Uso di Spin in un approccio probabilistico alla verifica automatica di sistemi concorrenti

Igor Melatti 131422

Indice

Capitolo 1

Introduzione

Il problema principale di chi vuole effettuare del *model checking* su una struttura software concorrente, ovvero verificare se essa contiene errori, come la non terminazione, o non rispetta in pieno i requisiti (tipicamente, quelli funzionali) che il sistema finale deve avere, è quello dell'"esplosione degli stati". Capita spesso, infatti, che un sistema concorrente abbia un numero relativamente basso di stati in ciascuna delle sue componenti, ma che poi la quantità totale di stati del sistema nel suo complesso sia una funzione esponenziale di tale numero. Dal momento che un qualsiasi approccio automatico alla verifica deve necessariamente memorizzare gli stati (non necessariamente tutti, come si vedrà) che vengono via via esplorati, ciò comporta che alcuni processi di verifica non riescano a terminare per l'esiguità della memoria disponibile. I tentativi di risolvere questo problema hanno portato all'ideazione di modalità di compressione (cfr) o di procedimenti che non garantiscono più la completa copertura dell'insieme degli stati raggiungibili, ma solo di una sua parte, tipicamente nota a verifica terminata (cfr), usando ovviamente meno memoria di quella necessaria alla verifica esaustiva; in queste ultime procedure, sostanzialmente, può accadere che uno stato venga riconosciuto come già visitato anche se non è così, e quindi non venga esplorato anche se dovrebbe esserlo.

Nel presente lavoro di tesi si tratterà di un altro possibile approccio: uno stato che sia già stato visitato non verrà più mantenuto in memoria dal momento della sua prima visita fino al termine della verifica, ma potrà accadere che venga "sovrascritto" da un altro stato, di modo che diventerà possibile che uno stato sia visitato più di una sola volta[1]. È chiaro che tale procedimento permette di risparmiare memoria, e sarà necerrario più tempo di esecuzione per portare a termine la stessa verifica; è inoltre possibile, ovviamente, che l'intero processo non termini. È da notare che, con questo approccio, quando una verifica termina, essa è esaustiva, ovvero copre totalmente l'insieme degli stati raggiungibili; tuttavia, si vedrà che essa è applicabile anche ad alcune tecniche di ricerca non esaustiva, ottenendo (sinonimo) una copertura dell'insieme degli stati raggiungibili uguale a quella che si otteneva senza "sovrascrivere".

Si vedrà come tale approccio possa essere implementato, modificando opportunamente il *model checker* Spin in quattro delle sue modalità di verifica, e si analizzeranno i risultati sperimentali ottenuti su alcuni particolari protocolli da verificare.

[1] In realtà, ciò può accadere anche nelle verifiche tradizionali, quando si usino tecniche di esplorazione *nested*, ovvero inglobate (sinonimo) l'una nell'altra, per trovare dei cicli di esecuzione; qui chiaramente si intende che uno stato viene visitato più di una volta all'interno di ogni singola esplorazione.

Capitolo 2

Il *model checker* Spin

Spin è un programma ideato da G. J. Holtzmann ai Bell Labs a partire dagli anni '80 come esempio concreto per un articolo sui protocolli di comunicazione (cfr), e successivamente perfezionato fino a divenire uno dei migliori *model checkers* in circolazione, tanto da essere già stato utilizzato per verificare la correttezza del software di alcuni sistemi distribuiti, come ad esempio i sistemi operativi o i cosiddetti *embedded systems* (sistemi incorporati, come quelli installati sulle automobili per controllarne la velocità).

Più precisamente, si tratta di un tool ideato per analizzare la consistenza logica dei sistemi concorrenti, specialmente dei protocolli di comunicazione. Il sistema da analizzare viene descritto in un linguaggio di specifica chiamato Promela (PROcess MEta LAnguage, mentre Spin sta per Simple Promela INterpreter), che permette la creazione dinamica di processi concorrenti, nonché la comunicazione tra di essi. Quest'ultima avviene tramite canali, che possono essere sia sincroni, implementando così una comunicazione a *rendez-vous*, sia asincroni, accodando i messaggi prima dello smistamento; è comunque possibile anche la comunicazione a memoria condivisa.

Data una descrizione Promela di un modello di sistema, Spin può o effettuare una

esecuzione randomizzata del sistema così come è stato descritto, oppure generare un programma C che sia in grado di effettuare una efficiente verifica delle proprietà di correttezza che si richiede che il sistema abbia. Più dettagliatamente, questo programma può controllare che non esistano possibili esecuzioni del sorgente Promela nelle quali:

- non venga rispettata una *assertion* che si trova in un certo punto del sistema
- il sistema non cicla su delle istruzioni che devono invece essere eseguite infinite volte (*non-progress cycles*) o, viceversa, cicla su istruzioni che non devono essere eseguite infinite volte (*acceptance cycles*); tuttavia, in una singola verifica non si può controllare la presenza di entrambi i tipi cicli, ma occorre eventualmente effettuare due verifiche distinte
- ci siano *deadlocks*, che nella terminologia di Spin si intendono essere situazioni in cui tutti i processi sono bloccati in attesa di un messaggio che solo uno di questi processi in attesa può mandare
- non risultino verificate proprietà di correttezza espresse in formule di logica temporale lineare.

Inoltre, una verifica esaustiva eseguita da Spin può stabilire con certezza matematica se un dato comportamento di un sistema sia o no esente da errori.
Il verificatore (chiamato Pan, ovvero Protocol ANalyzer) è costruito in modo da essere efficiente sia in termini di tempo di esecuzione che di spazio di memoria. Tuttavia, per la verifica di sistemi molto grandi e complicati ciò può non bastare, e così capita che Pan esaurisca la memoria a sua disposizione (per il problema dell'"esplosione degli stati", cfr); anche in questi casi è comunque possibile effettuare una verifica, usando varie tecniche di compressione (minore spazio di memoria utilizzato, ma maggiore tempo di esecuzione richiesto) o tecniche che forniscono risultati approssimati: ciò significa che, se viene rilevato un errore, allora tale errore

è effettivamente presente nel sistema così com'è descritto, mentre una verifica che affermi l'assoluta correttezza del sistema deve essere considerata come passibile di errore (seppur molto basso), in quanto è possibile che alcuni stati raggiungibili non siano stati visitati; tali tecniche di approssimazione sono principalmente il *bit state storage* (o *supertrace*) e l'*hash compaction*, e verranno descritte con qualche dettaglio in più nel seguito.

2.1 Il linguaggio Promela

In questo paragrafo si cercherà di spiegare brevemente quali comportamenti si possono modellare con Promela; la conoscenza dell'esatta e completa grammatica di Promela non è necessaria alla comprensione di questo lavoro di tesi. Si tenga presente, in ogni caso, che tale grammatica assomiglia molto a quella del C, dalla quale mutua vari costrutti; è inoltre utile sottolineare che Spin è stato ideato principalmente per il comportamento di quelle parti di sistemi concorrenti delle quali si sospetti un malfunzionamento (soprattutto per quanto riguarda le comunicazioni tra componenti), e dunque Promela è stato pensato per astrarre dai dettagli implementativi e concentrarsi nel nocciolo del problema.
Tre sono gli elementi fondamentali di Promela: i *processi*, le *variabili* e i *canali*. Le variabili non sono diverse da quelle del C, anche se esistono meno tipi base (solo quattro: `bit`, `byte`, `short` ed `int`), e possono essere globali, ovvero accessibili a più processi, o locali, cioè visibili al solo processo nel corpo del quale sono dichiarate. Si possono usare anche strutture e tipi enumerativi, ma di questi ultimi se ne può dichiarare solo uno per ogni descrizione Promela.

2.1.1 I processi

I processi sono degli oggetti globali che specificano il comportamento delle componenti del sistema concorrente in considerazione. Essi sono infatti costituiti da una

serie di istruzioni, che possono venire eseguite in *interleaving* con quelle degli altri processi, il che rappresenta l'opzione di *default*, oppure in maniera atomica, e per far ciò occorre usare la parola chiave `atomic`.

Tutti i processi, per diventare attivi, devono essere mandati in esecuzione da un altro processo, mentre inizialmente è attivo un solo processo, quello chiamato `init` (ciò è ovviamente analogo al ruolo della funzione `main` nel C); un processo, invece, termina quando raggiunge la fine del proprio corpo (non è detto che ciò accada, naturalmente), e sono già terminati tutti i processi che lui stesso ha attivato. È da notare che è possibile anche che un processo chiami se stesso, realizzando una chiamata ricorsiva; va comunque detto che Promela non permette che ci siano più di 256 processi attivi contemporaneamente.

Per quanto riguarda le istruzioni, sono contemplati gli assegnamenti, il costrutto condizionale e quello iterativo, un'istruzione di `print`, nonché i salti incondizionati a determinate etichette; le istruzioni di invio e ricezione messaggi verranno trattate nel paragrafo seguente. Promela permette inoltre di specificare comportamenti non deterministici: ad esempio, nel seguente frammento di processo,

```
if
::a=3
::a=4
fi;
```

alla variabile `a` potrà essere assegnato il valore 3 o 4 in maniera non deterministica. Se al posto delle parole chiave `if` e `fi` ci fossero state `do` e `od`, allora il processo avrebbe continuato ad assegnare ad `a` ora il valore 3 ora 4, sempre in maniera non deterministica, senza uscire mai: comportamenti simili sono permessi da Promela, che prevede, per uscire da un ciclo `do-od` (l'unico costrutto iterativo presente in Promela), l'uso di un `break`, un `goto` o una condizione d'uscita finale (come in un `repeat-until`).

È infine da notare che Promela considera alla stessa stregua istruzioni e condizioni,

che hanno una sintassi ancora una volta simile a quella del C. Ciò vuol dire, da un lato, che una condizione è anche un'istruzione, e quindi, se in un processo si trova

```
(a==b);
```

l'effetto sarà quello di bloccare l'esecuzione del processo stesso finché `a` non diventa uguale a `b`; dall'altro che le istruzioni possono anche non essere eseguibili, causando il blocco del processo finché non lo diventano: in particolare, gli assegnamenti sono sempre eseguibili, mentre possono non esserlo le istruzioni di invio e ricezione messaggi, delle quali si parlerà nel prossimo paragrafo.

2.1.2 I canali

Promela permette due modi di comunicazione tra processi: si possono infatti usare o variabili globali visibili ai pocessi che devono comunicare (memoria condivisa) oppure si possono definire dei canali di comunicazione lungo i quali inviare dei valori di un tipo predefinito o ridefinito dall'utente; a loro volta, i canali possono essere sia sincroni che asincroni, simulando così sia lo scambio di informazioni a *rendez-vous* che quello a *buffer*. Ogni canale può essere di uno solo dei due tipi, e può essere dichiarato sia locale che globale. Anche qui c'è un limite: non possono essere attive più di 256 code contemporaneamente.

Qualsiasi processo può usare un canale che sia nel suo *scope* sia in invio che in ricezione, ma è anche possibile dichiarare canali a sola lettura (ricezione) o a sola scrittura (invio); è inoltre possibile sapere quanti messaggi sono accodati su un dato canale in un certo momento, come anche se esso è pieno o vuoto. Le istruzioni di invio o ricezione, tuttavia, possono essere bloccanti: nel caso di comunicazione asincrona, una ricezione su un canale vuoto ha l'effetto di bloccare il processo che effettua tale ricezione finché un altro processo non manda un messaggio su quel canale; analogamente, può essere bloccante l'invio di un messaggio su un canale pieno (i canali vanno dichiarati con il massimo numero di messaggi che possono

contenere), ma in questo caso Spin permette anche di specificare, al momento di avviare una verifica o una simulazione, che eventuali messaggi mandati su un canale pieno debbano considerarsi persi, e l'esecuzione proseguire.
Nel caso della comunicazione sincrona, invece, il processo che invia su un canale si blocca finché un altro non effettua una ricezione su quel canale, e viceversa, un processo che riceve si blocca finché un altro non invia (in effetti un canale è dichiarato sincrono semplicemente dicendo che il massimo numero di messaggi che può contenere è 0).
Un caso interessante si ha quando si fanno delle ricezioni o degli invii su canali sincroni dentro un pezzo di codice che debba essere eseguito come un'istruzione atomica. In generale, la presenza di un'istruzione non eseguibile in un pezzo di codice da eseguire in `atomic` porta alla perdita dell'atomicità, e quindi si permette ad altri processi di eseguire delle loro istruzioni; pertanto, se un invio o una ricezione su un canale sincrono sono all'interno di un'`atomic` e risultano non eseguibili, il controllo passa agli altri processi, finché il *rendez-vous* non diventa possibile; allora, il controllo passa in maniera atomica dal processo che invia a quello che riceve. Per riacquistare il controllo, il processo che aveva la sequenza atomica con l'invio del messaggio deve competere con gli altri processi, seguendo le normali regole dell'*interleaving*; lo stesso accade nel caso generale: quando l'istruzione che aveva causato il blocco e la perdita dell'atomicità ridiventa eseguibile, il rispettivo processo rientra nella competizione con gli altri processi per l'esecuzione delle istruzioni.

2.1.3 Altre particolarità di Promela

Tra le altre prerogative di Promela vanno ricordati:

il *timeout*: espressione booleana vera se tutti i processi sono bloccati; può essere utile per rimediare a situazioni di stallo

il ***progess label***: etichetta un'istruzione che deve essere eseguita infinitamente spesso; usata per definire i *non-progress cycles*

l'***acceptance label***: etichetta un'istruzione che non deve essere eseguita infinitamente spesso; usata per definire gli *acceptance cycles* (presenti soprattutto nei *never claims*)

l'***end state label***: etichetta un'istruzione come stato finale valido (sono in stati finali validi anche i processi che hanno raggiunto la fine del loro corpo di istruzioni); si può decidere se occorra o no che, per considerare valido uno stato finale, tutte le code dei messaggi del rispettivo processo siano vuote

l'***assertion***: indica una condizione che deve essere vera in un certo momento dell'esecuzione di un processo.

2.2 I *temporal claims*

Come già detto, Spin (e Pan) non consentono solamente di controllare la presenza di deadlock o di altre proprietà "classiche", ma anche di definire, usando le formule LTL (Linear Temporal Logic), delle proprietà temporali (*temporal claims*) che il sistema deve rispettare, come, ad esempio, il fatto che l'invio di un certo messaggio sia sempre preceduto dalla ricezione di un altro messaggio (tipicamente, ciò viene fatto usando delle variabili di appoggio, il cui valore viene poi controllato nelle formule LTL); tali formule vengono trasformate da Spin in automi di Büchi e quindi tradotte in particolari processi Promela, che vengono aggiunti a quelli creati dall'utente e sono caratterizzati dalla parola chiave `never`. È possibile verificare una sola formula LTL per volta.

Ovviamente, le formule logiche lineari permettono di esprimere proprietà di un singolo cammino (o esecuzione); Spin estende a tutti i cammini tali proprietà, ed emette un messaggio di errore qualora trovi un'esecuzione che non le rispetti. Per ragio-

ni di efficienza, infatti, le formule devono essere in forma di *never claims*: devono cioè specificare un comportamento scorretto, che mai deve essere riscontrato. Esiste tuttavia una versione grafica di Spin, chiamata Xspin, che si occupa di facilitare il passaggio delle opzioni a Spin, e permette di fornire formule LTL anche nella versione positiva, che specifica cioè un comportamento desiderato: tali formule verranno automaticamente trasformate in un *never claim* equivalente.

2.3 Tipico uso di Spin

L'uso tipico di Spin nella sua versione non grafica è questo: innanzitutto si descrive in Promela il sistema, con eventuali etichette speciali (*end state* e *progress*) se si è interessati anche a proprietà di *liveness*; tale descrizione va fatta in un *file* di testo da passare a Spin.
Il passo successivo è quello di simulare un'esecuzione random del sorgente Promela stesso per vedere se è effettivamente conforme al sistema che si vuole testare; per fare ciò, conviene usare la versione grafica di Spin, che ha una visualizzazione delle simulazioni più intuitiva e più ricca di informazioni, e prevede anche il tracciamento del diagramma MSC dei messaggi scambiati durante l'esecuzione. Una volta che si sia sicuri della conformità del modello Promela al sistema in analisi, si può passare alla verifica, chiedendo a Spin di generare il codice del verificatore, Pan, che sarà ovviamente dipendente dalla descrizione Promela data. Se si vuole verificare anche una formula LTL (che deve essere espressa in forma negata), occorre scriverla su un *file* a parte, che poi va passato a Spin; il *never claim* corrispondente viene dato in *standard output*, e conviene pertanto redirigerlo in *append* al *file* del sorgente Promela, che ovviamente dovrà essere ricompilato da Spin. Tali operazioni sono comunque notevolmente semplificate nella versione grafica, dove basta scrivere la formula LTL in un apposito campo, scegliere se essa è negata o no e poi far partire la verifica.

2.3.1 Direttive di compilazione per Pan

Tornando alla versione non grafica, a questo punto è necessario compilare Pan (che è scritto in C, quindi bisogna avere un compilatore C), ma occorre stare attenti alle direttive da dare al compilatore: all'inizio si può compilare dando semplicemente l'ammontare della memoria RAM disponibile (`-DMEMLIM=`N, dove N è il numero di Megabytes che si vogliono destinare alla verifica), non specificando il quale si rischia il *paging* della memoria e lo stallo dell'elaboratore, qualora la memoria stessa non sia sufficiente a terminare la verifica; può però accadere che, per sistemi abbastanza grandi, Pan esaurisca la memoria a sua disposizione e termini avvisando di non essere stato in grado di portare a compimento la verifica per l'esiguità di memoria disponibile. Si può allora provare a ricompilare Pan chiedendogli di usare una tecnica di compressione migliore (`-DCOLLAPSE`), anche se ciò richiederà un tempo leggermente maggiore; se neanche questo dovesse essere sufficiente, si può comunque effettuare una verifica che dia risultati approssimati, usando o la tecnica del *bit state storage* (`-DBITSTATE`) o quella dell'*hash compaction* (`-DHC`), che necessitano di molta meno memoria. Sono inoltre previste molte altre direttive, che permettono, per lo più, di sfruttare la conoscenza del sorgente Promela per ottimizzare la verifica: ad esempio, se non si vogliono trovare cicli di accettazione o di non progresso, si può eliminare dal verificatore la parte di codice che si occupa della rilevazione di tali cicli (`-DSAFETY`). Infine, talvolta può anche capitare che i 1024 *bytes* che Pan riserva alla rappresentazione del vettore dello stato globale non bastino; in tal caso la verifica si interrompe chiedendo di aumentare il valore di `VECTORSZ`, cosa che può essere facilmente fatta con la direttiva `-DVECTORSZ=`N, dove N sarà il nuovo numero di *bytes* per lo stato globale[1].

Riassumendo, le direttive più importanti nel presente lavoro di tesi sono:

[1] In realtà, questi N *bytes* non vengono usati per le variabili globali del protocollo Promela, i *bytes* per le quali non richiedono interventi esterni.

Direttive generali

`-DMEMLIM=`N Sono disponibili N *Megabytes* di memoria

`-DSAFETY` Non effettua i controlli di *liveness*

`-DVECTORSZ=`N Per memorizzare lo stato globale vengono usati N *bytes*

`-DSC` Abilita lo *stack cycling*, vedi il paragrafo seguente.

Verifica esaustiva

`-DNOCOMP` Nessuna compressione, poco usato

`-DCOLLAPSE` Modalità di compressione

`-DJOINPROC` Possibile variante (cfr)

`-DSEPQS` Possibile variante (cfr)

Verifica non esaustiva

`-DHC` *hash compaction*

`-DBITSTATE` *bit state hashing*

2.3.2 Opzioni di esecuzione per Pan

Si può ora mandare in esecuzione il risultato della precedente compilazione, e anche qui sono disponibili varie opzioni: si può dare una stima di quanto è grande lo spazio degli stati raggiungibili, dicendo qual è l'ordine di grandezza del numero di *entries* nella *hash table*, ovvero la struttura usata per memorizzare gli stati raggiunti nella verifica; tale informazione è molto importante con il *bit state storage*, con il quale, per avere la miglior approssimazione possibile, è bene fare in modo che la *hash table* sia il più grande possibile, mentre, con le altre modalità, una *hash table* grande fa solo risparmiare tempo di esecuzione. Per attivare tale opzione, occorre

digitare `-w`N, e la *hash table* avrà 2^N *entries*[2]. Si può inoltre scegliere se effettuare controlli su proprietà di *fairness*, come la presenza o di *acceptance cycles* (`-a`) o di *non-progress cycles* (`-l`), alla quale aggiungere eventualmente la proprietà di *fairness* (`-f`); oppure se verificare proprietà di *safety* (che è l'opzione di *default*), come il controllo dell'assenza di *deadlocks*. Si può inoltre specificare a che profondità debba limitarsi la verifica (`-m`N, dove N è il numero di passi al quale arrestarsi); se tale profondità non fosse sufficiente, Pan porterà comunque a termine la verifica avvisando di aumentare la profondità stessa, onde avere una totale copertura dello spazio di stati raggiungibili; tale opzione è utile soprattutto se si sospetta che un errore possa verificarsi entro un certo numero di passi, e quindi si può limitare la ricerca a tale numero, con risparmio di tempo a volte notevole. A tal proposito, occorre dire che ci sono sistemi nei quali la profondità della verifica esaustiva è talmente grande da far sì che il *trail*, che è una delle tre strutture dati più importanti del codice di Pan (le altre sono la *hash table* e la *transition table*, vedi più avanti) e ha N *entries* se si è data l'opzione `-m`N, occupi anch'esso molta memoria RAM: una profondità di 4000000 passi porta ad un'occupazione di 100 Megabyte di memoria. In questi casi, se si vuole una verifica completa, è preferibile compilare Pan con la direttiva `-DSC`, attivando la modalità dello *stack cycling*: l'opzione `-m`N viene ora interpretata nel senso che il *trail* ha N *entries* in memoria centrale, e usa il disco come area di *swap* quando la profondità di esplorazione supera N; tale meccanismo verrà ripreso più avanti. (cfr)
Riassumendo:

`-w`N L'*hash table* deve avere 2^N *entries*

`-m`N La verifica sarà limitata ad un massimo di N passi atomici in profondità. Il *trail* avrà N locazioni (cfr).

`-a` Occorre verificare se ci sono degli *acceptance cycles*

[2]Ciò non è del tutto vero nella modalità *supertrace*, cfr

`-l` Occorre verificare se ci sono dei *non-progress cycles*

`-f` Occorre verificare la proprietà di *fairness* (da usare in combinazione alle due precedenti)

`-c`N La verifica termina dopo N errori (di *default*, $N = 1$)

Infine, in caso di errore, Pan scrive in un *file* la serie di passi che ha causato l'errore (nella terminologia di Spin, anch'essa viene chiamata *trail*, come la struttura dati usata nella verifica; ciò perché è proprio tale struttura dati a contenere la serie di passi cercata). Questo *file* non è leggibile direttamente, ma deve essere passato a Spin, che eseguirà quelle istruzioni nello stesso modo in cui esegue le simulazioni; a questo punto è facile capire cosa ha causato l'errore e cercare di porvi rimedio.
Ancora una volta, la versione grafica di Spin permette di rendere più semplici questi passi, automatizzando il passaggio delle direttive al compilatore e delle opzioni di Pan; inoltre, la visualizzazione delle simulazioni di cui si è già parlato può essere efficacemente usata per la "lettura" dei *trail* risultanti da verifiche terminate con errori: si tratta della *guided simulation*, che si avvia automaticamente non appena una verifica è terminata con esito negativo.

Capitolo 3

Il codice di Spin (cenni)

Il codice di Spin ruota tutto intorno al *file* YACC "spin.y", in cui è definita la grammatica di Promela; a coadiuvare questo *file* ce ne sono altri 37, grazie ai quali Spin prima costruisce l'albero di parsing del sorgente Promela, e poi, nel momento in cui si debba lanciare una simulazione, gli assegna una semantica.

Per quanto riguarda invece le verifiche, si è già detto che Spin genera il codice C di un programma, Pan, nel quale traduce l'albero di parsing ottenuto dal sorgente Promela (aggiungendo anche i moduli per l'analisi dello spazio di stati) e che una volta compilato e lanciato esegue la verifica. Tale codice è in gran parte fisso, cioè indipendente dalla particolare specifica Promela data, ed è definito nel *file* "pangen1.h" come un *array* di stringhe statiche; ad esso vengono aggiunte le altre parti del codice, che dipendono invece dalla specifica Promela, e che si trovano nei *file* "pangen1.c", "pangen2.h", "pangen2.c", "pangen3.c", "pangen3.h", "pangen4.c", "pangen4.h", "pangen5.c", "pangen5.h". Nei *file header* si trovano definiti segmenti di codice memorizzati nello stesso formato di "pangen1.h", mentre nei *file* con l'estensione ".c" ci sono le funzioni che combinano questi segmenti, aggiungendo anche qualche altro piccolo pezzo, fino ad ottenere il codice di Pan.

Tale codice, una volta messo insieme, viene memorizzato nella *directory* che conte-

neva il sorgente Promela, ed è diviso in 5 *files*: "pan.c", "pan.h", "pan.b", "pan.m" e "pan.t"; di questi, il primo è in gran parte indipendente dal sorgente Promela, e contiene tutto il codice elencato in "pangen1.h", "pangen4.h" e "pangen5.h" (ma questi due ultimi *files* contengono una parte di codice che non verrà trattata nel presente lavoro di tesi, cfr. anche il capitolo seguente), più una parte dipendente, comprendente le dichiarazioni delle variabili globali, contenute in "pangen2.h", e le funzioni che implementano le modifiche allo stato globale (quelle dovute ad esempio all'invio di un messaggio o al lancio di un nuovo processo), contenute in "pangen3.h"; il secondo proviene da "pangen3.h" e contiene la definizione di quasi tutte le strutture, la maggior parte delle quali sono dipendenti dal sorgente Promela, e le `define` di alcune costanti, alcune delle quali servono ad abilitare o disabilitare determinate porzioni di codice in funzione delle direttive di compilazione; gli ultimi 3 dipendono fortemente dal sorgente Promela, tanto da non avere una parte di codice fissa: essi vengono costruiti dai file "pangen1.c", "pangen2.c" e "pangen4.c".

Nel capitolo seguente si analizzerà più in dettaglio il codice di Pan, che è di importanza centrale nel presente lavoro di tesi.

Capitolo 4

Il codice di Pan

Nella figura seguente sono raffigurati schematicamente le tre strutture dati più importanti di Pan: la *transition table*, la *hash table* e il *trail*, insieme con le funzioni principali con le quali interagiscono.

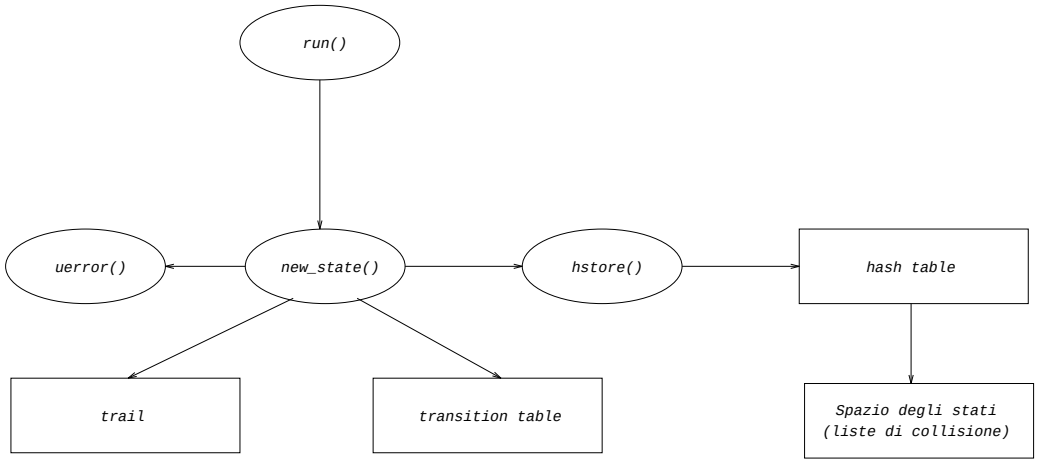

Figura 4.1: Schema del codice di Pan

In tale schema è presente la funzione `run()`, che ha il compito di inizializzare tutte le strutture dati che serviranno alla verifica, tra le quali, appunto, la *transition table* e la *hash table*, con l'unica eccezione del *trail*, inizializzato poco prima che il `main` chiami

la `run()`. A parte ciò, il `main` controlla anche che le opzioni da riga di comando siano esatte e prepara eventuali *files* di appoggio per la verifica. Un'altra importante struttura inizializzata dalla `run()` è quella che mantiene lo stato attualmente sotto esame: `now`; inizialmente essa è totalmente settata a zero, e ciò rappresenta lo stato iniziale dal quale comincerà la verifica. In effetti, gli stati attraversati dalla verifica e memorizzati di volta in volta in `now`, sono costituiti dai valori delle variabili locali e globali, dal numero di processi in esecuzione e dallo stato delle code dei messaggi; tali stati vengono opportunamente modificati dalle istruzioni Promela che vengono man mano simulate, usando per ciò la *transition table*.

Dopo aver effettuato le inizializzazioni, la funzione `run()` chiama, tramite la funzione `do_the_search()` (che ha anche il compito di settare alcuni valori iniziali del *trail* e dello stato globale) la funzione `new_state()`, che effettua, insieme con varie funzioni ausiliari, la parte centrale della verifica. Più dettagliatamente, tramite questa funzione si realizza una *depth-first search* (DFS nel seguito) di tutti comandi eseguibili del sorgente Promela in tutti gli *interleaving* possibili, usando in ciò la *hash table* per accedere allo spazio degli stati già visitati, i quali vi vengono memorizzati insieme con altre informazioni che serviranno per verificare proprietà di *liveness* o di *fairness* (sempre che tale parte di codice sia abilitata); la *transition table*, invece, viene usata per generare il prossimo stato da visitare, apportando allo stato corrente le modifiche necessarie per simulare l'effetto dell'istruzione Promela che è stata correntemente schedulata; in maniera simmetrica, essa viene usata anche per annullare gli effetti dell'ultima istruzione Promela simulata, e realizzare così il *backtracking* necessario nella procedura di verifica (per maggiori dettagli sulla *transition table*, cfr. il paragrafo ....). Per questa maniera di procedere, che prevede di generare di volta in volta lo stato da visitare, la tecnica di verifica di Spin è detta *on-the-fly*.

La verifica termina quando non ci sono più stati da visitare, ovvero si sono analizzati tutti gli stati che sono eseguibili allo stato iniziale, o anche quando sono state visitate tutte le transizioni che partono dallo stato iniziale; in questo caso l'esito della verifica

è positivo, e l'output di di Pan, in *standard output*, riporta le statistiche sulla verifica appena terminata, elencando tra l'altro numero di stati raggiunti e di transizioni effettuate, nonché informazioni sulla quantità di memoria usata, suddivisa in quella servita per le tre strutture principali; se la ricerca era in *supertrace*, riporta anche una stima della copertura media dello spazio degli stati raggiungibili.
In caso di errore viene invece chiamata la funzione `uerror()`, che scrive il contenuto del *trail* in un *file* (cfr), stampa su *standard output* un messaggio simile a quello di prima ma nel quale si avverte che è stato rilevato un errore ed esce, a meno non aver dato l'opzione `-c`N (cfr). Come già detto (cfr...), il file con il *trail* può essere usato per determinare cosa ha provocato l'errore.
Per quanto riguarda la struttura globale di Pan, è anche da notare come le variabili che puntano alle tre strutture dati principali siano globali, e tutto il codice di Pan, in generale, è basato sugli effetti collaterali che le funzioni che lo compongono hanno sulle variabili globali, tra le quali figurano anche i contatori per le statistiche e, soprattutto, la struttura `now`. Tale struttura può crescere e restringersi dinamicamente durante la verifica, a seconda che ci siano processi che entrino in esecuzione o che terminino, oppure che vengano inviati o ricevuti messaggi (in quanto ciò comporta un accrescimento o restringimento delle code dei messaggi, che sono codificate nella `now`); il tutto è gestito, anche se in maniera nascosta, dalla `new_state` (cfr...), che si serve a questo proposito di alcune funzioni ausiliarie. Le uniche modifiche non nascoste a `now` sono quelle che riguardano i suoi primi campi, che servono per le informazioni su *liveness* e *fairness*.
Si tenga comunque presente, infine, che lo schema della figura (cfr...) non è sempre valido: nel *bit state storage* lo spazio degli stati può non essere presente (cfr par), ed esiste inoltre una procedura di verifica completamente diversa da quella appena descritta; essa viene detta "esplicita" e consiste nell'utilizzo di strutture *BDD-like*, ma non verrà trattata in questo lavoro di tesi.

4.1 La *transition table*

La *transition table*, ovvero matrice di transizione, svolge un ruolo centrale nel processo di verifica. Essa viene memorizzata come una matrice, ogni singolo nodo della quale è un puntatore ad una struttura del tipo:

```
typedef struct Trans {
   short atom;   /* istruzione atomica o no */
   short st;   /* stato successivo */
   int forw;   /* indice della transizione in avanti */
   int back;   /* indice della transizione all'indietro */
   struct Trans *nxt; /* lista di transizioni eseguibili */
} Trans;
```

Ogni processo del sorgente Promela definisce un insieme di entrate in questa matrice, per la precisione una per ogni stato di flusso di esecuzione di quel processo. Una variabile globale punta alla matrice:

```
Trans ***trans;
```

La matrice di transizione viene effettivamente creata dalla chiamata che la funzione `run()` effettua alla funzione `settable()`; quest'ultima funzione, che si trova nel file "pan.t", è a sua volta creata dai file "pangen1.c" e "pangen2.c". Per quanto riguarda le modalità di questa creazione, vale la pena dire che ogni istruzione Promela viene numerata, di modo che ad ognuna di queste istruzioni corrisponde un identificatore unico; tali identificatori sono usati come indici nella matrice. Nella terminologia di Spin, inoltre, questi indici sono intesi come veri e propri indici di stati (piuttosto che di istruzioni che li generino), e in tal senso verranno usati nel seguito.

Per quanto riguarda più dettagliatamente i campi, `atom` indica che una istruzione è atomica se il suo secondo bit è a 1; `st` indica l'indice dello stato che viene raggiunto se viene effettuata con successo la transizione indicizzata da `forw`, ed indica sostanzialmente qual è l'istruzione Promela che segue quella da simulare; `forw` (ri-

spettivamente, `back`) è un intero che permette ad uno `switch` C, che è contenuto nel *file* "pan.m" ("pan.b"), di localizzare le istruzioni C che devono essere eseguite per simulare l'effetto (per annullare l'effetto) dell'istruzione Promela corrispondente alla locazione della matrice in cui esso si trova. Infine, `nxt` punta ad altre locazioni della stessa matrice: in tali locazioni ci sono le informazioni riguardanti le istruzioni Promela con le quali l'istruzione alla quale la corrente locazione fa riferimento deve competere, nel caso si trovi in un costrutto di non-determinismo. Nella terminologia di Spin, tali istruzioni in costrutti di non-determinismo vengono chiamate "opzioni" (*options*), e con tale termine saranno indicate in seguito.

4.2 Il nucleo di Pan: la funzione `new_state()`

Come detto, la `new_state()` è la funzione centrale del processo di verifica: più di metà dell'intero processo si svolge in questa procedura, e molto del tempo rimanente viene usato per calcolare i valori delle funzioni hash.
Molte porzioni di codice al suo interno sono abilitate o disabilitate a seconda che ci siano o no *rendez-vous*, *temporal claims*, *acceptance* o *progress states*, e anche a seconda del tipo di memorizzazione dello spazio di stati che si è scelto (*bit state storage*, *hash compaction* etc...). Ignorando tutte queste opzioni, la funzione `new_state()` può essere schematizzata in pseudo-codice nel seguente modo:

```
void new_state()
{
Down:
   if (la profondita' e' superiore alla massima)
      vai a "Up";
   if (lo stato corrente e' gia' stato visitato)
      vai a "Up";
   for (ogni processo attivo x)
```

```
      for (ogni "opzione" y di x da simulare)
         if (y eseguibile) simula y e aggiorna lo stato corrente;
         else passa all'opzione successiva;
/* se non ci sono piu' opzioni si passa ad un altro processo */
         salva x e y nel trail;
         incrementa la profondita';
         vai a "Down";
Up:
         recupera x e y dell'ultima transizione simulata dal trail;
         annulla l'effetto di y e aggiorna lo stato globale;
         decrementa la profondita';
      end for
   end for
   analizza lo stato corrente;
   if (c'e' un errore)
      termina;
   if (la profondita' e' maggiore di zero)
      vai a "Up";
}
```

e semplificata con qualche dettaglio in più così:

```
1   new_state()
2   {   register Trans *t, *ta;
3       char n, m, ot, lst;
4       short II, tt;
5       short From = now.nr_pr-1;
6       short To = 0;
7   Down:
```

```
8       if (depth >= maxdepth)
9          {  truncs++;
10            goto Up;
11         }
12      if (To == 0)
13         {  if (hstore((char *)&now, vsize))
14               {   truncs++;
15                   goto Up;
16               }
17             nstates++;
18         }
19      if (depth > mreached)
20         mreached = depth;
21      n = timeout = 0;
22
23  Again:
24      for (II = From; II >= To; II -= 1)
25         {   this = pptr(II);
26             tt = (short) ((P0 *)this)->_p;
27             ot = (unsigned char) ((P0 *)this)->_t;
28             for (t = trans[ot][tt]; t; t = t->nxt)
29                {
30  #include "pan.m"
31  P999:
32                   if (m>n||(n>3&&m!=0)) n=m;
33                   depth++; trpt++;
34                   trpt->pr = II;
35                   trpt->st = tt;
```

```
36                  if (t->st)
37                     {   ((P0 *)this)->_p = t->st;
38                         reached[ot][t->st] = 1;
39                     }
40                  trpt->o_t = t; trpt->o_n = n;
41                  trpt->o_ot = ot; trpt->o_tt = tt;
42                  trpt->o_To = To;
43                  if (t->atom&2)
44                     {   From = To = II; nlinks++;
45                     } else
46                          {   From = now.nr_pr-1; To = 0;
47                          }
48                  goto Down;     /* pseudo-ricorsione */
49  Up:
50                  t = trpt->o_t; n = trpt->o_n;
51                  ot = trpt->o_ot; II = trpt->pr;
52                  tt = trpt->o_tt; this = pptr(II);
53                  To = trpt->o_To;
54  #include "pan.b"
55  R999:
56                   depth--; trpt--;
57                   ((P0 *)this)->_p = tt;
58               }  /* tutte le "opzioni" */
59         } /* tutti i processi */
60      if (n == 0)
61         {   if (!endstate() && now.nr_pr)
62               {
63                  if (!timeout)
```

```
64                      {timeout=1;
65                       goto Again;
66                      }
67                   uerror("deadlock");
68               }
69         }
70      if (depth > 0) goto Up;
71  }
```

La `new_state()` viene chiamata una sola volta e non termina fino a che non ha completato il processo di ricerca o trova un errore; in questa versione semplificata, l'unico errore presente è quello dovuto ad un *deadlock* (o *invalid end-state*).
Scendendo più in dettaglio, si può notare che il grosso del lavoro è svolto da due cicli `for`. Il primo di essi, quello alla linea 24, effettua un ciclo su tutti i processi attualmente in esecuzione, mentre il secondo, alla linea 28, scorre tutte le "opzioni" (cfr) del processo corrente. Gli identificativi del processo corrente e dello stato (cfr) all'interno di quel processo vengono presi dallo stato globale `now` (`pptr(x)` è una macro C che punta alla parte di `now` relativa al processo `x`), dove erano stati messi, rispettivamente, dalla funzione `addproc` e dalla linea 37, e memorizzati nelle due variabili `ot` e `tt` rispettivamente; esse vengono usate per indirizzare la *transition table* (cfr... paragrafo precedente). I comandi C che simulano l'esecuzione dell'istruzione Promela sono nascosti, come già detto, nel *file* "pan.m", che contiene uno `switch` indicizzato da `trans[ot][tt]`; tali comandi accedono alla variabile globale `now` e la modificano di conseguenza. Se l'istruzione Promela risulta eseguibile, viene effettuato un `goto P999`, altrimenti un viene eseguito un `continue`, che porta ad esaminare l'"opzione"successiva.
Se una transizione ha avuto successo, è necessario che il nuovo stato così creato sia visitato in maniera ricorsiva dalla `new_state()`; tuttavia, per evitare le inefficienze

di una chiamata ricorsiva vera e propria, Pan effettua una pseudo-ricorsione, utilizzando dei salti incondizionati e manipolando in proprio lo *stack* necessario per gestire le chiamate ricorsive: a tal proposito, utilizza opportuni campi del *trail* per la memorizzazione degli indici della matrice di transizione e di altre informazioni necessarie direttamente alla funzione `new_state` (come ad esempio il campo `o_To`, che permette di ripristinare il limite inferiore del `for` di linea 24); mentre un'altra struttura, `stack`, che qui non compare e che è una pila vera e propria, serve per l'effettivo ripristino dello stato globale `now`, quando occorre annullare l'effetto della cancellazione di un processo o di una coda (cfr paragrafo seguente). Per quanto riguarda, invece, il *trail*, si tratta sostanzialmente di un vettore di strutture, alle locazioni del quale punta la variabile `trpt`.

Si ha, pertanto, che le linee da 32 a 42 effettuano il salvataggio sul *trail* delle informazioni che permetteranno l'esplorazione delle transizioni dello stato appena generato: il contatore di profondità `depth` viene incrementato, `trpt` punta alla struttura successiva nel vettore del *trail* (in effetti punta sempre alla `depth+1`-esima struttura). Se la transizione inizia o fa parte di una sequenza atomica, la linea 43 si assicura che il processo corrente continuerà ad essere in esecuzione anche nel passo successivo, forzando il `for` di linea 24 ad assumere, al passo successivo, solamente il valore del processo corrente. Se invece la transizione non è atomica, allora la linea 46 fa sì che tutti i processi correnti debbanmo essere presi in considerazione al prossimo passo. La ricorsione è data dal `goto Down` di linea 48.

Al ritorno dalla pseudo-ricorsione, che avviene con un `goto` all'etichetta `Up` di linea 49, tutte le variabili locali più importanti vengono ripristinate al loro valore precedente il `goto Down` per mezzo di `trpt`, il quale, a sua volta, punta esattamente alla locazione del vettore di *trail* alla quale puntava immediatamente prima del `goto Down`. A questo punto, lo stato globale `now` viene ripristinato effettuando delle istruzioni che annullano l'effetto dell'ultima istruzione esplorata; il codice corrispondente è nascosto nel file "pan.b", che ha una struttura in tutto e per tutto simile a quella

di "pan.m". Una volta terminato il segmento di codice che annulla l'operazione, viene effettuato un `goto R999`, mentre non hanno senso, qui, i `continue` presenti in "pan.m". È da notare che i `goto Up` sono tre: quello di riga 10 impedisce che si vada al di là della profondità massima, quello di riga 15 impedisce di visitare uno stato già esaminato (la funzione `hstore` controlla se lo stato da visitare è già presente nella *hash table*; se così è restituisce un valore maggiore di zero, altrimenti inserisce lo stato nella *hash table* e restituisce zero, cfr), mentre quello di riga 70 effettua il passo di pseudo-ricorsione che sarebbe stato implicito con una ricorsione vera e propria, tornando al punto dal quale la chiamata pseudo-ricorsiva era stata effettuata; inoltre, il fatto che, per arrivare a riga 70, occorra passare prima per la riga 56, permette che `trpt` punti esattamente alla locazione dove puntava prima del `goto Down`, mentre la riga 57 ripristina in `now` il corretto indice di stato.

Infine, l'etichetta `Again` di linea 23, insieme con il rispettivo `goto` di linea 65, è usata per implementare il meccanismo del `timeout` (cfr...). Se la verifica, ad un certo punto, non trova più transizioni effettuabili, ciò viene innanzitutto notato alla linea 60, dove `n` varrà zero. A questo punto, si controlla velocemente se non si sia in uno stato finale valido (cfr...); se così non è , si setta a 1 la variabile `timeout`, che consentirà ad una eventuale condizione `timeout` di Promela di essere abilitata, e si ritorna all'etichetta `Again` per effettuare un secondo tentativo di effettuare una transizione. Se anche ciò fallisce, viene chiamata la funzione `uerror()`, che salva su un file i campi `pr` e `o_t->t_id` delle locazioni del vettore del *trail* che vanno dalla prima alla `depth`-esima (sarà questo il *file* del *trail*, quello che permette di scoprire l'errore commesso, cfr...); fatto ciò , stampa le statistiche su *standard output* (cfr...) e termina la verifica.

La verifica termina senza errori nel momento in cui si arrivi alla linea 70 con `depth` che vale 0: ciò equivale a dire che tutte le transizioni dello stato iniziale sono state esplorate, e il controllo a questo punto può tornare al `main`, che stampa le statistiche ed esce.

4.2.1 Alcune funzioni ausiliarie

I file "pan.b" e "pan.m", per svolgere i loro compiti e modificare opportunamente `now`, fanno uso di alcune funzioni standard, che risultano però dipendenti dal sorgente Promela; tali funzioni riguardano la gestione dei processi e delle code dei messaggi.

Per quanto riguarda le code, tali funzioni sono:

`addqueue()`

`delq()`

`q_restor()`

`qsend()`

`qrecv()`

`unsend()`

`unrecv()`

La prima crea una coda quando viene dichiarato un canale Promela; la seconda distrugge una coda esistente non appena il processo che l'ha creata termina; la terza, usata nelle transizioni all'indietro di "pan.b", annulla l'effetto di una `delq()` ripristinando l'ultimo stato noto (per far questo usa la variabile `stack`); la quarta e la quinta, rispettivamente, accodano e tolgono da una coda un messaggio in seguito ad un invio o ad una ricezione; la sesta e la settima annullano (in "pan.b") gli effetti di `qsend()` e `qrecv()`.

Per quanto riguarda i processi, si ha invece:

`addproc()`

`delproc()`

`p_restor()`

La prima crea un processo quando il sorgente Promela ne manda in esecuzione uno; la seconda distrugge un processo che ha terminato la sua esecuzione (ma può essere anche usataper annullare l'effetto di una `addproc()`); la terza annulla l'effetto di una `delproc()`.

Tali funzioni, al loro interno, chiamano anche quelle relative alle code: ad esempio, una `addproc()` può chiamare una `addqueue()` se all'interno del processo da creare è dichiarata una coda.
Infine, a proposito delle funzioni che creano processi o code, occorre dire che, per ogni processo e per ogni diferente tipo di canale Promela, vengono dichiarate in "pan.h" delle opportune strutture, che vengono poi usate per gestire correttamente le modifiche a `now`.

4.3 La *hash table*

Dal punto di vista del codice C, la *hash table* è un vettore la cui dimensione viene scelta con un opzione da riga di comando (cfr...). A tale vettore punta la variabile `H_tab`; come detto (cfr...), la funzione che effettua l'interfacciamento tra la funzione *new_state* e la *hash table* è la *hstore*. Essa può essere schematizzata in pseudo-codice in questo modo:

```
int hstore(char *vin, int nin)
{
#if (non comprimere)
   v = vin;
#else
   v = comprimi(vin);
/* nel caso della hash compaction, calcola anche j1 */
#endif
#if (non definita hash compaction)
   j1 = valore della funzione hash della chiave v;
#endif
   if (H_tab[j1] e' libera)
      crea in H_tab[j1] una lista di collisione con un nodo;
```

```
   else
      for (ogni nodo n nella lista di collisione di H_tab[j1])
#if (collapse compression)
          if (n e' un frammento di stato)
             if (n e' l'ultimo elemento della lista)
                inserisci un nuovo nodo alla fine della lista ed esci dal for;
             end if
             passa direttamente al nodo successivo;
          end if
#endif
          if (v e' uguale allo stato in n)
#if (gestisci proprieta' di liveness, con o senza fairness)
             modifica opportunamente i primi byte dello stato in n;
#endif
             return (un valore maggiore uguale di 1);
          else if (v e' lessicograficamente minore dello stato in n)
             inserisci un nuovo nodo prima di n;
             esci dal for;
               else if (n e' l'ultimo nodo)
                  inserisci un nuovo nodo alla fine della lista;
                  esci dal for;
/* in questo modo la lista e' lessicograficamente ordinata
   sui byte degli stati memorizzati */
                    end if
               end if
          end if
      end for
   end if
```

```
    copia v nell'apposito campo del nodo appena creato;
    return 0; /* stato nuovo */
}
```

Come si vede, la *hash table* di Pan può essere usata in almeno tre maniere differenti, a seconda delle direttive che sono state date al compilatore; esse saranno esaminate nei paragrafi seguenti.

4.3.1 Le modalità di non compressione e di *byte masking*

Nelle modalità di non compressione (`-DNOCOMP`), o di compressione col *byte masking* (*default*), la *hash table* viene gestita con le tradizionali liste di trabocco: lo stato globale viene o no compresso nella maniera scelta, e il risultato viene passato ad una funzione, `s_hash()`, che calcola qual è la sua locazione nel vettore dell'*hash table*; se tale locazione risulta la lista di collisione vuota, si crea una lista di collisione con un elemento e lo stato, compresso o no, vi viene memorizzato; altrimenti, si scorre la lista per vedere se lo stato che si vuole inserire è già presente: se così è non sono necessarie modifiche (o almeno non lo sono se non si è interessati alle proprietà di *liveness*), altrimenti lo stato viene inserito nella lista. La lista, per intuibili motivi di efficienza, viene mantenuta lessicograficamente ordinata.

4.3.2 La *hash compaction*

Con la *hash compaction* (`-DHC`) la gestione resta la stessa, tranne che per il fatto che compressione e calcolo della funzione *hash*, che questa volta è *r_hash*, avvengono insieme; inoltre la compressione è fatta in modo che non vengano superati un certo numero di *bytes* (48 di *default*). Questa costrizione porta talvolta a perdita di informazione, per cui può succedere che due stati diversi vengano ritenuti invece uguali, causando in tal modo una non perfetta copertura dello spazio degli stati raggiungibili.

4.3.3 La *collapse compression*

L'ultima modalità è quella della *collapse compression* (`-DCOLLAPSE`), che è la migliore modalità di verifica, in termini di occupazione di memoria, tra le possibili modalità di esplorazione esaustiva dello spazio degli stati raggiungibili, e si basa sull'assunto che il grande numero di stati raggiungibili generato da Pan sia dovuto alle molteplici combinazioni di pochi stati locali esistenti nei singoli processi. In essa uno stato, quando deve essere memorizzato nella *hash table*, viene spezzato in più frammenti, che vengono inseriti in vari punti della *hash table* stessa. Per essere più precisi, si memorizzano separatamente nelle liste di collisione dell'*hash table* gli stati locali dei singoli processi[1] (questa è l'opzione di *default*, ma con la direttiva `-DJOINPROCS` tutti i processi vengono memorizzati insieme in un'unico nodo di una lista di collisione), poi tutti insieme quelli delle code dei messaggi (o separandoli, con la direttiva `-DSEPQS`), poi quello delle variabili globali, e infine si memorizza un vettore che contiene degli identificatori unici dei frammenti di stato memorizzati finora, identificatori che vengono generati al momento dell'inserzione nell'*hash table*. Tale vettore è memorizzato sempre nella stessa *hash table*, ma un campo del nodo della lista di collisione in cui si trova permette di distinguerlo dai frammenti di stato: questi ultimi hanno in tale campo il numero di byte che occupano (occorre precisare che essi sono anche compressi ulteriormente col *byte masking*), mentre il vettore degli identificatori vi ha il valore zero; è da notare che ciò permette di ordinare queste informazioni decrescentemente anche in base alla loro grandezza, facendo sì che i vettori degli identificatori, che vengono confrontati meno spesso dei frammenti di stati, siano sempre in fondo alla lista. Se uno stato è già stato messo nell'*hash table*, vuol dire che tutte le sue singole parti sono già state memorizzate, pertanto la fase di inserimento dei frammenti dello stato terminerà avendo rilevato che essi sono tutti già presenti e ritornando i rispettivi identificatori unici; il vettore di questi identifi-

[1] Tali stati sono costituiti, sostanzialmente, dal valore delle variabili locali dei processi stessi.

catori verrà quindi trovato come già memorizzato, e si potrà concludere che lo stato è già stato visitato.
Buona parte di queste operazioni sono effettuate nella funzione `compress`, che si occupa dei frammenti di stato, sfruttando anche la funzione `ordinal`, che li memorizza effettivamente; nella funzione `hstore`, invece, vengono memorizzati solo i vettori di identificatori.

4.3.4 La modalità *supertrace* (cenni)

Nella modalità di *supertrace* (`-DBITSTATE`), l'uso della *hash table* è molto diverso: non ci sono più liste di trabocco (sempre che non si vogliano anche proprietà di *liveness*), e il vettore viene visto come un insieme di bit che indicano se uno stato è stato visitato o no. Il procedimento è pertanto questo: si calcola una doppia funzione *hash*, `d_hash`, e se entrambi i bit che occupano le posizioni risultanti sono settati a 1, si assume che lo stato sia stato visitato.
Il funzionamento della funzione `hstore` verrà ripreso con più dettagli nel paragrafo cfr.

Capitolo 5

Una nuova versione di Spin

Si è detto nel primo capitolo (cfr) che ciò che si intende fare è introdurre un approccio grazie al quale sia possibile usare meno memoria per portare a termine una verifica, pagando tale miglioria con un allungamento dei tempi di esecuzione. Come accennato, ciò sarà fatto permettendo che si possa "dimenticare" di aver visitato alcuni stati, che verranno perciò processati più d'una volta. In termini di codice, ciò si concretizza nel limitare ad uno la lunghezza delle liste di collisione: se uno stato deve essere messo in una locazione dell'*hash table* già occupata, lo stato in questione andrà a sovrascrivere quello già presente; se poi quest'ultimo stato verrà ritrovato in seguito durante il processo di verifica, sarà esaminato come se fosse nuovo. Ovviamente, affinché la verifica termini è necessario che ci sia un numero di collisioni piccolo rispetto al numero di stati del protocollo da verificare; per ottenere ciò, si può semplicemente aumentare la dimensione dell'*hash table*, cosa che Spin permette di fare facilmente (cfr). Tuttavia, così facendo si rischia che la *hash table* stessa occupi molta memoria, laddove solitamente il maggior consumo di memoria è dovuto alle liste di collisione, che, nelle modalità esaustive, arrivano a contenere l'intero insieme degli stati raggiungibili. È pertanto necessario, per questo tipo di verifica, partire con una dimensione dell'*hash table* piccola, e poi aumentarla finché non si

riesce a completare la verifica (la versione di Spin qui proposta include un'efficace euristica che permette di rilevare i casi in cui la verifica non può terminare); tale procedimento è stato effettivamente usato nella fase di *testing* (cfr).
L'idea non sarà però applicata a tutte le modalità di verifica:

Non trattate:

***bit state storage*:** non prevede liste di collisione (cfr), e pertanto una modifica come quella che si intende fare non ha senso;

`-DMA`: è una verifica cosiddetta "esplicita", basata su particolari strutture dati chiamate *BDD*, e segue uno schema completamente diverso dalla procedura descritta nel capitolo (cfr)

Trattate:

Nessuna compressione è di uso assai raro e non verrà trattata nel *testing* (cfr)

Byte masking

Hash compaction

Collapse compression

Le prime tre modalità tra quelle trattate richiedono modifiche comuni, mentre la *collapse compression* è la più complessa da modificare, e anzi richiede qualche correzione allo schema visto.
Per quanto riguarda i file sorgente, ne saranno modificati solo 3: "pangen1.h" per il codice delle funzioni, "pangen2.h" per alcune dichiarazioni e "pangen3.h" per alcune modifiche al codice dipendente da Promela.
Nel seguito, si farà riferimento anche a parti del codice trascurate nell'analisi del paragrafo (cfr...); le modifiche sono attivabili con la direttiva al compilatore `-DRANDOM_SPIN`.

5.1 Le modifiche al codice di Pan

Per realizzare quanto detto non è sufficiente "tagliare" le liste di trabocco della *hash table*: occorre anche tener presente degli effetti che tale modifica ha sull'intero processo di verifica.

Il più evidente di tali effetti lo si ha sul *trail*: infatti, la struttura `trail[depth]` (`trail` è una variabile globale che punta alla seconda locazione del vettore del *trail*[1]) ha un campo, `ostate`, che punta, per questioni di efficienza nelle verifiche di proprietà di *liveness* o di un *never claim*, al nodo della lista di collisione della *hash table* che contiene lo stato visitato a profondità `depth` dell'attuale DFS; con la modifica che ci si propone di fare, questo puntatore, in seguito ad un conflitto sulla *hash table* e alla sovrascrittura dello stato, potrebbe puntare ad uno stato sbagliato. Inoltre, tale puntatore viene sfruttato sia in lettura che in scrittura dell'area di memoria corrispondente, e in questo secondo caso le modifiche apportate possono essere accedute da una successiva chiamata di `hstore`.

Un altro problema è dato dalla gestione della memoria: Pan, ancora una volta per questioni di efficienza, usa una funzione, `emalloc`, che alloca la memoria "per sempre", senza permetterne il rilascio; inoltre, questa funzione alloca più memoria di quanta non se ne chieda, e poi usa la memoria in eccesso per servire le richieste di memoria successive, risparmiando così sul fattore tempo.

Questi due problemi verranno affrontati nei due paragrafi seguenti.

5.1.1 Le modifiche per il *trail*

Si è detto che il campo `ostate` del *trail* modifica dei nodi delle liste di trabocco dell'*hash table*. Più in dettaglio, data la struttura di un componente delle liste di trabocco[2]:

[1] La seconda locazione del vettore è `trail[0]`, la prima è `trail[-1]`, in quanto può succedere che ci sia una lettura in `trpt-1`; è una invariante del codice di Pan che `trpt` punti a `trail[depth]`

[2] Ovviamente, il vettore dell'*hash table* contiene puntatori a strutture di questo tipo.

```
struct H_el {
   struct H_el *nxt; /* lista di collisione */
#ifdef FULLSTACK
   unsigned tagged; /* serve per le proprieta' di liveness */
#if defined(BITSTATE) && !defined(NOREDUCE) && !defined(SAFETY)
   unsigned proviso;
#endif
#endif
#if defined(CHECK) || (defined(COLLAPSE) && !defined(FULLSTACK))
   unsigned long st_id;
#endif
#ifdef COLLAPSE
#if VECTORSZ<65536
   unsigned short ln; /* distingue frammenti di processi
                         da vettori di identificatori (cfr)*/
#else
   unsigned long ln;
#endif
#endif
#ifdef REACH
   unsigned D;
#endif
   unsigned state; /* stato (compresso) */
} **H_tab; /* puntatore all'intera hash table */
```

i campi modificati sono il secondo e l'ultimo[3]; la dichiarazione del campo `ostate` è ovviamente

[3]In realtà, viene modificato anche il terzo, ma ciò avviene solo nella modalità `BITSTATE`, alla quale non è applicabile la modifica di Spin proposta

```
struct H_el *ostate;
```

In particolare, il puntatore `ostate` accede al campo `state`, sia in lettura che in scrittura, solamente nel seguente modo:

```
((char *)&(trpt->ostate->state))[0]
```

Per quanto riguarda il passaggio ad `ostate` dell'indirizzo a cui deve puntare, esso avviene tramite una variabile globale, `Lstate`, che ha lo stesso tipo di `ostate` e che viene settata opportunamente dalla funzione `hstore` una volta che sia stato trovato il giusto posto dello stato corrente nella lista di collisione; una volta che il controllo è ritornato alla funzione `new_state`, se lo stato risulta essere non visitato viene effettuata una chiamata alla macro `onstack_put()`, che si limita ad assegnare `Lstate` a `ostate`. La modifica al campo `tagged` avviene tramite un'altra macro, `onstack_zap()`:

```
#define onstack_zap()   { \
   if (trpt->ostate) \
      trpt->ostate->tagged = \
      (S_A)? (trpt->ostate->tagged&~V_A) : 0; \
   }
```

dove `S_A` e `V_A` sono una variabile ed una costante usate nel controllo delle proprietà di *liveness* e *fairness*. È importante notare che `onstack_put()`, ad un dato livello di profondità nella DFS, viene chiamata sempre prima di `onstack_zap`, e pertanto quest'ultima non modifica solo il campo `tagged` di `ostate`, come potrebbe sembrare a prima vista, ma anche quello della struttura `H_el` in cui lo stato compresso è stato memorizzato.

Il problema è stato pertanto risolto nella seguente maniera: al posto del puntatore `ostate`, nella struttura (chiamata `Trail`) che compone il *trail* è stata messa una struttura che contiene lo stato in esame (non compresso), un `char`, uno `short int` e un *unsigned int*:

```
#ifdef RANDOM_SPIN
struct H_el_rid {
   unsigned tagged;
   short used;
   char cvo;
   State state;
} H_el_rid;
#endif /* RANDOM_SPIN */
```

Il primo campo rifletterà le modifiche all'omonimo della struttura `H_el`, il terzo quelle del campo `state` di `H_el`; il secondo serve per capire se i dati della struttura sono validi, mentre il quarto campo, come si vedrà poco più avanti, è necessario per sapere se e dove effettuare le modifiche nell'*hash table*; a tal proposito, si precisa che `State` è la struttura dello stato globale, ed è quindi il tipo di `now`.
La struttura `Trail` contiene pertanto una modifica:

```
#ifndef RANDOM_SPIN
   struct H_el *ostate;
#else /* RANDOM_SPIN */
   struct H_el_rid ostate;
#endif /* RANDOM_SPIN */
```

di modo che `ostate` è ora una struttura statica. Ciò , naturalmente, aumenta di molto l'occupazione di memoria dovuta al solo *trail*, dato che si è sostituito un puntatore con una struttura. Per evitare ciò, la direttiva `-DRANDOM_SPIN` attiva la costante `SC`, che abilita la parte di codice che permette di sfruttare il disco come area di *swap*; in tal modo, nell'opzione da riga di comando `-m`N, il valore di N (che ora determina quante caselle del *trail* vano mantenute in memoria centrale) potrà essere molto basso, riducendo l'uso di memoria centrale per il *trail*. Le modifiche alla struttura `H_el_rid` sono effettuate negli stessi punti in cui, nel codice "normale" di

Pan, si modifica `ostate`. A tal proposito, le macro `onstack_put()` e `onstack_zap()` sono diventate funzioni:

```
#ifdef RANDOM_SPIN
void onstack_zap()
{
   if (trpt->ostate.used)
      {
         trpt->ostate.tagged = (S_A)? (trpt->ostate.tagged&~V_A) : 0;
         if (hpresent((char *)&(trpt->ostate.state), trpt->ostate._vsz, 1))
            H_tab[j1]->tagged=trpt->ostate.tagged;
      }
}

void onstack_put()
{
   trpt->ostate.tagged = Lstate->tagged;
   trpt->ostate.cvo = ((char *)&(Lstate->state))[0];
   memcpy((char *)&(trpt->ostate.state), (char *)&now, vsize);
   trpt->ostate.used = 1;
}
#endif /* RANDOM_SPIN */
```

In `onstack_put()`, come si vede, sono state salvate le due informazioni che vengono modificate tramite `ostate`; in più è stato salvato lo stato globale (`vsize` è una variabile globale che mantiene il numero effettivo di *bytes* occorrenti per memorizzare lo stato globale `now`; è replicata nel campo `_vsz` di `State`), e si è abilitato il *flag* di validità.

In `onstack_zap()`, invece, la modifica al campo `tagged` viene propagata anche alle strutture anche all'omonimo campo della locazione di `H_tab` nella quale è stato memorizzato lo stato; per determinare questa locazione, è stata creata una nuova funzione, `hpresent`, che ricalca il codice della funzione `hstore` (cfr...), limitandosi però a dire se lo stato è o no presente nella *hash table*, senza apportare alcuna modifica a quest'ultima; la variabile globale `j1` è quella in cui le funzioni `s_hash` e `r_hash` (cfr...) memorizzano l'indice della tabella da loro calcolato.
Ovviamente, anche tutte le altre istruzioni che riguardassero `ostate` sono state modificate. L'inizializzazione è stat modificata nel seguente modo:

```
#ifndef RANDOM_SPIN
   trpt->ostate = (struct H_el *) 0;
#else /* RANDOM_SPIN */
   memset((char *)&trpt->ostate, 0, sizeof(H_el_rid));
#endif /* RANDOM_SPIN */
```

Un pezzo di codice con una modifica a `ostate` è invece diventato:

```
#ifdef VERI /* se occorre verificare una formula LTL */
#ifndef RANDOM_SPIN
   if ((trpt-1)->ostate)
      ((char *)&((trpt-1)->ostate->state))[0] |= 128;
#else /* RANDOM_SPIN */
   if ((trpt-1)->ostate.used)
      {
         (trpt-1)->ostate.cvo |= 128;
         if (hpresent((char *)&((trpt-1)->ostate.state),
                     (trpt-1)->ostate.state._vsz, 1))
            ((char *)&(H_tab[j1]->state))[0] = (trpt-1)->ostate.cvo;
      }
```

```
#endif /* RANDOM_SPIN */
#else /* !VERI */
#ifndef RANDOM_SPIN
   ((char *)&(trpt->ostate->state))[0] |= 128;
#else /* RANDOM_SPIN */
   trpt->ostate.cvo |= 128;
   if (hpresent((char *)&(trpt->ostate.state), trpt->ostate.state._vsz, 1))
      ((char *)&(H_tab[j1]->state))[0] = trpt->ostate.cvo;
#endif /* RANDOM_SPIN */
#endif /* !VERI */
```

dove la funzione `hpresent` è usata nello stesso modo di sopra.
Un pezzo di codice con un accesso in lettura a `ostate`, facente parte della condizione di un `if`, è stato modificato così :

```
#ifdef VERI
#ifndef RANDOM_SPIN
    (trpt-1)->ostate &&
   !(((char *)&((trpt-1)->ostate->state))[0] & 128))
#else /* RANDOM_SPIN */
    (trpt-1)->ostate.used &&
   !((trpt-1)->ostate.cvo & 128))
#endif /* RANDOM_SPIN */
#else
#ifndef RANDOM_SPIN
   !(((char *)&(trpt->ostate->state))[0] & 128))
#else /* RANDOM_SPIN */
   !(trpt->ostate.cvo & 128))
#endif /* RANDOM_SPIN */
```

Tutti questi pezzi di codice erano nella funzione `new_state`; altre due modifiche sono state fatte nella funzione `checkcycles()`:

```
#ifndef RANDOM_SPIN
   struct H_el *sv = trpt->ostate; /* save */
#else /* RANDOM_SPIN */
   struct H_el_rid sv;
   memcpy((char *)&sv, (char *)&trpt->ostate, sizeof(H_el_rid));
#endif /* RANDOM_SPIN */
```

⋮

```
#ifndef RANDOM_SPIN
   trpt->ostate = sv;   /* restore */
#else /* RANDOM_SPIN */
   memcpy((char *)&trpt->ostate, (char *)&sv, sizeof(H_el_rid));
#endif /* RANDOM_SPIN */
```

5.1.2 Le modifiche alla funzione `emalloc`

Ogni volta che sia necessario allocare memoria, Pan usa la funzione `char *emalloc(unsigned long n)`, dove l'argomento, ovviamente, indica la quantità di memoria da allocare. Questa funzione, dopo aver "allineato" il valore di `n` con `sizeof(void *)`, prova a vedere se la memoria in più allocata in una chiamata precedente è di almeno `n` *bytes*; se così è, toglie `n` *bytes* dal contatore (`left`) di questa memoria rimasta e sposta in avanti, sempre di `n` *bytes*, una variabile, `have`, che punta all'interno dell'area di memoria allocata; il valore restituito dalla funzione è quello di `have` precedente quest'ultimo spostamento. Se invece la memoria già allocata non basta, allora il frammento di questa memoria che non è stato usato viene considerato perso, e si alloca una nuova area di memoria; prima di far ciò, però, si controlla che `n` sia almeno 100 volte la dimensione massima dello stato globale, contenuta nella costante

`VECTORSZ` (cfr anche...); se così è, verranno allocati `n` *bytes*, altrimenti i *bytes* allocati saranno `100*VECTORSZ`: l'idea è evidentemente quella di evitare di fare troppe allocazioni (ad ogni stato inserito nella *hash table* corrisponde una chiamata alla funzione `emalloc`), allocando sempre almeno la memoria che serve per 100 stati. L'area di memoria allocata viene poi totalmente settata a zero.
L'allocazione vera e propria viene fatta dalla funzione `Malloc`, che controlla che la quantità di memoria allocata fino a quel momento, con in più quella che occorre allocare adesso, non esca dal limite imposto a tempo di compilazione (`-DMEMLIM`, cfr...); se così è, stampa una statistica sull'uso della memoria fino a quel momento, fa stampare le statistiche sullo stato della verifica fino a quel momento e poi fa terminare il programma. Se invece la richiesta di memoria rientra nei limiti imposti, viene chiamata o la funzione C `malloc` o la funzione C `sbrk`, a seconda che a tempo di esecuzione sia stata o no specificata la direttiva `-DPC`; viene poi opportunamente incrementato il valore del contatore globale della memoria utilizzata (`memcnt`) e restituito un puntatore all'area di memoria appena allocata.
Le modifiche a queste funzioni sono semplici: nella funzione `emalloc`, ci si limita a fare l'allineamento, a chiamare la funzione `Malloc` ed effettuare il settaggio dell'area di memoria restituita; nella funzione `Malloc`, invece, si alloca solo con la funzione `malloc` (la funzione `sbrk` non permette deallocazioni), ed inoltre vi è un altro contatore globale, `memtot`, che mantiene il massimo della memoria usata fino a quel momento (senza deallocazioni, il valore di questo contatore coincide con quello di `memcnt`).

5.1.3 Le modifiche per la *hash table*

Le modifiche più importanti sono ovviamente quelle che riguardano la *hash table*. Innanzitutto, occorre modificare la definizione della struttura `H_el`[4] (cfr):

[4]Vengono riportati solo i campi che interessano

```
struct H_el {
   struct H_el *nxt; /* lista di collisione */
#ifdef FULLSTACK
   unsigned tagged; /* serve per le proprieta' di liveness */
#endif
#ifdef COLLAPSE
#if VECTORSZ<65536
   unsigned short ln; /* distingue frammenti di stato
                         da vettori di identificatori (cfr) */
#else
   unsigned long ln;
#endif
#endif
#ifdef RANDOM_SPIN
#if VECTORSZ<65536
   unsigned short vsize;
#else
   unsigned long vsize;
#endif
#endif /* RANDOM_SPIN */
   unsigned state; /* stato (compresso) */
} **H_tab; /* puntatore all'intera hash table */
```

Come si può vedere, la modifica consiste nell'aggiungere il campo `vsize`, che tornerà utile nella gestione del contatore `memcnt` (cfr...) nelle deallocazioni.
Le altre modifiche riguardano tutte la funzione `hstore`. La prima parte della funzione non cambia: restano intatte le chiamate alle funzioni che comprimono lo stato e a quelle che calcolano la locazione *hash* dello stato stesso; occorre però dire che nella *collapse compression*, la compressione avrà anche l'effetto di memorizzare nell'*hash*

table, tramite la funzione `ordinal`, i frammenti dello stato in esame (cfr), mentre la funzione `hstore` si occuperà della memorizzazione del vettore degli identificatori. Le modifiche alla funzione `ordinal` verranno esposte più avanti (cfr). Nel seguito, quando si parlerà di stato da inserire, si intenderà la sua versione compressa con una delle modalità trattate.

Dopo la compressione ed il calcolo della funzione *hash*, si ha:

```
   tmp = H_tab[j1]; /* j1: locazione calcolata dalle funzioni hash */
   if (!tmp)               /* locazione libera */
   {  tmp = grab_state(n); /* n: bytes occorrenti per lo stato compresso */
      H_tab[j1] = tmp;
#ifdef RANDOM_SPIN
      tmp->vsize=n;
#endif /* RANDOM_SPIN */
   } else                  /* locazione occupata */
   {
#ifndef RANDOM_SPIN
    for (;; hcmp++, olst = tmp, tmp = tmp->nxt)
      {
#endif /* RANDOM_SPIN */
#ifdef COLLAPSE
      if (tmp->ln != 0)
      {
#ifndef RANDOM_SPIN
         if (!tmp->nxt) goto Append;
         continue;
#else /*RANDOM_SPIN*/
         goto Alloc;
#endif /*RANDOM_SPIN*/
```

```
}
```

A commento di questa porzione di codice è utile ricordare che `H_tab` è il puntatore all'intero vettore della *hash table*, e precisare che la macro `grab_state(n)` alloca un'area di memoria per una struttura di tipo `H_el`, ma togliendo la quantità di memoria necessaria per l'ultimo campo e aggiungendo poi `n` *bytes*, ovvero quanti ne servono per memorizzare lo stato (compresso) nella *hash table*. Le modifiche comuni alle quattro modalità di compressione consistono nel salvare nel campo `vsize` il numero di *bytes* necessari per lo stato (compresso) e nel disabilitare il `for` che scorre la lista di collisione. Nel caso della modalità *collapse compression*, occorre, se la locazione della *hash table* risulti già occupata, controllare anche il campo `ln`: se non è zero, vuol dire che la locazione dell'*hash table* è usata per un frammento di stato e non per un vettore di identificatori (cfr), e dato che nella `hstore` vengono memorizzati solo questi ultimi, si può andare direttamente all'etichetta `Alloc`, dove il frammento di stato verrà sovrascritto; altrimenti, si va avanti come nella altre modalità.

Segue una porzione di codice che resta invariata, nella quale avvengono confronti tra lo stato che si vuole inserire con quelli già presenti (naturalmente, con le modifiche apportate questi stati si riducono ad uno); se uno di essi risulta uguale allo stato da inserire, la funzione `hstore`, effettuate alcune modifiche allo stato già presente, ritorna il controllo alla funzione `new_state`. Tali modifiche sono attivate solo nel caso di verifiche con proprietà di *liveness*, e riguardano i primi *bytes* dello stato stesso, nonché il campo `tagged`; è da notare che i *bytes* così modificati non prendono parte al confronto. Inoltre il valore di ritorno della funzione `hstore` sarà maggiore di zero, e varrà 1, 2 o 3 in funzione, tra le altre cose, del valore di *tagged* Tutta questa porzione di codice si trova nel ramo `else` del pezzo di codice riportato sopra, ovvero nel caso che la locazione della *hash table* in esame non sia libera.

Dopodiché, si ha:

```
   } else /* lo stato da inserire non e' uguale a quello puntato da tmp */
#ifndef RANDOM_SPIN
       if (m < 0) /* se e' "minore" */
          {   /* inserzione dello stato prima di tmp */
             ntmp = grab_state(n);
             ntmp->nxt = tmp;
             if (!olst)
                H_tab[j1] = ntmp;
             else
                olst->nxt = ntmp;
             tmp = ntmp;
             break; /* dal for di cui sopra */
          } else if (!tmp->nxt)
          {   /* aggiunta dello stato alla fine della lista */
Append:      tmp->nxt = grab_state(n);
             tmp = tmp->nxt;
             break; /* idem */
      }   }/* for tmp */
#else /*RANDOM_SPIN*/
Alloc:
      {
         if (tmp->vsize!=n)
            {
#ifdef MEMCNT
               memcnt -= sizeof(struct H_el)-sizeof(unsigned)+tmp->vsize;
#endif
               free(tmp);
               tmp=grab_state(n);
```

```
                tmp->vsize = n;
                H_tab[j1]=tmp;
            }
#ifdef COLLAPSE
          else tmp->ln = 0;
#endif
          hcmp++;
       }
#endif /*RANDOM_SPIN*/
```

Questo segmento di codice si trova all'interno del `for` di cui si è parlato prima; in tale segmento, la variabile `m` è il risultato di una chiamata alla funzione C `memcmp`, con la quale si confrontano lo stato attuale compresso e gli stati della lista di collisione (ora ridotti ad uno); sfruttando il valore di ritorno memorizzato in `m`, la lista viene mantenuta ordinata nella versione di Pan non modificata. La modifica consiste qui nel preparare la sovrascrittura dello stato già presente nella *hash table* con il nuovo stato, dato che essi sono diversi: se lo stato da inserire necessita della stessa quantità di memoria di quello già presente non occorre far nulla, altrimenti si dealloca l'area di memoria presente e se ne alloca un'altra, su misura per il nuovo stato, aggiornando il contatore di memoria `memcnt` (cfr...). In ogni caso, viene incrementato il contatore delle collisioni, `hcmp` e viene posto a zero il campo `ln` nella *collapse compression*, ad indicare che si tratta di un vettore di identificatori: se avviene la riallocazione, ciò è garantito dal fatto che la macro `grab_state`, tramite la funzione `emalloc`, setta a zero tutta l'area di memoria appena allocata (cfr), altrimenti occorre farlo esplicitamente.

La funzione `hstore` prosegue infine senza altre modifiche, memorizzando nell'area di memoria allocata lo stato e le altre informazioni; questo, naturalmente, se lo stato non era già presente nell'*hash table*, altrimenti la funzione `hstore` sarebbe terminata prima.

5.1.4 Altre modifiche

È stato infine necessario modificare la funzione che rintraccia, sul *trail*, il punto dal quale comincia un ciclo di non-progresso o di accettazione, una volta che tale ciclo sia stato trovato (questa informazione servirà alla costruzione del *file* di *trail*):

```
int
depth_of(struct H_el *s)
{  Trail *t; int d;
#ifdef RANDOM_SPIN
   char *v;
   int n;

   v = (char *) &comp_now; /* solo cosi' la funzione "compress" funziona */
   n = compress((char *)&(t->ostate.state), t->ostate.state._vsz);
   /* v punta ora allo stato compresso */
#endif /* RANDOM_SPIN */
   for (d = 0; d <= A_depth; d++)
   {   t = getframe(d);
#ifndef RANDOM_SPIN
      if (s == t->ostate)
#else /* RANDOM_SPIN */
      if (t->ostate.used && (s->tagged == t->ostate.tagged) &&
          (!memcmp((char *)&s->state, v, n)))
#endif /* RANDOM_SPIN */
         return d;
   }
   printf("pan: cannot happen, depth_of\n");
   return depthfound;
```

```
}
```

Come si può vedere, nella versione di Pan non modificata bastava fare un confronto tra due puntatori, `s` e `t->ostate`; ora, però, il campo `ostate` non è più un puntatore. È pertanto necessario, sempre che il campo `used` indichi che i dati sono validi, confrontare i campi di della struttura di `ostate` (ovvero la `H_el_rid`, cfr...) con gli omologhi puntati da `s`. Per confrontare il campo `tagged` non ci sono problemi, mentre per il campo `state` occorre ricordare che lo stato nel *trail* non è compresso, mentre quello nella *hash table* lo è; pertanto, prima di effettuare il confronto occorre comprimere il campo `state` di `ostate`. È da notare che il campo `cvo` non viene controllato: occorrerebbe confrontarlo con `((char *)&(s->state))[0]`, ma l'esito positivo di tale confronto è implicato dall'esito positivo di quello appena descritto tra i campi `state`. Nel caso della *collapse compression*, usare la funzione `compress` vuol dire, potenzialmente, modificare la *hash table* (cfr), ma in questo caso ciò non avviene, in quanto lo stato da trovare è stato l'ultimo immesso (altrimenti la funzione `depth_of` non sarebbe stata chiamata), e quindi i suoi frammenti ed il suo vettore degli identificatori sono sicuramente ancora presenti nella *hash table*.

5.1.5 Il test di terminazione

La nuova verisione di Pan ora proposta, se lasciata così com'è, potrebbe non terminare mai: i conflitti nella *hash table* potrebbero infatti impedire di riconoscere dei cicli sufficientemente lunghi, mandando in *loop* perpetuo il programma. Per evitare ciò, è stato inserito un controllo di terminazione basato sul rapporto di collisione (*collision rate*):

$$coll\_rate = \frac{\#collisioni}{\#inserzioni\ \ nell'hash\ \ table}$$

Dato che le collisioni sono particolari inserzioni, questo rapporto è sempre minore o uguale a 1; tuttavia, quando si avvicina troppo a questo valore vuol dire che per ogni inserzione c'è una collisione, e si può concludere che il programma è in *loop*, in

quanto perde di continuo le informazioni che cerca di memorizzare. Nel `testing` si è usato il valore 0.9.

Per attivare questo controllo di terminazione è necessario compilare con la direttiva `-DBLOCKCOLL`.

Capitolo 6

Testing

6.1 La strategia di *testing*

Per il *testing* della nuova versione di Spin sono stati usati sette sorgenti Promela. Eccone una breve descrizione (col termine "parametrico" si intende che il protocollo in questione è dipendente da una qualche costante interna, cambiando la quale si ottiene una modifica dell'insieme di stati raggiungibili):

Protocollo	**Descrizione**	**Parametrico**
Erathostenes	Algoritmo di Eratostene per il calcolo dei numeri primi	SI
Leader	Algoritmo di Dolev, Klawe & Rodeh per l'elezione del leader in un anello unidirezionale	SI
Leader_ltl	Stesso algoritmo di prima, ma con una formula LTL da verificare: c'è un cammino nel quale ci sia semper uno e un solo leader?	SI
Mobile	Modello per la gestione del segnale in una cella per la telefonia mobile; ha una formula LTL	NO
Peterson	Algoritmo di Peterson per la mutua esclusione tra processi concorrenti	SI
Pftp	Protocollo di controllo di flusso	SI
Sort	Generazione di numeri "random"	SI

Nella procedura di testing, si è dapprima fatta una verifica con la versione di Pan non modificata, usando i vari algoritmi di compressione presenti. I risultati sono raccolti nella tabelle che seguono, nella quale il numero accanto al nome di un protocollo parametrico indica il valore che si è dato alla costante che lo rendeva tale; per quanto riguarda le altre colonne, si ha:

M: quantità totale di memoria RAM necessaria per terminare la verifica (in *Megabytes*)

Mh: quantità di memoria RAM necessaria per la sola *hash table* (in *Megabytes*)

Mt: quantità di memoria RAM necessaria per il solo *trail* (in *Megabytes*)

Ms: quantità di memoria RAM necessaria per le liste di collisione, inutile nella modalità *supertrace* (in *Megabytes*)

S: numero finale di stati memorizzati nella *hash table*

T: numero finale di transizioni attivate nella verifica

***Sv*:** numero di *bytes* che sono stati effettivamente necessari, al massimo, per memorizzare lo stato globale `now`; tale valore resta uguale in tutti i tipi di verifica, perciò viene riportato solo nella prima tabella

P: massimo livello di profondità raggiunto nella verifica

`-w`: logaritmo della dimensione dell'*hash table* (cfr...)

`-m`: massima profondità raggiungibile (cfr...)

Hf: in modalità *supertrace*, indica il rapporto tra la dimensione dell'*hash table* e il numero di stati memorizzati in essa; deve essere maggiore di 100 per garantire una buona copertura dell'insieme degli stati raggiungibili.

6.1.1 Esiti delle verifiche della versione non modificata

Direttive di compilazione: `-DSC`						
Protocollo	**M**	**Ms**	**S**	**T**	***Sv***	**P**
Erathostenes50	25.456	23.963	112551	161370	444	690
Erathostenes51	28.631	27.137	125791	180802	444	695
Erathostenes52	33.546	32.052	146409	211195	444	704
Erathostenes53	34.980	33.486	152279	219894	444	709
Erathostenes54	40.714	39.221	175446	253350	472	772
Erathostenes55	44.503	43.010	191002	276133	472	777
Erathostenes56	53.207	51.714	225609	326393	472	790
Erathostenes57	57.099	55.708	241534	350079	472	795
Erathostenes58	63.038	61.647	265083	384807	472	804
Erathostenes59	64.574	63.183	271055	393443	472	809
Erathostenes60	73.176	71.785	304194	440794	500	876
Erathostenes65	148.136	146.745	581452	843333	528	971
Erathostenes70	285.050	283.659	1.06325e+06	1.54677e+06	556	1078
Leader	4.054	2.662	14578	14809	208	152
Leader6	27.299	25.908	110194	111553	272	182
Leader7	278.496	277.105	927538	938145	344	210
Leader_ltl	6.306	4.915	26690	89923	212	282
Leader_ltl6	48.907	47.515	201602	678547	272	338
Leader_ltl7	508.496	507.104	1.69448e+06	5.70199e+06	344	390
Mobile	4.770	3.379	44455	186456	128	1833
Pftp	7.433	5.939	47356	64970	28	1923
Pftp6	66.520	65.129	464546	668681	36	4813
Pftp8	462.004	460.511	2.9491	4.3077	148	12874
Peterson3	1.903		16720	30322	164	4914
Peterson4	116.903	115.409	3.18584e+06	6.85332e+06	180	913656
Sort50	32.049	29.993	5252	5252	5868	5352
Sort75	144.000	139.938	11627	11627	12392	11777
Sort100	440.000	428.927	20502	20502	21720	20702
Mh: 1.049 M; **Mt**: 0.240M; `-w18`; `-mi0000`						

Direttive di compilazione: `-DSC -DHC`					
Protocollo	**M**	**Ms**	**S**	**T**	**P**
Erathostenes50	3.234	1.843	112551	161370	690
Erathostenes51	3.439	2.048	125791	180802	695
Erathostenes52	3.746	2.355	146409	211195	704
Erathostenes53	3.849	2.458	152279	219894	709
Erathostenes54	4.258	2.867	175446	253350	772
Erathostenes55	4.463	3.072	191002	276133	777
Erathostenes56	5.078	3.687	225609	326393	790
Erathostenes57	5.282	3.891	241534	350079	795
Erathostenes58	5.692	4.301	265083	384807	804
Erathostenes59	5.794	4.403	271055	393443	809
Erathostenes60	6.306	4.915	304194	440794	876
Erathostenes65	10.710	9.319	581452	843333	971
Erathostenes70	18.493	17.101	1.06325e+06	1.54677e+06	1078
Leader	1.698	0.307	14578	14809	152
Leader6	3.234	1.843	110194	111553	182
Leader7	16.240	14.848	927538	938145	210
Leader_ltl	2.210	0.819	48719	186456	1833
Leader_ltl6	7.843	6.451	402483	678547	338
Leader_ltl7	55.563	54.171	3.38392e+06	5.70199e+06	390
Mobile	2.210	0.819	48719	186456	1833
Pftp	2.313	0.819	47356	64970	1923
Pftp6	8.969	7.475	464546	668681	4813
Pftp8	48.804	47.311	2.9491	4.3077	12874
Peterson3	1.698	0.307	16400	29508	4815
Peterson4	51.467	49.973	3.08465e+06	6.57339e+06	846600
Sort50	3.049	0.993	5252	5252	5352
Sort75	2.000	0.000	11627	11627	11777
Sort100	10.000	0.000	20502	20502	20702
Mh: 1.049 M; **Mt**: 0.240M; `-w18`; `-m10000`					

Direttive di compilazione: `-DSC -DCOLLAPSE`					
Protocollo	**M**	**Ms**	**S**	**T**	**P**
Erathostenes50	5.590	4.199	112551	161370	690
Erathostenes51	6.102	4.711	125791	180802	695
Erathostenes52	6.921	5.530	146409	211195	704
Erathostenes53	7.228	5.837	152279	219894	709
Erathostenes54	8.150	6.759	175446	253350	772
Erathostenes55	8.764	7.373	191002	276133	777
Erathostenes56	10.198	8.807	225609	326393	790
Erathostenes57	10.915	9.524	241534	350079	795
Erathostenes58	11.836	10.445	265083	384807	804
Erathostenes59	12.143	10.752	271055	393443	809
Erathostenes60	13.577	12.186	304194	440794	876
Erathostenes65	25.661	24.270	581452	843333	971
Erathostenes70	47.370	45.979	1.06325e+06	1.54677e+06	1078
Leader	2.005	0.614	14578	14809	152
Leader6	5.282	3.891	110194	111553	182
Leader7	32.522	31.130	927538	938145	210
Leader_ltl	2.722	1.331	44455	186456	1833
Leader_ltl6	8.355	6.963	201602	678547	338
Leader_ltl7	61.809	60.418	1.69448e+06	5.70199e+06	390
Mobile	2.722	1.331	44455	186456	1833
Pftp	2.825	1.331	47356	64970	1923
Pftp6	14.192	12.698	464546	668681	4813
Pftp8	80.652	79.158	2.9491	4.3077	12874
Peterson3	1.801	0.410	16720	30322	4914
Peterson4	65.906	64.412	3.18584e+06	6.85332e+06	913656
Sort50	3.049	0.993	5252	5252	5352
Sort75	4.000	0.938	11627	11627	11777
Sort100	10.000	0.000	20502	20502	20702
Mh: 1.049 M; **Mt**: 0.240M; `-w18; -m10000`					

Nelle tabelle viste finora, l'opzione `-w` non riveste molta importanza: avere una *hash table* con molte *entries* vuol dire solamente abbassare il tempo di esecuzione, in quanto la lunghezza media delle liste di collisione è ovviamente più bassa, rendendo più facili le inserzioni e le ricerche. Pertanto, per rendere più facili i raffronti, tutte le verifiche sono state fatte con lo stesso numero di *entries*.

Fatta questa precisazione, resta da dire che la colonna **Ms** rappresenta sostanzialmente la memoria minima che occorre avere per poter completare la verifica sul protocollo corrispondente: si potrebbe sempre infatti ridurre la *hash table* fino a lasciarle una sola *entry*, e lo stesso fare con il *trail*, che comunque già nelle tabelle di sopra occupa una quantità di memoria irrisoria.

Nel paragrafo seguente si vedrà come tale memoria minima venga ulteriormente

limitata dalla versione modificata di Spin.

6.1.2 Esiti delle verifiche della versione modificata

In questa seconda serie di *testing* il parametro `w` assume un'importanza fondamentale: infatti, essendo le liste di collisione lunghe al massimo 1, il numero di *entries* dell'*hash table* limita di fatto l'uso della memoria. I valori di `w` riportati nelle tabelle sono i minimi per i quali la verifica termina con un *collision rate* inferiore al 90%. Anche il parametro `-m` è ora importante: infatti nel *trail* non ci sono più semplici puntatori come prima (tant'è vero che, nelle tabelle del paragrafo precedente, la memoria occupata dal *trail* era sempre la stessa), ma vi viene anche memorizzato lo stato globale analizzato in quel momento. Ciò implica che, per avere una occupazione di memoria pari a quella di prima, occorre che il *trail* sia molto più piccolo; inoltre, per protocolli con grande stato globale come i *Sort*, è necessario rimpicciolirlo ancora di più . Ciò non procura tuttavia inconvenienti alla verifica, in quanto la direttiva di *stack cycling* è sempre abilitata. (cfr)

Direttive di compilazione: `-DRANDOM_SPIN`									
Protocollo	**M**	**Mh**	**Mt**	**Ms**	**S**	**T**	**P**	`-w`	`-m`
Erathostenes50	11.567	0.262	0.245	11.057	116694	171521	690	16	227
Erathostenes51	11.567	0.262	0.245	11.057	116694	171521	690	16	227
Erathostenes52	12.909	0.262	0.245	12.399	153554	228891	704	16	227
Erathostenes53	13.059	0.262	0.245	12.550	159930	238879	709	16	227
Erathostenes54	13.764	0.262	0.245	13.255	187451	281649	772	16	227
Erathostenes55	14.240	0.262	0.245	13.731	221384	337141	777	16	227
Erathostenes56	24.713	0.524	0.245	23.942	238411	355445	790	17	227
Erathostenes57	25.474	0.524	0.245	24.702	254688	380272	795	17	227
Erathostenes58	26.442	0.524	0.245	25.670	280697	421133	804	17	227
Erathostenes59	26.690	0.524	0.245	25.918	286651	429798	809	17	227
Erathostenes60	27.898	0.524	0.245	27.126	327605	493844	876	17	227
Erathostenes65									
Erathostenes70									
Leader	0.993	0.016	0.246	0.723	14583	14809	153	12	227
Leader6	4.215	0.066	0.246	3.896	110200	111553	183	14	227
Leader7	40.369	0.524	0.247	39.589	927548	938145	211	17	227
Leader_ltl	1.697	0.033	0.246	1.410	73689	176587	284	13	227
Leader_ltl6	15.456	0.262	0.246	14.939	214949	723758	338	16	227
Leader_ltl7									
Mobile	2.283	0.131	0.245	1.893	129320	427253	1958	15	227
Pftp	2.269	0.066	0.246	1.944	63865	87121	1923	14	227
Pftp6	19.030	0.524	0.246	18.246	668056	965446	4813	17	227
Pftp8									
Peterson3	0.868	0.262	0.246	0.357	26408	46830	5084	16	227
Peterson4									
Sort50	17.078	0.016	0.511	16.539	5252	5252	5352	12	81
Sort75	47.326	0.016	0.577	46.715	11627	11627	11777	12	41
Sort100									

Direttive di compilazione: `-DRANDOM_SPIN -DHC`									
Protocollo	**M**	**Mh**	**Mt**	**Ms**	**S**	**T**	**P**	`-w`	`-m`
Erathostenes50	3.395	2.097	0.245	1.050	122587	191862	690	19	227
Erathostenes51	3.395	2.097	0.245	1.050	122587	191862	690	19	227
Erathostenes52	3.492	2.097	0.245	1.148	162218	258113	704	19	227
Erathostenes53	3.507	2.097	0.245	1.162	168898	268919	709	19	227
Erathostenes54	3.549	2.097	0.245	1.204	278572	454878	772	19	227
Erathostenes55	6.409	4.194	0.245	1.967	206656	326266	777	20	227
Erathostenes56	6.554	4.194	0.245	2.113	245732	390030	790	20	227
Erathostenes57	6.612	4.194	0.245	2.171	265359	423671	795	20	227
Erathostenes58	6.681	4.194	0.245	2.240	293034	469434	804	20	227
Erathostenes59	6.699	4.194	0.245	2.257	301738	483920	809	20	227
Erathostenes60	6.779	4.194	0.245	2.338	344151	551836	876	20	227
Erathostenes65	13.246	8.389	0.245	4.610	672362	1.08778e+06	971	21	227
Erathostenes70	26.017	16.777	0.245	8.993	1.15652e+06	1.85355e+06	1078	22	227
Leader	0.360	0.066	0.246	0.041	14590	14810	153	14	227
Leader6	1.106	0.524	0.246	0.327	110198	111553	182	17	227
Leader7	7.068	4.194	0.247	2.619	928861	939969	211	20	227
Leader_ltl	1.095	0.524	0.246	0.316	59706	101051	284	17	227
Leader_ltl6	6.943	4.194	0.246	2.494	448753	758259	340	20	227
Leader_ltl7									
Mobile	1.809	1.049	0.245	0.501	79628	299368	2325	18	227
Pftp	1.096	0.524	0.246	0.312	67163	92497	1923	17	227
Pftp6	6.996	4.194	0.246	2.542	763873	1.08258e+06	4607	20	227
Pftp8									
Peterson3									
Peterson4									
Sort50	0.582	0.033	0.511	0.027	5252	5252	5352	13	81
Sort75	0.711	0.066	0.577	0.051	11627	11627	11777	14	41
Sort100	0.785	0.131	0.531	0.095	20502	20502	20702	15	21