nuova computazione.

2.2 Architettura proposta

La struttura logica del sistema prevede di replicare tanti EngineManager quanti sono i thread pronti per l'esecuzione e di modificare i restanti manager in modo da renderli thread-safe, permettendo la gestione di accessi contemporanei da parte di diversi processi.

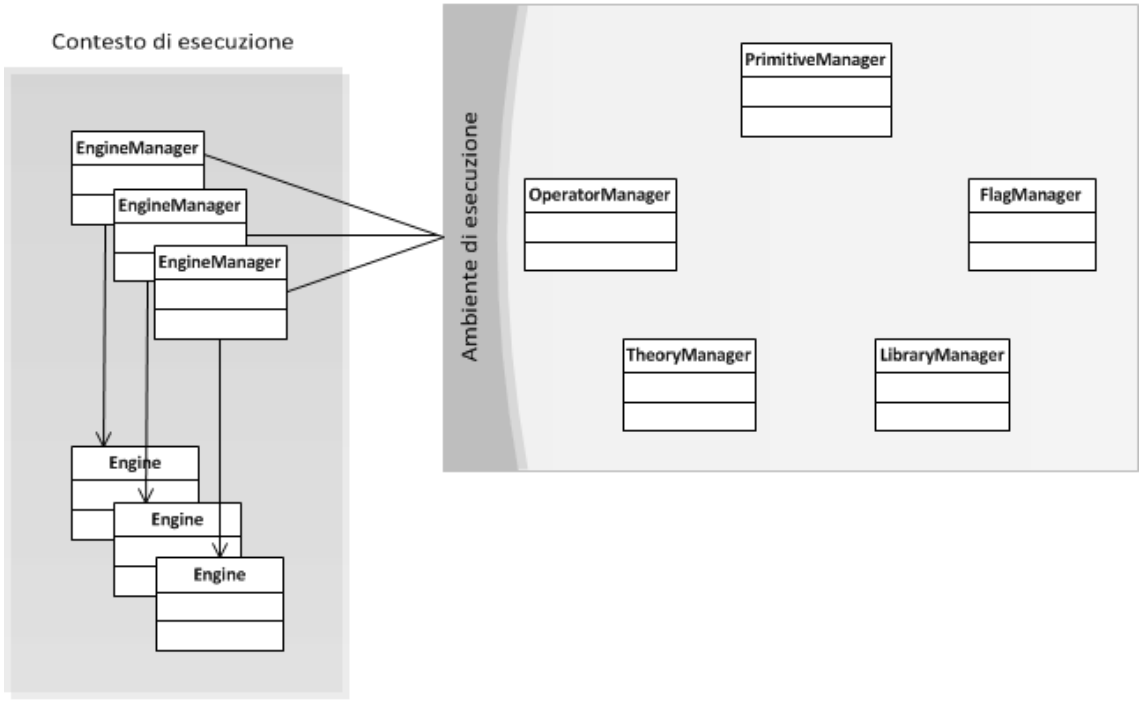

Figura 2.2–1 L'ambiente di esecuzione costituito dai cinque manager è unico all'interno del sistema, mentre gli EngineManager sono replicati, uno per ogni processo.

Per gestire la creazione dei processi e la computazione parallela di query, viene definito un nuovo componente chiamato *ThreadManager*, poiché non si vogliono attribuire ulteriori responsabilità alla classe Prolog, bensì rendere la concorrenza il più trasparente possibile al motore.

ThreadManager si occupa di generare un nuovo thread e un nuovo contesto di esecuzione per ogni richiesta di risoluzione di una query.

I thread sono rappresentati dalla classe *EngineRunner* che possiede sia un riferimento a EngineManager che a ThreadManager, il quale è quindi composto da un'aggregazione di EngineRunner, come mostrato nella figura 2.2-2. Ogni thread è identificato all'interno del sistema attraverso un *id* univoco necessario per risalire al giusto contesto di esecuzione durante il processo di unificazione: ThreadManager deve contenere una struttura dati che mappi l'identificatore di un thread nel corrispondente EngineRunner, in modo tale che sia possibile recuperare il contesto di esecuzione corretto a partire dall'id del thread corrente.

La presenza di un ulteriore manager suggerisce di realizzare un'astrazione per la classe *EngineManager* e di definirne due diverse implementazioni: la prima corrisponde a

ThreadManager, mentre la seconda coincide con la classe concreta EngineManager preesistente che viene rinominata in *BasicEngineManager*.

Inoltre viene aggiunta un'estensione a Prolog, la classe *CProlog* (*ConcurrentProlog*), che rappresenta il modello concorrente: se il motore viene istanziato tramite la classe Prolog, l'architettura risulta essere esattamente quella dell'implementazione standard dell'interprete, mentre se viene istanziato mediante la classe CProlog, si fa riferimento alla nuova architettura introdotta.

Poiché si tratta di un'architettura basata sul paradigma del parallelismo esplicito, devono essere definiti gli strumenti di alto livello per la gestione del multithreading. Viene introdotto un componente che incapsula tutte le funzionalità messe a disposizione dell'utente: la *ThreadLibrary*. Ogni chiamata di libreria fa direttamente riferimento al ThreadManager che si occupa della reale esecuzione di ciascuna di esse.

Nella nuova architettura è previsto anche un sistema di protezione in cui ogni processo deve essere dotato dei diritti di lettura o modifica per poter accedere alle informazioni relative ad altri thread. Il meccanismo di protezione non è concentrato all'interno di un unico componente, bensì distribuito in diverse parti del sistema.

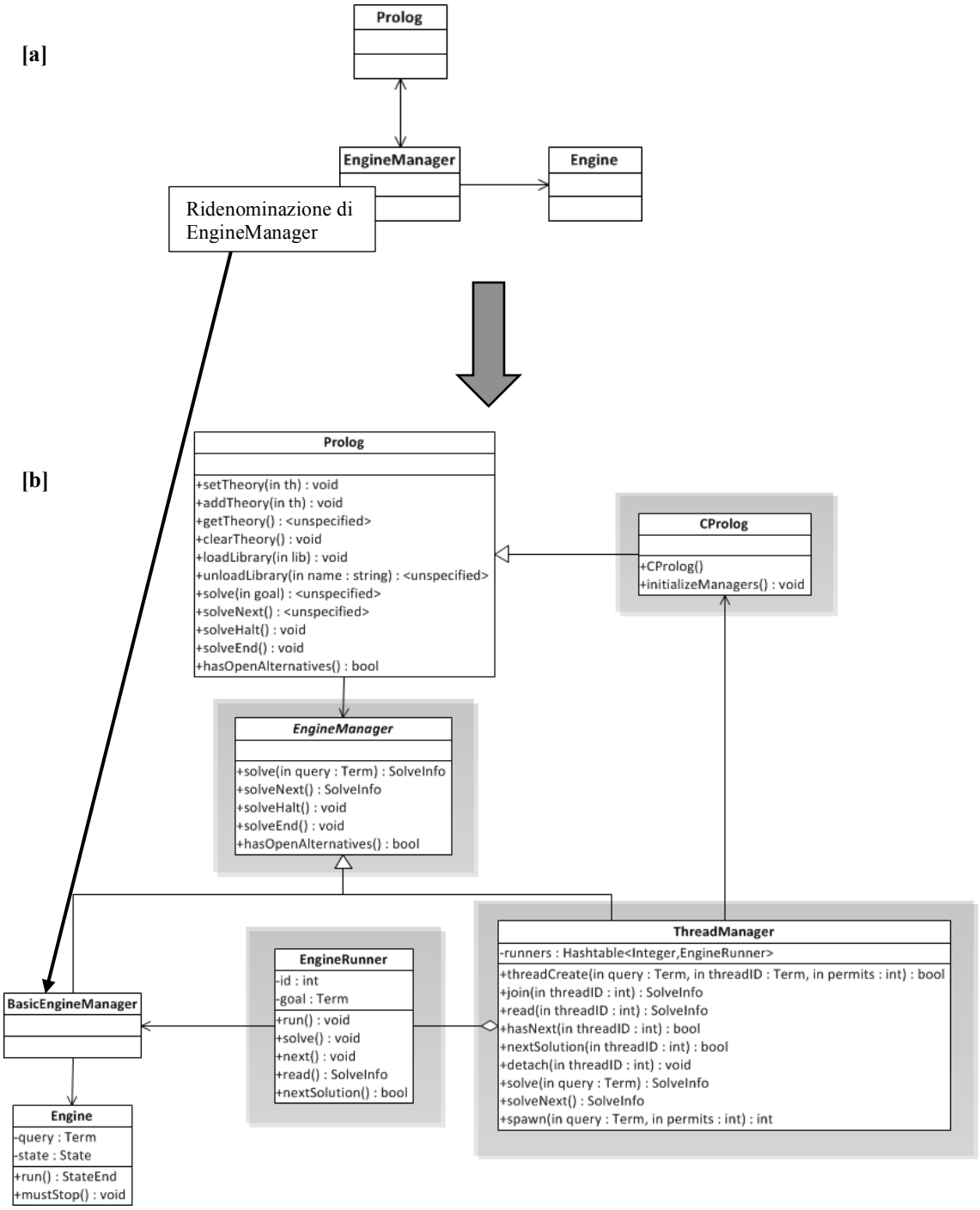

Figura 2.2–2 Confronto tra la struttura dell'interprete nella versione 2.6 [a] e il precedente prototipo [b]. Gli elementi evidenziati in grigio corrispondono ai nuovi componenti aggiunti.

Come discusso nel paragrafo 2.1, in questo modello più processi *cooperano* (in tempi diversi) per trovare le soluzioni alternative di una query: una volta che un processo ha terminato la computazione della prima soluzione, esso non viene sospeso in attesa di una richiesta di soluzione alternativa, bensì conclude la sua esecuzione e all'interno del

sistema viene tenuta traccia del suo contesto di esecuzione. Se poi si presenta un'ulteriore richiesta di computazione relativa alla stessa query, per servirla (se la soluzione alternativa esiste) viene generato un nuovo thread a cui viene associato il contesto di esecuzione già esistente. Quando l'utente fa riferimento all'identificativo del thread generato per risolvere quella determinata query, in realtà non si sta riferendo a un singolo thread, ma all'*insieme dei processi* che cooperano per risolvere la query e quindi a un determinato contesto di esecuzione.
Riassumendo, durante il processo di computazione di una query, all'interno del sistema, sono presenti tanti EngineRunner quante sono le soluzioni richieste ed un unico BasicEngineManager per quella query che rappresenta il riferimento al contesto di esecuzione, trasferito da un processo al successivo. ThreadManager gestisce tutti gli EngineRunner e implementa i metodi che consentono di associare il corretto EngineRunner al thread corrente.

2.3 Strumenti di alto livello

Gli strumenti di cui tipicamente è dotato un modello multithreading riguardano la comunicazione e la sincronizzazione tra i processi, i quali possono interagire per due principali ragioni:

- per accedere a risorse comuni coordinandosi tra di loro → *sincronizzazione*
- cooperando per il raggiungimento di un unico obiettivo → *comunicazione*

L'accesso a una risorsa comune deve essere possibile per un solo processo alla volta e per evitare conflitti è necessario che le interazioni tra i thread siano regolate in modo da serializzare gli accessi a tale risorsa. A questo scopo vengono definiti dei mutex, attraverso i quali l'utente può gestire in maniera esplicita le interazioni tra i processi che sono in competizione, ordinando gli accessi alle risorse.
La cooperazione, invece, è caratterizzata dalla comunicazione tra i processi: si sfrutta il modello delle code, secondo il quale i thread possono scambiarsi dei messaggi mediante delle operazioni di deposito e prelievo. Ad ogni processo è associata una coda e l'utente ha la possibilità di definirne di nuove. L'accesso alle code non avviene liberamente, in quanto il sistema di protezione verifica che il thread possieda i diritti di modifica o lettura: se un processo deve depositare un messaggio in una coda, necessita dei diritti di modifica per quella coda, mentre il prelievo del messaggio richiede i diritti di lettura.

Per quanto riguarda le funzionalità di base richieste da un modello multitasking, sono previsti predicati di creazione e distruzione di thread, per la lettura di soluzioni e per effettuare richieste di computazioni successive.
Di seguito sono brevemente presentati alcuni dei predicati implementati che svolgono le funzioni basilari di gestione dei processi (per un approfondimento si veda il capitolo 5).

- `thread_create/2(Query, ThreadId)`
  Crea un nuovo thread di identificatore `ThreadId` e comincia ad eseguire la `Query` data.

- `thread_id/1(ThreadId)`
  Tenta di unificare l'identificativo del thread corrente a `ThreadId`.

- `thread_join/2(ThreadId, Result)`
  Aspetta la terminazione del thread di identificatore `ThreadId` e ne raccoglie il risultato, unificando il goal risolto a `Result`. Il thread viene eliminato dal sistema.

- `thread_read/2(ThreadId, Result)`
  Come `thread_join/2`, ma il thread di identificatore `ThreadId` non viene eliminato dal sistema. Esso rimane disponibile per altre letture successive o per ulteriori computazioni.

- `thread_detach/1(ThreadId)`
  Pone il thread di identificatore `ThreadId` nello stato *detached*. I thread in questo stato non possono essere attesi dagli altri thread mediante `thread_join/2` o `thread_read/2`, né possono essere oggetto di `thread_next_sol/2`; quando terminano non lasciano alcuna traccia della loro computazione.

- `thread_next_sol/1(ThreadId)`
  Forza il thread `ThreadId` a riprendere la computazione, esplorando soluzioni alternative della query iniziale.

- `thread_has_next/1(ThreadId)`
  Ha successo se il goal del thread di identificatore `ThreadId` ha soluzioni rimanenti.

- `thread_execute/2(Query, ThreadId)`
  Come `thread_create/2`, ma tale predicato è backtrackabile.

CAPITOLO 3

IL NUOVO PROTOTIPO

Il prototipo precedente presenta diversi punti deboli sia a livello logico che strutturale. Si consideri, ad esempio, il flusso di esecuzione dei thread nell'architettura appena analizzata: viene creato un processo per ogni richiesta di soluzione di una query. Appare chiaro come il numero dei thread generati nel sistema sia eccessivo: il modello computazionale è decisamente pesante rispetto al livello di astrazione iniziale, in cui un thread doveva essere associato all'insieme delle soluzioni di una query. La direzione che si vuole prendere per il nuovo prototipo mira a colmare questo gap logico-implementativo, favorendo un approccio che tenga in considerazione le prestazioni del sistema.

3.1 Possibili approcci

Il nuovo requisito non comporta alcuna variazione dei concetti di ambiente e contesto di esecuzione, in quanto il processo di risoluzione è comunque dipendente dalle informazioni condivise tra tutti i thread e l'effetto del processo di unificazione rimane circoscritto al contesto di esecuzione del thread che l'ha effettuato.
È possibile pensare a diverse soluzioni compatibili con l'esigenza di generare un numero di thread limitato, il cui ciclo di vita inizi con la computazione della prima soluzione della query ed eventualmente prosegua con l'esplorazione delle successive soluzioni fino al loro esaurimento.

Una prima possibilità è quella di mantenere la struttura del prototipo precedente. In tal caso, il nuovo requisito potrebbe essere soddisfatto semplicemente rendendo biunivoca l'associazione tra EngineRunner e contesto di esecuzione, cambiando l'implementazione di ThreadManager. Quest'ultimo dovrebbe generare un thread solo quando si presenta l'iniziale richiesta di risoluzione di una query e mantenerlo in vita in attesa di altre richieste, senza dover trasferire il contesto di esecuzione da un processo al successivo.

Per regolare l'esecuzione parallela di thread è necessario introdurre un modello di sincronizzazione all'interno di ThreadManager. Bisogna assicurare che i thread siano sospesi una volta computata la soluzione e risvegliati all'arrivo di una nuova richiesta di computazione per quella query.

Questa prima soluzione risulterebbe essere molto accattivante, in quanto non incide in maniera profonda sulla struttura del precedente prototipo, bensì effettua un intervento solo sul ThreadManager. L'architettura risultante avrebbe però diversi problemi a livello concettuale.

Infatti, l'ereditarietà nella gerarchia dei manager viola il principio di sostituibilità di Liskov, secondo cui gli oggetti appartenenti a una sottoclasse devono essere in grado di esibire tutti i comportamenti e le proprietà esibiti da quelli appartenenti alla superclasse. Teoricamente il cliente (in questo caso Prolog) dovrebbe essere in grado di utilizzare istanze di classi derivate dalla classe base EngineManager senza conoscerne il tipo.

Questo concetto non verrebbe assolutamente rispettato in questa architettura, in quanto le due classi che ereditano, BasicEngineManager e ThreadManager, dovrebbero esporre a Prolog le stesse funzionalità. Al contrario, esse lavorano in due ambiti completamente separati: il primo manager si occupa della gestione dei thread, mentre l'altro della computazione vera e propria delle query.

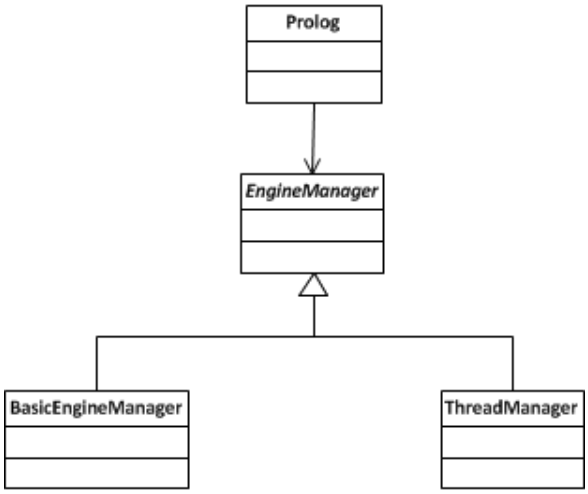

Figura 3.1–1 La gerarchia dei manager.

Inoltre, poiché BasicEngineManager fa parte del contesto di esecuzione del thread, non è opportuno che la classe Prolog sia in diretto contatto con tutta la struttura relativa alla computazione di query perché il compito di ThreadManager è proprio quello di mascherare i dettagli del singolo processo a Prolog.

L'unico vantaggio di questo genere di gerarchia è la possibilità di tornare al modello sequenziale dell'interprete nel caso in cui venga istanziato il motore Prolog piuttosto che CProlog, durante l'inizializzazione del sistema. Si torna esattamente all'attuale architettura sequenziale perché tutta la parte relativa al parallelismo non viene più

sfruttata. La decisione sul tipo di motore da istanziare, però, viene presa a livello di compilazione e non risulta essere una funzionalità utile perché non è a disposizione dell'utente.
Nel nuovo prototipo sarà necessario decidere come gestire le due classi, se farne una unica oppure tenerle separate e rendere la funzionalità di più alto livello, così che in base alla scelta dell'utente si possa istanziare un motore piuttosto che l'altro.

Questi problemi suggeriscono di identificare un'architettura diversa, che confini le modifiche internamente a ThreadManager. A fronte di queste considerazioni, risulta quindi evidente la necessità di sviluppare un diverso modello strutturale.

Una prima possibilità potrebbe essere quella di adottare un approccio più conservativo nel quale tornare alla struttura dell'interprete attuale dove tutta la concorrenza sia incapsulata all'interno di ThreadLibrary. La figura del ThreadManager verrebbe inglobata all'interno della libreria che si occuperebbe della reale esecuzione di ogni singola chiamata da parte del programmatore.
In questo scenario occorrerebbe prevedere una struttura che tenga traccia della corrispondenza tra thread (attraverso il suo identificatore) ed EngineManager, alla quale Prolog possa accedere durante l'esecuzione per risalire al contesto di esecuzione corretto.

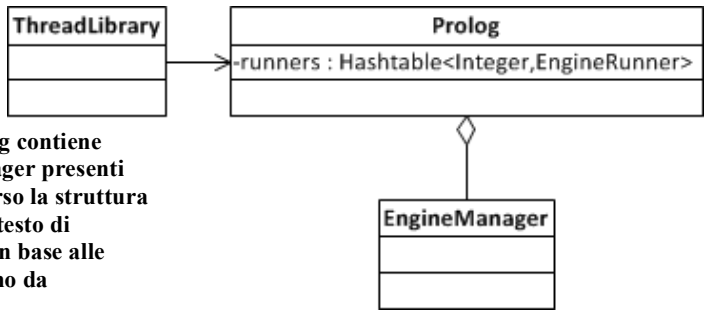

Figura 3.2.1–2 Prolog contiene tutti gli EngineManager presenti nel sistema. Attraverso la struttura *runners* risale al contesto di esecuzione corretto in base alle richieste che giungono da ThreadLibrary.

In alternativa alla struttura dati, la classe Prolog potrebbe contenere un riferimento alla libreria, la quale avrebbe il compito di fornirgli l'istanza del manager che deve occuparsi della computazione.

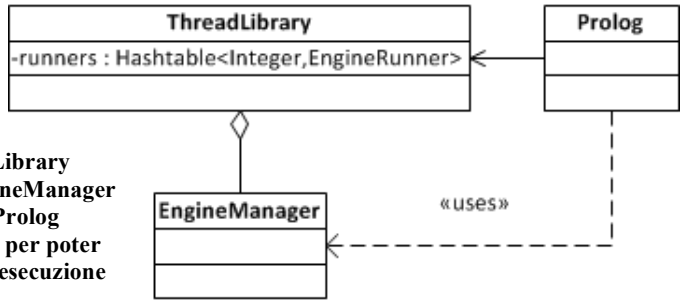

Figura 3.1–3 ThreadLibrary contiene tutti gli EngineManager presenti nel sistema. Prolog dipende dalla libreria per poter risalire al contesto di esecuzione corretto.

Il grande svantaggio di questo scenario che prevede che tutto il parallelismo sia inglobato nella ThreadLibrary, è che la concorrenza non sarebbe più trasparente a Prolog, poiché quest'ultimo dovrebbe mantenere, nel primo caso, un riferimento alla struttura dati che riconosce i contesti di esecuzione oppure, come alternativa, un riferimento a ThreadLibrary. Inoltre, appare inopportuno incapsulare tutta la concorrenza in un unico componente in quanto ciò non rispetterebbe il ruolo che le librerie hanno in tuProlog e contemporaneamente verrebbe meno tutto il sistema di divisione delle responsabilità tra manager.

Per questi motivi, sembra in definitiva più conveniente sviluppare il nuovo prototipo a partire da quello precedente, cercando di risolvere in maniera opportuna i diversi problemi che attualmente presenta.

3.2 Architettura

Innanzitutto è necessario ragionare sulla gerarchia dei manager: considerando la separazione delle responsabilità all'interno del sistema e il ruolo che gioca ogni singolo manager, è opportuno delegare a uno solo di essi il compito di occuparsi della computazione di query. Di conseguenza le relazioni che intercorrono tra Prolog e i due manager dovranno essere riviste e sostituite con un modello più valido, facendo scomparire la gerarchia.

Nel prototipo precedente, l'assenza di un'associazione biunivoca tra EngineRunner e BasicEngineManager, ha portato a una separazione tra le due classi: esternamente il flusso di esecuzione risulta omogeneo, ma in realtà più thread condividono lo stesso contesto di esecuzione senza sovrapporsi nel tempo.

Nel nuovo prototipo invece, si vuole che tra processo e contesto di esecuzione esista un'associazione biunivoca: per questo motivo ogni thread dovrà disporre di una propria unità di elaborazione indipendente dalle altre per poter risolvere una query.

Per coniugare l'eliminazione della gerarchia e l'esigenza di un unico manager con la dipendenza reciproca tra BasicEngineManager ed EngineRunner, la scelta più logica appare quella di riunire le due classi: una versione riprogettata di EngineRunner implementerà tutte le funzionalità di BasicEngineManager e in particolare la capacità di computare query e di gestire la macchina a stati. Questa unione è giustificata dal fatto che ogni contesto di esecuzione *appartiene* ad uno specifico thread.

Per quanto riguarda la classe ThreadManager, essendo il *motore* di tutta la computazione parallela di query, è stata rinominata in *EngineManager* come nell'interprete attuale, anche se a livello concettuale sono differenti.

Naturalmente il nuovo manager dovrà essere adattato alla nuova architettura e saranno apportate modifiche rispetto alla classe ThreadManager del precedente prototipo.

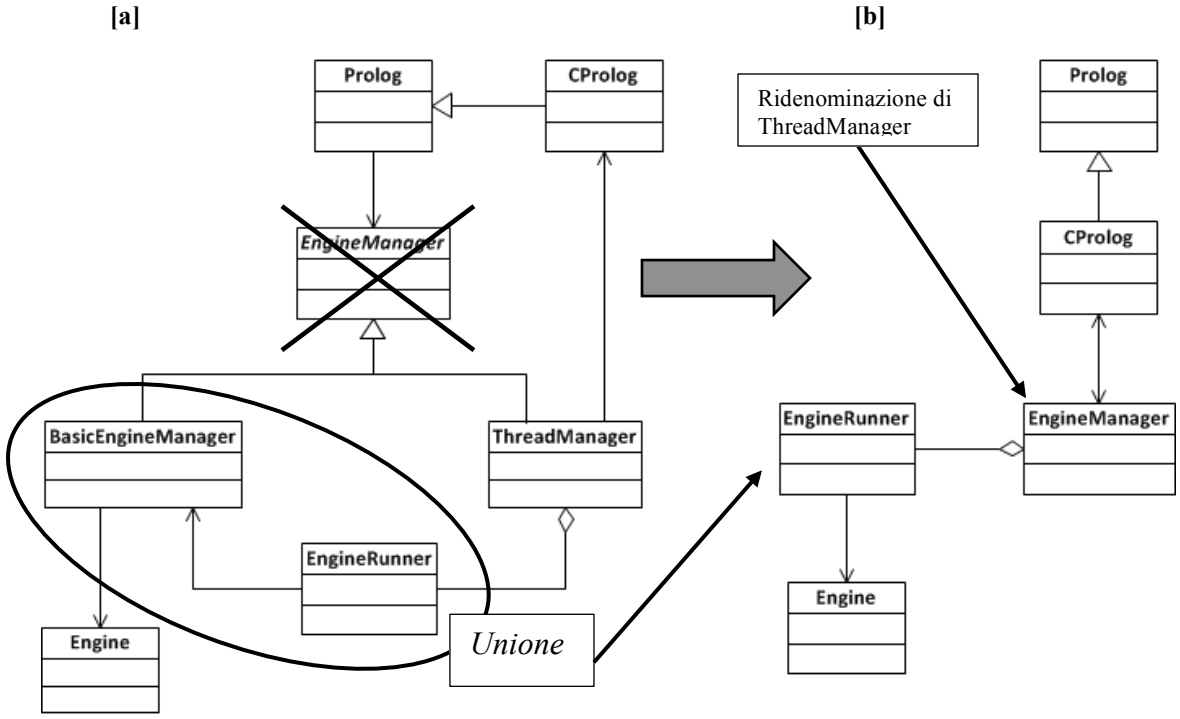

Figura 3.2–1 Confronto tra l'architettura del precedente prototipo [a] e quella nuova [b].

- **La gerarchia viene eliminata.**
- **BasicEngineManager viene inglobato all'interno di EngineRunner.**
- **ThreadManager viene rinominato in EngineManager.**

La nuova struttura appare più lineare, priva di ridondanze.

3.2.1 Il processo di unificazione

In tuProlog l'unificazione viene eseguita all'interno di uno dei due termini, al quale il motore Prolog deve fornire il corretto contesto di esecuzione.
Nella nuova architettura, Prolog delega tale ricerca al manager di competenza, cioè EngineManager. Quest'ultimo deve essere in grado di reperire l'istanza corretta di EngineRunner su cui eseguire l'unificazione in maniera trasparente al motore Prolog.

Un processo può essere riconosciuto attraverso due tipi di identificativi:

- L'*id* che corrisponde all'identificativo numerico che viene assegnato al processo nel momento della sua creazione tramite `thread_create/2` e con il quale l'utente può far riferimento al processo nelle chiamate di libreria.
- Il *pid* che è l'identificativo del processo assegnato dalla macchina virtuale java.

Sfruttando questa conoscenza, EngineManager può risalire al thread di cui si cerca il contesto di esecuzione, attraverso l'introduzione di due diverse strutture dati:

- *Runners* mappa l'id di un certo thread nel relativo EngineRunner; mantiene le istanze di EngineRunner che rappresentano i processi in esecuzione, cioè quei thread che sono disponibili per essere manipolati dall'utente. L'id viene rimosso dalla struttura solo nel caso in cui il thread si trovi in stato detached oppure in seguito ad un istruzione di join, per cui il processo viene rimosso dal sistema.
- *Threads* mappa il pid di un certo processo nel relativo id; mantiene l'identificativo di tutti i processi che sono stati generati, sia quelli in esecuzione che quelli che hanno concluso il loro servizio. Attraverso tale struttura, si può ricavare l'istanza di EngineRunner che corrisponde al processo in esecuzione a partire dal pid del thread corrente, ottenibile usando il metodo statico `Thread.currentThread().getId()`, e risalendo all'id corrispondente.

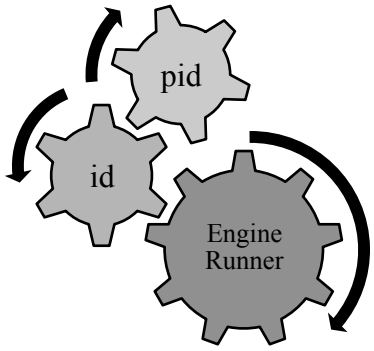

**Figura 3.2.1–1 Combinazione delle strutture *threads* e *runners*.
A partire dalla conoscenza del pid di un processo, si può risalire al suo id e di conseguenza all'istanza di EngineRunner corrispondente.**

Con la creazione di un nuovo thread, il suo id e il suo pid vengono inseriti nelle strutture dati corrispondenti.
In conclusione, quando si cercherà di effettuare un'unificazione, sarà possibile risalire al contesto di esecuzione corretto semplicemente interrogando la macchina virtuale circa il pid del thread corrente.

3.2.2 La modalità sequenziale

I principali compiti di EngineManager riguardano:

- la gestione delle strutture dati runners e threads;
- l'implementazione dei metodi che consentono di associare il corretto EngineRunner al thread corrente o a quello che è stato specificato dall'utente;
- l'esecuzione di tutte le chiamate che provengono da Prolog o da ThreadLibrary.

In base alla provenienza della chiamata, EngineManager gestisce la modalità sequenziale o parallela dell'interprete, in modo trasparente al motore Prolog. Se l'utente non sfrutta le librerie adibite al multitasking, il sistema lavora in maniera sequenziale esattamente come in tuprolog classico, senza overhead e le chiamate in tal caso provengono da Prolog. A tale scopo la modalità sequenziale viene mascherata mediante la creazione di un unico processo: EngineManager genera un EngineRunner che rappresenta il processo di base, chiamato *root,* con id predefinito pari ad uno. Esso viene poi aggiunto ad entrambe le strutture runners e threads per poter identificare successivamente il processo.

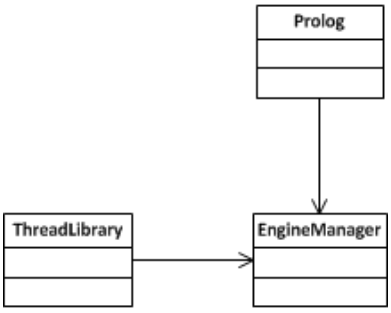

Figura 3.2.2–1 Prolog e ThreadLibrary dipendono da EngineManager.

Per com'è stata progettata la nuova architettura, la definizione di due motori separati (Cprolog e Prolog), che ereditano l'uno dall'altro, risulta ridondante dato che l'unica differenza tra le due classi riguarderebbe a questo punto solo il caricamento della ThreadLibrary. Per questo motivo si è deciso, analogamente a quanto fatto per BasicEngineManager e EngineRunner, di unire le funzionalità in un'unica classe Prolog che, insieme a tutte le librerie di default, carichi anche la ThreadLibrary.
L'architettura finale è illustrata in figura 3.2.2-2.

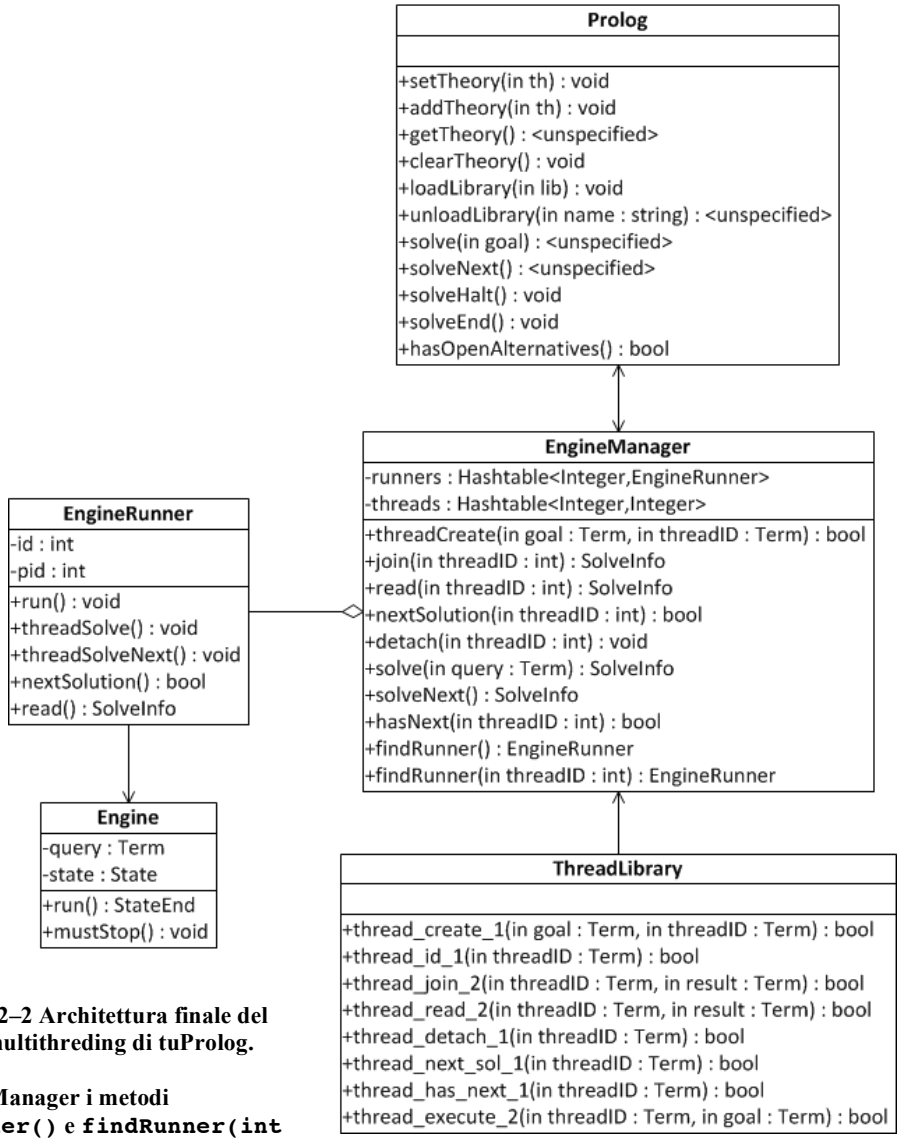

Figura 3.2.2–2 Architettura finale del supporto multithreding di tuProlog.

In EngineManager i metodi `findRunner()` e `findRunner(int threadID)` servono per trovare il corretto EngineRunner, a partire rispettivamente dalle strutture threads e runners.

Della ThreadLibrary sono mostrati solo le primitive di base.

3.2.3 Possibili estensioni future

Come già accennato, il precedente prototipo è dotato di un sistema di sicurezza per regolare gli accessi in lettura e modifica di un thread nei confronti di un altro. In realtà, queste funzionalità non risultano gestite da un vero e proprio componente che incapsuli tali responsabilità: al contrario, il controllo degli accessi è sparso in tutto il sistema.

Un simile approccio appare chiaramente in contrasto con i principi strutturali alla base di tuProlog.

Nell'architettura del prototipo precedente, inoltre, ThreadLibrary offre delle primitive con le quali l'utente può definire i diritti per un processo durante la sua creazione e altre primitive per la cessione di diritti ad altri processi. Pertanto il sistema di controllo si occupa solamente di eseguire ciò che viene definito dall'utente, lasciando sulle sue spalle tutta la responsabilità della sicurezza.

L'utente, quindi, detta le leggi di accesso che regolano le interazioni tra i thread e il sistema che le esegue: la considerazione che sorge spontanea è che l'utente non ha in realtà bisogno del supporto da parte del sistema per gestire questo basso livello di sicurezza. Pertanto il meccanismo di protezione è stato escluso dall'implementazione del nuovo prototipo.

Un'estensione futura che vada verso questa direzione, potrebbe prevedere la realizzazione di uno specifico componente che abbia la responsabilità di gestire la sicurezza attraverso la definizione di specifici ruoli, i quali verrebbero assegnati ai vari thread. Per realizzare una struttura di questo tipo, è necessario conoscere a priori quale sarà la natura dei singoli processi e in alcuni ambiti può anche rivelarsi un limite per il sistema.

CAPITOLO 4

INTERAZIONI TRA I PROCESSI

Esistono due forme di interazione tra i processi:

a) I processi che interagiscono in maniera *indiretta* sono quei processi che potrebbero evolvere in modo indipendente rispetto all'esecuzione di altri thread, ma che non lo possono fare perché competono per l'uso di risorse che devono essere utilizzate da ciascuno in modo esclusivo. Si parla di *competizione* e in generale un solo processo alla volta deve avere accesso agli oggetti condivisi (*mutua esclusione*).
b) I processi che interagiscono in maniera *diretta* sono quei processi che *cooperano* per raggiungere un determinato obiettivo. Se tale cooperazione prevede uno scambio di messaggi, l'interazione consiste anche in una *comunicazione* tra i processi.

Esiste un'altra forma di interazione tra due processi che va sotto il nome di *interferenza* provocata da una erronea soluzione a problemi di cooperazione e competizione. Caratteristica di ogni forma di interferenza è che il manifestarsi dei propri effetti erronei dipende dai rapporti di velocità tra i processi. Una scorretta sincronizzazione dei processi può dar luogo ad errori che dipendono dal tempo e con ciò si intende dire che tali errori non si riproducono necessariamente riavviando il sistema con le stesse condizioni iniziali. Come si vedrà, obiettivo fondamentale degli strumenti di sincronizzazione è proprio quello di evitare condizioni di interferenza tra i processi.

4.1 Sincronizzazione dell'ambiente di esecuzione

L'accesso a un ambiente condiviso da parte di diversi processi rende necessaria l'introduzione di vincoli per l'esecuzione delle operazioni.

Nell'interprete tuProlog l'ambiente di esecuzione è stato individuato come l'insieme dei manager che includono le informazioni condivise tra tutti i thread.

La sincronizzazione degli accessi nell'intero ambiente nasce dall'esigenza di garantire una corretta competizione tra i processi, evitando che si verifichino condizioni di deadlock.

Per prevenire situazioni di errore a causa delle interazioni tra i processi all'interno dell'ambiente di esecuzione, si deve intervenire su:

- il tipo di strutture dati utilizzate per garantire la consistenza dei dati;
- l'ordine con cui i processi eseguono le operazioni, imponendo dei vincoli per evitare condizioni di deadlock. Ad esempio, se un qualsiasi blocco di codice sincronizzato (all'interno di un manager) richiama un metodo "synchronized" di un manager che nell'ordine stabilito lo precedeva, si crea un circolo per cui i due processi che stanno eseguendo si bloccano a vicenda, aspettando che uno esegua l'operazione che serve all'altro per proseguire e viceversa.

4.1.1 Sincronizzazione dei manager

Di seguito viene effettuata l'analisi dei singoli componenti che formano l'ambiente di esecuzione al fine di definire le modifiche necessarie in ciascuno di essi per far fronte agli eventuali problemi di sincronizzazione.

`TheoryManager`

Una delle funzioni del manager riguarda la creazione e la gestione del database delle clausole che formano la teoria caricata nel sistema. Le clausole sono mantenute in una struttura dati di tipo *ClauseDatabase*, basata sulla classe *java.util.HashMap* che è priva di sincronizzazione interna.

È necessario quindi sincronizzare tutti i metodi che modificano la teoria, quelli che aggiornano lo stato della struttura e i metodi di lettura dal database per garantire la consistenza dei dati presenti nel manager.

`LibraryManager`

L'insieme delle librerie caricate nel sistema è contenuto all'interno di una struttura di tipo *ArrayList<Library>* i cui accessi sono già sincronizzati.

Per quanto riguarda i metodi che richiamano altri manager, i riferimenti a PrimitiveManager e TheoryManager non possono creare condizioni di deadlock, in quanto nessuna chiamata da parte dei due manager può condurre a LibraryManager.

`OperatorManager`

La struttura destinata a contenere la lista degli operatori definiti è di tipo *OperatorRegister*, che si basa sulla classe *java.util.LinkedHashSet*, priva di sincronizzazione: i metodi che prevedono un accesso in lettura o scrittura all'insieme degli operatori devono perciò essere sincronizzati.
Tuttavia, non essendovi alcun riferimento ad altri manager, non si corre alcun rischio di deadlock.

`PrimitiveManager`

Avendo il compito di gestire le primitive presenti nella teoria in base alla loro natura di predicati, funtori o direttive, il manager deve avere a disposizione più strutture dati, ciascuna destinata a una diversa tipologia di primitiva. L'attuale implementazione sfrutta le classi non sincronizzate *java.util.HashMap* e *java.util.IdentityHashMap* ma, al contrario di TheoryManager, in questo caso non è possibile sincronizzare direttamente l'accesso alle strutture a causa della presenza dei metodi di valutazione delle primitive, cioè `evalAsPredicate()`, `evalAsFunctor()` e `evalAsDirective()` , che possono invocare i metodi di qualsiasi libreria. Una possibile soluzione consiste nell'inglobare tali strutture all'interno di oggetti internamente già sincronizzati, come *java.util.Collections.synchronizedMap*.
Non essendo presenti riferimenti ad oggetti esterni, non possono verificarsi situazioni di deadlock.

I problemi legati alla sincronizzazione hanno effetto anche sulla classe *PrimitiveInfo*, utilizzata da PrimitiveManager per rappresentare le informazioni su ogni predicato, funtore o primitiva che gli viene fornita dal sistema: queste informazioni saranno utilizzate per la valutazione dei predicati nello stato GoalEvaluation. Durante il processo di risoluzione, se più thread analizzano lo stesso predicato, l'operazione di valutazione viene eseguita chiamando il metodo `evalAsPredicate()` della stessa istanza di PrimitiveInfo che rappresenta quello specifico predicato, e lo stesso accade per `evalAsFunctor()` e `evalAsDirective()`. Poiché questi metodi, insieme a quello che si occupa di invalidare le primitive, contengono variabili d'ambiente, è necessario sincronizzarne l'accesso.

`FlagManager`

È necessario sincronizzate le operazioni di accesso in lettura e scrittura alla struttura dati di tipo *ArrayList<Flag>* che contiene l'insieme dei flag definiti nel sistema.

4.1.2 Sincronizzazione dei mediatori

Oltre all'ambiente di esecuzione formato dai cinque manager, devono essere sottoposti ad analisi anche EngineManager e lo stesso motore Prolog, poiché entrambi sono soggetti a molte chiamate provenienti dall'esterno e per questo ricoprono il ruolo di mediatori all'interno del sistema. In particolare, il primo fa da tramite per le chiamate provenienti da ThreadLibrary e i diversi EngineRunner presenti nel sistema, mentre il secondo gestisce le chiamate di libreria che sono dirette verso i manager.

`EngineManager`

Le quattro strutture dati che servono rispettivamente per l'identificazione dei thread (*runners* e *threads*) e la gestione di code e semafori (*queues* e *locks*), derivano tutte dalla classe *java.utile.Hashtable.* Come più volte accennato, essendo tale classe priva di sincronizzazione interna, è necessario regolare gli accessi in ciascuna struttura sia in lettura che in scrittura per garantire la consistenza dei dati.

Inoltre, per evitare interferenze durante la creazione di un thread, visto che si tratta di un operazione atomica, è necessario sincronizzare anche il metodo `threadCreate(Term goal, Term threadID)`, facendo in modo che prima che l'utente abbia la possibilità di interagire con il thread, il sistema abbia completato il processo di generazione.

Anche i metodi `solve()` e `solveNext()` che agiscono sulla computazione del processo base root, a cui si fa riferimento per la modalità sequenziale, necessitano di sincronizzazione perché attualmente non sono supportate richieste contemporanee di risoluzione da parte dell'utente.

`Prolog`

Nell'attuale versione di tuProlog, per non rischiare che lo stato dei manager risulti inconsistente a causa delle diverse chiamate che provengono dall'esterno, i metodi della classe Prolog risultano tutti sincronizzati.

La nuova architettura esclude a priori la possibilità di incorrere in tale rischio perché, come si è visto, la sincronizzazione viene gestita a livello più basso, cioè nei singoli manager.

Concentrare tutta la sincronizzazione all'interno del motore in un modello concorrente, porterebbe a un'eccessiva sequenzializzazione durante l'esecuzione: essendo Prolog il nodo centrale della comunicazione tra i manager, se ogni metodo del motore fosse sincronizzato, tutte le interazioni dovrebbero avvenire in maniera sequenziale e ciò condurrebbe a una completa paralisi del sistema.

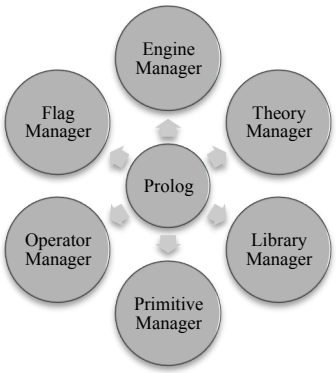

Figura 4.1.2–1 La sincronizzazione non è più concentrata nel motore Prolog, ma viene trasferita all'interno dei singoli manager.

4.2 Sincronizzazione delle esecuzioni parallele di thread

Si consideri il caso di due processi P1 e P2 che si scambiano informazioni tramite un'area di memoria comune (un buffer, una variabile …). Il processo P1 provvede ciclicamente a produrre un'informazione e ad inserirla nell'area di memoria prestabilita; P2, sempre ciclicamente, preleva l'informazione dalla memoria e la utilizza per i suoi scopi. Le fasi di produzione, inserimento, prelievo e utilizzo risultano essere l'una propedeutica all'altra e ogni volta che è necessario scambiarsi una nuova informazione, le quattro fasi si ripetono nuovamente: il processo di comunicazione è ciclico.

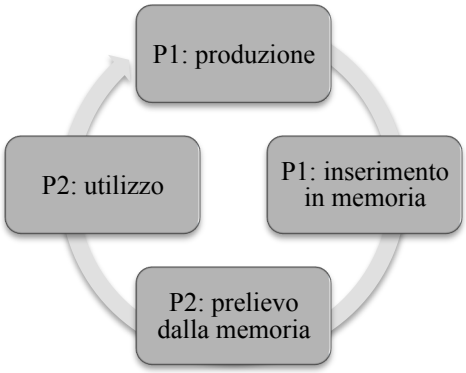

Figura 4.2–1 Cooperazione tra i processi P1 e P2: processo ciclico per lo scambio di informazioni.

Per ottenere una corretta cooperazione tra i processi che interagiscono, occorre introdurre dei vincoli attraverso i quali imporre che le operazioni di un processo non possano avere inizio prima di (o dopo) determinate operazioni di altri processi. In altre parole è necessario sincronizzare le esecuzioni parallele dei thread: ad esempio, bisogna garantire che l'operazione di prelievo sia sempre preceduta dall'operazione di inserimento.

4.2.1 Specifiche del protocollo

Rifacendosi al modello appena descritto relativo ai due processi P1 e P2, nel caso specifico del nuovo prototipo, i ruoli di *produttore* e *lettore* possono essere identificati in quei processi che producono le diverse soluzioni di una query e nei thread che prelevano le soluzioni e le sfruttano per i propri scopi.

Nel caso di computazioni Prolog, però, non esiste soltanto l'interazione tra queste due tipologie di processi: bisogna considerare anche l'interazione necessaria per forzare la computazione di una nuova soluzione nei confronti di un produttore.

Per questo motivo, i processi possono assumere tre differenti ruoli, variabili nel tempo, durante il loro flusso di esecuzione:

- Il *lettore* si occupa di prelevare le soluzioni (una per ogni istruzione di lettura) rese disponibili in seguito alla computazione di un produttore.
- Il *committente* ha il compito di riavviare l'esecuzione di un processo produttore, forzando una nuova computazione.
- Il *produttore* è il thread che sta risolvendo o ha risolto una query. In base alle richieste che giungono dal committente, cerca una nuova soluzione che viene eventualmente prelevata da un lettore interessato.

Si noti come le operazioni dei lettori e dei committenti facciano sempre riferimento ai produttori: le uniche due relazioni previste tra i processi, infatti, riguardano la cooperazione tra lettori e produttori oppure tra committenti e produttori. Il ruolo rappresentato dall'utente consisterà nell'interfacciarsi con i processi lettori e committenti per poter risalire, attraverso i primi, alle soluzioni computate e mediante i secondi l'utente può gestire le computazioni dei produttori. Dovranno, quindi, essere messe a disposizione dell'utente delle primitive con le quali possa dare istruzioni precise: `thread_read/2` e `thread_join/2` per l'interfacciamento con i lettori e `thread_next_sol/1` per i committenti. Naturalmente le chiamate di lettura si riferiscono all'ultima soluzione computata dal produttore.

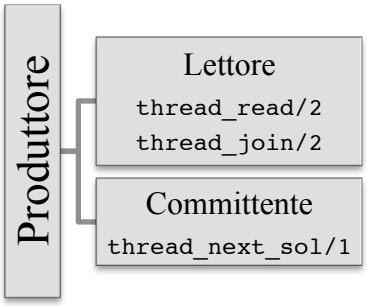

Figura 4.2.1–1 Relazioni tra processi (lettore/produttore e committente/produttore) e le relative primitive.

La necessità di regolare le interazioni tra i processi in base ai ruoli che possono assumere nel corso della computazione, porta alla definizione di diversi vincoli.

- Se il produttore sta eseguendo, esso non può compiere alcuna operazione.
- Se un lettore tenta di prelevare una soluzione, esso deve attendere la terminazione della computazione del produttore e la sua esecuzione è momentaneamente sospesa.
- Se un committente tenta di accedere al produttore, esso deposita la richiesta per la computazione successiva e prosegue con la sua esecuzione, senza essere sospeso.
- I lettori hanno sempre la precedenza sui committenti al termine di una computazione del produttore.

Per una corretta computazione, il produttore non può essere interrotto in nessun caso e solo quando il processo di risoluzione è concluso il lettore preleva la soluzione appena computata e il produttore prende in considerazione l'eventuale richiesta del committente, richiesta che comporta la ripresa della sua esecuzione.
Non è necessario sospendere anche i committenti, poiché la loro esecuzione non dipende dal risultato della computazione del produttore, come invece accade per i lettori.

Appena un produttore finisce di computare una soluzione, sono previsti i seguenti quattro casi.

1. Non sono presenti né lettori in attesa né richieste di successive computazioni. In questo caso:
   1.1. Se ci sono soluzioni alternative, il produttore si sospende in attesa di essere risvegliato per una nuova computazione.
   1.2. Se non ci sono soluzioni alternative, il produttore termina la sua esecuzione.

2. Sono presenti solo dei lettori in attesa di prelevare la soluzione appena computata:
   2.1. Il produttore li risveglia e subito dopo, in base alla presenza o meno di soluzioni alternative, percorre il punto 1.1 o 1.2.

2.2. I lettori prelevano contemporaneamente la soluzione senza il rischio di interferire tra loro.

3. Sono presenti solo richieste di nuove computazioni:
   3.1. Se ci sono soluzioni alternative, il produttore riprende immediatamente la sua esecuzione, servendo una tra le richieste dei committenti.
   3.2. Se non ci sono soluzioni alternative, il produttore termina la sua esecuzione ignorando la richiesta del committente.

Al contrario di quanto avviene con i lettori, le richieste dei committenti devono essere eseguite una per volta.

4. Sono presenti sia lettori in attesa che richieste di computazione da parte di committenti:
   4.1. Il produttore risveglia tutti i lettori in attesa di prelevare la soluzione appena computata e, in base alla presenza o meno di soluzioni alternative, percorre il punto 3.1 o 3.2.

I lettori, dunque, hanno sempre la precedenza sui committenti perché in caso contrario, rischierebbero di prelevare una soluzione sbagliata.

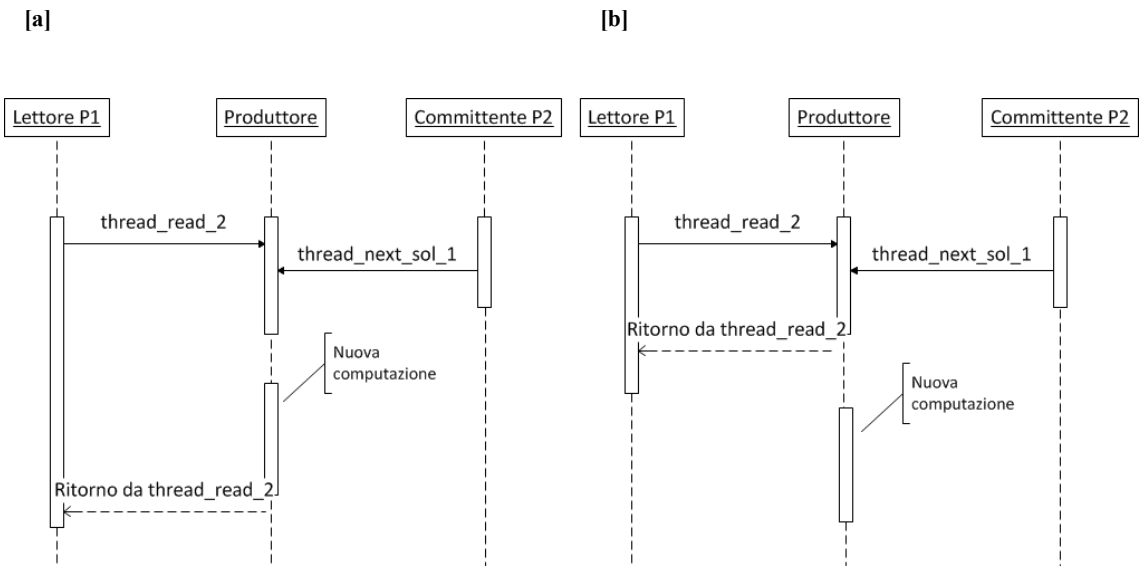

Figura 4.2.1–2 Confronto tra il caso in cui la precedenza è data ai committenti [a] e quello in cui sono serviti prima i lettori [b].

Si considerino un produttore che sta computando, un thread P1 che è in attesa di prelevare la soluzione corrente dal produttore mediante `thread_read/2` e un secondo thread P2 che richiede al produttore una nuova computazione con un'istruzione di `thread_next_sol/1`. Se la richiesta del committente avesse la precedenza rispetto al lettore, P1 preleverebbe la soluzione relativa alla computazione richiesta da P2 e non la soluzione corrente perché il produttore riprenderebbe immediatamente la sua esecuzione senza risvegliare P1. Inoltre, si dà la precedenza ai lettori, in quanto l'operazione di lettura è molto più veloce se confrontata con il processo di computazione di una soluzione.

Ultimo caso da considerare riguarda il produttore che non sta eseguendo alcuna computazione ed è in attesa della richiesta di un committente. In questo caso non c'è priorità tra lettori e committenti, i quali entrano in servizio in base all'ordine di chiamata:

- I lettori prelevano l'ultima soluzione computata dal produttore senza alterare il suo stato.
- Il committente risveglia il produttore sospeso per forzare la ripresa dell'esecuzione.

Per rendere più chiari i flussi di esecuzione di produttori, lettori e committenti, nelle figure 4.2.1 rispettivamente numero 3, 4 e 5 sono mostrati i diagrammi di stato che riassumono le loro fasi di esecuzione.
Lo stato *Termina* non si riferisce al thread, ma alla conclusione del *ruolo* assunto dal thread in quel momento, mentre lo stato *In esecuzione* comporta che il processo stia eseguendo la funzionalità prevista dal ruolo assunto: il produttore *sta computando* una soluzione, il lettore *sta prelevando* una soluzione e il committente *sta depositando* la richiesta di computazione.

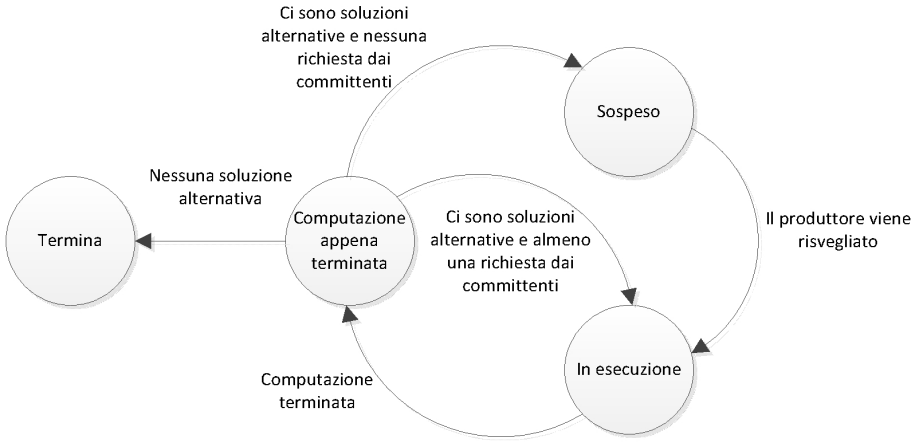

Figura 4.2.1–3 Diagramma degli stati del produttore.

Essendo un diagramma che si riferisce agli stati del produttore, non appaiono i legami che intercorrono con lettori e committenti, fondamentali per la sincronizzazione tra i processi.

Nello stato *Computazione appena terminata*, il produttore, prima di qualunque altra operazione, risveglia tutti i lettori in attesa di prelevare la soluzione e quando si trova nello stato *Sospeso*, è il committente che, prima di effettuare la richiesta di computazione, si occupa del suo risveglio.

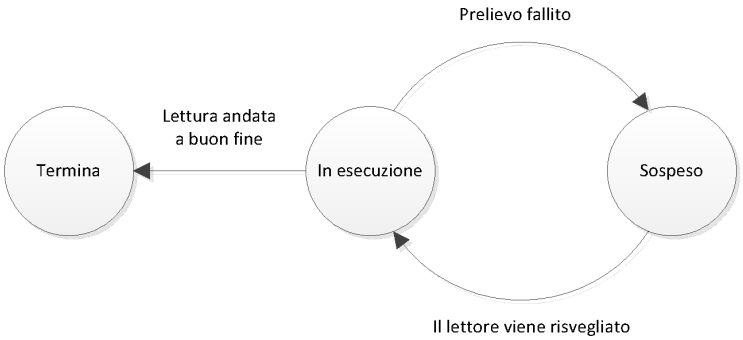

Figura 4.2.1–4 Diagramma degli stati del lettore.

Il prelievo della soluzione fallisce se il produttore è nello stato di esecuzione e il lettore deve attendere che il produttore in questione lo risvegli.

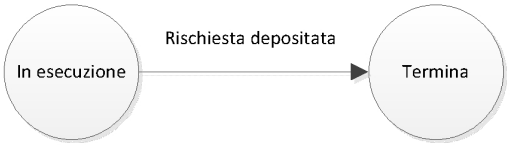

Figura 4.2.1–5 Diagramma degli stati del committente.

Ogni volta che si presenta un'istruzione di `thread_next_sol/1`, il committente inizia il processo per depositare la richiesta di computazione. Non è detto che la richiesta sia servita perché questo dipende dalla presenza o meno di soluzioni alternative nella query. In ogni caso il committente deposita la sua richiesta ed eventualmente risveglia il produttore per poi terminare l'esecuzione.

4.2.2 Realizzazione del protocollo

Per realizzare il modello di interazione appena descritto è necessario disporre di idonei meccanismi di sincronizzazione. In Java tali meccanismi sono disponibili a livello di linguaggio e sono supportati dalla JVM.
Dei tre ruoli delineati, solo i committenti sono indipendenti dall'esecuzione degli altri due: quando si presenta un'istruzione di `thread_next_sol/1`, il committente deposita semplicemente la richiesta di una nuova computazione e termina.
Lettori e produttori, invece, sono le due categorie di thread rispettivamente in attesa:

- di leggere il risultato della computazione corrente.
- che un committente riavvii l'esecuzione.

Per risvegliare i lettori si utilizza una variabile condizione (*condition*) cui è associato un lock, noto strumento di sincronizzazione che permette ai thread di sospendere la propria esecuzione in attesa che sia soddisfatta una data condizione logica. Le operazioni fondamentali sulle condition sono la sospensione (*await*) e il risveglio di thread (*signal*).
Nel caso del modello di sincronizzazione considerato, la condizione di sospensione per i lettori dipende dallo stato del produttore di cui stanno attendendo la soluzione: *se* il produttore è in esecuzione *oppure* si sta cercando di prelevare la prima soluzione di una query di cui il produttore non ha ancora cominciato la computazione, *allora* il lettore si

sospende. Quando il produttore termina la sua computazione, risveglia tutti i lettori attraverso un'istruzione di *signalAll* sulla condition.
Per quanto riguarda i produttori, invece, essi si mettono in attesa nel caso in cui ci siano soluzioni alternative della query *e contemporaneamente* nessuna richiesta di risoluzione.
Il loro risveglio dipende dall'arrivo di una richiesta da parte dei committenti e il meccanismo di sincronizzazione è regolato attraverso un'istanza di *Object*. Per questa tipologia di semaforo viene utilizzato il meccanismo nativo di sincronizzazione di thread presente in Java, ovvero l'uso di *wait/notify* all'interno di blocchi sincronizzati sul lock dell'oggetto che rappresenta il semaforo.

Nel nuovo prototipo, la classe che implementa tutta la sincronizzazione che regola le interazioni dirette tra i processi è EngineRunner.
Essendo la gestione dei thread trasparente al resto del sistema, l'effetto della sincronizzazione di lettori, committenti e produttori rimane limitato a tale classe.

CAPITOLO 5

THREADLIBRARY

Per il supporto alla programmazione concorrente sono messe a disposizione dell'utente diverse primitive, molte delle quali rispecchiano le funzionalità proposte dal prototipo precedente. Queste ultime primitive sono state sottoposte a un processo di attualizzazione per renderle compatibili con il nuovo modello.
Nel complesso la ThreadLibrary comprende tutta una serie di utilità. Tra queste:

- Funzioni di creazione e distruzione di thread.
- Istruzioni di prelievo e per la ricerca di nuove soluzioni.
- Predicati backtrackabili.
- Costrutti e funzioni per la comunicazione tra thread.
- Costrutti e funzioni di mutua esclusione (*mutex*) per garantire che un solo thread possa eseguire un blocco di codice in un certo istante.

Tutte queste funzionalità sono inglobate all'interno della classe ThreadLibrary, la quale delega a EngineManager l'esecuzione vera e propria dei predicati: la libreria, infatti, si limita a controllare la correttezza degli argomenti passati al predicato. Essa viene caricata di default nel sistema dal TheoryManager durante la fase di inizializzazione del motore.

5.1 Primitive di base per la gestione dei thread

- `thread_create/2(Query, ThreadId)`
  Crea un nuovo thread di identificatore `ThreadId` e comincia ad eseguire la `Query` data.

- `thread_id/1(ThreadId)`
  Tenta di unificare a `ThreadId` l'identificativo del thread corrente.

- `thread_join/2(ThreadId, Result)`

  Aspetta la terminazione del thread di identificatore `ThreadId` e ne raccoglie il risultato, unificando il goal risolto a `Result`. Il thread viene poi eliminato dal sistema. La chiamata fallisce se il processo è in stato *detached* (per maggiori informazioni si veda la primitiva `thread_detach/1`).

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal,ArgNo,ValidType,Culprit))` se `ThreadId` non è un intero. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero), `Culprit` è il termine errato trovato.

- `thread_read/2(ThreadId, Result)`

  Come `thread_join/2`, ma il thread `ThreadId` non viene eliminato dal sistema: esso rimane disponibile per altre letture successive o per ulteriori computazioni.

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal,ArgNo,ValidType,Culprit))` se `ThreadId` non è un intero. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero), `Culprit` è il termine errato trovato.

- `thread_detach/1(ThreadId)`

  Pone il thread di identificatore `ThreadId` nello stato *detached*.

  I thread in questo stato non possono essere attesi dagli altri thread mediante `thread_join/2` o `thread_read/2`, né possono essere oggetto di `thread_next_sol/1`.

  Quando terminano, non lasciano alcuna traccia della loro computazione.

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal,ArgNo,ValidType,Culprit))` se `ThreadId` non è un intero. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero), `Culprit` è il termine errato trovato.

- `thread_next_sol/1(ThreadId)`

  Forza il thread `ThreadId` a riprendere la computazione, esplorando soluzioni alternative della query iniziale.

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal,ArgNo,ValidType,Culprit))` se `ThreadId` non è un intero. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero), `Culprit` è il termine errato trovato.

- `thread_has_next/1(ThreadId)`

  Ha successo se il goal del thread di identificatore `ThreadId` ha soluzioni rimanenti.

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal,ArgNo,ValidType,Culprit))` se `ThreadId` non è un intero. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero), `Culprit` è il termine errato trovato.

- `thread_execute/2(Query, ThreadId)`

  Come `thread_create/2`, ma tale predicato è backtrackabile.

  Deve essere seguito da un'istruzione di lettura che serve per prelevare le diverse soluzioni computate dal thread di identificatore `ThreadId`. Attenzione all'uso di `thread_join/2` per il prelievo di soluzioni alternative alla prima: dopo il backtracking si incorre nel fallimento del predicato `thread_execute/2` a causa dell'eliminazione del thread dal sistema (provocata dall'istruzione di `thread_join/2`) che non consente di prelevare la soluzione computata da tale thread. In questo caso il comportamento di `thread_execute/2` coincide con quello di `thread_create/2`.

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal,ArgNo,ValidType,Culprit))` se `ThreadId` non è un intero. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero), `Culprit` è il termine errato trovato.

- `thread_sleep/1(Timeout)`

  Il thread in esecuzione si sospende per l'intervallo di tempo in millisecondi specificato in `Timeout`.

  Exception: `error(type_error(ValidType, Culprit), type_error(Goal,ArgNo,ValidType,Culprit))` se `Timeout` non è un intero. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero), `Culprit` è il termine errato trovato.

5.2 Primitive per le code di messaggi

La comunicazione tra i processi avviene attraverso la condivisione di aree di memoria: le *code*. Tali code seguono una politica di tipo FIFO, secondo la quale il primo messaggio introdotto è anche il primo a essere estratto. Esse possono essere di due differenti tipologie:

- *code pubbliche* create in modo esplicito e referenziate attraverso l'identificativo stabilito al momento della creazione;
- *code private* associate ad uno specifico thread e immediatamente disponibili specificando l'id del thread stesso.

Le primitive non variano per le due tipologie, che vengono distinte in base al tipo di identificativo specificato: un intero per le code private di uno specifico thread e un atomo per quelle pubbliche, indipendenti dai singoli thread.

L'invio di messaggi è sempre *asincrono*: il processo prosegue la sua esecuzione immediatamente dopo aver depositato il messaggio, senza attendere alcuna conferma di ricezione da parte del destinatario.

Per l'estrazione dei messaggi dalle code, invece, vi sono due alternative: la prima è di tipo *bloccante* e provoca la sospensione del processo all'interno della coda se non ci sono messaggi; in tal caso, il thread sarà risvegliato all'arrivo del primo messaggio utile. Nel caso *non bloccante*, invece, il thread prosegue l'esecuzione anche in assenza di messaggi sulla coda: in tal caso, l'esito della primitiva sarà negativo. Inoltre, quando il processo estrae il messaggio, c'è sia la possibilità di rimuoverlo dalla coda sia di lasciarlo a disposizione di altri processi.

Le corrispondenti primitive sono le seguenti:

- `thread_send_msg/2(Msg, QueueOrThreadId)`
  Inserisce un messaggio in una specifica coda `Queue` o nella coda associata ad un certo thread di identificatore `ThreadId`. Eventuali thread in attesa su quella coda vengono risvegliati.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `QueueOrThreadId` non è né un intero né un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero o atomo), `Culprit` è il termine errato trovato.

- `thread_get_msg/2(Msg, QueueOrThreadId)` – `thread_wait_msg/2(Msg, QueueOrThreadId)`
  Entrambi i predicati esaminano la coda di messaggi specificata alla ricerca di un termine che unifichi con `Msg` e sospendono l'esecuzione fino a quando tale termine non viene trovato. Una volta trovato il messaggio, `thread_get_msg/2` lo preleva rimuovendolo dalla coda, mentre `thread_wait_msg/2` lo legge solamente.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `QueueOrThreadId` non è né un intero né un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero o atomo), `Culprit` è il termine errato trovato.

- `thread_remove_msg/2(Msg, QueueOrThreadId)` – `thread_peek_msg/2(Msg, QueueOrThreadId)`
  Entrambi i predicati esaminano la coda di messaggi specificata alla ricerca di un termine che unifichi con `Msg`. Se raggiungono la fine della coda senza trovare tale termine, i predicati falliscono. Nel caso la ricerca si concluda con successo, `thread_remove_msg/2` preleva il messaggio rimuovendolo dalla coda, mentre `thread_peek_msg/2` lo legge solamente.
  Exception: `error(type_error(ValidType, Culprit),`

`type_error(Goal, ArgNo, ValidType, Culprit))` se `QueueOrThreadId` non è né un intero né un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero o atomo), `Culprit` è il termine errato trovato.

- `msg_queue_create/1(Queue)`
  Crea una nuova coda di nome `Queue` che non è associata ad alcun thread.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `Queue` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `message_queue_destroy/1(Queue)`
  Distrugge la coda di nome `Queue`. Non è possibile distruggere code associate a thread.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `Queue` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `msg_queue_size/2(Size, QueueOrThreadId)`
  Unifica a `Size` il numero di elementi della coda.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `QueueOrThreadId` non è né un intero né un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (intero o atomo), `Culprit` è il termine errato trovato.

5.3 Primitive di sincronizzazione

Gli strumenti di sincronizzazione consistono in semplici semafori: dispositivi mutuamente esclusivi (*MUTual EXclusive device*) che possono essere occupati da un

solo processo alla volta. Tali semafori sono *rientranti*, ovvero uno stesso semaforo può essere bloccato dallo stesso thread più di una volta, e risulta occupato fino a quando il thread possessore non effettua altrettante operazioni di liberazione. Questo permette di entrare dinamicamente in regioni critiche innestate, protette dallo stesso mutex, evitando che un thread sia causa di deadlock per se stesso.
L'utilizzo scorretto dei mutex però può portare a situazioni di deadlock (stallo) del sistema: la responsabilità di garantire la correttezza dell'utilizzo di tali semafori è lasciata all'utente.

- `mutex_create/1(MutexId)`
  Crea un mutex di nome `MutexId`. Se esiste nel sistema un mutex con lo stesso identificativo, viene mantenuto quello definito in precedenza.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `MutexId` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `mutex_destroy/1(MutexId)`
  Distrugge il mutex di nome `MutexId`.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `MutexId` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `mutex_lock/1(MutexId)`
  Blocca `MutexId` o si sospende se bloccato. Se `MutexId` non esiste viene creato.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `MutexId` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `mutex_trylock/1(MutexId)`
  Blocca `MutexId`. Tale predicato fallisce se `MutexId` è già in possesso di un altro thread.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `MutexId` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `mutex_unlock/1(MutexId)`
  Sblocca `MutexId`.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `MutexId` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `mutex_isLocked/1(MutexId)`
  Ha successo se `MutexId` è attualmente bloccato da un thread.
  Exception: `error(type_error(ValidType, Culprit), type_error(Goal, ArgNo, ValidType, Culprit))` se `MutexId` non è un atomo. `Goal` è il goal nel quale si presenta il problema, `ValidType` è il tipo di dato atteso (atomo), `Culprit` è il termine errato trovato.

- `mutex_unlock_all/0()`
  Sblocca tutti i mutex di cui è in possesso il thread.

5.4 Esempi di uso della libreria

Gli esempi di seguito illustrati mostrano alcuni programmi Prolog che utilizzano ThreadLibrary mostrando sia l'uso delle primitive di base, sia le più tipiche problematiche di sincronizzazione e comunicazione fra thread.

5.4.1 Calcolo del fattoriale: sequenziale e concorrente

Si supponga di voler calcolare il fattoriale di due numeri, e si supponga che la tipica implementazione dell'algoritmo sia espressa dal predicato `fact/2` sotto riportato.

```
fact(0,1):-!.
fact(N,X):-M is N-1,fact(M,Y),X is Y*N.
```

Per calcolare il fattoriale ad esempio di 7 e 8 in modo concorrente, si possono attivare due thread, dedicati ciascuno al calcolo di un valore, come nell'esempio che segue:

```
start(N,X,M,Y):- thread_create(fact(N,X),ID1),
                 thread_create(fact(M,Y),ID2),
                 thread_read(ID1, fact(N,X)),
                 thread_read(ID2, fact(M,Y)).
```

Come si vede, i due risultati parziali vengono recuperati in modo sincrono tramite due chiamate a `thread_read/2` con l'identificativo del thread fornito dalla corrispondente `thread_create/2`.

```
?- start(7,X,8,Y).
yes.
X / 5040  Y / 40320
Solution: start(7,5040,8,40320)
```

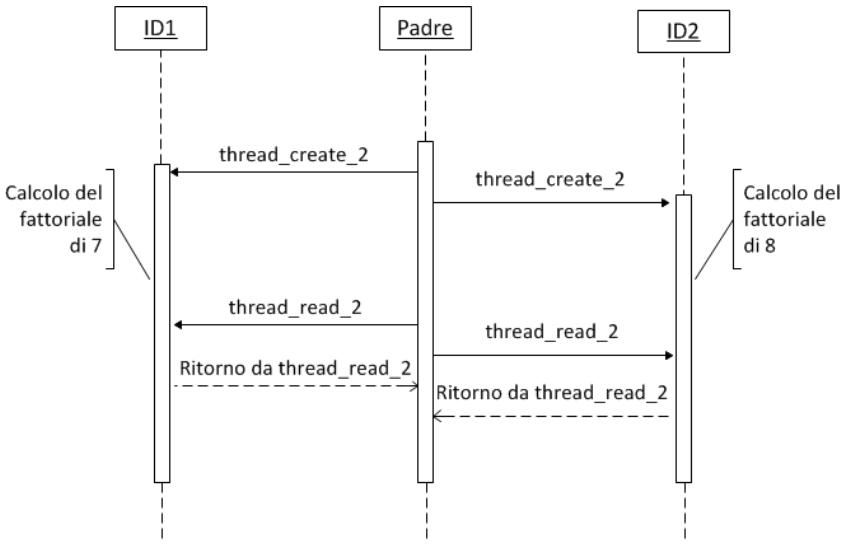

Figura 5.4.1–1 Interazioni tra padre e figli per la soluzione del problema sul calcolo fattoriale dei numeri 7 e 8.

Nel diagramma di sequenza mostrato in figura 5.4.1-1, il thread iniziale crea due figli produttori, con identificatori rispettivamente `ID1` e `ID2`, che iniziano a computare le soluzioni: il goal di `ID1` corrisponde al predicato `fact(7,X)`, mentre `ID2` deve risolvere la query data dal predicato `fact(8,Y)`. Il processo padre si comporta poi come lettore nei confronti dei figli ed estrae uno dopo l'altro i risultati da essi forniti.

Ovviamente, questo stesso problema può essere risolto anche in modo sequenziale, attraverso una doppia chiamata alla funzione per il calcolo del fattoriale `fact(N,X)`: in tal caso vi è un unico processo in esecuzione che si occupa di calcolare sequenzialmente i due valori, senza l'ausilio di nessun altro processo.
Ciò permette di effettuare un semplice confronto di prestazioni fra le due versioni, per avere una prima, seppur rozza, stima dell'eventuale miglioramento introdotto dall'approccio multi-threaded.
Per misurare il tempo di esecuzione nei due casi, è stato sfruttato l'ambiente di testing di JUnit che specifica per ogni singolo modulo eseguito il tempo in secondi. Dai dati rilevati emerge che la differenza tra le due modalità di esecuzione del calcolo fattoriale rientra nell'ordine di decine di millisecondi, con incertezza di qualche millisecondo.
La media dei due tempi, infatti, è di 0,282 s per il caso sequenziale e di 0,329 s per quello parallelo, come si può notare dal grafico di figura 5.4.1-2, che si basa su un campione di una decina di misure di esempio per entrambi i modelli.
Come si nota, in questo caso molto semplice la versione multi-threaded risulta peggiore (più lenta) di circa il 14% rispetto a quella sequenziale, a causa dell'overhead dovuto alla gestione della concorrenza.
Naturalmente, ci si attende che il risultato sia piuttosto diverso in situazioni dove la concorrenza possa realmente "fare la differenza" nella computazione.

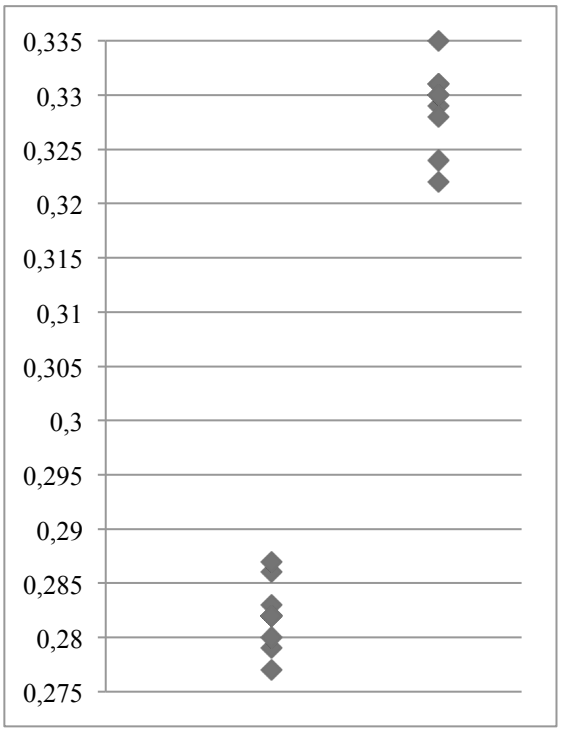

Figura 5.4.1–2 Dispersione delle misure effettuate con JUnit per i due modelli, sequenziale e parallelo.

5.4.2 Predicati bloccanti e non bloccanti

Di seguito sono riportati due diversi esempi di utilizzo di code pubbliche e private con predicati bloccanti e non bloccanti.

1. Primo esempio: coda pubblica con identificativo `CODA` e predicato di ricezione bloccante `thread_get_msg/2`.

Si vuole che il figlio si metta in attesa di ricevere un messaggio dal padre; quest'ultimo deve inviare tale messaggio dopo 500 ms. dalla generazione del figlio.

```
start(X) :- msg_queue_create('CODA'),
            thread_create(thread1(X),ID),
            thread_sleep(500),
            invio('CODA','messaggio molto importante'),
            lettura(ID,X).
thread1(X) :- thread_get_msg(a(X),'CODA').
invio(ID, M):- thread_send_msg(a(M),ID).
lettura(ID, X):- thread_join(ID,thread1(X)).

?- start(X).
yes.
X / 'messaggio molto importante'
Solution: start('messaggio molto importante')
```

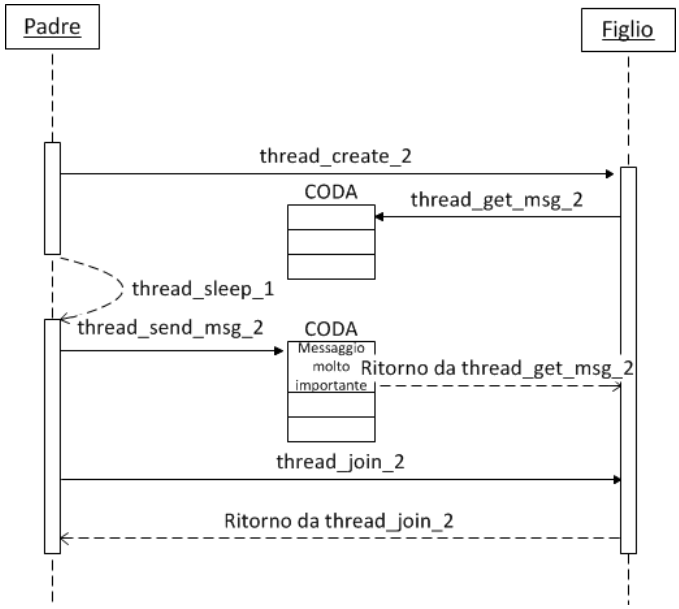

Figura 5.4.2–1 Interazione tra padre e figlio attraverso lo scambio di messaggi con coda pubblica.

Il funzionamento del programma, mostrato in figura 4.5.2-1, è molto semplice: il processo iniziale, dopo aver creato la coda pubblica per permettere la comunicazione, genera un figlio che tenta di prelevare un messaggio dalla coda ancora vuota. Essendo la `thread_get_msg/2` un'istruzione bloccante, il figlio si mette in attesa del messaggio. In seguito, il padre si sospende per 500 ms. e al suo risveglio deposita in maniera asincrona il messaggio nella coda: il processo figlio ora può procedere con l'operazione di prelievo e concludere la chiamata di `thread_get_msg/2` rimasta precedentemente in sospeso.
Si noti come la comunicazione tra padre e figlio renda evidente la sincronizzazione presente nei predicati di prelievo bloccanti.

2. Secondo esempio: coda privata con identificativo del processo e predicato di ricezione non bloccante `thread_peek_msg/2`.

Il processo padre deve creare tre figli: il primo si mette in attesa di un messaggio sulla propria coda, il secondo è il mettente di tale messaggio, mentre il terzo esplora la coda del primo figlio senza sospendersi nel caso la ricerca fallisse. Si vuole che solo il primo figlio riesca a prelevare il messaggio.

```
start(X) :- thread_create(thread1(ID),ID),
            thread_sleep(200),
            thread_create(thread2(ID), ID2),
            thread_create(thread3(ID,X), ID3),
            lettura(ID3,X).
thread1(ID) :- tell("threadLog.txt"),
               write("ID1: attende il messaggio"),
               thread_get_msg(a(X),ID),
               write("ID1: messaggio prelevato dalla propria
               coda").
thread2(ID) :- invio(ID, 'messaggio molto importante').
invio(ID, M) :- thread_send_msg(a(M),ID),
                write("ID2: messaggio inviato").
thread3(ID,X) :- thread_peek_msg(a(X),ID),
                 write("ID3: tenta di prelevare il messaggio").
lettura(ID, X) :- thread_read(ID, thread1(X)),
```

```
                write("Padre: leggo la soluzione").

?- start(X).
no.
```

Il processo principale crea tre diversi thread con identificativi pari a `ID1`, `ID2`, `ID3`: il primo ha il compito di attendere un messaggio sulla propria coda, il secondo si occupa di depositare tale messaggio sulla coda del thread `ID1` ed il terzo processo deve cercare di prelevare sempre lo stesso messaggio, ma attraverso un predicato non bloccante.
La particolarità di questo esempio sta nel fatto che il processo `ID1` preleva il messaggio e contemporaneamente lo rimuove dalla sua coda per effetto del predicato `thread_get_msg/2`. Questo comporta che il predicato non bloccante `thread_peek_msg/2` eseguito dal thread `ID3`, fallisca poiché raggiunge la fine della coda di `ID1` senza trovare alcun messaggio depositato. L'esito negativo della computazione è dovuto alla `thread_read/2` del padre verso il figlio `ID3` che non va a buon fine a causa del fallimento del figlio.
Per registrare le operazioni man mano che sono eseguite, si utilizza un file di log di nome *ThreadLog.txt* sul quale memorizzare tali informazioni.

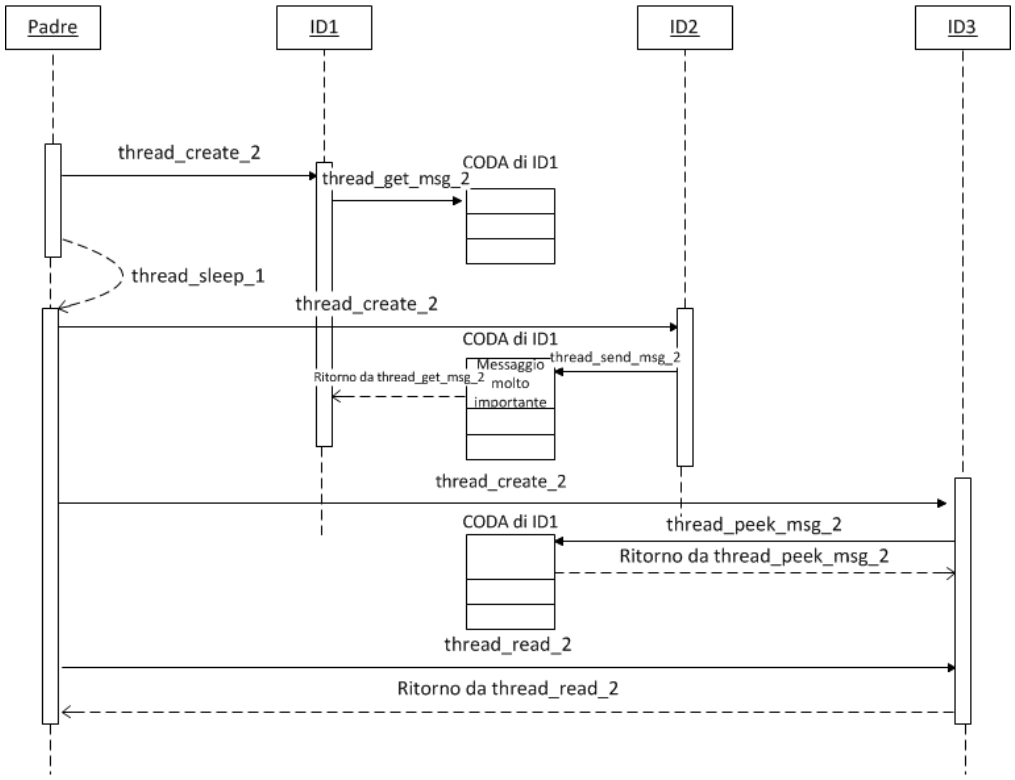

Figura 5.4.2–2 Interazione tra padre e figli attraverso lo scambio di messaggi con coda privata del thread ID1. Il predicato `thread_peek_msg/2` fallisce perché non essendo bloccante, scorre tutta la coda senza trovare alcun messaggio e termina.

5.4.3 Sincronizzazione delle interazioni tra i processi

Come esempio di utilizzo degli strumenti di sincronizzazione, sono messi a confronto due programmi di cui solo il secondo è dotato di sincronizzazione esplicita.

L'obiettivo è prelevare l'ultima soluzione alternativa della query `figlio(bob,X)`, a partire da una serie di fatti. Il thread produttore deve computare tutte le soluzioni alternative, di cui il processo lettore dovrà prelevare solo l'ultima. Il thread iniziale, che è committente, effettua richieste di nuove computazioni per il produttore fino a quando sono disponibili delle soluzioni alternative della query.

```
start :- thread_create(figlio(bob,X), ID1),
         loop(1,5,1,ID1),
         thread_create(lettura(ID1,X), ID2).
loop(I, To, Inc, ThreadId) :- Inc >= 0, I > To, !.
loop(I, To, Inc, ThreadId) :- Inc < 0,  I < To, !.
loop(I, To, Inc, ThreadId) :- (thread_has_next(ThreadId) ->
                               thread_next_sol(ThreadId),
                               Next is I+Inc,
                               loop(Next, To, Inc, ThreadId); !).
lettura(ID, X) :- thread_read(ID,X).
figlio(bob,alex).
figlio(bob,anna).
figlio(bob,maria).

?- start.
```

Il thread iniziale crea un figlio produttore con identificativo pari a `ID1` che inizia a cercare la prima soluzione. Fino a quando non arrivano ulteriori richieste di computazione, il processo produttore `ID1` si sospende.

La chiamata alla funzione `loop(from,to,inc,info)` consiste in un ciclo iterativo in cui bisogna specificare l'indice da cui partire, il suo valore iniziale, l'incremento desiderato e i parametri che eventualmente devono essere trasferiti da una chiamata all'altra, come in questo caso l'identificatore del thread. Il ciclo ha lo scopo di chiamare delle `thread_next_sol/1` ripetutamente fino a quando le soluzioni della query non sono esaurite.

Il thread padre poi genera un secondo figlio con il ruolo di lettore dell'ultima soluzione computata dal thread `ID1`.
In questo scenario non è possibile prevedere quale soluzione sarà prelevata, in quanto tale soluzione risulta essere dipendente dalle velocità relative del processo lettore e del padre.
La soluzione, infatti, potrebbe essere una qualunque tra `alex`, `anna` o `maria` visto che, non appena viene generato, il lettore attende la prima soluzione disponibile che dipende dalla velocità con cui il produttore computa le soluzioni all'interno del ciclo. Il problema può sorgere a causa del ruolo del committente che, non dovendo aspettare che il figlio computi tutte le soluzioni, deposita le richieste di ulteriori computazioni e continua la sua esecuzione. Questo comporta che il figlio lettore venga generato durante la fase di computazione del produttore e di conseguenza prelevi la prima soluzione disponibile.
Di seguito è mostrato un possibile utilizzo degli strumenti di sincronizzazione mediante i quali si può prevenire questo genere di incertezze.

```
start :- thread_create(figlio(bob,X), ID1),
         mutex_lock('mutex'),
         thread_create(lettura(ID1,X), ID2),
         loop(1,5,1,ID1),
         mutex_unlock('mutex').
loop(I, To, Inc, ThreadId) :- Inc >= 0, I > To, !.
loop(I, To, Inc, ThreadId) :- Inc < 0,  I < To, !.
loop(I, To, Inc, ThreadId) :- (thread_has_next(ThreadId) ->
                               thread_next_sol(ThreadId),
                               Next is I+Inc,
                               loop(Next, To, Inc, ThreadId); !).
lettura(ID, X) :- mutex_lock('mutex'),
                  thread_read(ID,X),
                  mutex_unlock('mutex').
figlio(bob,alex).
figlio(bob,anna).
figlio(bob,maria).
```

```
?- start.
```

Gli accessi al thread produttore per il prelievo della soluzione sono sincronizzati attraverso il semaforo *mutex* che regola le interazioni: fino a quando il mutex è in mano al padre, il lettore non può prelevare alcuna soluzione. Non appena il produttore esce dal ciclo, il padre rilascia il lock di mutex, così che il lettore possa appropriarsene e prelevare la soluzione, avendo ora la certezza che si tratti dell'ultima soluzione alternativa per la query `figlio(bob,X)`, cioè `X / maria`.

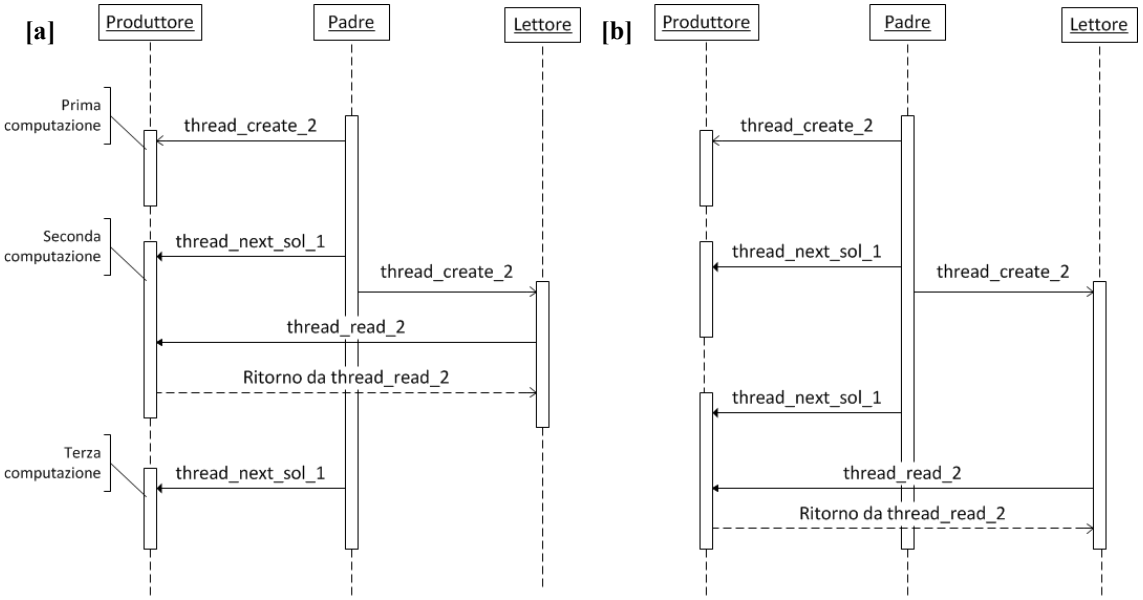

Figura 5.4.3–1 Confronto tra due diversi approcci per la risoluzione dello stesso problema: nessun uso di strumenti di sincronizzazione [a] e uso di mutex per regolare le interazioni tra i processi [b].

L'esempio di utilizzo dei mutex, mette in luce la necessità di introdurre sincronizzazione esplicita nel flusso di esecuzione dei thread e soprattutto come cambino le interazioni tra i processi, in presenza o in assenza di questa.

Inoltre, senza un'adeguata sincronizzazione tra i processi, è possibile che essi non vengano schedulati nel quanto di tempo che intercorre tra la computazione della soluzione di interesse e la successiva. L'esecuzione parallela di più thread all'interno della stessa applicazione, infatti, è vincolata dall'architettura della macchina su cui gira: se la macchina ha un unico processore, in un determinato momento può essere in esecuzione un unico thread, con il grande vantaggio che tutti i processi vengono fatti evolvere contemporaneamente anche se in ogni istante solo uno di essi è in reale

esecuzione. Sfruttando i semafori, si dà così un ordine alle esecuzioni parallele dei thread.

5.5 Collaudo

Per collaudare il nuovo prototipo si è sfruttata l'unità di testing JUnit, con lo scopo di testare tutte le funzionalità della ThreadLibrary, predicato per predicato.
In generale si è seguito lo stesso procedimento per ogni metodo di test: si è caricata nel sistema una teoria con diverse query da risolvere. Attraverso tale teoria vengono creati numerosi processi che interagiscono tra di loro in diversi modi, cercando di identificare eventuali malfunzionamenti. Bisogna prevedere il comportamento di ciascun processo e i risultati delle computazioni devono essere ben definiti a livello teorico. Se esiste corrispondenza tra la previsione e il risultato (verificato attraverso numerose asserzioni), il test si considera concluso con successo.

Il collaudo non ha effetto solamente sulla libreria dei thread, in quanto verifica anche altri aspetti del sistema: i test della libreria fallirebbero se i manager o le relazioni tra lettori, produttori e committenti non fossero correttamente sincronizzate. Ad esempio, come discusso nel capitolo 4, a un'istruzione di lettura proveniente dalla ThreadLibrary corrisponde l'immediato intervento di un lettore con determinate caratteristiche e vincoli descritti nella classe EngineRunner.
Tutti i test fatti per ciascun predicato della ThreadLibrary hanno avuto esito positivo: di conseguenza, è possibile ritenere ragionevolmente verificati sia il funzionamento dei singoli predicati, sia la correttezza dell'infrastruttura di sincronizzazione che regola le interazioni di coordinamento e competizione dei processi.

Un'altra parte importante della fase di collaudo è consistita nel verificare che le modifiche effettuate non causassero problemi in altre parti del sistema: a tale scopo, si è sfruttata la vasta suite di test già presente in tuProlog, con esito finale positivo.

5.5.1 Prestazioni

Nell'esempio sul calcolo fattoriale del paragrafo 5.4.1, il confronto tra la versione sequenziale di tuProlog e il nuovo prototipo sviluppato, evidenziava un peggioramento delle prestazioni nel modello multi-threaded rispetto a quello sequenziale. Ciò corrispondeva al risultato aspettato, perché in un caso così semplice tutto il vantaggio che può portare la concorrenza non è assolutamente tangibile e durante l'esecuzione si rischia di sentire solo il peso dell'overhead.

Il test descritto di seguito ha lo scopo di dimostrare la validità del modello concorrente del nuovo prototipo in termini di efficienza e di prestazioni del sistema.

Si supponga di dover manipolare una lista L che contiene N altre sottoliste numeriche, per ognuna delle quali si vuole poter effettuare alcune operazioni:

- contare il numero di volte in cui il termine C compare nella sottolista;
- appiattire la sottolista: ad esempio, l'appiattimento della lista [1,[2,3,[4]],5,[6]] deve produrre [1,2,3,4,5,6];
- in seguito all'operazione di appiattimento, ordinare in maniera crescente la sottolista.

Il problema può essere risolto sia seguendo un modello sequenziale che concorrente. Nel primo caso si esegue un'operazione alla volta e si procede in maniera ricorsiva agendo su ogni sottolista contenuta in L: inizialmente si effettua l'appiattimento, poi si ordinano gli elementi ed infine si guarda la ricorrenza del termine specificato.

Dati la lista L, il numero di sottoliste in essa contenuta ed il termine da cercare, il programma è il seguente:

```
? – start([[[2,2],2,2,1],[4,[3],2],[9,8,9,2]], 3, 2).

start(L, 0, T) :- !.
start([H|Tail], N, T) :- plain(H,L_plain),
                         bubble(L_plain,L_ord),
                         occorr(T,H,Count),
                         C is N-1,
                         start(Tail,C, T).
plain(L1,L2) :- plain(L1,[],L2).
```

```
plain([],ACC,ACC).
plain([H|REST],ACC,L2) :- H = [_|_],
                          plain(H,ACC,ACC1),
                          plain(REST,ACC1,L2).
plain([H|REST],ACC,L2) :- append(ACC,[H],ACC1),
                          plain(REST,ACC1,L2).
plain(X,ACC,L2) :- append(ACC,[X],L2).
bubble(L1,L2) :- bubble(L1,0,L2).
bubble(L1,0,L2) :- sweep(L1,0,L2).
bubble(L1,0,L2) :- sweep(L1,1,LTMP),
                   bubble(LTMP,0,L2).
sweep([X|[]],0,[X|[]]).
sweep([X,Y|REST1],CHANGED,[X|REST2]) :- X =< Y,
                                        sweep([Y|REST1],CHANGED,REST2).
sweep([X,Y|REST1],1,[Y|REST2]) :- X > Y,
                                  sweep([X|REST1],_,REST2).
occorr(T,L,N) :- occorr(T,L,0,N).
occorr(_,[],ACC,ACC).
occorr(T,[T|REST],ACC,N) :- ACC1 is ACC+1,
                            occorr(T,REST,ACC1,N).
occorr(T,[_|REST],ACC,N) :- occorr(T,REST,ACC,N).
```

Non interessa capire nel dettaglio il funzionamento di questo programma, piuttosto modificarlo sfruttando le potenzialità derivanti dal modello concorrente sviluppato per il nuovo prototipo.

L'idea è quella di distribuire le responsabilità tra vari processi. Essendo l'operazione di conteggio indipendente dalle altre, si è deciso di assegnare i compiti di manipolazione di ogni singola sottolista a due processi distinti, in modo da non lasciare tutto il peso dell'esecuzione a un unico thread: il processo iniziale si occuperà di contare le ricorrenze in ogni sottolista del termine specificato, mentre il secondo processo dovrà appiattire le sottoliste e gestirne l'ordinamento.

Di seguito non viene riportato l'intero programma nella versione multi-threaded, ma soltanto le istruzioni iniziali con le parti modificate:

```
start(L, N, T) :- thread_create(firstResp(L, N),ID),
                  secondResp(L, N, T).
secondResp(L, 0, T):- !.
secondResp([H|Tail], N, T) :- occorr(T,H,Count),
                              C is N-1,
                              secondResp(Tail,C, T).
firstResp(L, 0) :- !.
firstResp([H|Tail], N) :- plain(H,L_plain),
                          bubble(L_plain,L_ord),
                          C is N-1,
                          firstResp(Tail,C).
```

Eseguendo i due test nella versione sequenziale e concorrente, rispettivamente su tuProlog 2.6 e sul nuovo prototipo, si ottengono dei risultati piuttosto interessanti: il tempo di esecuzione risulta essere di molto inferiore nel caso multi-threaded, con un guadagno percentuale pari quasi al 16%.

Per avere un'idea più precisa sul margine di guadagno che può offrire il supporto al multi-threading di tuProlog, si vuole mostrare almeno un altro esempio che possa mettere in luce questo aspetto.
Si definisce un predicato chiamato `study` che applicato a un numero di matricola `Matr` di uno studente e a una lista di esami `L_exams`, dà come risultato la media `AV` dei suoi voti. Ogni esame è rappresentato da un termine della lista `L_exams` nella forma `exam(Matr, ExamName, Vote)`.
Dati una lista di matricole `L_stud`, il numero `Num` di studenti corrispondenti a tale lista e una lista di esami `L_exams`, si vuole calcolare la media dei voti di ogni studente e il numero di esami presenti nella lista.
Di seguito viene definito il predicato `study`:

```
?- study([s1,s2,s3,s4,s5], Num, [exam(s2,f1,30), exam(s1,f1,27),
exam(s3,f1,25), exam(s1,f2,30), exam(s4,f1,25), exam(s3,f2,20),
exam(s5,f1,20), exam(s2,f5,30), exam(s1,f5,27), exam(s3,f5,25),
exam(s1,f4,30), exam(s4,f5,25), exam(s3,f8,20), exam(s5,f7,20)],
5).
```

```
study(L_stud,N,L_exams,Num) :- length(L_exams,Num),
                               loop(1,N,1, L_stud, L_exams).
length([],0).
length([_|Queue],N):-length(Queue,Nqueue),
                     N is Nqueue + 1.
loop(I, To, Inc, L_stud, L_exams) :- Inc >= 0, I > To, !.
loop(I, To, Inc, L_stud, L_exams) :- Inc < 0,  I < To, !.
loop(I, To, Inc, [H|Tail], L_exams) :-  start(H,L_exams),
                                  Next is I+Inc,
                                  loop(Next,To,Inc,Tail,L_exams).
start(S,L):- totStud(S,L,N,T),
             N > 0,
             AV is T/N.
totStud(_,[],0,0) :- !.
totStud(S,[exam(S,_,V)|R],N,T) :- !, totStud(S,R,NN,TT),
                                   N is NN + 1,
                                   T is TT + V.
totStud(S,[_|R],N,T) :- totStud(S,R,N,T).
```

Nella versione concorrente di questo programma, il processo iniziale si occuperà di contare il numero degli esami contenuti nella lista `L_exams` e un altro processo gestirà il ciclo che ha lo scopo di generare tanti thread quanti sono i numeri di matricola specificati. Tali thread avranno il compito di calcolare la media dei voti di ciascuno studente.

Come nell'esempio precedente, sono riportate solo le parti modificate rispetto alla versione sequenziale.

```
study(L_stud, N, L_exams, Num) :-
               thread_create(loop(1, N, 1, L_stud, L_exams),ID),
               length(L_exams,Num).
loop(I, To, Inc, L_stud, L_exams) :- Inc >= 0, I > To, !.
loop(I, To, Inc, L_stud, L_exams) :- Inc < 0,  I < To, !.
loop(I, To, Inc, [H|Tail], L_exams) :-
                         thread_create((totStud(H,L_exams,N,T),
```

```
                    N > 0, AV is T/N),ID),
                    Next is I+Inc,
                    loop(Next, To, Inc, Tail, L_exams).
```

Il confronto tra la versione concorrente e sequenziale del programma appena mostrato sul calcolo della media dei voti, ha avuto esito positivo a favore dell'interprete dotato di supporto al multithreading: il guadagno in termini di prestazioni risulta essere circa dell'8%.
Si noti come questo risultato sia diverso rispetto a quello ottenuto nel precedente esempio di manipolazione delle liste, pari al 16%. Questo è dovuto al fatto che la percentuale di guadagno dipende molto dal livello di concorrenza del problema stesso e soprattutto da come l'utente sia in grado di esprimerlo, fattori di cui è necessario tenere conto quando si lavora con supporti per il multithreading *esplicito*.

Questi esempi dimostrano quanto, in generale, un modello concorrente possa essere vantaggioso in casi in cui è possibile sfruttarne tutte le potenzialità.

CAPITOLO 6

CONCLUSIONI

Con questo lavoro si è cercato di contribuire allo sviluppo dell'interprete tuProlog nella direzione del parallelismo esplicito.

Per il raggiungimento dell'obiettivo è stato necessario effettuare uno studio preliminare relativo al linguaggio Prolog, concentrandosi, in particolar modo, sulle strategie operative che caratterizzano la ricerca delle soluzioni di un problema in un paradigma di programmazione logica. Lo studio è proseguito con l'analisi dell'architettura dell'attuale motore, facendo emergere tutta la struttura che sta dietro all'elaborazione di una query: la divisione delle responsabilità, il funzionamento del processo di risoluzione, in che modo i manager sono coinvolti in tale processo e in quali termini, ed infine, focalizzando l'attenzione su EngineManager, in quanto componente che incapsula tutta l'unità di elaborazione di una query. È stata poi analizzata la precedente versione prototipale del supporto al multithreading, i cui concetti di ambiente di esecuzione e contesto di esecuzione si sono rivelati ancora validi e da mantenere alla base della nuova architettura. Sono quindi state valutate diverse possibilità di riuso/adattamento del precedente prototipo per la realizzazione del nuovo: da tale studio di fattibilità è emersa con chiarezza la possibilità di sottoporre il prototipo precedente a un processo di reingegnerizzazione. Partendo dalla definizione di nuovi requisiti si è quindi giunti a sviluppare un nuovo prototipo caratterizzato da profonde differenze strutturali rispetto al precedente. In particolare, la nuova architettura tiene conto dei problemi di sincronizzazione che possono nascere da un modello di computazione concorrente e dei diversi modi con cui possono interagire i processi, competendo per l'accesso a risorse condivise o cooperando per il raggiungimento di un obiettivo. Si è intervenuti sui manager per rendere thread-safe l'ambiente di esecuzione ed è stato definito un protocollo per la gestione delle esecuzioni parallele di thread.

Essendo tuProlog un software scritto interamente in Java, sono stati introdotti nell'architettura dell'interprete alcuni concetti legati alla programmazione ad oggetti, come semafori e strumenti di comunicazione, con lo scopo di poterne sfruttare i vantaggi.

Per rendere disponibili all'utente Prolog le nuove funzionalità di gestione dei thread, di comunicazione e sincronizzazione, è stata realizzata una nuova libreria, ThreadLibrary, che definisce un insieme compatto ma ragionevolmente completo di primitive. La successiva fase di collaudo, funzionale e prestazionale, ha evidenziato non solo il corretto funzionamento in tutti i casi di test considerati, ma anche un interessante incremento delle prestazioni rispetto al modello sequenziale.

Il risultato è una nuova versione di tuProlog, in grado di offrire un pieno supporto per il multithreading esplicito.

Bibliografia

Ancilotti Paolo, Boari Maurelio, Ciampolini Anna, Lipari Giuseppe, *Sistemi Operativi*, McGraw-Hill, Milano 2008.

Console Luca, Lamma Evelina, Mello Paola, Milano Michela, *Programmazione Logica e Prolog*, UTET, Milano 1997.

Larman Craig, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Prentice Hall PTR, Upper Saddle River, NJ, USA 2004.

tuProlog
tuProlog Manual, Università di Bologna 2012.
http://alice.unibo.it/xwiki/bin/view/Tuprolog/
http://code.google.com/p/tuprolog/

Swi-Prolog 6.3.9
http://www.swi-prolog.org/