A.A. 2006-2007

Seminario per il corso di Metodi Formali per l'Ingegneria del Software

Docente: prof. Toni Mancini

Verifica formale del software: SPIN

Davide Nifosi

Matricola: 797199

SPIN (Simple Promela INterpreter)

Spin è un *verification tool* usato per analizzare la consistenza logica di sistemi concorrenti e, in particolare, dei protocolli di comunicazione in essi utilizzati. Una volta definito formalmente un sistema attraverso il linguaggio di modellazione Promela (PROcess MEta LAnguage), è possibile eseguire simulazioni su di esso oppure analizzarne le proprietà di correttezza. Spin offre inoltre un buon numero di ottimizzazioni nella creazione delle strutture dati per l'analisi, che consentono di ridurre significativamente la memoria utilizzata e il tempo di esecuzione.

Introduzione a Promela

La descrizione di un sistema in Promela ha generalmente questa struttura:

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;
proctype Sender() {
  ...
      /* corpo del processo */
}
proctype Receiver() {
  ...
}
init {
  ...
    /* crea processi */
}
```

La sintassi è molto simile a quella del linguaggio C; un modello Promela incorpora le seguenti sezioni:

1. Dichiarazione dei tipi di dato
2. Dichiarazione dei canali di comunicazione
3. Dichiarazione delle variabili globali
4. Dichiarazione dei tipi di processi
5. Eventuale processo di inizializzazione

Tipi di dato

Promela eredita dal C la possibilità di includere macro (#define) e definizione di tipi (typedef); c'è invece un tipo di dichiarazione particolare per i tipi di messaggi che i processi si possono scambiare. Ad esempio, scrivendo

```
mtype = {MSG, ACK};
```

si vuole indicare che i messaggi scambiabili avranno esclusivamente le etichette MSG e ACK. In questo modo si facilita la tracciabilità degli errori e si astrae dalla specifica di un valore concreto a queste etichette.

Canali

Un canale è l'entità attraverso la quale dei processi possono scambiarsi dei messaggi; il modello concettuale su cui è basata l'idea è molto simile al modello Task-Channel. La definizione di un canale è come segue:

```
chan nomecanale = [buffer] of { tipo1, tipo2 ...};
```

Ogni canale è strutturato a FIFO, con una dimensione massima indicata da 'buffer'; ogni messaggio che transita nel canale deve avere il formato indicato dai tipi tra parentesi graffe, nello stesso ordine. Generalmente il primo tipo è 'mtype', per consentire a chi riceve messaggi sul canale di filtrarli secondo l'etichetta, che funge da header del messaggio.
Se 'buffer' equivale a 0, allora il canale opera in modo sincrono (bloccante).

Alternativamente alla dichiarazione di canali globali, è possibile definirli localmente ad un processo per poi passarli come parametri ad un processo figlio.

Variabili

I tipi di dato associabili a variabili sono:

- bit, bool (0 o 1, false o true)
- byte (0..255)
- short (intero a 16 bit)
- int (intero a 32 bit)
- unsigned (specifica del numero di bits usati per il dato)
- pid (il tipo di dato ritornato da '_pid', che fornisce il pid del processo corrente)
- ogni tipo definito con 'typedef'

Inoltre è possibile definire array per ogni tipo di variabile ed anche per ogni tipo di canale, utilizzando la sintassi C. Nella dichiarazione di un array, la dimensione deve essere una costante; l'indice può invece essere anche un'espressione contenente variabili.

Tipi di processo

Una definizione di processo è specificabile con questa sintassi:

```
proctype nomeprocesso(parametri) {
    /* corpo del processo */
}
```

Per attivare un'istanza di questo processo si fa uso dell'operatore 'run' all'interno di un altro processo, includendo eventuali parametri attuali; tale operatore restituisce il pid del processo generato. Possono essere passati come parametri qualsiasi tipo di dato o di canale, ma non array o tipi di processo.
Ogni istanza di processo conserva il proprio spazio dei dati e il proprio process counter.
Come alternativa a 'run', è possibile fare in modo che una o più istanze di un processo vengano attivate all'avvio del sistema; tramite la dichiarazione:

```
active [2] proctype nomeprocesso(parametri) {
    /* corpo del processo */
}
```

si attivano 2 istanze del processo nomeprocesso, con i valori degli eventuali parametri impostati a default.

Un particolare tipo di processo è 'init': definibile senza alcun parametro, esso è automaticamente attivato all'avvio ed è in genere usato per inizializzare gli altri processi.
Un processo può essere attivato in qualsiasi punto del corpo di un altro processo, comprese le espressioni logiche e i cicli. Tuttavia **il numero di processi eseguibili contemporaneamente deve essere limitato**: Spin impone un limite di 255 processi.

Una volta avviata una simulazione del sistema, i processi attivati sono eseguiti in concorrenza, con pianificazione non deterministica, indipendentemente dalla velocità di esecuzione.

Corpo di un processo

Il corpo di un processo è essenzialmente composto da una sequenza di 'statements': ogni statement può essere 'eseguibile' o 'bloccato'. Intuitivamente, se in uno statement è espressa una qualche condizione che non sia soddisfatta, esso è bloccato. Ogni singolo statement è atomico.

Gli statements derivati dalla sintassi C sono:

- le assegnazioni (sono sempre eseguibili)
- le espressioni:
  - aritmetiche (sono eseguibili solo se il risultato è diverso da 0)
  - logiche (eseguibili solo se 'true', ovvero diverse da 0, come in C)
  - di confronto (come sopra)
  - di stato e di ricezione (descritti più avanti)
- la funzione 'printf' (sempre eseguibile, usata come aiuto alla verifica)

Gli altri statements propri di Promela sono i seguenti.

'skip': sempre eseguibile, causa solo l'incremento del process counter.

'run': sempre eseguibile, tranne quando il numero massimo di processi attivabili è stato raggiunto.

'assert': lo statement 'assert(espressione)' è sempre eseguibile, ma qualora l'espressione sia valutata a 0, Spin termina la simulazione con un errore.

'if': questo statement è strutturato così

```
if
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
fi;
```

Ogni 'choice' è a sua volta uno statement e viene chiamato 'guardia'. L'if statement è eseguibile solo se almeno una delle guardie è eseguibile: se questa esiste, il processo proseguirà sugli statements successivi alla freccia. In presenza di più guardie eseguibili, Spin sceglierà a caso tra di esse quella su cui proseguire.
Una guardia speciale è 'else' e viene eseguita solo se non ci sono altre guardie eseguibili nel blocco if.

'do': analogo a if, con la differenza che la scelta si ripete indefinitamente, a meno che non venga eseguito all'interno del blocco lo statement 'break' o che il flusso di esecuzione non venga alterato da 'goto'.

I blocchi 'if' e 'do' possono essere annidati.

'goto': fa proseguire l'esecuzione del processo dal punto indicato dall'etichetta che segue il comando. Esistono prefissi di etichette riservati per Spin che assistono l'analisi dei processi:

- 'end:' sta ad indicare che lo statement seguente è accettabile dal simulatore come stato finale; utile per far sì che Spin non segnali la non terminazione di un processo che non raggiunge normalmente la fine del suo corpo
- 'progress:' se Spin è impostato per segnalare il 'no progress' dei processi, esso lo notificherà nel caso lo statement successivo a 'progress:' non venga eseguito un numero indefinito di volte.
- 'accept:' se Spin è impostato per segnalare gli 'acceptance-cycles', esso lo notificherà nel caso lo statement successivo a 'accept:' venga eseguito un numero indefinito di volte.

Statements per scambiare messaggi:

Invio: `nomecanale ! <expr1>, <expr2>, … <exprn>;`

Ricezione dati: `nomecanale ? <var1>, <var2>, … <varn>;`

Validazione messaggi: `nomecanale ? <const1>, <const2>, … <constn>;`

Un invio è eseguibile se il canale su cui si spedisce il messaggio non è pieno, mentre una ricezione è eseguibile se il canale non è vuoto; nella validazione l'eseguibilità è condizionata anche dal confronto dei dati ricevuti con i corrispondenti campi costanti elencati. I tipi di dato scambiati devono rispettare il formato specificato dal canale usato.
Validazione e ricezione possono essere combinati: viene eseguito il confronto dei valori ricevuti con i rispondenti campi delle costanti e, se tutti hanno successo, i valori ricevuti in corrispondenza dei campi delle variabili sono assegnati ad esse.

Altri statements utilizzabili

'atomic': la sequenza di statements, compresa tra parentesi graffe, che segue questa parola è eseguibile in modo atomico; 'atomic' è eseguibile se lo statement in testa alla sequenza è eseguibile. Tuttavia Spin, nella sua simulazione, genera comunque stati di esecuzione separati per ogni singolo statement in quanto, se uno di essi (escluso il primo) è bloccato, il processo può essere interrotto in favore di un altro processo. Per garantire l'esecuzione strettamente atomica si può usare 'd_step': con esso la sequenza di statemnts viene vista esattamente come un singolo statement e gli stati intermedi non sono generati (importante per le ottimizzazioni). Nel caso uno statement successivo al primo sia bloccato, si genera un errore.

'timeout': eseguibile solo se nessun altro processo nel sistema è eseguibile; evita deadlocks.

'unless': strutturato come segue

```
...
{
  ...
} unless { guardia; ... }
```

Finché la guardia dopo 'unless' è bloccata, gli statements nel blocco precedente possono essere eseguiti; se e quando la guardia diventa eseguibile, vengono eseguiti gli statements successivi ad essa.
Può simulare la ricezione di segnali o i blocchi 'try/catch' delle eccezioni.

Costrutti addizionali

Esternamente alle dichiarazioni di processi (o anche in files separati) è possibile definire un 'never claim', ovvero una sequenza di espressioni che rappresenta un certo comportamento (finito o infinito) del sistema e che si desidera non si verifichi mai.

Durante l'analisi di un sistema, presa in esame una possibile esecuzione, le espressioni nel never claim sono valutate di pari passo con essa: se si incontra un'espressione valutata come falsa, detta esecuzione è permessa, quindi si passa ad un'altra; altrimenti si prosegue con quella successiva. Qualora la valutazione del never claim giungesse alla fine della sequenza prima del termine dell'esecuzione corrente, si riscontrerebbe una violazione.

All'interno del never claim sono ammessi tutti gli statements che non producano effetti collaterali (assegnazioni, run, ricezioni di dati...); sono quindi consentite espressioni, statements speciali (skip, break, goto) e validazioni di dati ricevuti. Di particolare interesse sono i seguenti tipi di espressioni, utili a controllare facilmente il flusso di esecuzione di un processo:

- 'nomeprocesso[p]@label': l'espressione risulta vera se il processo di tipo 'nomeprocesso' e pid 'p' si trova nell'esecuzione allo statement contrassegnato dall'etichetta 'label'; simula il controllo dello stato di un processo senza ricorrere a variabili di stato.
- nomecanale?[mess, ...]: l'espressione risulta vera se lo statement di ricezione corrispondente è eseguibile; funziona solo con canali asincroni.

Spin incorpora un traduttore automatico da formula LTL a never claim; la sintassi ammessa include solo una parte degli operatori temporali:

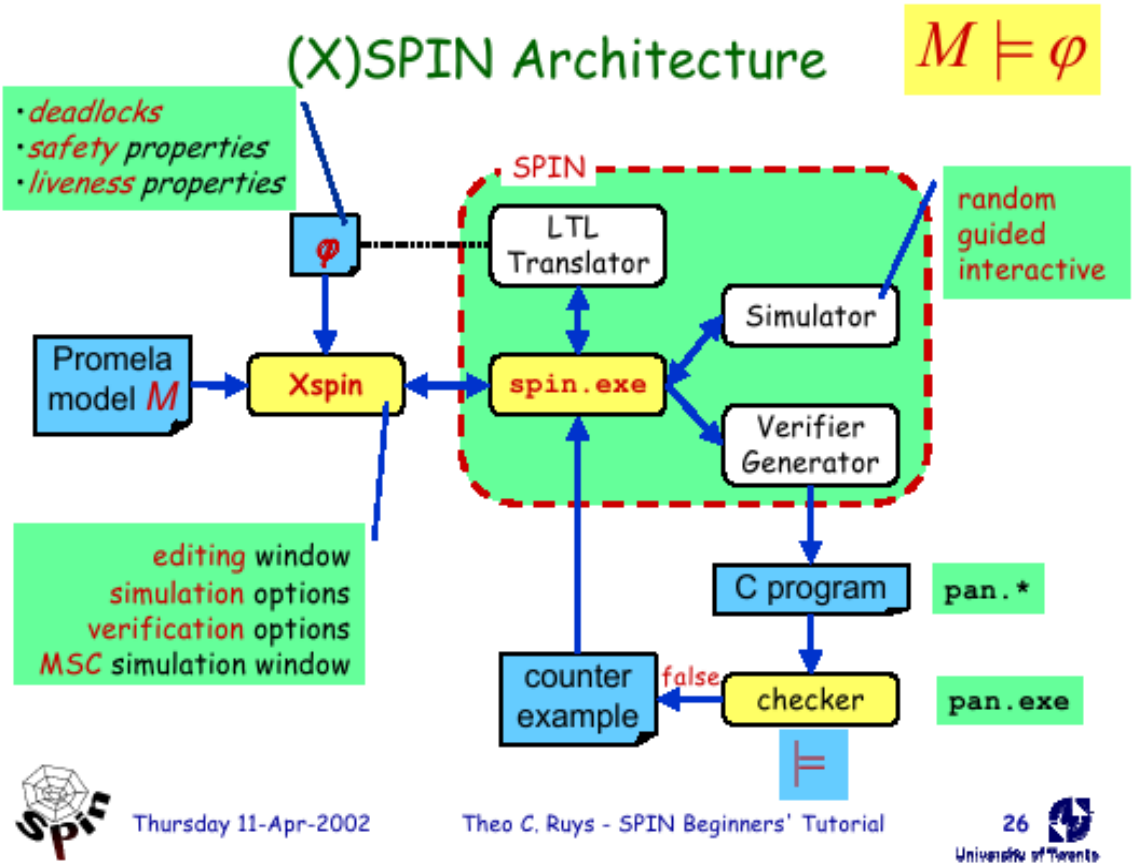
- '[]', che corrisponde all'operatore Globally
- '<>', che corrisponde all'operatore Future
- U (Strong until)
- X (Next) *

* In Spin l'operatore X è disabilitato di default, poiché il suo utilizzo potrebbe portare alla creazione di stutter-closed never claims (ossia con proprietà che dipendono dal numero di passi di esecuzione in cui rimangono vere o false): in

queste situazioni verrebbero a mancare le ipotesi sotto le quali l'algoritmo di riduzione dello spazio degli stati usato di norma da Spin può essere applicato.

Esecuzione di Spin

L'architettura di Spin è riassumibile in questo schema:



Essenzialmente, Spin può essere utilizzato in tre modalità: simulazione, analisi e traduzione LTL. L'analisi deve necessariamente appoggiarsi ad un compilatore C.
Xspin è un'interfaccia grafica per il programma, che consiste di un editor testuale per i modelli, finestre di opzioni per verifica e simulazione (con uno schema grafico per rappresentare il flusso di esecuzione), assistenza alla creazione di never claims ed altre funzionalità per logging e suggerimenti.

Simulazione

Invocando Spin senza opzioni su un file contenente un modello Promela, verrà eseguita una simulazione di esecuzione, con scelte casuali ad ogni passo non deterministico; con l'opzione '-nN' si specifica N come 'seed'. Altre opzioni sono usate per la verbosità e la formattazione del risultato.
Oltre alla simulazione casuale:

- '-i': consente di effettuare una simulazione interattiva, in cui le scelte nei punti di non determinismo sono a carico dell'utente.
- '-t': con il comando 'spin -t nomefile' si richiede a Spin di eseguire una simulazione guidata, seguendo le scelte indicate nel file 'nomefile.trail' che rappresenta una traccia di esecuzione valida; solitamente questa traccia è prodotta in modalità di analisi in presenza di proprietà non verificate e si può usare, congiuntamente all'opzione '-p' (stampa tutti gli statements eseguiti), per visualizzare la possibile esecuzione che viola tali proprietà.

Analisi

Con l'opzione '-a' Spin produce una serie di files 'pan.*'; compilando il file pan.c, ad esempio con il comando

```
cc –o pan pan.c
```

si crea il Process ANalyzer per il sistema definito. Eseguendo 'pan' viene effettuata la verifica delle proprietà richieste nel modello (asserzioni, never claims); in caso di violazioni, verrà scritta la traccia di esecuzione che ha generato errore nel file '.trail'. Di default, Spin usa una DFS per l'esplorazione dello spazio degli stati, il partial order reduction algorithm e lo state hashing.
Nella compilazione è possibile specificare delle opzioni:

- '-DNOREDUCE': disabilità il partial order reduction algorithm (creazione di tutti gli stati di esecuzione)
- '-DNP': abilita la ricerca di non-progress cycles (cicli non eseguibili indefinitamente)
- '-DSAFETY': se la verifica è solo su proprietà di safety, questa opzione ottimizza il tempo di esecuzione
- '-DMEMLIM=X': specifica il numero massimo di MB di RAM da utilizzare per l'analisi
- '-DCOLLAPSE': attiva la compressione della hash table degli stati (con collision detection)
- '-DBITSTATE': attiva la riduzione ad un solo bit di memoria per gli stati (senza collision detection, perdita di completezza)
- '-DBFS': usa una Breadth-First Search al posto della DFS

Nell'esecuzione di 'pan' è possibile richiedere, oltre ad opzioni di verbosità, la verifica di particolari proprietà o modificare i limiti della ricerca:

- '-a': tiene traccia degli 'acceptance cycles' (gli statements contrassegnati da un'etichetta 'accept:' non possono essere raggiunti un numero indefinito di volte; attivabile se non è stato passato '-DNP' al compilatore)
- '-l': tiene traccia di 'no-progress cycles' (gli statements contrassegnati da un'etichetta 'progress:' devono poter essere raggiunti un numero indefinito di volte nei cicli infiniti; ogni ciclo infinito senza progress statement è segnalato; attivabile solo se abilitata con '-DNP')
- '-mN': fissa il limite massimo di profondità di ricerca a N passi (default: 10000)
- '-i', '-I': ricerca del percorso più breve per raggiungere una violazione (attivabili solo de abilitate con '-DREACH')
- '-cN': termina dopo N errori al massimo (default: 1); scrivendo '-c0' verranno ricercati tutti gli errori. Con l'opzione '-e' è possibile memorizzare una traccia per ogni errore (di default si genera solo la prima)

Traduzione LTL

Con le opzioni '-f formula' e '-F file', Spin stampa su stdout il never claim corrispondente alla formula LTL ritrovata rispettivamente sulla riga di comando o nel file indicato (legge solo la prima riga). Il risultato prodotto può essere scritto in un file ed associato ad un modello da analizzare tramite il comando

```
spin −a −N file_never_claim file_promela
```

come alternativa all'inserimento nello stesso file. L'opzione '-F' va utilizzata quando per aggirare i conflitti sintattici con la shell utilizzata.

Uso di Spin nelle verifiche formali viste nel corso

Promela è studiato appositamente per rappresentare sistemi concorrenti ed è particolarmente indicato per analizzare le interazioni tra processi a livello puramente formale.
Nel modellare stati e transizioni di istanze di classi UML, il linguaggio fornisce agevolazioni per rappresentare semplicemente le loro interazioni, ovvero far sì che l'azione compiuta da una particolare istanza possa diventare un evento per un'altra istanza. Questo perché c'è un buon supporto allo scambio di messaggi: per questo motivo è stata scelta come 'traduzione' da UML-STD a Promela la metodologia descritta nel seguito.

Da States and Transitions Diagram a Promela

Per rappresentare uno stato si è scelto di sfruttare le etichette: una dichiarazione di processo rappresenta una classe (con eventuali parametri di inizializzazione) ed ogni statement al suo interno rappresenta uno stato, identificabile dall'etichetta che gli sarà preposta. In questo modo una transizione potrà essere rappresentata semplicemente da un 'goto etichetta'.

Per una corretta rappresentazione, ad ogni stato (non transitorio) dovrà essere associato un 'do statement', che simulerà l'attesa per un evento; evento che può essere rappresentato dalla ricezione di un messaggio di un particolare tipo. Così uno stato dovrebbe risultare come segue:

```
...
mtype = { evento1, evento2, ... };
chan ev = [N] of { mtype, ... };
...
proctype classe(parametri) {
...
stato1: do
        :: ev? evento1 -> azioni... ; goto stato2;
        :: ev? evento2 -> azioni... ; goto stato3;
           ...
        :: ev? eventon -> azioni... ; goto stato3
        od;
...
}
```

Il canale su cui arrivano gli eventi può essere sia dichiarato come globale, sia passato come parametro dal processo che genererà gli eventi (ad esempio init); a questo punto la guardia eseguibile nel blocco rappresenterà la transizione da effettuare (o la permanenza nello stato in caso di 'skip') e gli statements successivi le azioni da compiere.

Si noti che:

- è possibile aggiungere condizioni ad ogni guardia ponendole in AND con lo statement di ricezione interessato;
- è possibile modellare transizioni non deterministiche, poiché in presenza di più guardie eseguibili ne verrà scelta una a caso;
- è necessario specificare il comportamento del processo per ogni tipo di evento che è possibile ricevere, altrimenti eventi non applicabili allo stato corrente non verranno scartati;
- le azioni eseguibili per ogni transizione possono comprendere l'invio di messaggi ad altri processi.

Per una verifica generale di una classe, è possibile definire all'interno di un altro processo (init nei casi più semplici) un generatore casuale di eventi

```
...
init {
...
ripeti: do
        :: ev! evento1;
        :: ev! evento2;
           ...
        :: ev! eventon
...
}
```

Alternativamente sarebbe possibile rappresentare lo stato con una variabile.

Per la verifica di proprietà del sistema, le si può descrivere in LTL seguendo la sintassi accettata da Promela (operatori [], <>...), per poi tradurle in never claim tramite il traduttore integrato in Spin. Si ricorda che **per la verifica di proprietà positive è necessario negare la formula**, poiché un never claim rappresenta un comportamento indesiderato.
Il never claim ottenuto si può trascrivere nel modello, oppure in un altro file associabile al modello con l'opzione '-N' di Spin. Quindi si crea il Process ANalyzer e lo si esegue con l'opzione '-a'(un never claim prodotto automaticamente fa uso di etichette 'accept:', perciò per avere una verifica completa devono essere controllati gli acceptance cycles).

Spin nella fase di verifica degli algoritmi

La sintassi di Promela è molto vicina a quella del C: è inoltre possibile inserire porzioni di vero e proprio codice C all'interno di un modello (costrutto 'c_code'), ma qui non verrà discusso.
Quindi per la verifica di funzioni scritte in C o linguaggi simili sarebbe sufficiente adattare il codice, riportarlo in una dichiarazione di processo ed aggiungere gli statements di verifica delle proprietà all'interno del modello.

- Per la verifica di precondizioni e postcondizioni di una funzione è sufficiente porre delle asserzioni rispettivamente all'inizio e al punto di terminazione della funzione (se ne esiste più d'uno, essi sono sempre facilmente ricongiungibili tramite goto statements). Questo sistema è applicabile anche per proprietà da verificare in punti specifici del processo.
- Per verificare proprietà che devono risultare vere durante tutta l'esecuzione (invarianti del sistema) si può
  - definire un processo 'monitor' contenente un'asserzione ed attivarlo all'avvio del sistema: dovendo essere possibile eseguire questo processo in qualsiasi momento durante l'esecuzione, durante l'analisi tutte le esecuzioni esplorate dovranno verificare questa asserzione. Il problema con questo sistema è che in pratica si raddoppia il numero degli stati del sistema; ciò è aggirabile condizionando l'asserzione con la verifica della proprietà stessa

```
active proctype monitor() {
  atomic { !proprietà -> assert(false) }
}
```

- oppure si può definire un never claim

```
never {
  do
  :: assert(proprietà)
  od
}
```

Tuttavia, rispetto alla sintassi C, ci sono alcune differenze che possono rendere difficile una traduzione semanticamente equivalente. Al di là del formato dei cicli differente, Promela ha diverse limitazioni:

1. I puntatori non sono supportati (mancanza aggirabile con l'inserimento di blocchi 'c_code' e affini)
2. Gli array non possono essere passati ai processi come parametri; l'uso di typedef wrappers per array non è consentito
3. Nella dichiarazione di un array la dimensione deve essere una costante (limitazione condivisa da vecchie versioni di C)
4. Un processo non ha un valore di ritorno: è però possibile simulare un return con un canale
5. I tipi di dato utilizzabili sono pochi (float...)

Comunque la maggior parte delle espressioni è facilmente riconducibile a statements corretti; qui seguono alcuni casi di interesse:

Costrutto 'if'

C:

```
...
if (condizione) {
  ...
} else {
  ...
}
...
```

Promela:

```
...
if
:: condizione ->
   ...
:: else ->
   ...
fi;
```

Nota: anche se 'else' non è presente nel codice C, in Promela va comunque aggiunto (seguito da uno 'skip') per evitare che l'if statement si possa bloccare.

Ciclo 'while'

C:

```
...
while (condizione) {
  ...
}
...
```

Promela:

```
...
do
:: condizione -> ...
:: else -> break
od;
```

Funzioni con side-effect sui parametri

Un modo sarebbe quello di usare variabili globali al posto dei parametri.

Controllo sulle variabili

Perché un never claim possa far riferimento a variabili locali, è necessario utilizzare la dicitura 'nomeprocesso[pid]:nomevar' per indicare a quale istanza di 'nomeprocesso' appartiene la variabile 'nomevar'; è bene ricordare però che così facendo si perdono le ipotesi di funzionamento del partial order reduction algorithm. Per quanto riguarda gli indici degli array e i puntatori, l'unico modo per controllarle è incorporare blocchi di c_code nel processo (eventualmente condizionati).

Espressione delle pre/postcondizioni

In Promela è possibile verificare

- espressioni di logica proposizionale
- espressioni di confronto
- condizioni sui canali
- condizioni sui processi attivi (stato di esecuzione, pid) o sulla loro attivabilità

Per effettuare un'analisi completa del sistema modellato si dovrà creare un generatore casuale dell'input della funzione da verificare: tipicamente ciò viene eseguito nel processo 'init'. Così facendo, in fase di analisi saranno generati stati di esecuzione in proporzione a quanto l'input della funzione può variare.
La crescita del numero degli stati generati è esponenziale, quindi l'analisi può

facilmente raggiungere requisiti di tempo e memoria insostenibili.

Correttezza parziale e totale

Spin, essendo un model checker esplicito, incorpora già funzionalità adeguate alla verifica di programmi, come la gestione del Program Counter e degli stati di terminazione. Tuttavia, esprimere condizioni sulla terminazione di un processo è leggermente macchinoso: tramite la variabile globale predefinita '_nr_pr', è possibile sapere in ogni momento il numero di processi correntemente attivi e la cosa può essere sfruttata per controllare la terminazione di processi. In certi casi (ad esempio con un solo processo) è però più semplice utilizzare una variabile globale di terminazione che viene posta a 'true' alla fine del codice del processo.

Le precondizioni di una funzione si possono esprimere in logica proposizionale, eventualmente sfruttando dei 'do statements' per simulare formule al primo ordine, e si possono associare ad una variabile di tipo bool all'inizio del codice del processo.
Per quanto riguarda le postcondizioni, invece, bisogna evidenziare alcuni problemi:

1. per poter fare riferimento ai valori originari dei parametri della funzione, qualora essa producesse side-effects, è necessario effettuarne comunque la copia all'inizio del processo: ciò perché in questi casi si usano variabili globali al posto dei parametri, che sono passati per valore
2. poiché può essere d'interesse verificare le postcondizioni anche in caso di non terminazione, esse non potranno essere precalcolate per associarle ad una variabile bool, bensì dovranno essere scritte come una singola proposizione; per maggiore leggibilità si può far uso di una macro

A questo punto è possibile sfruttare il traduttore LTL di Spin per esprimere la correttezza parziale e totale sotto forma di never claim (ovviamente usando la versione negata delle formule LTL).
Purtroppo si deve constatare che, in presenza di postcondizioni espresse in logica del primo ordine, esse andranno riscritte in logica proposizionale: difatti l'unico modo per ovviare a questo sarebbe utilizzare dei cicli 'do', ma essi richiederebbero un contatore da incrementare ad ogni passo, cosa non possibile in quanto in un never claim non si può fare side-effect.

Strutture dati, algoritmi e prestazioni

Spin si basa principalmente sulla ricerca in profondità per esplorare tutte le possibili esecuzioni di un sistema; è però possibile usare anche una ricerca in ampiezza (utile per trovare i percorsi all'errore più brevi).
Come strutture base per l'analisi, Spin mantiene

- un state vector per ogni processo (variabili locali, Program Counter)
- uno stack per la Depth-First Search
- una hash table per rintracciare gli state vectors (con collision detection) e per tenere memoria di tutti gli stati già visitati nella DFS

Ricerca nello spazio di esecuzione

Il DFS stack non mantiene gli state vectors, quindi per effettuare backtracking vengono eseguite le transizioni inverse; inoltre, in caso di errore durante l'analisi, Spin deve eseguire una simulazione guidata dalla traccia generata per ricostruire il percorso e gli stati generati.

In presenza di più percorsi d'errore possibili è preferibile cercare quelli più brevi: Spin permette di limitare la profondità di ricerca sia manualmente (opzione '-m'), sia automaticamente: con l'opzione '-i', una volta trovato un errore, la profondità massima viene ridotta e la ricerca rieseguita (può essere molto lento); con l'opzione '-I' viene effettuata una sorta di ricerca binaria, con risultati più rapidi ma approssimati.

A causa delle restrizioni su numero massimo di processi e dimensione delle variabili, grazie alla memorizzazione degli stati già visitati anche sistemi non terminanti possono essere interamente esplorati. In questi casi è comunque consigliabile far uso di etichette 'end:' per segnalare stati che non rappresentino un potenziale stallo: se infatti si giunge ad una terminazione (nessun processo può eseguire) e qualche processo non è in uno stato finale, Spin segnala un 'invalid end-state' , che può essere interpretato come deadlock.

Ottimizzazioni

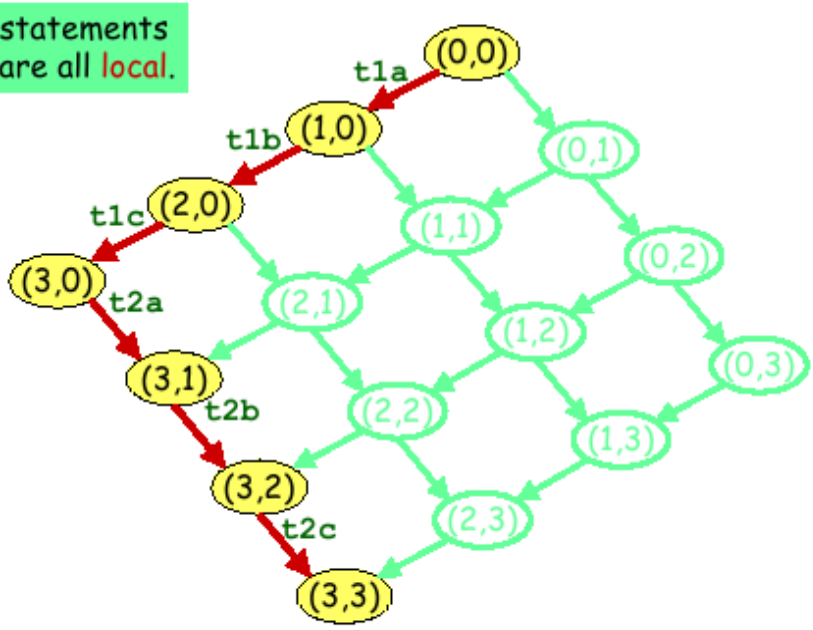
L'algoritmo primario di ottimizzazione in Spin è il 'partial order reduction algorithm': è utile a ridurre la memoria utilizzata senza overhead significativi sul tempo di esecuzione.

In pratica, prima della selezione del processo da eseguire si effettua un ordinamento dei processi secondo una proprietà di 'safety': sono considerati 'conditionally safe' processi che, finché è verificata una data condizione, effettuano solo accessi a variabili locali, statements atomici non osservabili da altri processi, invio e ricezione su canali ad accesso esclusivo.
Lo scopo è privilegiare i processi conditionally safe, così da poter ignorare le transizioni degli altri (sarebbe indifferente seguirle prima o dopo) e ridurre il numero di percorsi esplorati.

Reduction Algorithms (3)

- Partial Order Reduction (cont.)

Suppose the statements of P1 and P2 are all local.



La caratteristica dell'implementazione in Spin è che la ricerca dei processi 'safe' è effettuata prima di eseguire la verifica: l'ordinamento parziale indotto sui processi è riportato in una hash table specifica per questa proprietà, quindi la scelta durante l'esecuzione è molto veloce.

L'effetto primario di questo algoritmo è ridurre drasticamente il numero degli stati in sistemi concorrenti. È bene però notare che la sua correttezza è garantita solo se le proprietà che si vogliono verificare non contemplino un ordinamento temporale stretto dei processi: ad esempio, se si vuole imporre in LTL che un processo venga eseguito immediatamente dopo un altro (operatore X), la presenza di safe statements potrebbe influenzare la valutazione.

Altre ottimizzazioni per ridurre la memoria usata consistono nella compressione degli state vectors (o in casi estremi nella creazione di un automa minimizzato degli stati) e nella riduzione della hash table: in particolare, per quest'ultima sono utilizzati due metodi:

- hash compaction: si basa sulla simulazione di una hash table molto grande (quindi con pochissime collisioni) memorizzandone una molto più piccola; una funzione hash a 64 bit è calcolata su ogni stato, ma il risultato è memorizzato in una hash table di dimensioni 64 x (#stati) con collision detection. Risultati sperimentali mostrano che questo sistema è molto efficace, affidabile e poco dispendioso.
- bitstate hashing: usato solo in caso di requisiti di memoria estremi, questo sistema fornisce solo un risultato approssimato; consiste nel memorizzare un solo bit per ogni stato raggiungibile (che indica se è stato visitato o meno), senza gestire collisioni.

Infine, Spin include lo 'slicing algorithm', che suggerisce eventuali riduzioni da applicare al modello Promela sulla base delle proprietà che si vogliono verificare.

Spin vs. NuSMV

La differenza fondamentale tra i due model checkers è che Spin è un verificatore esplicito, mentre NuSMV è simbolico: ciò comporta vantaggi e svantaggi dal punto di vista della flessibilità e degli usi tipici.

Un punto a favore di Spin è sicuramente la sintassi Promela, che rende semplice la traduzione del codice di un sistema da verificare, o che comunque è più vicina alla logica algoritmica di quanto non lo sia quella di NuSMV, più orientata agli stati ed alle transizioni. Inoltre, sempre nella verifica di programmi, vi sono diversi aspetti gestiti implicitamente da Spin (PC, valutazione array, solo transizioni esplicitate...) che in NvSMV devono essere regolati manualmente.

D'altra parte, una rappresentazione simbolica risulta molto flessibile, sia per i tipi di dato utilizzabili, sia per la modellazione degli States and Transitions Diagrams, sia per la generazione implicita di eventi casuali (che in Spin va specificata manualmente). NuSMV inoltre accetta direttamente la sintassi LTL per esprimere proprietà.
Si noti anche come sia possibile utilizzare NuSMV come solutore di un problema, date le specifiche; con Spin non è in generale cosa attuabile.

Per quanto riguarda la struttura, entrambi consentono un approccio modulare: i processi Promela equivalgono sostanzialmente ai moduli NuSMV. Tuttavia la comunicazione tra moduli avviene in modo diverso: tramite canali configurabili (o variabili globali) nel primo, attraverso riferimenti condivisi nel secondo.

Infine si ha che SPIN è stato progettato in modo particolare per la verifica di protocolli di comunicazioni, quindi sono previste opzioni adatte al rilevamento di situazioni di starvation, deadlock e livelock; questo tramite il controllo di stati finali non validi e non-progress cycles.

Riferimenti

- *Theo C. Ruys “SPIN Beginners' Tutorial”, 2002*
- *Matt Dwyer, John Hatcliff “Introduction to SPIN”, 2001*
- *Promela and SPIN man pages and guidelines for verification (http://spinroot.com)*
- *G.J. Holzmann “An analysis of bitstate hashing”, 1998*
- *G.J. Holzmann, D. Peled “An Improvement in Formal Verification”, 1994*
- *Pierre Wolper, Dennis Leroy “Reliable hashing without collision detection”, 1993*