Metodi per la Verifica del Software
Progetto Finale

Marco Grandi

L'obiettivo di questa relazione è di presentare il modello sviluppato per ogni scenario richiesto del nostro caso di studio: "un veicolo equipaggiato con un sistema di diagnostica digitale di supporto alle situazioni di emergenza".

Per ognuno dei tre scenari proposti verranno formalizzate le proprietà rilevanti richieste ed indicato quali il modello verifica e quali no. Infine, si darà una stima dello spazio degli stati.

Una scelta comune a tutti e tre i modelli è stata quella di utilizzare delle macro, riportate nel Listato 1, per quelle costanti utilizzate nelle comunicazioni tra i processi che modellano le relative entità del caso di studi. Le prime due macro sono quelle utilizzate da un sistema di emergenza per richiedere o rilasciare una prenotazione, mentre le ultime due sono quelle utilizzate da un servizio per rispondere.

```
1 /* tipo di richiesta */
2 #define REQUEST 0
3 #define RELEASE 1
4
5 #define BUSY 0
6 #define AVAILABLE 1
```

Listato 1: macro per costanti

I modelli utilizzano solo 2 valori per esprimere il tipo di richiesta/risposta e quindi è sufficiente una variabile di tipo `bit` per contenere tale informazione.

Scenario A

Lo scenario A è quello più semplice, dove si modella la rottura di un singolo veicolo in presenza di un servizio di carta di credito, di uno o più servizi rispettivamente per l'officina e per il carro attrezzi e di un servizio per il noleggio. Il comportamento del sistema non è richiesto essere ciclico, ma comunque i singoli servizi sono stati modellati da processi ciclici che non terminano. Vediamo ora nel dettaglio tutte le variabili globali ed i `proctype` definiti.

Dato che lo scenario prevede uno o più servizi officina e carro attrezzi, si è deciso di utilizzare una macro per distinguere il modello con singola istanza da quello con più istanze delle stesso processo (due nello specifico). Il test visibile nel Listato 2 controlla se la macro `MULTI` è definita[1] e differenzia il modello creato in base a questa. La macro `NPROC` indica il numero di processi attivi per i servizi della tipologia officina e carro attrezzi.

```
40 /* numero di processi attivi per le tipologie
41    di servizio officina e carro attrezzi */
42 #ifdef MULTI
43 #define NPROC 2
```

[1] per definire la macro basta aggiungere l'opzione `-DMULTI` al comando `spin`

```
44 #else
45 #define NPROC 1
46 #endif
```

Listato 2: macro per numero di processi

Il sistema di emergenza ed i servizi sono rappresentati da `proctype`. Per far comunicare i processi prodotti dall'esecuzione[2] di tali `proctype` si è scelto di utilizzare i canali messi a disposizione da Promela. Si sarebbero potuti utilizzare degli array glocabli in alternativa, ma la scelta dei canali meglio cattura la natura di sistema distribuito del nostro problema. I canali dei servizi sono definiti globalmente, canali che saranno utilizzati dal sistema di emergenza per comunicare con i servizi. Per ogni processo, che modella un relativo servizio, è previsto un canale di tipo rendezvous. Tale tipologia per i canali è stata preferita perché richiede la sincronizzazione dei processi comunicanti e provoca l'avanzamento atomico di entrambi. La dichiarazione e l'inizializzazione dei canali è visibile nel Listato 3.

```
49 /* canale carta di credito */
50 chan creditcard_ch = [0] of {bit};
51
52 /* canali officine */
53 chan garage_chs [NPROC] = [0] of {bit, byte};
54
55 /* canali carro attrezzi */
56 chan towtruck_chs [NPROC] = [0] of {bit, byte};
57
58 /* canale noleggio auto */
59 chan rentalcar_ch = [0] of {bit, byte};
```

Listato 3: canali dei servizi

Nel dettaglio, per i servizi di officina e carro attrezzi sono previsti due array di canali contenti il numero di istanze richiesto. Quindi un array di uno o due canali.

Per quanto riguarda il tipo dei canali, questo è stato scelto in base ai vincoli sui servizi. Il servizio di carta di credito è sempre disponibile ed ha sempre successo, quindi anche se potremmo astrarre completamente il modello evitando di rappresentare questo servizio, siamo interessati a tracciare la comunicazione che questo avrà con un sistema di emergenza e dunque il canale della carta di credito ha un solo campo di tipo `bit` per trasmettere l'informazione richiesta/risposta. I restanti canali, utilizzati delle altre tre tipologie di servizi, hanno due campi: uno di tipo `bit` e l'altro di tipo `byte`. Il primo campo di ogni messaggio è utilizzato per indicare l'informazione richiesta/risposta (ovvero dal sistema per richiedere/rilasciare e da un servizio per rispondere) mentre il secondo è utilizzato per inviare il pid del processo che modella il servizio.

La scelta di utilizzare un secondo campo è motivata dai vincoli imposti dallo scenario. È richiesto infatti che un servizio disponibile debba rispondere con successo ad una richiesta di prenotazione e venga considerato prenotato finché non esplicitamente rilasciato. Si è quindi immaginato che ogni servizio possa restituire, insieme all'informazione di successo, un ticket (unico del servizio) che il sistema di emergenza dovrà riconsegnare in fase di rilascio della prenotazione. L'astrazione naturale di questo ticket è rappresentato dalla variabile `_pid`, locale ad ogni `proctype` di Promela che indica il numero di istanziazione del processo. Ulteriori considerazioni di questa scelta sono riportate dove viene presentata la modellazione dei servizi.

A livello globale sono dichiarate anche alcune variabili di tipo `byte`, modificate dall'`init`, che servono per memorizzare i pid dei vari processi lanciati. Come visibile dal Listato 4 nelle dichiarazioni è stato utilizzato il modificatore `local` per indicare che le variabili, anche se definite globalmente, saranno accedute da un solo processo (`init` nello specifico). Questa scelta

[2] grazie al comando `run`

permette di aumentare l'effetto della partial order reduction durante la verifica, ma permette comunque di utilizzare tali variabili all'interno di formule LTL e never claim. Anche in questo caso si testa se la macro `MULTI` è definita e se lo è si dichiarano altre variabili per i processi aggiuntivi.

```
62 /* variabili globali accedute localmente all'interno dell'init:
63    indicano il pid dei servizi */
64 local byte g_pid1 = 0;
65 local byte tt_pid1 = 0;
66 local byte rc_pid = 0;
67
68 /* se MULTI è definito considero anche il secondo servizio */
69 #ifdef MULTI
70 local byte g_pid2 = 0;
71 local byte tt_pid2 = 0;
72 #endif
```

Listato 4: variabili per i pid dei processi

Prima di entrare nel dettaglio dei servizi, è necessario commentare il Listato 5, dove sono dichiarate delle variabili che riguardano i servizi prenotabili e che saranno utilizzate dai sistemi di emergenza. Lo scopo di queste variabili è quello di indicare quali sono i servizi già prenotati. Sono state introdotte già nello scenario A anche se servono a risolvere un problema che emerge nello scenario B, a cui si rimanda per la motivazione dettagliata. Queste variabili sono utilizzate da un sistema di emergenza per acquisire in mutua esclusione il canale con cui comunicherà con un servizio, mutua esclusione che si manterrà se la prenotazione avrà successo. Altrimenti la risorsa verrà rilasciata ed altri sistemi di emergenza potranno comunicare con quel servizio.

Inizialmente era stato preferito un array per le officine ed uno per i carri attrezzi entrambi di tipo `bit`, ma SPIN memorizza tali array come array di `byte`. Ciò causa lo spreco di memoria. Si è quindi cambiata l'implementazione per utilizzare una variabile di tipo `byte` come array di otto bit. Questa scelta limita il numero massimo di servizi di una tipologia a 8, ma per le altre scelte fatte non è un problema. Per i servizi che invece prevedono una singola istanza viene utilizzata una variabile di tipo `bool`.

```
80 byte garage_booked = 0;
81
82 byte towtruck_booked = 0;
83
84 bool rentalcar_booked = false;
```

Listato 5: variabili per lo stato dei processi

Nel Listato 6 sono riportate le macro utilizzate per testare e modificare tali variabili di tipo `byte` utilizzate come array di bit. Ovviamente si opera utilizzando operazioni bitwise.

```
33 /* macro per testare se un servizio è non prenotato */
34 #define notBooked(v, n) (!(v >> (n) & 1))
35 /* macro per prenotare e rilasciare un servizio */
36 #define Book(v, n)     v = v | (1 << (n))
37 #define Cancel(v, n)   v = v & ~(1 << (n))
```

Listato 6: macro per le variabili "booked"

Il `proctype` che modella il servizio della carta di credito è riportato nel Listato 7. Il comportamento è alquanto semplice dato che all'infinito attende una richiesta e risponde positivamente. Si sarebbe potuto ottenere tale comportamento ciclico anche utilizzando il costrutto `do...od` presente in PROMELA ma, dato questo processo (come altri riportati nel seguito) non deve

terminare ed ammette un unica sequenza di comandi eseguita all'infinito, il salto incondizionato `goto` è sufficente. La scelta nondeterministica tra le guardie attive del costrutto `do...od` non rappresenta un vantaggio in questo caso: se la receive non è eseguibile il processo deve bloccarsi finché qualcuno non si sincronizzerà per inviare una richiesta.

La scelta di utilizzare il costrutto `atomic`, per rendere atomica la sequenza di operazioni di ricezione ed invio, è stata fatta per evitare interleaving non desiderati. Questi effetti indesiderati non si verificano nello scenario A, dove è previsto un solo sistema di emergenza, ma in quelli successivi, dove invece ne sono previsti almeno due. Pensiamo per esempio ad un modello con due sistemi di emergenza ed un servizio di carta di credito. In tale modello può succedere che un sistema invii la richiesta al servizio ma riceva poi la risposta dall'altro sistema che è convinto di inviare una richiesta al servizio quando, in realtà, sta rispondendo all'altro car emergency system. L'utilizzo di `atomic` nel servizio, invece, provoca l'avanzamento del solo sistema che ha inviato la richiesta e quindi riceverà la risposta dalla carta di credito[3].

```
214 /* servizio carta di credito
215    sempre disponibile e che ha sempre successo */
216 proctype creditcard(chan ch)
217 {
218 endCC:  atomic { ch ? REQUEST;
219      /* le richieste hanno sempre successo */
220      ch ! AVAILABLE };
221      goto endCC
222 }
```

Listato 7: proctype della carta di credito

Nel Listato 8 è invece riportato il `proctype` che modella un generico servizio. Nella descrizione dello scenario A infatti si richiede lo stesso comportamento per ogni processo delle tipologia officina, carro attrezzi e noleggio auto. Quello che differenzia un processo da un altro è il canale su cui comunica, in base a quale canale viene utilizzato il generico servizio rappresenta un servizio officina o carro attrezzi o noleggio. Quindi, invece di fare un particolare `proctype` per ogni tipologia di servizio si è preferito avere un unico `proctype` parametrico rispetto al canale con cui comunica. Se per esempio in fase di istanziazione di un processo che modella un'officina si passa come parametro attuale un canale di quelli presenti nell'array di canali per le officine, tale processo risulterà essere un'officina per il sistema d'emergenza.

Un servizio all'inizio sceglie nondeterministicamente[4] se è disponibile o occupato (grazie al costrutto `if...fi`) e successivamente il suo comportamento è ciclico: attende una richiesta ed in base allo stato agisce. Se è disponibile la prenotazione ha successo e quindi il processo attende il rilascio, altrimenti restituisce un fallimento. Le macro utilizzate nella ricezione e nell'invio sul canale, presenti anche nel Listato 7, sono delle costanti e quindi le receive (indicate dal simbolo `?`) sul canale risultano bloccanti nel caso che le send (indicate dal simbolo `!`) con cui si siano sincronizzate non trasmettano la stessa costante. Oltre al fatto che entrambe le operazioni risultino bloccanti perché tutti i canali sono di tipo rendezvous. In questo modo si forza il protocollo della comunicazione che due processi stabiliscono: prima la fase di richiesta, seguita dalla risposta e nel caso di servizio disponibile, la successiva attesa del rilascio da parte del richiedente ed un ack finale inviato dal servizio.

È importante notare che il processo non risponde a nessuna richiesta di prenotazione quando si trova nello stato "prenotato". Semplicemente attende il rilascio. Per implementare tale comportamento bisognerebbe aggiungere un costrutto `do...od`, in coda alla dichiarazione, al quale saltare dopo la ricezione della richiesta di prenotazione, nel caso il servizio sia disponibile.

[3] questa soluzione potrebbe rappresentare un'alternativa alle variabili presentate nel Listato 5 per risolvere il problema dell'accesso unico al canale

[4] si sarebbe potuto utilizzare alternativamente il costrutto `select` di PROMELA

All'interno del costrutto di ripetizione si gestirebbe sia il rilascio sia eventuali richieste. Questa aggiunta, però, risulta inutile nel programma perché, grazie alle variabili introdotte nel Listato 5, se un servizio è prenotato nessun sistema di emergenza, escluso quello che detiene la prenotazione, può comunicare con il servizio. Si rimanda al Listato 9 per i dettagli della comunicazione dal lato di un sistema di emergenza.

```
225 /* generico servizio
226    sceglie nondeterministicamente se è disponibile od occupato */
227 proctype service(chan ch)
228 {
229     /* indica se il servizio è disponibile o meno */
230     bool state;
231
232     /* scelta nonderminisica dello stato */
233     if
234     :: state = false
235     :: state = true
236     fi;
237
238 endS: ch ? REQUEST, _;
239     if
240     /* se è disponibile allora viene prenotato */
241     :: state ->
242       ch ! AVAILABLE, _pid;
243       /* il messaggio di release deve avere il pid del servizio:
244          ovvero il ticket di prenotazione */
245 bookS:    ch ? RELEASE, eval(_pid);
246       /* annullo il ticket di prenotazione, inviando 0 */
247       ch ! AVAILABLE, 0
248     /* altrimenti è occupato */
249     :: !state -> ch ! BUSY, 0
250     fi;
251     goto endS
252 }
```

Listato 8: proctype del generico servizio

Dal codice si capisce chiaramente l'utilizzo del secondo campo dei canali per i servizi di tipo officina, carro attrezzi e noleggio auto. Il processo infatti invia il proprio pid sono nel caso sia disponibile, altrimenti invia il valore `0`. Quest'ultimo valore è quello assegnato alla variabile `_pid` del primo processo creato, che in questo caso è l'`init` e dato che è questo a creare tutti gli altri processi, tutti i servizi avranno pid diverso da zero. Nel modello si assume di conseguenza di indicare con lo zero la mancanza di ticket, mentre con tutti gli altri 255 valori i diversi ticket di altrettanti possibili servizi. Come già detto, il ticket serve a rappresentare il fatto che un servizio è stato prenotato ed inoltre permette di capire quale specifico processo si è prenotato. Nella fase di rilascio, tale processo (che modella un servizio) si aspetta di ricevere il ticket che ha inviato e dal codice soprastante è visibile che la receive di rilascio, ovvero quella con la macro `RELEASE`, si aspetta esattamente il valore della variabile `_pid`. Il costrutto `eval` infatti trasforma un'espressione in una costante, rendendo non eseguibile il rilascio con ticket (e quindi pid) diverso da quello del precedentemente inviato.

L'ultimo `proctype` presente nel programma che modella lo scenario A è quello riguardante il sistema d'emergenza ed è riportato nel Listato 9. Tale `proctype` non prevede parametri ed

utilizza alcune variabili per memorizzare le informazioni importanti del workflow: se l'utente è registrato o meno, il risultato di ogni richiesta, i ticket per ogni tipologia di servizio e delle variabili di comodo per indicare quale canale di un array di canali. Il comportamento del processo creato a partire dal `proctype` non è ciclico e prevede una serie di passi. Come prima cosa viene scelto nondeterministicamente se l'utente è registrato o meno e successivamente si inizia il workflow previsto in caso di guasto. Stiamo ovviamente astrando dal caso concreto in cui il sistema periodicamente controlla lo stato dell'autoveicolo e procede se rileva un guasto. Siamo interessati a capire il comportamento del sistema in caso di guasto e quindi consideriamo il veicolo danneggiato e necessario di riparazione.

Il workflow prevede inizialmente di differenziare il comportamento in base al fatto che l'utente sia abbonato o meno. Se lo è si procede con il resto, altrimenti si richiede l'autorizzazione al servizio della carta di credito. Nel caso che l'autorizzazione sia negata il processo passa nella fase di fallimento, altrimenti prova a prenotare un'officina. L'idea è che il processo provi a richiedere la prenotazione a tutte le officine non già prenotate e la prima che risponde positivamente lo faccia avanzare alla richiesta successiva. Se tutte le richieste falliscono, invece, si ha il fallimento del workflow. In caso di successo, lo scenario prevede la richiesta parallela di carro attrezzi e noleggio, ma PROMELA non permette di esprimere questo tipo di parallelismo. Si è quindi scelto di trasformare la richiesta parallela in una richiesta sequenziale dove l'ordine non conta perché non c'è dipendenza tra le due sotto-richieste. La decisione di richiedere il carro attrezzi prima del noleggio permette solo di evitare la richiesta al noleggio se la prenotazione di un carro attrezzi fallisce. La richiesta del carro attrezzi è simile a quella dell'officina, anche in questo caso se tutte le richieste falliscono il workflow fallisce. Per quanto riguarda il noleggio invece, il modello prevede un unico processo per questo servizio e quindi non viene utilizzato il costrutto `for`. Inoltre, il fallimento della richiesta non provoca il fallimento del workflow e quindi l'unica cosa prevista è il rilascio del canale del noleggio. Concludendo la descrizione astratta del comportamento, è da notare che da entrambi gli stati raggiunti dopo la fase delle richieste, indicati dalle etichetta `succ` e `fail`, il processo procede allo stato in cui rilascia le eventuali prenotazioni acquisite e termina.

I dettagli importanti di questo `proctype` sono il modo in cui avvengono le richieste dei servizi ed il rilascio finale. Per la richiesta di officina e carro attrezzi si è utilizzato il costrutto `for`[5] in maniera da far variare una variabile sugli indici di un array, che nello specifico è l'array di canali di officine o carro attrezzi. Dato il valore della variabile, ovvero l'indice, si testa se il relativo servizio non sia già prenotato, questo utilizzando le macro definite nel Listato 6. Si è utilizzato il costrutto `atomic` come guardia del comando condizionale per evitare l'interleaving tra il momento in cui si testa e quello in cui si ottiene in mutua esclusione il servizio (e quindi la capacità di comunicare con esso). Ovviamente se il test vale `0`, la prima guardia non è eseguibile e viene eseguita la seconda, passando all'indice successivo se presente. Se invece la prima guardia è eseguibile l'atomicità non può venire persa perché la prenotazione è un assegnamento, che è sempre eseguibile. Successivamente al blocco del canale si procede alla comunicazione con il servizio. Si invia e si riceve su un canale dell'array, grazie al valore delle variabili `idg` o `idtt`. Nell'invio della richiesta, il valore del secondo campo non è rilevante e quindi si può inviare un qualsiasi valore del tipo `byte`. La ricezione, invece, assegna i valori sul canale alle variabili `result` e `pid_garage` o `pid_towtruck`. A questo punto, se il risultato è positivo (ovvero `1`) si salta con l'istruzione `goto` alla richiesta successiva. Altrimenti si continua il normale control-flow dopo avere liberato il canale con il relativo assegnamento. La richiesta del noleggio è simile, a parte le particolarità già indicate.

L'etichetta `fail` indica il fallimento del workflow ed il processo salta a questa istruzione nel qual caso una richiesta fallisca (ad esclusione della richiesta del noleggio). Il rilascio finale delle prenotazioni, infine, segue l'ordine con cui queste vengono richieste (e possibilmente ottenute). Se la variabile `pid_garage` è diversa da zero, ovvero si ha il ticket di un'officina, si invia sul

[5] che in realtà è convertito nel corrispondente codice PROMELA prima di produrre il sorgente C

canale dell'officina prenotata il messaggio di release insieme al valore del pid. In questo caso siamo sicuri che la variabile `idg` contenga l'indice del canale dell'officina prenotata perché in precedenza il processo era saltato all'etichetta `reqTT` appena ricevuta la disponibilità dell'officina e successivamente il valore della variabile non è stato cambiato. L'istruzione seguente, quella alla riga 193, è invece uno statement di receive in cui si attende che il servizio confermi di essere nuovamente disponibile. Come effetto laterale, alla variabile `pid_garage` viene assegnato il valore ricevuto, ovvero `0`, che rappresenta la mancanza di ticket. Una considerazione importante è la seguente: tale variabile, come anche `pid_towtruck` e `pid_reantalcar`, è inizializzata a zero in fase di dichiarazione ed è utilizzata sono il lettura in tutto il `proctype`, ad esclusione delle istruzione di ricezione. Quindi, risulta essere una variabile locale al sistema di emergenza, modificata dal servizio remoto a cui si fa la richiesta, mediante le receive. La successiva ed ultima istruzione rilascia la mutua esclusione che il processo possedeva sul canale. Il rilascio del carro attrezzi ed del noleggio auto sono analoghi. L'etichetta `endCES` indica lo stato finale, ma non è indispensabile, dato che precede lo stato finale del processo, ovvero quello rappresentato dalla parentesi graffa chiusa.

```
87  /* car emergency system */
88  proctype ces()
89  {
90      /* ha un abbonamento? */
91      bool subscribing;
92      bool result;
93      /* pid dei servizi prenotati:
94         se pid == 0 allora il servizio non è prenotato
95         altrimenti il pid è usato come ricevuta di prenotazione */
96      byte pid_garage = 0, pid_towtruck = 0, pid_rentalcar = 0;
97      /* indice del canale dei servizi prenotati (per servizi con
           più istanze) */
98      byte idg = 0, idtt = 0;
99
100     /* scelta non deterministica se è abbonato o meno */
101     if
102     :: subscribing = false
103     :: subscribing = true
104     fi;
105
106 damage: if
107     /* è abbonato */
108     :: subscribing ->
109 sub:    skip
110     /* non è abbonato, richiedo l'autorizzazione per la
111        prenotazione con carta di credito */
112     :: !subscribing ->
113       creditcard_ch ! REQUEST;
114       creditcard_ch ? result;
115       if
116       /* autorizzazione ottenuta */
117       :: result ->
118 auth:    skip
119       /* autorizzazione negata */
120       :: !result -> goto fail
121       fi
```

```
122     fi;
123
124     /* tentativo di prenotazione dell'officina */
125 reqG: for(idg in garage_chs) {
126       if
127       /* se non è prenotata da un altro ces, blocco l'officina */
128       :: atomic { notBooked(garage_booked, idg) ->
129         Book(garage_booked, idg) };
130         garage_chs[idg] ! REQUEST, 0;
131         garage_chs[idg] ? result, pid_garage;
132         if
133         /* officina prenotata */
134         :: result -> goto reqTT
135         /* officina occupata, la rilascio e continuo ad iterare
              */
136         :: !result -> Cancel(garage_booked, idg)
137         fi
138       /* altrimenti continuo ad iterare */
139       :: else -> skip
140       fi
141     }
142     /* se viene raggiunto questo punto allora tutte le
           prenotazioni sono fallite */
143     goto fail;
144
145     /* tentativo di prenotazione del carro attrezzi */
146 reqTT:  for(idtt in towtruck_chs) {
147       if
148       /* se non è prenotato da un altro ces, blocco il carro
            attrezzi */
149       :: atomic { notBooked(towtruck_booked, idtt) ->
150         Book(towtruck_booked, idtt) };
151         towtruck_chs[idtt] ! REQUEST, 0;
152         towtruck_chs[idtt] ? result, pid_towtruck;
153         if
154         /* carro attrezzi prenotato */
155         :: result -> goto reqRC
156         /* carro attrezzi occupato, continuo ad iterare */
157         :: !result -> Cancel(towtruck_booked, idtt)
158         fi
159       /* altrimenti continuo ad iterare */
160       :: else -> skip
161       fi
162     }
163     /* se viene raggiunto questo punto allora tutte le
           prenotazioni sono fallite */
164     goto fail;
165
166     /* tentativo di prenotazione del noleggio auto */
167 reqRC:  if
168     :: atomic { !rentalcar_booked ->
```

```
169         rentalcar_booked = true };
170         rentalcar_ch ! REQUEST, 0;
171         rentalcar_ch ? result, pid_rentalcar;
172         if
173         :: result -> skip
174         :: !result -> rentalcar_booked = false
175         fi
176         /* sia in caso di fallimento della prenotazione, sia in
                caso di successo
177              procedo con la riparazione
178              nota: se result == true allora pid_rentalcar != 0,
                     ovvero vale
179              assert(!result || (pid_rentalcar != 0)); */
180          /* se il servizio è già prenotato, continuo con il
                workflow */
181         :: else -> skip
182         fi;
183
184         /* workflow è terminato con successo */
185 succ:   goto rel;
186
187 fail:   skip;
188       /* release dei servizi prenotati */
189 rel:    if
190       :: pid_garage != 0 ->
191         garage_chs[idg] ! RELEASE, pid_garage;
192         garage_chs[idg] ? AVAILABLE, pid_garage;
193         Cancel(garage_booked, idg);
194         if
195         :: pid_towtruck != 0 ->
196           towtruck_chs[idtt] ! RELEASE, pid_towtruck;
197           towtruck_chs[idtt] ? AVAILABLE, pid_towtruck;
198           Cancel(towtruck_booked, idtt)
199         :: else -> skip
200         fi;
201         if
202         :: pid_rentalcar != 0 ->
203           rentalcar_ch ! RELEASE, pid_rentalcar;
204           rentalcar_ch ? AVAILABLE, pid_rentalcar;
205           rentalcar_booked = false
206         :: else -> skip
207         fi;
208       :: else -> skip
209       fi;
210 endCES: skip
211 }
```

Listato 9: proctype del sistema di emergenza

Infine nel Listato 10 è presente l'`init` che istanzia i vari processi e memorizza i pid. Anche in questo caso se `MULTI` è definita vanno eseguite le operazioni riguardanti le altre istanze di officina e carro attrezzi.

```
261 /* inizializzazione */
262 init {
263   atomic {
264     run ces();
265     run creditcard(creditcard_ch);
266     g_pid1 = run service(garage_chs[0]);
267     tt_pid1 = run service(towtruck_chs[0]);
268 /* se MULTI è definito considero anche il secondo servizio */
269 #ifdef MULTI
270     g_pid2 = run service(garage_chs[1]);
271     tt_pid2 = run service(towtruck_chs[1]);
272 #endif
273     rc_pid = run service(rentalcar_ch)
274   }
275 }
```

Listato 10: init del programma

In ogni `proctype` è stata definita un etichetta che inizia con la parola chiave `end` per indicare uno stato terminale del processo ed evitare che la natura ciclica di alcuni processi provochi il fallimento della ricerca di endstate non validi.

Proprietà

Trattiamo ora le proprietà rilevanti dello scenario A. Dato che a partire dal programma sono ottenibili due modelli, in base al fatto che sia definita o meno la macro `MULTI`, per alcune proprietà sarà necessario distinguere i due casi.

a) "Per procedere nelle richieste dei servizi è necessario essere abbonati oppure avere l'autorizzazione per l'uso della carta di credito."

Questa proprietà esprime una precedenza temporale tra il momento in cui si controlla se l'utente è abbonato, e nel caso questo non lo sia si richiede l'autorizzazione per l'uso della carta di credito, e la fase delle richieste. Un traccia in cui si procede alle richieste dei servizi senza accertarsi dello stato di abbonamento rappresenta un comportamento non voluto del sistema.

La proprietà è di safety, in quanto esprime una situazione che non deve verificarsi nel modello. Il caso in cui il sistema di emergenza non faccia mai una richiesta non rappresenta una violazione della proprietà. Risulta quindi naturale esprimere questa proprietà utilizzando l'operatore weak until $\mathcal{W}$, presente in PROMELA, e delle remote reference. La relativa formulazione LTL è la seguente.

$$(\neg \text{ces@reqG})\,\mathcal{W}\,(\text{ces@sub} \vee \text{ces@auth})$$

Tale proprietà[6] è verificata dal modello e ci si può convincere di ciò in quanto l'etichetta `ces@reqG` è applicata ad uno stato che si può raggiungere solo se si è abbonati o se si ottiene l'autorizzazione per l'uso della carta di credito, nel caso non si fosse abbonati. Per distinguere le due possibilità viene utilizzato il costrutto `if...fi`. Se l'utente è abbonato semplicemente si procede alle richieste dei servizi, in caso contrario si procedere con la richiesta al servizio di carta di credito. Se tale richiesta ha successo si procede con la prenotazione dell'officina, altrimenti si fallisce. Anche se l'autorizzazione non avesse

[6] risulta essere di "precedence" secondo la raccolta di pattern `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

sempre successo, come richiesto nello scenario, la proprietà sarebbe verificata perché si salterebbe all'etichetta `fail` che identifica uno stato da cui non è raggiungibile quello delle richieste.

b) "Il workflow ha successo nel caso almeno un servizio di officina e uno di carro attrezzi siano disponibili, anche se il noleggio non è disponibile."

La proprietà esprime il fatto che sotto certe condizioni, ovvero la disponibilità di almeno un'officina e di un carro attrezzi, il workflow avrà successo. In altre parole esprime il fatto che un comportamento desiderato del sistema si verificherà. La proprietà è quindi di liveness e si può esprimere come segue, facendo uso di remote variable reference.

$$(\text{service[g\_pid1]:state} \wedge \text{service[tt\_pid1]:state}) \longrightarrow \Diamond \text{ces@succ}$$

Questa formulazione ha però un problema rispetto al programma presentato, legato al modo in cui la proprietà viene verificata. Spin traduce ogni formula LTL in un never claim e in questo caso specifico il never claim prodotto è quello riportato nel Listato 11. Tale never claim all'inizio controllerà il valore delle variabili `service[g_pid1]:state` e `service[tt_pid1]:state` (tralasciamo per il momento il remote reference a `ces`) e procederà solo se entrambe valgono `true`. Negli altri casi la proprietà è verificata perché la guardia del costrutto `do...od` vale `0` e il never claim è bloccato.

Durante la verifica della proprietà, il never claim viene fatto avanzare contemporaneamente al modello. Nello stato iniziale l'unico processo attivo è `init`, che deve eseguire la sua prima istruzione, e il never claim si trova nello stato etichettato con `T0_init`. L'unica guardia del costrutto iterativo vale `0` perché l'`init` non ha ancora creato i processi che modellano le varie entità dello scenario e quindi le relative variabili non esistono[7]. Di conseguenza il never claim è bloccato e la verifica termina con successo.

```
 0  never prop_b0 {    /* !((! ((service[g_pid1]:state) && (
        service[tt_pid1]:state))) || (<> ((ces@succ)))) */
 1  accept_init:
 2  T0_init:
 3    do
 4    :: ( ((service[g_pid1]:state) && (service[tt_pid1]:state))
        && ! (((ces@succ)))) -> goto accept_S3
 5    od;
 6  accept_S3:
 7  T0_S3:
 8    do
 9    :: (! (ces@succ)) -> goto accept_S3
10    od;
11  }
```

Listato 11: never claim proprietà

La formulazione della proprietà data controlla il valore delle variabili che indicano lo stato di un servizio prima che il relativo processo sia creato. Vorremmo invece essere in grado di scartare i primi passi della verifica, quelli in cui non sono ancora stati creati tutti i processi, e provare se vale la premessa quando effettivamente i processi che modellano i servizi sono attivi. Inoltre, nello stato iniziale di un processo creato a partire dal `proctype service`, la variabile `state` vale inizialmente `false`[8]. Solo dopo la scelta nondeterministica la

[7] questa condizione sembra essere equivalente a quella in cui entrambe le variabili valgono `false`

[8] perché il valore iniziale di una variabile dichiarata ma non inizializzata è `0`, come indicato nella pagina `http://spinroot.com/spin/Man/datatypes.html`

variabile potrebbe contenere il valore `true`. La formulazione corretta della proprietà deve quindi asserire che se il servizio di officina e quello di carro attrezzi saranno disponibili, allora il workflow avrà successo. La relativa formula LTL è la seguente[9].

$$\Diamond(\text{service[g\_pid1]:state} \wedge \text{service[tt\_pid1]:state}) \longrightarrow \Diamond \text{ces@succ}$$

Nel caso che la macro `MULTI` sia definita e pertanto il modello presenti più istanze di servizi di officina e carro attrezzi, almeno uno di questi deve essere disponibile.

$$\Diamond\left(\begin{array}{l}(\text{service[g\_pid1]:state} \vee \text{service[g\_pid2]:state}) \\ \wedge\ (\text{service[tt\_pid1]:state} \vee \text{service[tt\_pid2]:state})\end{array}\right) \longrightarrow \Diamond \text{ces@succ}$$

Il modello verifica questa proprietà perché, se la variabile `state` di un servizio vale `true`, la prima guardia del costrutto `if...fi`, presente nel Listato 8, è eseguibile e da questa si procede inviando il messaggio di prenotazione avvenuta con successo. Se si ha questo comportamento per il servizio di officina e di carro attrezzi il sistema di emergenza non può che avere successo, dato che non sono previsti salti dall'istruzione alla riga 167 alla riga 185 (che rappresenta lo stato di successo).

Una nota da fare riguardante la verifica di questa proprietà è di non utilizzare la partial order reduction perché questa non è compatibile con le "remote variable reference" utilizzate per formalizzare la proprietà[10].

c) "In nessuno stato terminale, né di successo né di fallimento, un servizio è prenotato e non rilasciato."

Anche in questo caso la proprietà descrive un comportamento non desiderato del sistema. Risulta di conseguenza essere una proprietà di safety. La sua formulazione in LTL, che fa uso di remote reference, è la seguente.

$$\Box\neg(\text{ces@endCES} \wedge (\text{service[g\_pid1]@bookS} \vee \text{service[tt\_pid1]@bookS} \vee \text{service[rc\_pid]@bookS}))$$

Ovvero non vogliamo che il sistema d'emergenza si trovi nello stato finale e contestualmente uno dei servizi si trovi nello stato di prenotazione avvenuta ma non rilasciata. Se la macro `MULTI` risulta definita dobbiamo estendere la formula per le istanze aggiuntive del modello.

$$\Box\neg\left(\text{ces@endCES} \wedge \left(\begin{array}{c}\text{service[g\_pid1]@bookS} \vee \text{service[g\_pid2]@bookS} \\ \vee\ \text{service[tt\_pid1]@bookS} \vee \text{service[tt\_pid2]@bookS} \\ \vee\ \text{service[rc\_pid]@bookS}\end{array}\right)\right)$$

Il modello verifica la proprietà perché sia dall'etichetta `succ`, che indica lo stato di successo, sia dall'etichetta `fail`, che indica lo stato di fallimento, il sistema d'emergenza procede al rilascio delle risorse prenotate, ovvero quelle indicate dalle variabili `pid_garage`, `pid_towtruck` e `pid_rentalcar` (nel caso siano diverse da zero).

Alternativamente si potrebbe provare questa proprietà utilizzando la verifica di safety "invalid endstates" presente in Spin, ma solo dopo aver etichettato correttamente quali sono gli stati finali desiderati. Riguardando il codice riportato nel Listato 8 si può osservare come l'etichetta che inizia con la parola chiave `end` sia stata posizionata nello stato in cui un servizio attende una richiesta e quindi non risulta prenotato. Se nello stato terminale del modello tutti i servizi si trovano nello stato etichettato con `endS`, allora nessuno di questi risulta prenotato. Di conseguenza se la verifica "invalid endstates" termina con successo, come effettivamente accade, la proprietà che ci interessa è verificata.

[9] nel programma è presente anche una formulazione analoga che rispetta il pattern "existence after Q" presentato nella pagina `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

[10] come riportato alla pagina `http://spinroot.com/spin/Man/remoterefs.html`

Spazio degli stati

Il programma presentato permette di ottenere due modelli che si differenziano per il numero di processi attivi. Siamo interessati ad una stima dello spazio degli stati di ognuno. Per calcolare tale stima generalmente bisogna considerare le variabili, i canali ed il numero di stati dei processi. In questo caso però possiamo fare delle osservazioni che ci permetteranno di semplificare il calcolo. La prima considerazione riguarda i canali usati. Questi sono tutti rendezvous e quindi non vanno ad incidere sullo spazio degli stati. Possiamo poi notare che il processo `init` è il primo ad essere eseguito e quello incaricato di creare gli altri processi. Tale processo procede nella sua computazione senza interleaving (ad esclusione della transizione nello stato terminale) e quindi il suo contributo allo spazio degli stati è additivo. Nello specifico si compone di solo 3 stati, dato che la creazione dei processi avviene in maniera atomica. A livello globale possiamo dunque trascurare le variabili riportate nel Listato 4 dato che il modificatore `local` ci permette di considerarle locali al processo dove sono usate, ovvero l'`init`.

L'utilizzo di canali rendezvous ha ripercussioni anche sui processi che modellano i servizi. Tale tipologia di canale fa infatti avanzare in maniera atomica entrambi i processi coinvolti nella comunicazione. In altre parole i due processi si sincronizzano e come risultato non si può avere l'interleaving tra questi (limitatamente agli statement di send e receive). Inoltre possiamo notare che i processi che modellano i servizi si compongono quasi totalmente di istruzioni di invio e ricezione sul canale. Queste due osservazioni ci permettono di trascurare gli stati dei processi `creditcard` e `service`, dato che entrambi avanzano contemporaneamente al processo `ces`. L'unica cosa che dobbiamo considerare è la variabile `state` (di tipo `bool`) presente nel `proctype` che modella un generico servizio, mentre il processo `creditcard` non ha variabili.

Le variabili globali di cui tenere conto sono quelle riportate nel Listato 5. Per tali variabili sono possibili ben 2^{17} ($= 256^2 \times 2$) configurazioni. Per sapere il numero di stati del processo che modella un car emergency system ricorriamo invece all'opzione[11] di Spin che produce l'automa di un `proctype`. Ricorrendo a tale strumento possiamo sapere che l'automa ha 51 stati, compreso quello in cui il processo attende che il suo pid sia reso nuovamente disponibile. Possiamo trascurare questo particolare stato e approssimare il numero di stati a 50. Il processo `ces` prevede inoltre 5 variabili di tipo `byte` e due di tipo `bool`. Le configurazioni possibili sono dunque 2^{42} ($= 256^5 \times 2^2$).

La dimensione dello spazio degli stati del modello con meno processi è di circa 50×2^{62} stati ($= 50 \times 2^{42} \times 2^{17} \times 2^3$, dove l'ultimo fattore indica le configurazioni della variabile `state` per i 3 servizi). Tale valore è maggiorato da $2^{68} \approx 256 \times 10^{18}$. L'altro modello, che si ottiene definendo la macro `MULTI`, ha una dimensione di circa 50×2^{64} ($= 50 \times 2^{42} \times 2^{17} \times 2^5$) stati. Valore che possiamo maggiorare con $2^{70} \approx 1 \times 10^{21}$.

Scenario B

Lo scenario B estende quello A, richiedendo che ci siano almeno due processi che modellano altrettanti sistemi di emergenza. Impone poi il vincolo di avere almeno due officine, ma un solo carro attrezzi. Rispetto a prima, queste due tipologie di servizi sono ora inizialmente liberi. Il servizio di noleggio, invece, continua a scegliere in modo nondeterministico se è libero o occupato. Viene poi richiesto che tutti i processi che modellano un'entità dello scenario siano ciclici. Rispetto allo scenario precedente, solo il `proctype` che modella il sistema di emergenza deve essere rivisto. Data la presenza di almeno due car emergency system, è necessario che ogni servizio sia prenotabile da uno solo alla volta.

Il modello che verrà presentato è stato ottenuto aggiungendo o rivedendo alcuni dettagli di quello implementato per lo scenario A. Nel seguito, quindi, si farà riferimento solo ai cambia-

[11] nello specifico l'opzione `-D` del comando `./pan`

menti apportati ed ai particolari aggiunti. Per quanto riguarda il numero di processi nel modello è stato deciso di avere due processi che modellano un car emergency system e due processi che modellano un'officina. Questa scelta assicura che ogni sistema d'emergenza riesca sempre ad ottenere la prenotazione di un'officina[12]. Se diversamente il numero di car emergency system fosse maggiore del numero di officine, quanto appena detto non sarebbe vero ed il workflow di alcuni sistemi fallirebbe durante la richiesta dell'officina. Lo scenario è comunque pensato in modo che un problema analogo possa succedere nella fase di richiesta del carro attrezzi. Si è preferito dunque modellare uno scenario più semplice, in cui il fallimento del workflow non sia provocato durante la fase di prenotazione dell'officina.

A livello globale è stata definita una macro, riportata nel Listato 12, per indicare il numero di officine presenti nel modello. Questa definizione va a sostituire quella riportata nel Listato 2.

```
32 /* numero di garage */
33 #define NGARAGE 2
```

Listato 12: macro per numero di officine

Rispetto al modello precedente è previsto ora un solo servizio di carro attrezzi ma due di officina. È stato di conseguenza dichiarato un array di canali per i servizi officina ed un singolo canale per il carro attrezzi. Le modifiche fatte alle dichiarazioni dei canali si posso vedere nel Listato 13.

```
39 /* canali officine */
40 chan garage_chs [NGARAGE] = [0] of {bit, byte};
41
42 /* canale carro attrezzi */
43 chan towtruck_ch = [0] of {bit, byte};
```

Listato 13: canali dei servizi

Anche in questo modello utilizziamo delle variabili globali per indicare se un servizio è prenotato o meno. Continuiamo ad utilizzare una variabile di tipi `byte` per i servizi della tipologia officina, ma ci limitiamo ad una variabile booleana per il servizio carro attrezzi dato che ne è previsto uno solo. Nel Listato 14 sono invece riportate le variabili globali, utilizzate solamente dal processo `init`, che contengono i pid dei vari processi creati. Queste dichiarazioni vanno a sostituire quelle presentate nel Listato 4.

```
49 /* variabili globali accedute localmente all'interno dell'init
50    indicano il pid dei servizi */
51 local byte ces_pid1 = 0;
52 local byte ces_pid2 = 0;
53 local byte g_pid1 = 0;
54 local byte g_pid2 = 0;
55 local byte tt_pid = 0;
56 local byte rc_pid = 0;
```

Listato 14: variabili per i pid dei processi

Il `proctype` della carta di credito non subisce cambiamenti, si rimanda al Listato 7 per il codice. Il resto dei servizi subisce invece qualche modifica rispetto allo scenario precedente. Adesso non abbiamo più lo stesso comportamento per officina, carro attrezzi e noleggio e ciò ci porta a dichiarare più `proctype`. Un servizio officina o carro attrezzi è inizialmente disponibile ed il comportamento poi è esattamente quello presentato nel Listato 8 nel caso si scelga nondeterministicamente di essere liberi. Quindi è stato previsto un `proctype` per il servizio generico,

[12] e dato che stiamo facendo model checking è naturale formalizzare questa proprietà e chiedersi se il modello la verifica: si veda la formula LTL `non_book` nel programma `CarEmB.pml`

inizialmente disponibile. La dichiarazione è riportata nel Listato 15 e viene utilizzata per istanziare sia processi che modellano officine, sia processi che modellano carro attrezzi. Come nello scenario A quello che differenzia le due istanze è il canale passato come parametro al momento della creazione.

```
205 /* generico servizio inizialmente disponibile */
206 proctype service(chan ch)
207 {
208 waitS:  ch ? REQUEST, _;
209     ch ! AVAILABLE, _pid;
210     /* il messaggio di release deve avere il pid del servizio */
211 bookS:  ch ? RELEASE, eval(_pid);
212     ch ! AVAILABLE, 0;
213     goto waitS
214 }
```

Listato 15: proctype del generico servizio inizialmente disponibile

Il comportamento descritto dal `proctype` prevede di attendere una richiesta, rispondere positivamente ed infine attendere il rilascio (inviando anche un ack al sistema di emergenza). Questo comportamento è ripetuto ciclicamente dato che alla fine della sequenza di istruzioni si salta nuovamente alla prima, grazie al costrutto `goto`.

Il noleggio auto invece ha un comportamento che risulta invariato rispetto al Listato 8 presentato nello scenario A. Nel Listato 16 è stata riportata solo la prima riga della dichiarazione, in cui si è semplicemente cambiato il nome del `proctype`. Il resto del codice è lo stesso.

```
217 /* servizio noleggio auto
218    sceglie nondeterministicamente se è disponibile od occupato */
219 proctype rentalcar(chan ch) { ... }
```

Listato 16: proctype del noleggio auto

Il sistema di emergenza è richiesto essere ciclico in questo modello. Dobbiamo quindi apportare alcune modifiche rispetto al Listato 9. Il modo più semplice di rendere ciclico il processo è quello di sostituire l'istruzione etichettata con `endCES` con un'istruzione `goto` dove la label è `damage`, ovvero quella che rappresenta il momento in cui l'auto si guasta (subito dopo aver scelto nondeterministicamente lo stato di abbonamento). Il codice del nuovo `proctype` è riportato nel Listato 17. In tale descrizione del comportamento del processo che modella un sistema d'emergenza si può notare anche com'è cambiato il modo di richiedere il carro attrezzi. Lo scenario infatti prevede un solo servizio di carro attrezzi e quindi nel codice non viene più utilizzato il costrutto `for`, utile nel caso ci siano più istanze di un servizio.

Nel dettaglio un processo prova ad ottenere in mutua esclusione il canale per la comunicazione con il processo che modella il carro attrezzi e, se riesce ad acquisire tale risorsa, invia la propria richiesta. In caso contrario il carro attrezzi risulta prenotato e quindi il processo fallisce il proprio workflow e deve procedere a rilasciare l'officina prenotata. Nel caso invece sia riuscito ad inviare la richiesta di prenotazione, attende la risposta ed in base a questa continua o meno nella sequenza di richieste. Se la prenotazione ha avuto successo il processo salta all'etichetta `reqRC` che indica la richiesta del noleggio auto, altrimenti rilascia il canale del carro attrezzi e salta all'etichetta `fail`, che rappresenta lo stato di fallimento del workflow. Per quanto riguarda il resto del codice, questo è rimasto invariato rispetto a quello presentato nello scenario A.

```
71 /* car emergency system */
72 proctype ces()
73 {
74     /* ha un abbonamento? */
75     bool subscribing;
```

```
76      bool result;
77      /* pid dei servizi prenotati:
78         se pid == 0 allora il servizio non è prenotato
79         altrimenti il pid è usato come ricevuta di prenotazione */
80      byte pid_garage = 0, pid_towtruck = 0, pid_rentalcar = 0;
81      /* indice del canale dei servizi prenotati (per servizi con
           più istanze) */
82      byte idg = 0;
83
84      /* scelta non deterministica se è abbonato o meno */
85  start:  if
86      :: subscribing = false
87      :: subscribing = true
88      fi;
89
90  damage: if
91      /* è abbonato */
92      :: subscribing ->
93  sub:  skip
94      /* non è abbonato, richiedo l'autorizzazione per la
95           prenotazione con carta di credito */
96      :: !subscribing ->
97        creditcard_ch ! REQUEST;
98        creditcard_ch ? result;
99        if
100       /* autorizzazione ottenuta */
101       :: result ->
102 auth:     skip
103       /* autorizzazione negata */
104       :: !result -> goto fail
105       fi
106     fi;
107
108     /* tentativo di prenotazione dell'officina */
109 reqG: for(idg in garage_chs) {
110       if
111       /* se non è prenotata da un altro ces, blocco l'officina */
112       :: atomic { notBooked(garage_booked, idg) ->
113         Book(garage_booked, idg) };
114         garage_chs[idg] ! REQUEST, 0;
115         garage_chs[idg] ? result, pid_garage;
116         if
117         /* officina prenotata */
118         :: result -> goto reqTT
119         /* officina occupata, la rilascio e continuo ad iterare
               */
120         :: !result -> Cancel(garage_booked, idg)
121         fi
122       /* altrimenti continuo ad iterare */
123       :: else -> skip
124       fi
```

```
125     }
126     /* se viene raggiunto questo punto allora tutte le
           prenotazioni sono fallite */
127     goto fail;
128
129     /* tentativo di prenotazione del carro attrezzi */
130 reqTT:  if
131     :: atomic { !towtruck_booked ->
132       towtruck_booked = true };
133       towtruck_ch ! REQUEST, 0;
134       towtruck_ch ? result, pid_towtruck;
135       if
136       /* carro attrezzi prenotato */
137       :: result -> skip
138       /* carro attrezzi occupato, prenotazione fallita */
139       :: !result -> towtruck_booked = false;
140         goto fail
141       fi
142     /* se il servizio è già prenotato, la richiesta è fallita */
143     :: else -> goto fail
144     fi;
145
146     /* tentativo di prenotazione del noleggio auto */
147 reqRC:  if
148     :: atomic { !rentalcar_booked ->
149       rentalcar_booked = true };
150       rentalcar_ch ! REQUEST, 0;
151       rentalcar_ch ? result, pid_rentalcar;
152       if
153       :: result -> skip
154       :: !result -> rentalcar_booked = false
155       fi
156       /* sia in caso di fallimento della prenotazione, sia in
             caso di successo
157           procedo con la riparazione
158           nota: se result == true allora pid_rentalcar != 0,
                 ovvero vale
159           assert(!result || (pid_rentalcar != 0)); */
160       /* se il servizio è già prenotato, continuo con il workflow
           */
161       :: else -> skip
162       fi;
163
164       /* workflow è terminato con successo */
165 succ: goto rel;
166
167 fail: skip;
168     /* release dei servizi prenotati */
169 rel:  if
170     :: pid_garage != 0 ->
171       garage_chs[idg] ! RELEASE, pid_garage;
```

```
172         garage_chs[idg] ? AVAILABLE, pid_garage;
173         Cancel(garage_booked, idg);
174         if
175         :: pid_towtruck != 0 ->
176           towtruck_ch ! RELEASE, pid_towtruck;
177           towtruck_ch ? AVAILABLE, pid_towtruck;
178           towtruck_booked = false
179         :: else -> skip
180         fi;
181         if
182         :: pid_rentalcar != 0 ->
183           rentalcar_ch ! RELEASE, pid_rentalcar;
184           rentalcar_ch ? AVAILABLE, pid_rentalcar;
185           rentalcar_booked = false
186         :: else -> skip
187         fi
188       :: else -> skip
189       fi;
190       goto damage
191   }
```

Listato 17: proctype del sistema di emergenza

Ed infine l'`init` del programma è riportato nel Listato 18. Come per lo scenario A, tale processo si limita a creare gli altri processi e memorizzare i loro pid nelle apposite variabili.

```
253 /* inizializzazione */
254 init {
255   atomic {
256     ces_pid1 = run ces();
257     ces_pid2 = run ces();
258     run creditcard(creditcard_ch);
259     g_pid1 = run service(garage_chs[0]);
260     g_pid2 = run service(garage_chs[1]);
261     tt_pid = run service(towtruck_ch);
262     rc_pid = run rentalcar(rentalcar_ch)
263   }
264 }
```

Listato 18: init del programma

Rispetto allo scenario A, nel programma sviluppato per questo scenario, sono state rinominate tutte le etichette che inviavano con la parola chiave `end`. Tale parola è stata rimpiazzata dalla parola `wait`, dato che tutti i processi sono ciclici e non è previsto che il modello termini.

Proprietà

Trattiamo ora le proprietà rilevanti dello scenario B.

d) "Ogni servizio è prenotato da un utente alla volta al massimo."

Questa proprietà richiede che nel modello non si verifichi mai che un servizio sia prenotato contemporaneamente da due utenti. Esprime quindi un comportamento non voluto del sistema e di conseguenza tale proprietà è di safety. Inoltre è richiesto che ogni stato del modello la rispetti e quindi nello specifico la proprietà è un invariante. Possiamo di fatto

notare che la richiesta è equivalente a quella che si ha nel problema della mutua esclusione. La formulazione LTL immediata che se ne darebbe è la seguente.

$$\Box\left(\begin{array}{l}\neg(\text{ces[ces\_pid1]:pid\_garage} == \text{g\_pid1} \wedge \text{ces[ces\_pid2]:pid\_garage} == \text{g\_pid1}) \wedge \\ \neg(\text{ces[ces\_pid1]:pid\_garage} == \text{g\_pid2} \wedge \text{ces[ces\_pid2]:pid\_garage} == \text{g\_pid2}) \wedge \\ \neg(\text{ces[ces\_pid1]:pid\_towtruck} == \text{tt\_pid} \wedge \text{ces[ces\_pid2]:pid\_towtruck} == \text{tt\_pid}) \wedge \\ \neg(\text{ces[ces\_pid1]:pid\_rentalcar} == \text{rc\_pid} \wedge \text{ces[ces\_pid2]:pid\_rentalcar} == \text{rc\_pid})\end{array}\right)$$

Tuttavia se si prova a verificare con SPIN se la proprietà vale, quello che si ottiene è un errore alla profondità zero. Questo errore è dovuto al fatto che il never claim generato prova a riferire variabili di processi non ancora creati. Analogamente a quanto fatto per la proprietà **b** dello scenario A bisogna, anche per questa proprietà, scartare i primi passi della verifica, quelli in cui i processi che modellano i sistemi d'emergenza non sono ancora stati creati. Seguendo il pattern "absence after Q" si può formulare correttamente la proprietà come segue.

$$\Box(\text{ces[ces\_pid1]@start} \wedge \text{ces[ces\_pid2]@start}) \longrightarrow \Box\left(\begin{array}{l}\neg(\text{ces[ces\_pid1]:pid\_garage} == \text{g\_pid1} \wedge \text{ces[ces\_pid2]:pid\_garage} == \text{g\_pid1}) \wedge \\ \neg(\text{ces[ces\_pid1]:pid\_garage} == \text{g\_pid2} \wedge \text{ces[ces\_pid2]:pid\_garage} == \text{g\_pid2}) \wedge \\ \neg(\text{ces[ces\_pid1]:pid\_towtruck} == \text{tt\_pid} \wedge \text{ces[ces\_pid2]:pid\_towtruck} == \text{tt\_pid}) \wedge \\ \neg(\text{ces[ces\_pid1]:pid\_rentalcar} == \text{rc\_pid} \wedge \text{ces[ces\_pid2]:pid\_rentalcar} == \text{rc\_pid})\end{array}\right)$$

Questa formulazione procede alla verifica della proprietà solo dopo che entrambi i processi che modellano un sistema d'emergenza hanno raggiunto l'etichetta `start`, che indica la prima istruzione del processo (prima di questa sono riportate solo le dichiarazioni e inizializzazioni delle variabili).

Il modello verifica questa proprietà perché grazie alle variabili che indicano se un servizio è prenotato, previste nel programma, solo chi ha effettivamente la prenotazione può comunicare con un servizio. Inoltre, ricordando quanto fatto notare nella descrizione dello scenario A, gli unici punti in cui si modificano le variabili che indicano i servizi prenotati sono le istruzioni di receive. Per il resto un sistema d'emergenza usa tali variabili in sola lettura. Quindi non è possibile che una di queste variabili sia modificata se non dal servizio che viene prenotato, che si blocca successivamente fino alla ricezione del messaggio di release.

e) "Ogni auto si guasta infinite volte."

Questa proprietà esprime una situazione che deve ripetersi all'infinito nel modello. Non è né una proprietà di safety né una proprietà di liveness. Si può facilmente esprimere utilizzano il pattern di "recurrence" come segue.

$$(\Box\Diamond\text{ces[ces\_pid1]@damage}) \wedge (\Box\Diamond\text{ces[ces\_pid2]@damage})$$

Ovvero vale sempre che prima o poi il processo `ces` con pid `ces_pid1` passi nello stato etichettato come `damage` e similmente che il processo `ces` con pid `ces_pid2` passi nello stesso stato. L'etichetta `damage` indica infatti il punto in cui un auto si guasta, quello in cui si verifica se un utente è abbonato o meno.

Il modello però non verifica questa proprietà perché la computazione in cui uno dei due processi `ces` rimane all'infinito fermo in un qualsiasi stato (che non sia `damage`) contraddice la richiesta. Per la verifica infatti non richiediamo che se un processo possa all'infinito eseguire un istruzione, allora l'esegua prima o poi. Quindi un processo potrebbe essere bloccato all'infinito in uno stato in cui la prossima istruzione è un assegnamento, statement che è sempre eseguibile.

f) "Assumendo weak fairness a livello di processi, ogni auto si guasta infinite volte."

Questa proprietà va ad arricchire quanto richiesto nella proprietà `e`, aggiungendo delle assunzioni di fairness. La formulazione LTL rimane la medesima, semplicemente in fase di verifica della proprietà dobbiamo dare al comando `./pan` l'opzione `-f` che indica l'assunzione di weak fairness.

Il modello verifica ora la proprietà e ciò è dovuto al fatto che tutte le istruzioni di un processo `ces` sono sempre eseguibili. Questa affermazione è però imprecisa perché alcune istruzioni presenti nel `proctype` del processo potrebbero risultare bloccanti, come per esempio le send e le receive. È proprio per fare in modo che la suddetta proprietà sia verificata che sono state introdotte le variabili presentate nel Listato 5, che servono ad indicare i servizi prenotati. Se tali variabili indicano che un servizio è già prenotato, un sistema d'emergenza non proverà a fare una richiesta, inviando un messaggio sul canale. Ciò permette al sistema che detiene la prenotazione di rilasciarla non appena possibile. Se infatti il servizio non fosse bloccato nell'attesa della receive e facesse nel frattempo delle altre operazioni, verrebbe meno il fatto che il messaggio di rilascio sia all'infinito eseguibile. Di conseguenza fallirebbe la verifica con uno dei due processi `ces` bloccato sull'istruzione di invio della release di un qualche servizio. C'è poi da notare che per ogni costrutto `if...fi` presente nel programma sono state previste due guardie. Quando non sono entrambe eseguibili, la seconda rappresenta la negazione della prima oppure è lo statement `else` e quindi una delle due è sempre eseguibile.

g) "Caratterizzazione di un "progress cycle" nel modello."

La richiesta è quella di etichettare, con etichette che iniziano con la parola chiave `progress`, alcuni stati del modello. Tali etichette servono per indicare gli stati dove il modello effettivamente progredisce. A partire da queste etichette, grazie alla verifica "non-progress cycles" presente in Spin, si riesce a stabilire se esistono o meno computazioni in cui il modello non progredisce: ovvero non passa infinite volte per uno degli stati di progresso. La stessa verifica può essere fatta utilizzando la seguente formula LTL.

$$\neg\Diamond\Box \mathrm{np\_}$$

In ogni caso, la particolarità di questo tipo di verifica è che, nonostante le etichette individuino uno stato locale ad un processo, il progresso riguarda l'intero sistema.

Nel modello sviluppato per lo scenario B si possono caratterizzare due "progress cycle" interessanti. Nonostante la richiesta fosse di un solo ciclo, nel seguito se ne presenteranno due. Il primo "progress cycle" rappresenta una computazione in cui il sistema d'emergenza riesce ad ottenere tutte le risorse necessarie al completamento del proprio workflow. Più precisamente possiamo dire che il modello progredisce ogni volta che un sistema d'emergenza ottiene l'autorizzazione per l'uso della carta di credito oppure risulta abbonato, quando la prenotazione dell'officina ha successo e quando riesce a prenotare il carro attrezzi. Il secondo "progress cycle" invece è pensato proprio per lo scenario B. Data la limitazione di un solo carro attrezzi è interessante capire come si comporta il modello sviluppato. In questo caso di progredisce solo quando la prenotazione del carro attrezzi ha successo.

Quella fin qui data, però, è solo la descrizione informale dei "progress cycle". La formalizzazione è data nei listati riportati sotto. A differenze di quanto fatto per le altre proprietà, formalizzate in coda al programma Promela scritto, per questa proprietà è stato creato un apposito file[13]. Per verificare entrambi i "progress cycle" evitando inutili

[13] `CarEmBprogress.pml`, ottenuto a partire dal file `CarEmB.pml`

modifiche manuali si è adottato lo stesso espediente presentato per lo scenario A, ovvero una apposita macro. Nello specifico si considera se è definita o meno la macro `ALL`[14]. Se la macro è definita si valuta il primo "progress cycle", in caso contrario si valuta il secondo.

Lo scenario prevede due processi che modellano altrettanti car emergency system. Se tali processi fossero creati dallo stesso `proctype`, adeguatamente etichettato con progress-label, il modello progredirebbe anche nel caso in cui uno dei due restasse sempre fermo nello stesso stato. Il processo che si muove infatti farebbe progredire il modello. Nel file creato per questa proprietà sono stati quindi previsti due `proctype` diversi per i due sistemi d'emergenza. Il primo `proctype` non è etichettato ed è esattamente quello riportato nel Listato 17, mentre il secondo è etichettato e la sua dichiarazione è quella presentata nel Listato 19.

```
178 /* car emergency system con progress label */
179 proctype cesP()
180 { ...
```

Listato 19: proctype generico servizio inizialmente disponibile

Nel corpo di tale `proctype` abbiamo inserito le apposite etichette che identificano gli stati di progresso. Il Listato 20 riporta il pezzo di codice dove viene aggiunta l'etichetta, se la macro `ALL` è definita, che si riferisce allo stato in cui si è verificato che l'utente è abbonato oppure si è ottenuta l'autorizzazione dal servizio della carta di credito (nel caso l'utente non fosse abbonato).

```
214       fi;
215 #ifdef ALL
216  progressCC: skip;
217 #endif
218
219       /* tentativo di prenotazione dell'officina */
220  reqGP:  for(idg in garage_chs) { ...
```

Listato 20: proctype generico servizio inizialmente disponibile

Allo stesso modo si etichetta lo stato che si raggiunge se la prenotazione dell'officina ha successo, come indicato nel Listato 21.

```
227           /* officina prenotata */
228           :: result ->
229 #ifdef ALL
230  progressG:       goto reqTTP
231 #else
232                  goto reqTTP
233 #endif
234           /* officina occupata, la rilascio e continuo ad
                 iterare */
235           :: !result -> Cancel(garage_booked, idg)
```

Listato 21: proctype generico servizio inizialmente disponibile

Il Listato 22 riporta l'unica etichetta che è sempre presente, ovvero quella che etichetta lo stato in cui è stato ottenuto il carro attrezzi.

```
251         /* carro attrezzi prenotato */
```

[14]per definire tale macro basta aggiungere l'opzione `-DALL` al comando `spin`

```
252         :: result ->
253  progressTT:    skip
254         /* carro attrezzi occupato, prenotazione fallita */
255         :: !result -> towtruck_booked = false;
```

Listato 22: proctype generico servizio inizialmente disponibile

Consideriamo ora la verifica di "non-progress cycles". Senza assunzioni di fairness il modello non verifica nessuno dei due cicli formalizzati. Questo perché la computazioni in cui il processo `cesP` resta all'infinito fermo nello stato etichettato `start` (ovvero il primo comando) rappresenta un ciclo del modello di non progresso. Se invece eseguiamo la verifica assumendo weak fairness a livello di processi, il risultato è differente per i due "progress cycle" formalizzati. Il primo ciclo, quello che si ottiene definendo nel modello la macro `ALL`, è verificato. Quindi non esistono nel modello cicli di non progresso in cui non si passa mai (da un certo punto in poi) in uno degli stati etichettati. Il fatto che la proprietà, assumendo weak fairness, sia verificata non è sorprendete. Infatti l'etichetta `progressCC` è sempre raggiunta da un processo se questo procede nella sua esecuzione. Diversamente il secondo ciclo non è verificato nemmeno assumendo weak fairness. Il controesempio riportato è quello in cui il processo `cesP` non riesce mai a prenotare il carro attrezzi. Questo significa che il modello sviluppato non presenta un limite al numero di volte che un sistema d'emergenza può prenotare il carro attrezzi e di conseguenza, se uno dei processi è più veloce dell'altro, non si riesce ad evitare il caso in cui un sistema non otterrà mai il carro attrezzi.

Spazio degli stati

Partendo dalle considerazioni fatte per la stima dello spazio degli stati dello scenario A possiamo dare una stima anche per questo scenario. L'automa prodotto per il `proctype` che modella un car emergency system prevede 44 stati. In questo caso il processo prevede però 4 variabili locali di tipo `byte` e 2 di tipo `bool`. Le possibili configurazioni di queste sono 2^{34} ($= 256^4 \times 2^2$). Anche le configurazioni possibili delle variabili globali sono diminuite, dato che una variabile che nello scenario A era di tipo `byte` in questo scenario è di tipo `bool`. I valori possibili sono quindi 2^{10} ($= 256 \times 2^2$). Rispetto a prima, in questo scenario solo il processo che modella il noleggio auto prevede una variabile locale (sempre di tipo `bool`).

La dimensione dello spazio degli stati del modello si può stimare in $44^2 \times 2^{79}$ stati ($= 44 \times 2^{34} \times 44 \times 2^{34} \times 2^{10} \times 2$, dobbiamo tenere conto delle due istanze dei processi `ces` e delle relative variabili). Possiamo maggiorare tale cifra con $2^{90} \approx 1 \times 10^{27}$.

Scenario C

Lo scenario C è l'ultimo scenario presentato ed è caratterizzato dall'aggiunta di alcune locazioni e dei vincoli su queste. Le possibili locazioni sono Pisa, Livorno, Lucca e Firenze. Ogni veicolo si guasta ora in una specifica locazione e, similmente, le officine sono contraddistinte da una determinata posizione. Carro attrezzi e noleggio auto devono invece coprire con il loro servizio almeno una zona, ma è possibile che l'area coperta comprenda più posizioni. Questo scenario arricchisce quello B e quindi il resto dei dettagli rimane invariato rispetto a prima. C'è però da notare che le posizioni introdotte inducono dei vincoli sul comportamento del modello: ora una prenotazione ha successo solo se il servizio è disponibile e la locazione dove si trova il veicolo è raggiungibile.

Consideriamo ora nello specifico tutti i servizi e valutiamo brevemente quali sono i vincoli indotti e come si caratterizza la "raggiungibilità" del servizio. Per quanto riguarda la carta di credito, questa non ha vincoli di posizione e quindi è sempre raggiungibile. Diversamente le

altre tre tipologie di servizio prevedono una locazione (o più). Mantenendo l'ordine richiesto nello scenario A, quello che un sistema di emergenza fa e procedere con la richiesta dell'officina. Se la prenotazione ha successo, si passa alle richieste di carro attrezzi e noleggio auto. Questo ordinamento delle richieste influisce su comportamento del modello. Infatti, la prenotazione di un'officina dipende solo dalla fatto che questa sia disponibile o meno e non dalla sua posizione. La raggiungibilità dell'officina dipende dal carro attrezzi che è incaricato di portare il veicolo dal luogo dove è avvenuto il guasto all'officina prenotata. Un servizio di officina deve quindi restituire la propria posizione se risponde positivamente ad una richiesta di prenotazione. Nella successiva richiesta del carro attrezzi, un sistema d'emergenza invia quindi la posizione ottenuta dall'officina e la propria posizione corrente. Il servizio di carro attrezzi risponde con successo solo se entrambe le locazioni sono tra quelle su cui interviene. Allo stesso modo, nella richiesta del noleggio, il sistema d'emergenza indica la posizione del veicolo e quella dell'officina. Il noleggio, se disponibile, risponde con successo se riesce a recapitare un veicolo sostitutivo in almeno una delle due locazioni indicate nella richiesta.

Nel programma che modella questo scenario si è arricchito l'insieme di macro di valori, presentato nel Listato 1, aggiungendo la macro `FARAWAY`. Tale macro, riportata nel Listato 23, a livello semantico precisa che un servizio non riesce a raggiungere la posizione specificata nella richiesta, ovvero che quella locazione non è tra quelle su cui opera. Come per la macro `BUSY`, la costante utilizzata è zero e quindi un sistema di emergenza non riesce a distinguere i due casi e capire di conseguenza perché la sua richiesta sia fallita. La scelta di utilizzare il valore zero però ci permette di continuare ad usare una variabile di tipo `bit` per contenere la risposta di un servizio.

```
5 #define BUSY 0
6 #define FARAWAY 0
7 #define AVAILABLE 1
```

Listato 23: macro per costanti

Nel Listato 24 è riportata la definizione dei nomi simbolici delle locazioni. È stato utilizzato il tipo `mtype` disponibile in Promela che permette di definire appunto dei nomi simbolici per costanti numeriche[15]. Tali nomi sono quindi stati utilizzati nel programma.

```
64 /* tipo locazione */
65 mtype = { Pisa, Livorno, Lucca, Firenze }
```

Listato 24: dichiarazione del tipo mtype

La tabella riportata sotto specifica per ogni nome simbolico la costante numerica che lo rappresenta. Si può notare che il valore `0` non viene utilizzato. Tale valore è stato impiegato nel modello per indicare "nessuna posizione". Si rimanda ai `proctype` dei servizi per l'utilizzo.

Nome simbolico	Costante numerica
Pisa	4
Livorno	3
Lucca	2
Firenze	1

L'aggiunta delle locazione va a modificare il tipo dei canali, dato che ora i sistemi d'emergenza ed i servizi devono comunicare anche le loro posizioni. Ai canali delle officine immettiamo in seconda posizione un campo `mtype` che serve ad un servizio per inviare la propria locazione al car emergency system. Il canale del carro attrezzi è stato modificato e prevede ora quattro

[15]il risultato è funzionalmente equivalente a quello che si potrebbe ottenere utilizzando una sequenza di macro

campi: il secondo ed il terzo sono di tipo `mtype` e servono ad un sistema di emergenza per inviare la propria locazione e quella dell'officina prenotata. Diversamente, per il noleggio auto si è preferito aggiungere un singolo campo di tipo `byte` invece di due di tipo `mtype` per le posizione di auto e officina. Tale campo viene trattato come un array di locazioni: il bit delle locazioni dell'auto e dell'officina è settato ad uno. Per capire a quale bit corrisponde una certa locazione basta fare riferimento alla tabella riportata sopra. Si evidenzia comunque che l'ultimo campo di ogni canale è di tipo `byte` e viene utilizzato per inviare il pid del processo che modella uno dei servizi richiesti nello scenario.

```
68 /* canale carta di credito */
69 chan creditcard_ch = [0] of {bit};
70
71 /* canali officine:
72    il campo mtype serve ad inviare la locazione dell'officina */
73 chan garage_chs [NGARAGE] = [0] of {bit, mtype, byte};
74
75 /* canale carro attrezzi:
76    i canali mype servono per indicare da dove a dove
77    il carro attrezzi deve operare */
78 chan towtruck_ch = [0] of {bit, mtype, mtype, byte};
79
80 /* canale noleggio auto:
81    il secondo campo byte rappresenta un array che
82    contiene la locazione dove è avvenuto il guasto
83    e dove è situata l'officina */
84 chan rentalcar_ch = [0] of {bit, byte, byte};
```

Listato 25: canali dei servizi

Il template del modello sviluppato per questo scenario prevede che siano indicate le posizione delle diverse entità in fase di creazione del processo che le modella. I vari `proctype` prevedono quindi un ulteriore parametro formale che serve ad indicarne la posizione o le posizioni. Sarà compito del processo `init` specificare tali posizioni in fase di istanziazione dei processi. Veicoli e officine hanno una (sola) specifica locazione e quindi il loro parametro formale ha tipo `mtype`. I servizi di carro attrezzi e noleggio auto invece possono operare su più province e quindi un parametro formale di tipo `mtype` non basta. Si potrebbe utilizzare un array di `bool` per indicare le locazioni dove il carro attrezzi può intervenire ed allo stesso modo un array di `bool` per le posizioni dove il noleggio può consegnare una macchina. In Promela c'è però la limitazione che un array non può essere passato come argomento ad un `proctype` e quindi deve essere definito localmente oppure globalmente. In entrambi questi casi non è facile avere una descrizione generica del servizio da istanziare con i parametri desiderati[16]. Inoltre, come già detto nella descrizione dello scenario A, gli array di `bool` sono memorizzati come array di `byte`. Nell'implementazione si è quindi scelto di adottare la stessa strategia adottata nel Listato 5, ovvero di utilizzare un parametro formale di tipo `byte` come array di otto bit. Dato che le locazioni possibili sono quattro, considerando anche il fatto che il valore zero non è utilizzato, una variabile di tipo `byte` è sufficiente per mantenere l'informazione di quali province sono coperte.

Per testare se una locazione è tra quelle servite, precisate con una variabile di tipo `byte`, oppure se tra un insieme di locazioni ce ne è almeno una raggiungibile sono state definite le macro `validLoc` e `oneValidLoc`, riportate nel Listato 26. La macro `toByteLoc` serve invece per ottenere un valore di tipo `byte` a partire da due locazioni (ovvero valori di tipo `mtype`).

[16] si potrebbe per esempio differenziare in base al pid, ma ciò complica la scrittura del `proctype`

```
52 /* macro per creare una variabile byte in cui sono ad uno
53    i bit delle locazioni passate come argomento */
54 #define toByteLoc(l1, l2) ((1 << (l1)) | (1 << (l2)))
55 /* macro per testare se una locazione è coperta */
56 #define validLoc(z, l)    (z >> (l) & 1)
57 #define oneValidLoc(z, l) ((z & l) != 0)
```

Listato 26: macro per le locazioni

Passiamo ora a descrivere i vari processi che modellano le entità dello scenario. Come già detto, il servizio di carta di credito non è soggetto a vincoli di locazioni e quindi il `proctype` definito nel Listato 7 non subisce cambiamenti. I restanti servizi prevedono però uno o più locazioni e quindi necessitano di aggiustamenti. Consideriamo il `proctype` definito nel Listato 27 utilizzato per creare un processo che modella un'officina. Tale definizione prevede due parametri: il canale utilizzato dal processo per comunicare e la specifica posizione dell'officina. Il comportamento è ciclico e si limita ad attendere una richiesta, rispondere positivamente (inviando anche la propria posizione ed il proprio pid insieme a tale risposta) ed attendere il rilascio della prenotazione. Da notare che la conferma di rilascio, da parte del servizio, invia la posizione `0`, ovvero non specifica nessuna locazione[17].

```
248 /* servizio officina, inizialmente disponibile
249    con specifica locazione */
250 proctype garage(chan ch; mtype loc)
251 {
252 start:  skip;
253 waitG:  ch ? REQUEST, _, _;
254     ch ! AVAILABLE, loc, _pid;
255     /* il messaggio di release deve avere la locazione ed il pid
           del servizio */
256 bookG:  ch ? RELEASE, eval(loc), eval(_pid);
257     ch ! AVAILABLE, 0, 0;
258     goto waitG
259 }
```

Listato 27: proctype dell'officina

Il comportamento del processo che modella il carro attrezzi è leggermente più complicato. Nel Listato 28 è riportata la definizione del `proctype` per questo servizio. Anche in questo caso i parametri formali attesi sono due: il canale per la comunicazione e le posizioni su cui il carro attrezzi interviene (specificate come `byte`). Localmente al processo sono definite altre due variabili, di tipo `mtype`, che contengono l'informazione di dove è avvenuto il guasto e della posizione dell'officina. Rispettivamente la variabile `from` e la variabile `to`. Il processo inizialmente attende una richiesta, copiando i valori inviati sul canale nelle proprie variabili locali, e successivamente risponde in base alle locazioni specificate. Se sono entrambe tra quelle dove il servizio interviene, allora la richiesta ha successo e procede all'attesa del rilascio. Altrimenti risponde negativamente. In ogni caso, dopo l'invio dell'ack del rilascio o della risposta negativa, il processo ritorna nella fase di attesa di una richiesta. Il controllo se entrambe le locazioni sono raggiungibili viene fatto con la macro `validLoc`, precedentemente introdotta.

```
262 /* servizio carro attrezzi, inizialmente disponibile
263    che può operare in una o più locazioni */
264 proctype towtruck(chan ch; byte locs)
265 {
```

[17] l'istruzione `skip` etichettata con `start` è stata aggiunta per esprimere una proprietà sulle locazioni

```
266       /* indica le locazioni della richiesta */
267       mtype from = 0, to = 0;
268
269   waitTT: ch ? REQUEST, from, to, _;
270       if
271       /* se le locazioni sono entrambe tra quelle su cui
272          il servizio opera, allora è prenotabile  */
273       :: (validLoc(locs, from) && validLoc(locs, to)) ->
274         ch ! AVAILABLE, from, to, _pid;
275         /* il messaggio di release deve avere le locazioni
               specificate nella richiesta ed il pid del servizio */
276   bookTT:    ch ? RELEASE, eval(from), eval(to), eval(_pid);
277         ch ! AVAILABLE, 0, 0, 0
278       /* altrimenti la prenotazione è fallita */
279       :: else ->
280         ch ! FARAWAY, 0, 0, 0
281       fi;
282       goto waitTT
283   }
```

Listato 28: proctype del carro attrezzi

È necessario infine modificare il `proctype` che modella il noleggio auto per gestire le locazioni. La nuova definizione di questo processo è riportata nel Listato 29. Anche in questo caso si aggiunge un secondo parametro alla dichiarazione. Parametro che indica le province dove il servizio può consegnare una macchina. Localmente al processo viene dichiarata la variabile `where`, di tipo `byte`, dove viene copiata l'informazione inviata sul canale da un sistema di emergenza. Tale informazione è l'array che indica la posizione del veicolo e quella dell'officina. Il comportamento del processo continua a prevedere l'iniziale scelta nondeterministica dello stato del servizio. Una volta che questa scelta è stata fatta, il processo attende una richiesta ed in relazione allo stato risponde. Se il servizio è occupato la richiesta fallisce, altrimenti si risponde in base alle posizioni specificate nella richiesta. Se almeno una delle due locazioni è tra quelle su cui il servizio può recapitare una macchina la prenotazione ha successo ed in caso contrario fallisce. La gestione della prenotazione, da parte del servizio, avviene nei soliti passi: attesa del rilascio e conferma. In tutti questi casi appena descritti, l'istruzione successiva è il salto incondizionato allo stato di attesa della richiesta successiva.

```
286   /* servizio noleggio auto
287      sceglie nondeterministicamente se è disponibile od occupato */
288   proctype rentalcar(chan ch; byte locs)
289   {
290       /* indica se il servizio è disponibile o meno */
291       bool state;
292       /* indica le locazioni della richiesta:
293          dove si trova il veicolo e dove si trova l'offina */
294       byte where = 0;
295
296       /* scelta nonderminisica dello stato */
297       if
298       :: state = false
299       :: state = true
300       fi;
301
```

```
302 waitRC: ch ? REQUEST, where, _;
303     if
304     /* se è disponibile allora potrebbe essere prenotato */
305     :: state ->
306       if
307       /* se la locazione è una di quelle su cui
308            il servizio opera, allora è prenotabile  */
309       :: oneValidLoc(locs, where) ->
310         ch ! AVAILABLE, where, _pid;
311         /* il messaggio di release deve avere la locazione ed il
               pid del servizio */
312 bookRC:      ch ? RELEASE, eval(where), eval(_pid);
313         ch ! AVAILABLE, 0, 0
314       /* altrimenti la prenotazione è fallita */
315       :: else ->
316         ch ! FARAWAY, 0, 0
317       fi
318     /* altrimenti è occupato */
319     :: !state -> ch ! BUSY, 0, 0
320     fi;
321     goto waitRC
322 }
```

Listato 29: proctype del noleggio auto

Il `proctype` che descrive il processo che modella un sistema di emergenza, riportato nel Listato 30, resta strutturalmente uguale. Quello che cambia è la dichiarazione, che prevede ora di specificare come parametro la posizione del veicolo, e l'utilizzo dei canali per comunicare con i servizi officina, carro attrezzi e noleggio auto. Per mantenere l'informazione della posizione, che un'officina potrebbe inviare in caso di prenotazione avvenuta con successo, viene dichiarata localmente al processo la variabile `loc_garage`. Tale variabile viene utilizzata nell'istruzione di receive durante la richiesta dell'officina (riga 156), ovvero viene copiato al suo interno il valore presente nel canale. Questo valore viene poi inviato (se il sistema riesce a prenotare un'officina) durante la richiesta del carro attrezzi sul terzo campo del canale. Come secondo campo viene invece inviata la posizione del veicolo. La posizione dell'officina serve anche per la richiesta del noleggio auto. Grazie alla macro `toByteLoc` si riesce ad inviare al processo che modella quest'ultimo servizio il valore corrispondente all'array in cui le posizioni dell'auto e dell'officina sono settate ad uno. In maniera analoga le posizioni note al sistema vengono inviate nella fase di rilascio delle prenotazioni.

```
109 /* car emergency system */
110 proctype ces(mtype loc)
111 {
112     /* ha un abbonamento? */
113     bool subscribing;
114     bool result;
115     /* pid dei servizi prenotati:
116        se pid == 0 allora il servizio non è prenotato
117        altrimenti il pid è usato come ricevuta di prenotazione */
118     byte pid_garage = 0, pid_towtruck = 0, pid_rentalcar = 0;
119     /* indice del canale dei servizi prenotati (per servizi con
           più istanze) */
120     byte idg = 0;
```

```
121     /* indica la posizione dell'officina prenotata */
122     mtype loc_garage = 0;
123
124     /* scelta non deterministica se è abbonato o meno */
125 start:  if
126     :: subscribing = false
127     :: subscribing = true
128     fi;
129
130 damage: if
131     /* è abbonato */
132     :: subscribing ->
133 sub:    skip
134     /* non è abbonato, richiedo l'autorizzazione per la
135          prenotazione con carta di credito */
136     :: !subscribing ->
137       creditcard_ch ! REQUEST;
138       creditcard_ch ? result;
139       if
140       /* autorizzazione ottenuta */
141       :: result ->
142 auth:    skip
143       /* autorizzazione negata */
144       :: !result -> goto fail
145       fi
146     fi;
147
148     /* tentativo di prenotazione dell'officina */
149 reqG: for(idg in garage_chs) {
150       if
151       /* se la idg-esima officina non è prenotata da un altro ces
152          ottengo il canala di comunicazione in mutua esclusione
               */
153       :: atomic { notBooked(garage_booked, idg) ->
154         Book(garage_booked, idg) };
155         garage_chs[idg] ! REQUEST, 0, 0;
156         garage_chs[idg] ? result, loc_garage, pid_garage;
157         if
158         /* officina prenotata */
159         :: result -> goto reqTT
160         /* officina occupata o locazione diversa, la rilascio e
               continuo ad iterare */
161         :: !result -> Cancel(garage_booked, idg)
162         fi
163       /* altrimenti continuo ad iterare */
164       :: else -> skip
165       fi
166     }
167     loc_garage = 0;
168     /* se viene raggiunto questo punto allora tutte le
          prenotazioni sono fallite */
```

```
169     goto fail;
170
171     /* tentativo di prenotazione del carro attrezzi */
172 reqTT:  if
173     :: atomic { !towtruck_booked ->
174       towtruck_booked = true };
175       towtruck_ch ! REQUEST, loc, loc_garage, 0;
176       towtruck_ch ? result, _, _, pid_towtruck;
177       if
178       /* carro attrezzi prenotato */
179       :: result -> skip
180       /* carro attrezzi occupato o locazione non coperta,
             prenotazione fallita */
181       :: !result -> towtruck_booked = false;
182         goto fail
183       fi
184     /* se il servizio è già prenotato, la richiesta è fallita */
185     :: else -> goto fail
186     fi;
187
188     /* tentativo di prenotazione del noleggio auto */
189 reqRC:  if
190     :: atomic { !rentalcar_booked ->
191       rentalcar_booked = true };
192       rentalcar_ch ! REQUEST, toByteLoc(loc_garage, loc), 0;
193       rentalcar_ch ? result, _, pid_rentalcar;
194       if
195       :: result -> skip
196       :: !result -> rentalcar_booked = false
197       fi
198       /* sia in caso di fallimento della prenotazione, sia in
             caso di successo
199           procedo con il workflow
200           nota: se result == true allora pid_rentalcar != 0,
                 ovvero vale
201           assert(!result || (pid_rentalcar != 0)); */
202       /* se il servizio è già prenotato, continuo con il workflow
           */
203       :: else -> skip
204       fi;
205
206       /* workflow è terminato con successo */
207 succ: goto rel;
208
209 fail: skip;
210     /* release dei servizi prenotati */
211 rel:  if
212     :: pid_garage != 0 ->
213       garage_chs[idg] ! RELEASE, loc_garage, pid_garage;
214       garage_chs[idg] ? AVAILABLE, _, pid_garage;
215       Cancel(garage_booked, idg);
```

```
216         if
217         :: pid_towtruck != 0 ->
218           towtruck_ch ! RELEASE, loc, loc_garage, pid_towtruck;
219           towtruck_ch ? AVAILABLE, _, _, pid_towtruck;
220           towtruck_booked = false
221         :: else -> skip
222         fi;
223         if
224         :: pid_rentalcar != 0 ->
225           rentalcar_ch ! RELEASE, toByteLoc(loc_garage, loc),
                pid_rentalcar;
226           rentalcar_ch ? AVAILABLE, _, pid_rentalcar;
227           rentalcar_booked = false
228         :: else -> skip
229         fi;
230         loc_garage = 0
231       :: else -> skip
232       fi;
233       goto damage
234   }
```

Listato 30: proctype del sistema di emergenza

L'`init` di questo scenario è riportato nel Listato 31 e risulta più complesso dei precedenti. Il suo scopo finale resta quello di istanziare tutti i processi che modellano un'entità dello scenario ma l'aggiunta delle locazioni produce diverse configurazioni possibili che vorremmo essere in grado di specificare. Come già detto, tutti i `proctype`, ad esclusione di quello della carta di credito, prevedono ora un parametro formale aggiuntivo per indicarne le posizioni. Localmente al processo sono dunque dichiarate la variabile `city` e la variabile `zone`, rispettivamente di tipo `mtype` e di tipo `byte`. La prima serve ad indicare una delle 4 possibili locazioni. La seconda serve invece a specificare una delle 15 possibili combinazioni delle quattro città (la combinazione vuota non è accettata). Quello che vorremmo è di far variare queste due variabili su tutti i valori interessanti per lo scenario e istanziare i diversi processi con tali valori.

L'esplosione combinatoria che si ha, se si considerano tutte le possibili configurazioni delle locazioni delle sei entità dello scenario, è apprezzabile e può risultare un problema ai fini della verifica. Si è quindi scelto di utilizzare sei macro per indicare quali processi istanziare con tutte le locazioni possibili: `CES1`, `CES2`, `GARAGE1`, `GARAGE2`, `TOWTRUCK` e `RENTALCAR`. Se una di queste macro non è definita si utilizzano dei valori scelti arbitrariamente. Per il carro attrezzi è stato scelto il valore 28 e per il noleggio auto il valore 20[18].

```
325 /* inizializzazione */
326 init {
327   /* variabile che contiene una delle posizioni dello scenario */
328   mtype city;
329
330   /* variabile che contiene una delle possibili combinazioni
        delle posizioni dello scenario;
331       la variabile è di tipo byte ed è usata come array di
            posizioni:
332           indica quali posizioni sono servite
```

[18] 28 indica le province di Pisa, Livorno e Lucca (dato che è la somma di $2^{\text{Pisa}} = 2^4 = 16$, $2^{\text{Livorno}} = 2^3 = 8$ e $2^{\text{Lucca}} = 2^2 = 4$) mentre 20 denota Pisa e Lucca

```
333       NOTA: deve essere diversa da zero perché vogliamo che almeno
              una posiziona sia servita
334       traduzione locazione/numero -> valore intero
335       Pisa = 4     -> 16
336       Livorno = 3   -> 8
337       Lucca = 2    -> 4
338       Firenze = 1   -> 2
339        Se l'i-esimo elemento dell'array vale 1, la relativa
              locazione è servita;
340          per indicare più locazione basta farne la somma dei valori
              interi oppure operare bitwise */
341     byte zone;
342
343     atomic {
344 #ifdef CES1
345       /* scelta non-deterministica della locazione del primo ces */
346       select(city : 1 .. 4);
347       ces_pid1 = run ces(city);
348 #else
349       ces_pid1 = run ces(Pisa);
350 #endif
351 #ifdef CES2
352       /* scelta non-deterministica della locazione del secondo ces
              */
353       select(city : 1 .. 4);
354       ces_pid2 = run ces(city);
355 #else
356       ces_pid2 = run ces(Livorno);
357 #endif
358       run creditcard(creditcard_ch);
359 #ifdef GARAGE1
360       /* scelta non-deterministica della locazione della prima
              officina */
361       select(city : 1 .. 4);
362       g_pid1 = run garage(garage_chs[0], city);
363 #else
364       g_pid1 = run garage(garage_chs[0], Pisa);
365 #endif
366
367 #ifdef GARAGE2
368       /* scelta non-deterministica della locazione della seconda
              officina */
369       select(city : 1 .. 4);
370       g_pid2 = run garage(garage_chs[1], city);
371 #else
372       g_pid2 = run garage(garage_chs[1], Livorno);
373 #endif
374
375 /* se una delle due macro è definita testo una sola
       configurazione per evitare
376       l'esplosione combinatoria */
```

```
377 #ifdef TOWTRUCK
378       /* scelta non-deterministica delle locazioni servite dal
              carro attrezzi */
379       zone = 2;
380       do
381       :: zone < 30 -> zone = zone + 2
382       :: break
383       od;
384       tt_pid = run towtruck(towtruck_ch, zone);
385 #else
386       /* 28 -> Pisa, Livorno e Lucca */
387       tt_pid = run towtruck(towtruck_ch, 28);
388 #endif
389
390 #ifdef RENTALCAR
391       /* scelta non-deterministica delle locazioni servite dal
              noleggio auto */
392       zone = 2;
393       do
394       :: zone < 30 -> zone = zone + 2
395       :: break
396       od;
397       rc_pid = run rentalcar(rentalcar_ch, zone)
398 #else
399       /* 20 -> Pisa e Lucca */
400       rc_pid = run rentalcar(rentalcar_ch, 20)
401 #endif
402    }
403 }
```

Listato 31: init del programma

Le configurazioni possibili che si devono verificare se tutte le macro sono definite sono 57600 (ovvero $4^4 \times 15^2$). Tale numero non è eccessivamente grande ma, dato che SPIN memorizza tutti gli stati che incontra durante la verifica di una proprietà, la memoria disponibile su un normale computer non risulta sufficiente per completare la verifica[19].

Versione "relabel"

Lo scenario C prevede 57600 configurazioni possibili e per ognuna di queste deve essere verificata la proprietà di interesse. Tale numero non risulta proibitivo ma vorremmo comunque ridurlo per limitare l'uso di memoria e tempo durante la verifica. Un'idea che risponde a questa esigenza è quella di considerare le configurazioni a meno di rietichettatura[20].

Lo scenario prevede quattro locazioni distinte e l'unica operazione che richiediamo su queste è l'identità, ovvero vogliamo capire quando due locazioni sono la stessa e quando non lo sono. In fase di modellazione abbiamo fatto un'astrazione sostituendo ai nomi delle locazioni un numero: ciò viene fatto nella definizione di `mtype` (riportate nel Listato 24). Il numero che corrisponde ad una locazione, come già indicato, dipende solo dall'ordine che abbiamo usato nella definizione di `mtype`. Possiamo notare poi che le posizioni non sono caratterizzate da una distanza che intercorre tra di loro e ciò ci permette di scambiarle come vogliamo. Pertanto, precisato che la sequenza di nomi di locazioni utilizzata nella definizione è stata del tutto arbitraria, possiamo

[19] bisogna quindi ricorrere alle opzioni che SPIN mette a disposizione per ridurre la memoria
[20] da qui il nome "relabel"

scambiare l'ordine dei nomi senza introdurre errori nella verifica. Eventuali errori che riguardano le locazioni, rilevati durante la verifica, dipendono solo dal fatto che due numeri sono diversi. Data la sequenza Pisa, Livorno, Lucca e Firenze supponiamo che il model checker trovi un errore quando entrambe le auto si trovano a Pisa ed entrambe le officine si trovano a Livorno. Lo stesso errore si sarebbe trovato se la sequenza fosse stata Firenze, Pisa, Livorno e Lucca ma in questo caso le auto sarebbero state a Firenze e le officine a Pisa. Abbiamo quindi fatto corrispondere diversamente le locazioni ai numeri ma l'errore viene comunque trovato perché non dipende dall'ordine, bensì dai numeri che indicano le locazioni. Possiamo quindi limitarci a considerare i numeri che rappresentano le locazioni e legare tali numeri alle posizioni a posteriori[21].

Quanto detto fino ad ora non riduce però il numero di configurazione da provare. L'osservazione che ci permette di ridurre i casi è quella che riguarda le configurazioni: esistono configurazioni che si possono ottenere da altre etichettando diversamente i numeri. Per esempio le configurazioni (1,1,1,1) e (4,4,4,4) rappresentano lo stesso scenario, ovvero quello dove i veicoli e le officine hanno la stessa locazione. Possiamo facilmente ottenere la prima dalla seconda sostituendo 4 ad 1 (e viceversa) e per entrambe possiamo etichettare il numero con una delle quattro possibili locazioni[22]. Ricordando che ciò che interessa è che la locazione sia o meno la stessa, possiamo considerare solo una delle due configurazioni ed ignorare l'altra. Questa considerazione vale anche per altre configurazioni è ciò ci permette di ridurre i casi da considerare.

L'idea è quindi di fissare la posizione del primo sistema d'emergenza e far variare la locazione dell'altro sistema d'emergenza e delle officine[23]. Il primo veicolo ha posizione 1 mentre il secondo può avere la stessa locazione (quindi 1) oppure averne un'altra diversa che indichiamo con 2. Analogamente la prima officina ha la stessa locazione di uno dei due veicoli (quindi 1 o 2) oppure un'altra locazione indicata con 3. Infine la seconda officina ha la stessa locazione di uno dei due veicoli o della prima officina (quindi 1, 2 o 3) oppure l'ultima locazione prevista, specificata dal 4. Questo ragionamento ci permette di esprimere tutte le configurazioni realmente interessanti e di passare da 256 configurazioni per sistemi d'emergenza e officine a solo 15 combinazioni riportate nelle seguente tabella.

ces1	ces2	garage1	garage2
1	1	1	1
1	1	1	2
1	1	2	1
1	1	2	2
1	1	2	3
1	2	1	1
1	2	1	2
1	2	1	3
1	2	2	1
1	2	2	2
1	2	2	3
1	2	3	1
1	2	3	2
1	2	3	3
1	2	3	4

Si ha quindi una riduzione di circa 17 volte dello spazio di ricerca e ciò permette di eseguire la verifica più velocemente e utilizzando meno memoria. Il processo `init` che realizza quanto

[21] questa operazione è concettuale: nel sorgente dobbiamo comunque indicare una definizione per il tipo `mtype`

[22] l'operazione di rietichettatura può essere pensata sia a livello di numeri che a livello di nomi

[23] per semplicità si considerano tutte le combinazioni di locazioni per il carro attrezzi ed il noleggio auto (ovvero 225, 15 per ogni servizio)

descritto è riportato nel Listato 32 e rappresenta l'unica differenza tra il primo sorgente presentato e quello della versione "relabel"[24]. Per ottenere le combinazioni interessanti si sono utilizzate quattro variabili locali (`city0`, `city1`, `city2`, `city3`) ed il costrutto `select` dove l'estremo superiore dell'intervallo è il massimo tra i valori contenuti nelle variabili già fissate a cui viene aggiunto uno.

```
325 /* inizializzazione */
326 init {
327   /* variabili che contengono una delle posizioni dello scenario
        */
328   mtype city0, city1, city2, city3;
329
330   /* variabile che contiene una delle possibili combinazioni
        delle posizioni dello scenario;
331      la variabile è di tipo byte ed è usata come array di
           posizioni:
332           indica quali posizioni sono servite
333      NOTA: deve essere diversa da zero perché vogliamo che almeno
           una posiziona sia servita
334     traduzione locazione/numero -> valore intero
335     Pisa = 4     -> 16
336     Livorno = 3   -> 8
337     Lucca = 2    -> 4
338     Firenze = 1   -> 2
339      Se l'i-esimo elemento dell'array vale 1, la relativa
           locazione è servita;
340        per indicare più locazione basta farne la somma dei valori
             interi oppure operare bitwise */
341   byte zone;
342
343   atomic {
344     city0 = 1;
345     ces_pid1 = run ces(city0);
346
347     /* scelta non-deterministica della locazione del secondo ces
          */
348     select(city1 : 1 .. (city0 + 1));
349     ces_pid2 = run ces(city1);
350
351     run creditcard(creditcard_ch);
352
353     /* scelta non-deterministica della locazione della prima
          officina */
354     select(city2 : 1 .. (city1 + 1));
355     g_pid1 = run garage(garage_chs[0], city2);
356
357     /* scelta non-deterministica della locazione della seconda
          officina */
358     select(city3 : 1 .. (city1 > city2 -> city1 + 1 : city2 + 1))
          ;
```

[24] chiamato `CarEmCrelabel.pml`

```
359         g_pid2 = run garage(garage_chs[1], city3);
360
361         /* scelta non-deterministica delle locazioni servite dal
                carro attrezzi */
362         zone = 2;
363         do
364         :: zone < 30 -> zone = zone + 2
365         :: break
366         od;
367         tt_pid = run towtruck(towtruck_ch, zone);
368
369         /* scelta non-deterministica delle locazioni servite dal
                noleggio auto */
370         zone = 2;
371         do
372         :: zone < 30 -> zone = zone + 2
373         :: break
374         od;
375         rc_pid = run rentalcar(rentalcar_ch, zone)
376     }
377 }
```

Listato 32: init del programma

Proprietà

Per lo scenario C si devono verificare le stesse proprietà dello scenario B e proporne poi delle altre. Tutte le proprietà sono state provate per la versione "relabel" del modello.

d) "Ogni servizio è prenotato da un utente alla volta al massimo."

La formulazione presentata per lo scenario B è valida anche per lo scenario C. Tale scenario è stato infatti ottenuto a partire dal precedente, modificando ed estendendo il programma per gestire le locazioni. Le variabili utilizzate nella formulazione LTL della proprietà non sono state modificate o eliminate.

Anche per quanto riguarda la verifica il risultato è lo stesso, ovvero il modello verifica la proprietà.

e) "Ogni auto si guasta infinite volte."

La formulazione LTL è quella presentata precedentemente nello scenario B. Il programma che descrive lo scenario C continua ad utilizzare l'etichetta `damage` per indicare lo stato in cui si controlla se l'utente è abbonato o meno.

La verifica della proprietà fallisce perché uno dei due processi `ces` che modellano il sistema d'emergenza può rimane fermo in uno stato e non procedere nell'esecuzione delle istruzioni. Tale computazione contraddice la proprietà.

f) "Assumendo weak fairness a livello di processi, ogni auto si guasta infinite volte."

Come per lo scenario B, anche in questo caso dobbiamo eseguire la verifica con l'apposita opzione. Rimandiamo alla medesima proprietà dello scenario precedente per i dettagli.

Nonostante l'aggiunta delle locazioni e la modifica dei vari `proctype` che modellano i servizi per gestire tali locazioni, il modello verifica la proprietà. Quindi, pure nello scenario C, la presenza delle variabili che indicano i servizi già prenotati permette di evitare che un servizio gestisca eventuali richieste. Può quindi aspettare il rilascio della prenotazione da parte del sistema d'emergenza che la detiene. Di conseguenza tutte le azioni che compie un processo `ces` sono (da un certo punto in poi) sempre eseguibili e quindi vengono sempre eseguite (prima o poi). Tra queste istruzioni c'è anche quella etichettata come `damage` che pertanto viene eseguita infinite volte.

g) "Caratterizzazione di un "progress cycle" nel modello."

Nel modello sviluppato sono stati formalizzati gli stessi "progress cycle" già discussi per lo scenario B. Un "progress cycle" etichetta gli stati in cui un processo, che modella un car emergency system, ha ricevuto un messaggio di successo come risposta alla propria richiesta ad un servizio. Vengono utilizzate tre etichette, una per ogni servizio necessario al completamento del workflow: carta di credito, officina e carro attrezzi. L'altro "progress cycle" invece etichetta il solo stato in cui un sistema d'emergenza ottiene il carro attrezzi, dato che lo scenario prevede un'unica istanza di questo servizio. Come fatto in precedenza, si è creato un apposito file[25] per la verifica di "non-progress cycles" ed i due cicli formalizzati si ottengono definendo o meno la macro `ALL`. Il programma PROMELA che viene proposto prevede due `proctype` strutturalmente uguali. L'unica differenza è che il primo non presenta etichette mentre il secondo si. Anche in questo caso la scelta è stata fatta per gli stessi motivi già esposti per lo scenario B, ovvero evitare che la computazione in cui il processo `ces` (quello non etichettato) procede nella sua esecuzione mentre quello `cesP` (etichettato) rimane fermo all'infinito in nello stesso stato. Nel seguito non si riporta il codice del programma, si rimanda direttamente al file per i dettagli.

La verifica di "non-progress cycles" produce gli stessi risultati dello scenario B, ma dei commenti a riguardo sono necessari. La verifica eseguita senza assunzioni di fairness fallisce per entrambi i cicli proposti perché la computazione prima descritta è possibile (e viene riportata come controesempio). Assumendo weak fairness a livello di processi la verifica del primo ciclo ha successo. Come per lo scenario precedente infatti lo stato etichettato `progressCC` viene visitato infinite volte. Non vale altrettanto per il secondo ciclo, quello con un unico stato di progresso (etichettato con `progressTT`). La verifica fatta assumendo weak fairness fallisce perché il processo `ces` può ottenere sempre il carro attrezzi. Questo controesempio viene riportato se il carro attrezzi riesce a raggiungere entrambi i veicoli e le relative officine. L'aggiunta delle locazioni produce però altri controesempi per il secondo "progress cycle" formalizzato. Se di fatto il processo `cesP` si trova in una provincia dove il carro attrezzi non interviene il controesempio prodotto è diverso. In questo caso il processo viene fatto avanzare, ma per i vincoli indotti dalle locazioni la sua richiesta fallisce sempre[26]. Un controesempio simile viene prodotto anche se non si assume weak fairness, ma in questo caso il processo `ces` rimane fermo nel suo stato iniziale.

h) "Nessuno dei workflow ha successo se tutte le officina hanno una locazione dove il carro attrezzi non interviene."

Questa proprietà, nel caso in cui nessuna delle officine sia raggiungibile dal carro attrezzi, descrive il comportamento atteso del modello: ovvero che nessun workflow abbia successo. La computazione in cui nessuna officina è servita dal carro attrezzi ma il workflow di

[25] `CarEmCprogress.pml`

[26] questo è esattamente il modello descritto nel file `CarEmCprogress.pml`

almeno un sistema d'emergenza ha successo rappresenta quindi un comportamento indesiderato che il modello non deve avere. Tale proprietà è quindi di safety. La formulazione LTL che presentiamo utilizza le variabili locali, ai processi che modellano le officine, che indicano la posizione del servizio. Nello stato iniziale della verifica però l'unico processo attivo è l'`init` e quindi non sono definite le variabili che ci interessano. Anche per questa proprietà utilizziamo il pattern "absence after Q" per verificare una condizione solo dopo che una premessa si è verificata. La premessa è che entrambi i processi officina nello stato iniziale (etichettato come `start`) siano in una posizione non raggiungibile dal carro attrezzi (per far ciò utilizziamo la variabile locale del servizio carro attrezzi che indica quali sono le locazioni dove interviene. Tale variabile sarà già inizializzata perché la fase di creazione dei processi è atomica). Dato che le locazioni dove interviene il carro attrezzi sono memorizzate in una variabile di tipo `byte` dobbiamo operare bitwise (utilizzando right shift e AND bitwise). La condizione che si deve verificare quando la premessa è vera asserisce che non succede mai che un processo che modella un sistema d'emergenza raggiunga lo stato etichettato `succ`.

$$\Box\left(\left(\begin{array}{c}\text{garage[g\_pid1]@start} \wedge \text{garage[g\_pid2]@start} \\ \wedge \\ \neg(\text{towtruck:locs} >> (\text{garage[g\_pid1]:loc}) \,\&1) \\ \wedge \\ \neg(\text{towtruck:locs} >> (\text{garage[g\_pid2]:loc}) \,\&1)\end{array}\right) \longrightarrow \Box\neg(\text{ces[ces\_pid1]@succ} \vee \text{ces[ces\_pid2]@succ})\right)$$

Il modello verifica la proprietà. Nel caso la premessa sia falsa la verifica non procede ulteriormente provando la conseguenza, dato che la proprietà è espressa come implicazione e quindi la formula risulta vera. Diversamente, se la premessa vale, la verifica procede in profondità. La proprietà risulta comunque dimostrata perché la guardia alla riga 273 (del Listato 28) non è mai eseguibile se nessuna delle officine è raggiungibile, in quanto la macro `validLoc(locs, to)` restituisce sempre zero. Viene quindi sempre eseguita la guardia `else` e il processo che modella il carro attrezzi invia un messaggio di prenotazione fallita. Di conseguenza il sistema d'emergenza che riceve tale messaggio passa nello stato di fallimento, dal quale non può raggiungere quello di successo.

i) "La prenotazione del carro attrezzi fallisce se la posizione del veicolo non è tra quelle dove il servizio interviene."

La proprietà descrive il comportamento del modello nel caso in cui un veicolo non sia raggiungibile dal carro attrezzi. Il sistema d'emergenza montato su tale veicolo non può ottenere la prenotazione del servizio perché altrimenti violerebbe i vincoli indotti dalle locazioni. Anche in questo caso la proprietà è di safety ed esprime il fatto che un comportamento non atteso del modello non si verificherà mai. Per la formulazione è utile usare la variabile locale `pid_towtruck` di un processo `ces`. Se il valore contenuto in tale variabile è diverso dal pid del processo che modella il carro attrezzi, tale servizio non è prenotato dallo specifico processo `ces`. Durante la verifica dobbiamo però scartare i primi passi, quelli in cui i processi `ces` non sono stati ancora creati. Utilizziamo quindi il pattern "absence after Q" per verificare la proprietà del singolo processo `ces` solo dopo che questo ha raggiunto l'etichetta `start`. La formulazione LTL, che prevede una sotto-formula per ogni processo che modella un sistema d'emergenza, è la seguente.

$$\Box\left(\left(\begin{array}{c}\text{ces[ces\_pid1]@start}\\ \wedge\\ \neg(\text{towtruck:locs} >> (\text{ces[ces\_pid1]:loc})\,\&1)\end{array}\right) \longrightarrow \Box(\text{ces[ces\_pid1]:pid\_towtruck} \mathrel{!}= \text{tt\_pid})\right)$$
$$\wedge$$
$$\Box\left(\left(\begin{array}{c}\text{ces[ces\_pid2]@start}\\ \wedge\\ \neg(\text{towtruck:locs} >> (\text{ces[ces\_pid2]:loc})\,\&1)\end{array}\right) \longrightarrow \Box(\text{ces[ces\_pid2]:pid\_towtruck} \mathrel{!}= \text{tt\_pid})\right)$$

Il modella verifica la proprietà. Le sotto-formule sono in congiunzione perché la richiesta riguarda ogni singolo sistema d'emergenza. Considerando solo una di queste, possiamo dire che l'implicazione evita di testare la conseguenza quando il veicolo si trova in una locazione raggiungibile dal carro attrezzi. Quando invece non è raggiungibile il model checker verifica che quel sistema d'emergenza non ottenga mai la prenotazione. In questo caso la verifica ha successo perché il sistema d'emergenza non raggiunto dal carro attrezzi riceverà un messaggio di prenotazione fallita, dato che la guardia alla riga 273 non è eseguibile (la macro `validLoc(locs, from)` restituisce zero per la specifica locazione ricevuta nella richiesta). Il sistema d'emergenza passerà quindi nello stato di fallimento.

j) "Il workflow non avrà mai successo se il veicolo non è in una provincia dove il carro attrezzi può intervenire."

La proprietà esprime un comportamento che non si deve mai verificare nel modello. Di conseguenza la proprietà è di safety. Inoltre si può notare che la premessa è la stessa della proprietà **i**, ovvero il veicolo non è raggiungibile dal carro attrezzi. Anche per questa proprietà facciamo uso delle variabili locali che indicano le locazioni delle diverse entità dello scenario. La formulazione LTL è la seguente.

$$\Box\left(\left(\begin{array}{c}\text{ces[ces\_pid1]@start}\\ \wedge\\ \neg(\text{towtruck:locs} >> (\text{ces[ces\_pid1]:loc})\,\&1)\end{array}\right) \longrightarrow \Box\neg(\text{ces[ces\_pid1]@succ})\right)$$
$$\wedge$$
$$\Box\left(\left(\begin{array}{c}\text{ces[ces\_pid2]@start}\\ \wedge\\ \neg(\text{towtruck:locs} >> (\text{ces[ces\_pid2]:loc})\,\&1)\end{array}\right) \longrightarrow \Box\neg(\text{ces[ces\_pid2]@succ})\right)$$

Il modello verifica questa proprietà. In questo caso però, invece di descrivere il comportamento che motiva l'esito della dimostrazione, siamo interessati a legare questa proprietà alla precedente. Infatti la proprietà **i** implica questa proprietà, dato che se la prenotazione del carro attrezzi fallisce allora il workflow fallisce (e quindi non ha successo). Il modello verifica la proprietà **i** e di conseguenza verifica anche questa.

k) "La prenotazione del carro attrezzi ha successo solo se questo può trasferire il veicolo dal luogo deve è avvenuto il guasto all'officina di riparazione."

Questa proprietà rappresenta un vincolo indotto dalle locazioni. È infatti necessario che sia la posizione del veicolo sia quella dell'officina siano tra quelle dove il carro attrezzi può intervenire. Se questo si verifica, il servizio può portare il veicolo guasto all'officina prenotata e quindi la prenotazione del carro attrezzi può avere successo. Dato che il

sistema è ciclico e che l'officina prenotata può cambiare ad ogni esecuzione del workflow (può essere una delle due istanze del servizio), dobbiamo verificare quanto richiesto per ogni ciclo. La computazione in cui il servizio viene prenotato ma almeno una locazione tra quella del veicolo e quella dell'officina non è raggiunta dal carro attrezzi rappresenta un comportamento indesiderato del modello. La proprietà è quindi di safety. Il caso in cui la prenotazione fallisca sempre non rappresenta un problema rispetto alla richiesta. La formulazione LTL data è la seguente[27].

$$
\Box\left(\begin{array}{l}(\text{ces[ces\_pid1]@damage} \wedge \neg \text{ces[ces\_pid1]@rel}) \longrightarrow \\ \left(\begin{array}{l}\\ (\text{ces[ces\_pid1]:pid\_towtruck} \mathrel{!}= \text{tt\_pid})\, \mathcal{W} \left(\begin{array}{c}(\text{towtruck:locs} >> (\text{ces[ces\_pid1]:loc})\,\&1) \\ \wedge \\ (\text{towtruck:locs} >> (\text{ces[ces\_pid1]:loc\_garage})\,\&1) \\ \vee \\ \text{ces[ces\_pid1]@rel}\end{array}\right) \\ \\ \end{array}\right)\end{array}\right)
$$

$$
\wedge
$$

$$
\Box\left(\begin{array}{l}(\text{ces[ces\_pid2]@damage} \wedge \neg \text{ces[ces\_pid2]@rel}) \longrightarrow \\ \left(\begin{array}{l}\\ (\text{ces[ces\_pid2]:pid\_towtruck} \mathrel{!}= \text{tt\_pid})\, \mathcal{W} \left(\begin{array}{c}(\text{towtruck:locs} >> (\text{ces[ces\_pid2]:loc})\,\&1) \\ \wedge \\ (\text{towtruck:locs} >> (\text{ces[ces\_pid2]:loc\_garage})\,\&1) \\ \vee \\ \text{ces[ces\_pid2]@rel}\end{array}\right) \\ \\ \end{array}\right)\end{array}\right)
$$

Considerata la complessità della formula è necessario spiegarla. Si può facilmente notare che le due sotto-formule in AND riguardano rispettivamente il primo ed il secondo processo `ces`. Nello specifico di un sistema d'emergenza, la richiesta è che deve valere sempre l'implicazione. Tale implicazione ha come premessa l'AND di due remote reference: lo stato di inizio del workflow e quello di rilascio. Se quindi un processo si trova nello stato di inizio e (contemporaneamente) non si trova in quello di rilascio (situazione che si verifica perché le due etichette riguardano stati diversi) viene provata la conclusione. Questa conclusione afferma che il processo non può ottenere la prenotazione del carro attrezzi, ovvero che la variabile `pid_towtruck` non contiene il pid di tale processo, a meno che la sua posizione e quella dell'officina che ha prenotato non siano entrambe tra quelle dove il servizio interviene. È però possibile il caso in cui in sistema d'emergenza non ottenga la prenotazione del carro attrezzi e quindi, in fase di rilascio delle risorse, non siamo più interessati al controllo sulla prenotazione. Il remote reference messo in OR con la condizione che permette la prenotazione del servizio serve proprio per bloccare il controllo della richiesta una volta raggiunto lo stato di release. Tale stato rappresenta infatti l'ultima fase del workflow. Come già detto questo controllo va fatto per ogni workflow.

Il modello verifica la proprietà. Se un sistema d'emergenza non ottiene la prenotazione del carro attrezzi per un dato workflow, allora il raggiungimento dello stato di release termina il controllo per quel workflow. Diversamente, se il carro attrezzi viene prenotato, il controllo fatto dal processo che modella il servizio alla riga 273, riportata nel Listato 28, garantisce che la condizione sia verificata.

[27] segue il pattern "precedence after Q until R" presentato nella pagina `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

l) "La prenotazione del noleggio ha successo solo se disponibile alla effettiva locazione del cliente (officina di riparazione o luogo dove è avvenuto il guasto)."

Anche quest'ultima proprietà rappresenta un vincolo indotto dalle locazioni. È esattamente l'esempio fatto nell'assunzione 14, riguardante lo scenario C. Risulta essere, come la proprietà **k**, una proprietà di safety. La formulazione LTL è la seguente.

$$\Box\left(\begin{array}{l}(\text{ces[ces\_pid1]@damage} \wedge \neg\text{ces[ces\_pid1]@rel}) \longrightarrow \\ \left(\begin{array}{l}\\ (\text{ces[ces\_pid1]:pid\_rentalcar != rc\_pid})\,\mathcal{W}\left(\begin{array}{c}(\text{rentalcar:locs} >> (\text{ces[ces\_pid1]:loc})\,\&1) \\ \vee \\ (\text{rentalcar:locs} >> (\text{ces[ces\_pid1]:loc\_garage})\,\&1) \\ \vee \\ \text{ces[ces\_pid1]@rel}\end{array}\right)\\ \\ \end{array}\right)\end{array}\right)$$

$$\wedge$$

$$\Box\left(\begin{array}{l}(\text{ces[ces\_pid2]@damage} \wedge \neg\text{ces[ces\_pid2]@rel}) \longrightarrow \\ \left(\begin{array}{l}\\ (\text{ces[ces\_pid2]:pid\_rentalcar != rc\_pid})\,\mathcal{W}\left(\begin{array}{c}(\text{rentalcar:locs} >> (\text{ces[ces\_pid2]:loc})\,\&1) \\ \vee \\ (\text{rentalcar:locs} >> (\text{ces[ces\_pid2]:loc\_garage})\,\&1) \\ \vee \\ \text{ces[ces\_pid2]@rel}\end{array}\right)\\ \\ \end{array}\right)\end{array}\right)$$

La struttura della formula è identica a quella presentata per la proprietà **k**, a parte l'OR che ha sostituito l'AND nella sotto-formula a destra dell'operatore $\mathcal{W}$. Ovviamente nella formula si fa riferimento a quelle variabili che riguardano il noleggio auto. Il cambio di operatore nella sotto-formula è dovuto invece al vincolo sul noleggio. Perché questo abbia successo basta che sia raggiungibile almeno una delle due locazioni: quella del veicolo o quella dell'officina.

Il modello verifica la proprietà. Nel caso il servizio di noleggio venga prenotato, il controllo alla riga 309 (del Listato 29) garantisce che la condizione richiesta sia verificata.

Spazio degli stati

Il modello sviluppato per lo scenario C prevede più variabili rispetto a quello presentato per lo scenario B. Ci aspettiamo quindi che lo spazio degli stati sia aumentato. In questo caso l'automa prodotto a partire dal `proctype ces` prevede 48 stati. Le configurazioni possibili delle variabili locali sono 2^{42} ($= 256^5 \times 2^2$), dato che ora il processo deve tener traccia della posizione dell'officina. Tale informazione, anche se di tipo `mtype`, viene memorizzata in una variabile di tipo `byte`[28]. Ai fine della stima dobbiamo tener conto anche delle variabili locali (di tipo `mtype` o `byte`) che rappresentano il parametro attuale e contengono l'informazione sulle locazioni. Ognuna di esse ha 2^8 ($= 256$) possibili valori. Dobbiamo poi considerare le 2^{16} ($= 256^2$) possibili configurazioni delle due variabili locali aggiunte al processo che modella il carro attrezzi. Il processo che rappresenta il noleggio auto prevede invece una nuova variabile rispetto allo scenario B. Le configurazioni delle due variabili locali di tale processo è 2^9 ($= 256 \times 2$). Infine, per quanto riguarda le variabili globali, abbiamo lo stesso numero di configurazioni dello scenario precedente.

La dimensione dello spazio degli stati si può stimare in $48^2 \times 2^{167}$ stati ($= 48 \times 2^{42+8} \times 48 \times 2^{42+8} \times 2^{10} \times 2^8 \times 2^8 \times 2^{16+8} \times 2^{9+8}$). Possiamo maggiorare tale cifra con $2^{179} \approx 5 \times 10^{53}$.

[28] stiamo sprecando memoria dato che i valori possibili sono solo 5, considerato anche il valore `0`.

Scenari alternativi

Sono stati sviluppati infine altri tre programmi PROMELA che rappresentano modelli alternativi rispettivamente per lo scenario B, per lo scenario C e per la sua versione "relabel"[29]. Questi tre programmi si differenziano da quelli prima presentati perché non utilizzano variabili (come quelle riportate nel Listato 5) per indicare i servizi già prenotati. È il servizio contatto che restituisce un messaggio di prenotazione fallita qualora sia già prenotato da un altro sistema d'emergenza. Si è quindi scelto di adottare la stessa tecnica utilizzata per la carta di credito (riportata nel Listato 7) per evitare che un messaggio inviato da un sistema d'emergenza sia ricevuto dall'altro. Nello specifico si sono rese atomiche le operazioni di receive e la successiva send eseguite dai processi che modellano un servizio. La conseguenza più importante di questa modifica è che ora un sistema d'emergenza non può bloccarsi sullo statement di receive dopo avere eseguito la send, dato che queste due operazioni risultano atomiche grazie alla modifica fatta ai processi che modellano i servizi. L'interesse di questa soluzione alternativa riguarda la proprietà **f**: ogni auto si guasta infinite volte, assumendo weak fairness. Tale proprietà risulta soddisfatta dal modello alternativo per lo scenario B. L'uso del costrutto `atomic` permette che la comunicazione richiesta/risposta e la successiva rilascio/ack avvengano senza che un altro sistema comunichi con il servizio, evento che bloccherebbe momentaneamente il primo sistema d'emergenza che aveva inviato una richiesta a tale servizio.

I modello alternativi dello scenario C non verificano però la proprietà. In tali modelli infatti bisogna gestire le locazioni, ovvero controllare se il veicolo e/o l'officina sono o meno raggiungibili. I processi che modellano il carro attrezzi ed il noleggio auto prevedono quindi istruzioni per verificare se devono o meno attendere il rilascio della prenotazione. Esistono quindi degli stati che servono a verificare se le locazioni sono raggiungibili. In tali stati il processo che modella il servizio non è quindi in ascolto sul canale ed un eventuale sistema d'emergenza che volesse fare una richiesta risulterebbe bloccato. Il verificarsi di questa situazione fa venire meno la premessa della weak fairness, ovvero che da un certo punto in poi ogni processo può sempre eseguire la sua prossima istruzione. Di conseguenza può capitare che uno dei due sistemi rimanga fermo in uno stato e ciò fa fallire la verifica. Lo scopo di questa considerazione è quello di legittimare l'uso delle variabili presentate nel Listato 5 se si vuole che la proprietà **f** sia verificata con assunzioni di weak fairness a livello di processi. Altrimenti bisognerebbe utilizzare un assunzione di strong fairness.

[29] i rispettivi file sono: `CarEmBalt.pml`, `CarEmCalt.pml` e `CarEmCaltrelabel.pml`