Università degli Studi di Firenze

Elementi di Software e Dependability

Modellazione di un sistema di Interlocking Ferroviario Distribuito tramite UML on-the-fly Model Checker

Simone Rossetto e Giovanni Rocciolo

Università degli Studi di Firenze

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

29 marzo 2011

---

30 marzo 2011

`interlocking.pdf`

realizzato con LaTeX

Indice

1 Introduzione

Lo scopo di questo elaborato consiste nello studiare, implementare e verificare una soluzione per il problema dell'*Interlocking Distribuito* in ambito ferroviario tramite lo strumento *UML on-the-fly Model Checker* (UMC) [1]. In particolare è stato esteso il lavoro di Marco Paolieri [2] introducendo nuove funzionalità e una configurazione ferroviaria più complessa.

Il resto del documento è così diviso: nella sezione 2 viene presentato il problema dell'Interlocking Distribuito in ambito ferroviario, con discussione su come sia possibile per i singoli treni prenotare un determinato itinerario. La sezione 3 contiene la descrizione del modello di simulazione da noi realizzato con particolare enfasi sulle novità introdotte rispetto al modello di M. Paolieri. Successivamente nella sezione 4 vengono presentate le proprietà base di *Safety* che il modello deve rispettare.

Infine nelle appendici riportiamo una breve descrizione del sistema UMC e il codice completo del nostro modello.

2 Interlocking ferroviario distribuito

Un sistema di *Interlocking Ferroviario* è un sistema in grado di gestire una porzione di tracciato su una rete ferroviaria, sulla quale transitano dei treni. Lo scopo del sistema è quello di ricevere e processare le richieste di prenotazione dei percorsi da parte dei treni e di accettarle o meno evitando gli incidenti.

Nella variante "distribuita", non è presente un solo centro di gestione ed elaborazione che invia comandi ad ogni sezione, ma ogni porzione del tracciato è a conoscenza delle operazioni che deve compiere in base al tipo di richiesta ricevuta e si comporta di conseguenza. Di fatto, quindi, ogni parte del tracciato eseguirà in modo indipendente il proprio compito inoltrando le eventuali richieste alle porzioni adiacenti del percorso richiesto.

2.1 Elementi di un tracciato

Il tracciato può essere visto con un grafo (V, E) dove V sono i vertici e E sono gli archi di collegamento. Su questo grafo è definita una relazione di adiacenza del tipo $R \subseteq V \times V$ dove un arco entrante in v da v' specifica che v' è un adiacente *sinistro* di v, mentre un arco uscente da v a v'' indica che v'' è un adiacente *destro* di v (si veda a tale proposito la figura 1).

I nodi possono essere di due tipi:

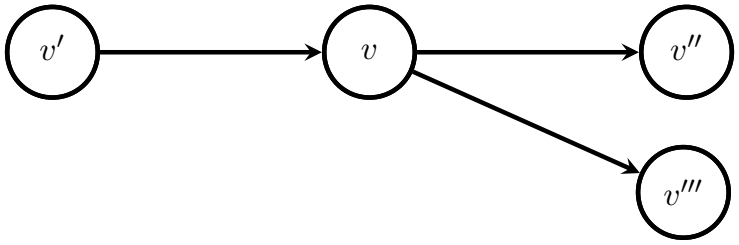

Figura 1: Esempio di grafo orientato

Circuiti di Binario Rappresenta un tratto di binario e può avere soltanto un adiacente destro ed un adiacente sinistro. Su tale tratto è possibile collocare anche una stazione e quindi prevedere la possibilità di fermarsi. In riferimento alla fig. 1, v', v'', v''' sono dei Circuiti di Binario.

Scambio Rappresenta un punto di diramazione o confluenza, e ha necessariamente tre nodi adiacenti. Nel caso di scambio destro (sinistro) questi sono detti: adiacente sinistro (destro), adiacente *normale* destro (sinistro), adiacente *rovescio* destro (sinistro). Dove per "normale" o "rovescio" si riferisce all'orientazione fisica dello scambio. In riferimento alla fig. 1, v è uno Scambio.

È importante notare inoltre che non possono esistere cappi, ossia un nodo non può avere come adiacente se stesso, e non esistono cicli nel grafo. Quindi tale grafo è un grafo orientato aciclico.

Un *itinerario* su questo tracciato è una sequenza di nodi in cui il primo e l'ultimo devono essere dei Circuiti di Binario e se il nodo v' segue v nella sequenza allora è necessario che $(v, v') \in R$, ossia v e v' devono essere adiacenti.

2.2 Funzionalità richieste al sistema di *Interlocking*

Le funzioni richieste ad un sistema di *Interlocking* sono:

- prenotazione di itinerari (anche multipli);
- cancellazione di un itinerario prenotato;
- liberazione dell'itinerario dopo che il treno lo ha percorso;
- liberazione dell'intero tracciato quando il treno arriva nelle stazioni terminali.

Vediamole in dettaglio.

Prenotazione Itinerario

La prenotazione dell'itinerario viene effettuata attraverso il protocollo *Two Phase Commit Protocol* (2PC) che prevede un doppio scambio di messaggi da parte di ogni nodo appartenente all'itinerario.

Ad ogni nodo è associato un identificativo ed ogni nodo conosce sia il nodo successivo sia il precedente lungo l'itinerario richiesto. Inoltre il primo e l'ultimo nodo dell'itinerario sanno di essere, rispettivamente, il primo e l'ultimo.

La prenotazione dell'itinerario parte dal treno che vi deve transitare sopra e viene inoltrata, tramite un messaggio di "request" al primo nodo (che deve essere necessariamente un Circuito di Binario). Ogni nodo, se libero, diviene "riservato" e propaga la richiesta al nodo successivo dell'itinerario. L'ultimo nodo, oltre a divenire "riservato", invia a ritroso un messaggio di "acknowledge" che sarà propagato fino al primo nodo.

Non appena il primo nodo riceverà questo "ack" (indice del fatto che tutti i nodi sull'itinerario sono disponibili), invierà un messaggio di "commit" il quale percorrerà nuovamente tutto il cammino per poi essere riscontrato da un messaggio di "agree" inviato dall'ultimo nodo.

Quando il primo nodo riceve l'"agree" comunica al treno che l'itinerario è disponibile, il treno potrà così iniziare a muoversi e una volta giunto alla fine potrà fermarsi oppure chiedere un nuovo itinerario eseguendo la stessa procedura.

Tutti i nodi *non* liberi rigetteranno ogni messaggio di "request", rispondendo con un "negative acknowledge". Ogni nodo che riceve tale messaggio torna ad essere libero e lo propaga.

Cancellazione Itinerario

La cancellazione di un itinerario viene eseguita con una versione a singola fase del 2PC questo perché i nodi risultano già "riservati", quindi è come se la prima fase del protocollo fosse già stata eseguita.

Il treno che aveva prenotato un itinerario e ora vuole cancellarlo invia un messaggio di "abort". Ogni nodo che lo riceve si porta in uno stato di "aborting" e inoltra la comunicazione al nodo successivo. Quanto tale messaggio raggiunge l'ultimo nodo, questo diviene nuovamente libero e invia a ritroso un messaggio di "cancel". I nodi che ricevono il secondo messaggio di cancellazione divengono a loro volta liberi e propagano il "cancel" ai nodi precedenti finché non si raggiunge il primo nodo, a questo punto l'itinerario risulta cancellato.

Liberazione Itinerario

Dopo che il treno si è mosso e ha raggiunto la destinazione è opportuno liberare l'itinerario per permettere ad altri treni di usufruirne. Questa liberazione può essere realizzata in due modi diversi: utilizzare una versione a singola fase del 2PC (come per la cancellazione dell'itinerario), oppure liberare ogni singolo nodo dopo il transito del treno.

La seconda soluzione risulta essere più semplice e meno dispendiosa perché evita lo scambio di messaggi e perché i nodi divengono liberi immediatamente e quindi permette una più veloce riprenotazione di nuovi itinerari. Nella nostra implementazione viene utilizzata questa seconda versione.

Liberazione dell'intero tracciato

Quando un treno giunge alla fine del suo ultimo percorso e si trova su un Circuito di Binario esterno è opportuno avere la possibiltà che il treno esca dal tracciato, sia per liberare una parte di circuito che rimarrebbe occupata, sia perché il treno potrebbe entrare in un altro tracciato.

Per questo motivo i Circuiti di Binario esterni sanno di essere esterni e quindi elaborano un comando aggiuntivo di "free" consentendo ai treni che vi transitano di muoversi anche verso l'esterno del tracciato liberandolo.

3 Modellazione del sistema

Il modello del nostro sistema segue quanto descritto nella sezione precedente implementando le varie funzionalità dei nodi del tracciato e le funzionalità del treno realizzando quindi tutto il sistema di *Interlocking Distribuito*.

3.1 Configurazione tracciato e itinerari

Il tracciato che abbiamo utilizzato per eseguire la simulazione del nostro sistema è formato da cinque Circuiti di Binario (`firenze`, `prato`, `porretta`, `bologna`, `pisa`) e da quattro Scambi (`switchFi1`, `switchFi2`, `switchBo1`, `switchBo2`) collegati come rappresentato in fig. 2.

Con questa configurazione esistono 18 possibili itinerari:

1. `firenze`, `switchFi1`, `switchFi2`, `prato`
2. `prato`, `switchBo2`, `switchBo1`, `bologna`
3. `firenze`, `switchFi1`, `switchFi2`, `prato`, `switchBo2`, `switchBo1`, `bologna`
4. `firenze`, `switchFi1`, `switchFi2`, `porretta`

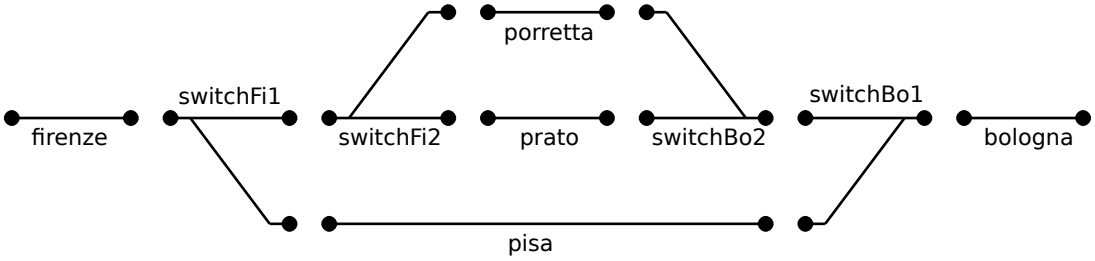

Figura 2: Tracciato ferroviario utilizzato per la simulazione

5. `porretta`, `switchBo2`, `switchBo1`, `bologna`
6. `firenze`, `switchFi1`, `switchFi2`, `porretta`, `switchBo2`, `switchBo1`, `bologna`
7. `prato`, `switchFi2`, `switchFi1`, `firenze`
8. `bologna`, `switchBo1`, `switchBo2`, `prato`
9. `bologna`, `switchBo1`, `switchBo2`, `prato`, `switchFi2`, `switchFi1`, `firenze`
10. `porretta`, `switchFi2`, `switchFi1`, `firenze`
11. `bologna`, `switchBo1`, `switchBo2`, `porretta`
12. `bologna`, `switchBo1`, `switchBo2`, `porretta`, `switchFi2`, `switchFi1`, `firenze`
13. `firenze`, `switchFi1`, `pisa`
14. `firenze`, `switchFi1`, `pisa`, `switchBo1`, `bologna`
15. `bologna`, `switchBo1`, `pisa`, `switchFi1`, `firenze`
16. `bologna`, `switchBo1`, `pisa`
17. `pisa`, `switchBo1`, `bologna`
18. `pisa`, `switchFi1`, `firenze`

La configurazione base considerata da Marco Palieri in [2] non aveva la stazione di Pisa e quindi era priva anche dei due Scambi aggiuntivi collegati con Firenze e con Bologna, di conseguenza gli itinerari possibili erano solo 12.

3.2 Modello del Circuito di Binario

Il Circuito di Binario è modellato con la classe `TrackCircuit` la quale è a conoscenza dei Circuiti di Binario precedenti e successivi (variabili `prev` e `next`) per ogni possibile itinerario, sa istantaneamente se è presente un treno (varibile `train`), inoltre utilizza la variabile booleana `outer_station` per sapere se è una stazione esterna nel tracciato considerato. Tutte queste varibili vanno assegnate in fase di creazione di ogni singolo oggetto.

Il comportamento dinamico del `TrackCircuit` è rappresentato nella figura 3 dove sono evidenziati sia gli stati in cui un Circuito di Binario può trovarsi sia quali segnali permettono le trasizioni.

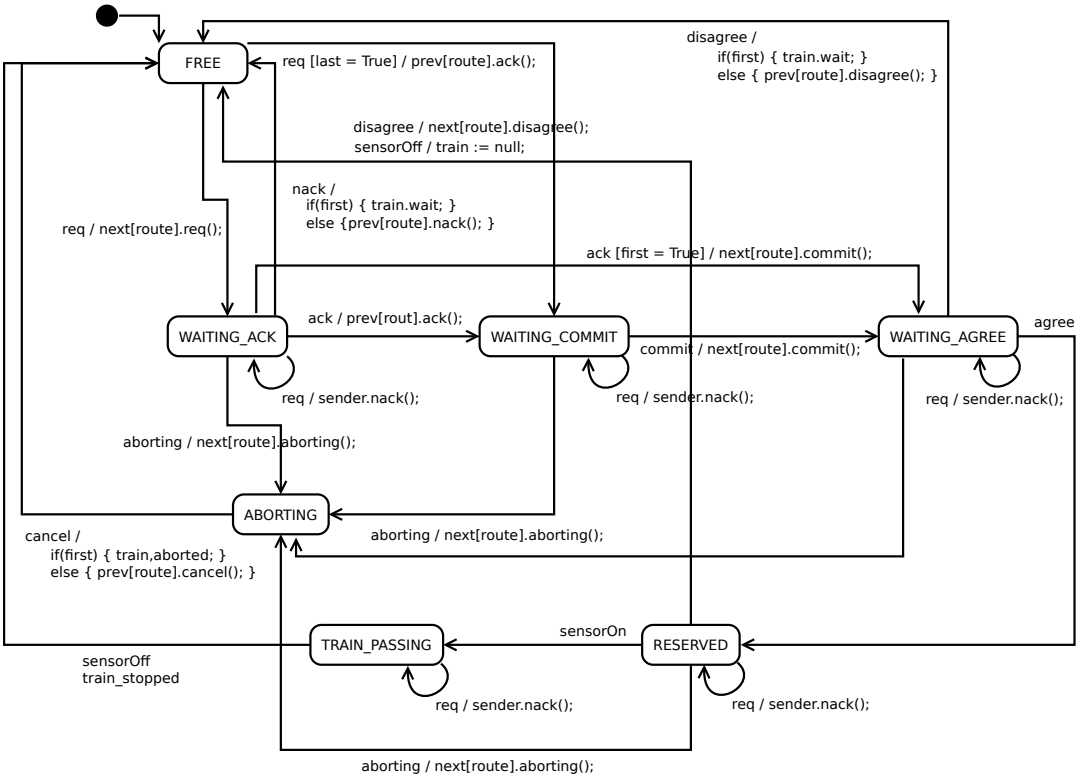

Figura 3: Comportamento dinamico del Circuito di Binario

I segnali che possono essere elaborati dal `TrackCircuit` sono:

Request Il segnale `req(sender, dest, route)` indica una richiesta di prenotazione per l'itinerario `route` da parte dell'oggetto `sender` (un Treno, uno Scambio oppure un altro Circuito di Binario). Quando viene ricevuto può essere rifiutato qualora lo stato corrente non sia `FREE` rispondendo con un segnale `nack` (o col segnale `wait` se è il primo nodo dell'itinerario), oppure può essere accettato eseguendo la transizione nello stato `WAITING_ACK` e propagando la richiesta al nodo successivo dell'itinerario. Se ci troviamo sul nodo finale lo stato diverrà `WAITING_COMMIT` e non viene propagato nessun messaggio, ma si risponde al mittente con un messaggio di `ack`.

Acknowledgement Il segnale `ack(sender, dest, route)` indica un riscontro positivo alla richiesta di prenotazione relativo all'itinerario `route` da parte dell'oggetto `sender`. Un nodo che riceve questo messaggio ha precedentemente propagato un messaggio di `req` e si trova nello stato `WAITING_ACK`, quindi propaga l'`ack` al nodo precedente sull'itinerario e si pone in attesa del `commit` eseguendo la transizione allo stato `WAITING_COMMIT`. Se a ricevere l'`ack` è il primo nodo dell'i-

tinerario questo si porta direttamente nello stato `WAITING_AGREE` e spedisce un messaggio di `commit` in avanti.

Negative Acknowledgement Il segnale `nack(sender, dest, route)` rappresenta un riscontro negativo alla richiesta di prenotazione per l'itinerario `route` da parte dell'oggetto `sender`. Un nodo che riceve questo messaggio ha precedentemente propagato un messaggio `req` e poiché la prima fase del 2PC è fallita, propaga il `nack` al nodo precedente sull'itinerario e ritorna allo stato `FREE` in attesa di una nuova richiesta di prenotazione.

Commit Il segnale `commit(sender, dest, route)` indica un commando di "conferma prenotazione" per l'itinerario `route` inoltrata dall'oggetto `sender`. Un nodo che riceve questo messaggio ha precedentemente propagato i messaggi `req` e `ack` dunque si trova in `WAITING_COMMIT` e alla sua ricezione lo propaga al nodo successivo e si pone nello stato `WAITING_AGREE`. Se il nodo è l'ultimo dell'itinerario invece che inoltrare il `commit` spedisce indietro un messaggio `agree` e si porta direttamente in `RESERVED` in attesa che il treno vi transiti sopra.

Agree Il segnale `agree(sender, dest, route)` indica la prenotazione finale per l'itinerario `route` da parte dell'oggetto `sender`. Un nodo che riceve questo messaggio ha precedentemente propagato i messaggi `req`, `ack` e `commit` quindi alla ricezione dell'`agree` lo propaga al nodo precedente sull'itinerario (o invia uno `start` al treno se è il primo nodo dell'itinerario) e si pone nello stato `RESERVED`.

Disagree Il segnale `disagree(sender, dest, route)` indica il fallimento di un `commit` per l'itinerario `route`. Nel nostro modello solo uno Scambio può sollevare il fallimento del `commit` quindi un Circuito di Binario può solo riceverlo. In tal caso, se si trova nello stato `WAITING_AGREE`, lo propaga al nodo precedente, se invece si trova nello stato `RESERVED` allora lo propaga al nodo successivo. In entrambi i casi, dopo l'inoltro, si porta allo stato `FREE`. La differenza di comportamento in base allo stato corrente è dovuta al fatto che il `disagree` è spedito da uno Scambio in entrambe le direzioni, quindi se il Circuito di Binario lo riceve dal nodo successivo vuol dire che ancora non ha ricevuto un `agree`, quindi si trova in `WAITING_AGREE` e deve inoltrarlo indietro, se invece lo riceve dal nodo precedente, allora vuol dire che ha già fatto passare un `agree` ed è quindi riservato, dunque dovrà inoltrare il `disagree` in avanti.

Abort Il segnale `aborting(sender, dest, route)` indica la richiesta di cancellazione dell'itinerario `route` per il quale era stata precedentemente inviata una richiesta di prenotazione. Il primo messaggio di questa richiesta parte sempre da un treno e ogni Circuito di Binario, alla ricezione di tale messaggio, inoltra la richiesta al nodo successivo e si porta nello stato `ABORTING`. Qual'ora il nodo

sia l'ultimo dell'itinerario allora passa direttamente allo stato `FREE` e spedisce indietro un messaggio di `cancel`.

Cancel Il segnale `cancel(sender, dest, route)` indica la conferma di cancellazione dell'itinerario `route`. Alla ricezione di questo messaggio un Circuito di Binario inoltra la richiesta al nodo precedente lungo l'itinerario e si porta nello stato `FREE` tornando ad essere libero e in attesa di una nuova prenotazione. Qual'ora il circuito sia quello su cui si trova il treno che ha richiesto la cancellazione allora, oltre ad eseguire la trasizione verso lo stato `FREE` spedisce il messaggio `aborted` al treno. In questa situazione l'itinerario è stato reso tutto libero e il treno può chiedere una nuova prenotazione.

Gli stati in cui può trovarsi un `TrackCircuit` sono:

- `FREE`: il circuito è libero in attesa di una prenotazione;
- `WAITING_ACK`: ha ricevuto il messaggio `req` ed è in attesa di un `ack` (questo stato non è possibile per l'ultimo nodo di ogni itinerario);
- `WAITING_COMMIT`: ha ricevuto il `ack` e lo ha inoltrato, quindi è in attesa di un `commit` (questo stato non è permesso al primo nodo dell'itinerario);
- `WAITING_AGREE`: ha ricevuto il `commit` e lo ha inoltrato, quindi è in attesa di un `agree` (questo stato non è permesso all'ultimo nodo dell'itinerario);
- `RESERVED`: il circuito è riservato ed è in attesa del transito del treno;
- `TRAIN_PASSING`: un treno si trova sul circuito;
- `ABORTING`: il circuito ha ricevuto un messaggio di `aborting` ed è in attesa di ricevere il `cancel`.

Rispetto a quanto sviluppato da Marco Palieri in [2] qui è stato aggiunta la possibilità di cancellare l'itinerario prenotato, quindi risultano nuovi i segnali `aborting` e `cancel` con il relativo stato `ABORTING`. Sono state quindi introdotte anche tutte le dovute transizioni per la gestione di questi nuovi segnali. Inoltre è stata implementata la liberazione di ogni singolo nodo dopo il passaggio del treno con la transizione automatica `TRAIN_PASSING` → `FREE` dopo il movimento del treno.

3.3 Modello dello Scambio

Lo Scambio viene modellato con la classe `Switch` rappresentata con i suoi stati e transizioni nella figura 4.

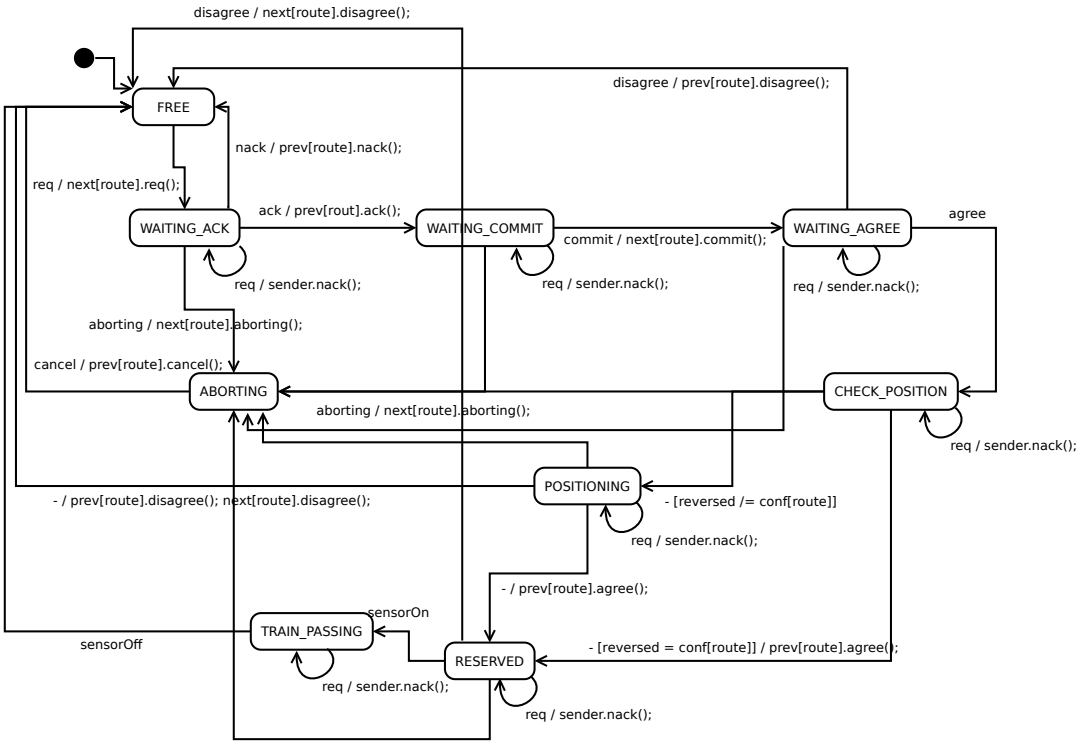

Figura 4: Comportamento dinamico di uno Scambio

L'algoritmo di prenotazione implementato da questa classe è lo stesso di quello della classe `TrackCircuit` con alcune varianti dovute al fatto che:

(a) uno Scambio non può mai essere il primo o l'ultimo nodo di un itinerario, quindi i messaggi possono essere solo ricevuti e spediti a Circuiti di Binario o altri Scambi, mai a treni, di conseguenza non sono presenti salti in avanti da `FREE` a `WAITING_COMMIT` e da `WAITING_COMMIT` a `RESERVED`, o da `WAITING_ACK` a `WAITING_AGREE` propri dei Circuiti di Binario iniziali e finali;

(b) i treni non possono fermarsi su uno Scambio, quindi la richiesta di cancellazione di un itinerario non può avvenire su uno Scambio;

(c) gli Scambi devono posizionarsi prima di permettere il passaggio del treno (il posizionamento può fallire, ed è in questo caso che viene spedito un messaggio di `disagree`).

Ogni oggetto della classe `Switch` ha due variabili aggiuntive rispetto ai Circuiti di Binario, ossia `reversed` e `conf` che rappresentano, rispettivamente, il posizionamento corrente (*normale* o *rovescio*) e quale deve essere il posizionmento corretto per ogni itinerario. Per permettere il posizionamento sono stati implementati due stati aggiuntivi, `CHECK_POSITION` e `POSITIONING`, il loro ruolo è il seguente:

- `CHECK_POSITION`: lo Scambio si sposta in questo stato subito dopo aver ricevuto un `agree` e controlla se il suo posizionamento è corretto per l'itinerario richiesto (ossia se `reversed == conf[route]`). Se il posizionamento è giusto inoltra il messaggio `agree` e si porta in `RESERVED` in attesa del transito del treno, se il posizionamento invece non è corretto allora passa allo stato `POSITIONING`;
- `POSITIONING`: in questo stato si aggiusta il posizionamento dello Scambio. Tale operazione può fallire, ossia non deterministicamente invece che passare allo stato `RESERVED` inoltrando l'`agree`, passa allo stato `FREE` spedendo in entrambe le direzioni un `disagree` che viene propagato su tutto l'itinerario cancellando, di fatto, la prenotazione.

Rispetto a quanto implementato da Marco Paolieri in [2], lo scambio è rimasto sostanzialmente lo stesso nel funzionamento, semplicemente abbiamo aggiunto le transizioni per la gestione dei messaggi di cancellazione dell'itinerario (`aborting` e `cancel`) e abbiamo creato lo stato `CHECK_POSITION` le cui funzioni venivano svolte in modo implicito nello stato `WAITING_AGREE`. La nostra versione risulta di più facile comprensione avendo diviso meglio i compiti di ogni stato.

3.4 Modello del Treno

Il treno è stato implementato nella classe `Train` rappresentata in figura 5.

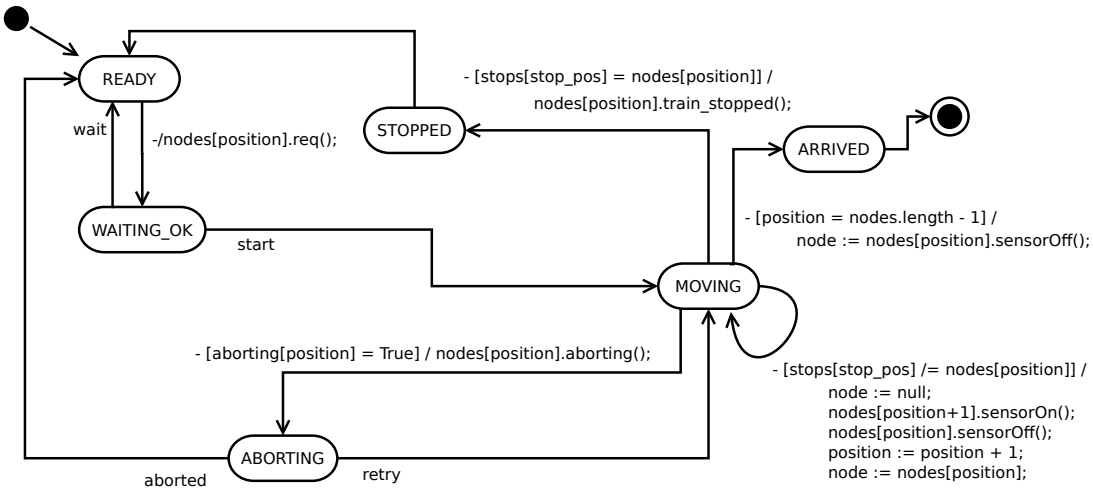

Figura 5: Comportamento dinamico della classe `Train`

Il suo compito è quello di richiedere un itinerario inviando al Circuito di Binario sul quale si trova un messaggio di `req`. Se la prenotazione ha successo, e quindi riceve in risposta un messaggio di `start`, si muove di nodo in nodo fino a raggiungere l'ultima

stazione. A questo punto può chiedere un nuovo itinerario oppure uscire dal tracciato, qual'ora si sia fermato su un nodo esterno (ossia un Circuito di Binario in cui la variabile `outer_station == True`).

In fase di istanziazione, ad ogni oggetto della classe `Train` sono assegnati:

- un vettore di interi `routes`, rappresentante gli itinerari che il treno deve percorrere prima di fermarsi;
- un vettore (`nodes`) contenente i nodi che compongono tutti gli itinerari da percorrere (tale vettore viene utilizzato durante la fase di movimento per sapere qual è il successivo nodo verso cui muoversi);
- un vettore di stazioni (`stops`) che indica su quali Circuiti di Binario il treno deve fermarsi;
- un vettore di nodi (`aborting`) che contiene i nodi sui quali il treno simula un malfunzionamente e invia la richiesta di cancellazione dell'itinerario.

Nella fase iniziale, il treno cicla fra gli stati `READY` ed `WAITING_OK` fino a quando non riceve una conferma positiva (messaggio `start`) alla richiesta di prenotazione del primo itinerario. Se la prenotazione non è andata a buon fine, ossia un nodo risulta già prenotato da un altro treno, il messaggio di ritorno sarà un `wait` e quindi il treno si riporta nello stato `READY` in attesa di tentare una nuova prenotazione.

Durante il movimento il treno si sposta da un nodo al successivo inviando messaggi di `sensorOn` al nodo su cui entra e `sensorOff` al nodo dal quale esce. La chiamata a `sensorOff` di fatto riporta il nodo di uscita nello stato `FREE` rendendolo disponibile per una nuova prenotazione.

Quando il treno giunge alla fine di un itinerario segnalerà la fermata al Circuito di Binario sul quale si trova per mezzo di un messaggio `train_stopped` e torna nello stato `READY` pronto per inviare la richiesta di prenotazione per l'itinerario successivo. Se siamo giunti alla fine dell'ultimo itinerario e la stazione è un nodo esterno il messaggio di `sensorOff` permetterà anche l'uscita del treno dall'intero tracciato.

Rispetto al modello di Marco Paolieri [2], la classe `Train` ha subito notevoli modifiche perché deve poter gestire itinerari multipli e non uno solo, deve potersi fermare alla fine di ogni itinerario, deve poter cancellare l'itinerario in caso di malfunzionamenti. Per questo lo schema del suo funzionamento dinamico è ben più complesso di quello

in [2].

4 Proprietà di *Safety*

Affinché il sistema da noi modellato sia un vero sistema di *Interlocking Ferroviario Distribuito* è necessario che siano garantite alcune proprietà di *Safety*. In particolare è importante che:

- non vi siano *incidenti*, ossia due treni non devono mai trovarsi contemporaneamente sullo stesso binario;
- non vi devono essere treni sugli Scambi durante la fase di posizionamento (quindi quando uno Scambio si trova in uno degli stati `CHECK_POSITION` e `POSITIONING` la sua variabile `train` deve essere `null`);
- i treni devono arrivare sempre a destinazione.

In *Computation Tree Logic* (CTL) [3, 4] queste proprietà sono così descritte:

```
AG NOT crash

AG (switch_positioning IMPLIES NOT train_on_switch)

EF AG train_arrived
```

dove `crash` rappresenta la condizione di incidente, ossia due treni sullo stesso nodo, `switch_positioning` rappresenta la situazione in cui lo Scambio è in fase di posizionamento, `train_on_switch` è asserito quando un treno si trova su uno Scambio e infine `train_arrived` indica che il treno ha compiuto tutti gli itinerari e si trova nello stato `ARRIVED`.

Ovviamente nel modello reale queste proprietà devono essere opportunamente modificate per controllare lo stato di ogni treno. Si veda a tal proposito il codice completo, listato B.6 a pagina 26.

5 Simulazione del modello

La simulazione del nostro modello è stata eseguita con diverse configurazioni iniziali degli oggetti coinvolti. Riportiamo qui di seguito una configurazione complessa e completa nella quale si vede come la funzionalità di liberazione di tutto il tracciato permette la circolazione anche di più treni con la stessa stazione di destinazione, situazione non permessa dal modello iniziale di Marco Paolieri [2].

La configurazione iniziale è riportata in figura 6 dove sono evidenziati i treni `marconi` e `meucci` alla partenza con i loro itinerari (per il codice completo si veda il listato B.4 a pagina 22).

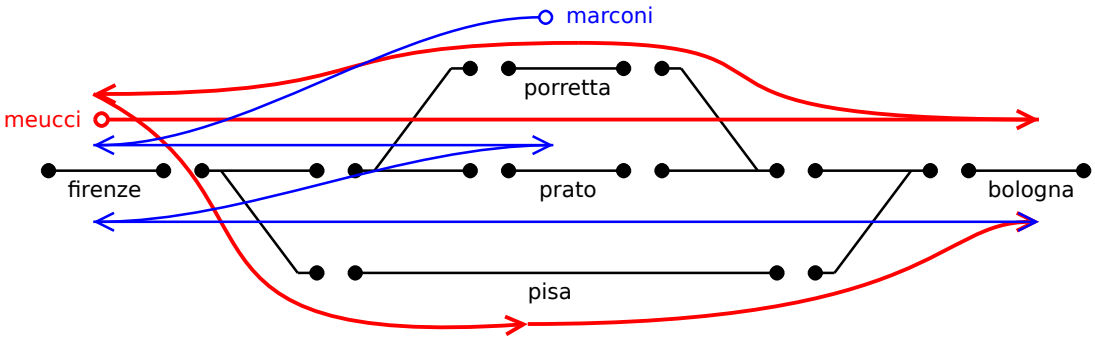

Figura 6: Configurazione iniziale e percorsi dei treni utilizzati per la simulazione

Si nota bene che in più di una circostanza gli itinerari dei due treni si incrociano, anche l'ultima stazione di arrivo è la stessa per entrambi, questo significa che un solo treno per volta ha l'opportunità di prenotare quel determinato itinerario, che sarà poi liberato, permettendo all'altro treno di utilizzarlo. Un'errata implementazione del sistema non avrebbe verificato la proprietà

```
AG NOT crash
```

in quanto due treni si sarebbero potuti trovare contemporaneamente sullo stesso nodo.

Ugualmente l'aver verificato che i treni arrivano sempre a destinazione è indice del fatto che non vi sono blocchi durante il tragitto ossia nodi del tracciato dal quale un treno non riesce più ad uscire per via di segnali mancanti o transizioni non deterministiche *unfair*.

Infine la proprietà secondo la quale i treni non devono trovarsi su uno scambio durante il posizionamento è praticamente sempre garantita dal fatto che gli Scambi si posizionano durante la fase di prenotazione di un itinerario, fase nella quale i treni sono sempre su Circuiti di Binario e quindi ogni Scambio è libero di muoversi.

6 Conclusioni

Complessivamente il modello da noi sviluppato riesce a simulare correttamente un sistema di *Interlocking Ferrioviario Distribuito* garantendo l'arrivo dei treni a destinazione senza incidenti e/o blocchi imprevisti. È però importante notare che le proprietà di *Safety* sono state testate solo per alcune configurazioni iniziali, sarebbe opportuno

implementare un sistema per verificarle con ogni possibile configurazione in modo tale da essere sicuri del totale e corretto funzionamento di tutto il sistema.

Inoltre potrebbe essere di notevole interesse studiare e implementare un sistema per poter generare in modo automatico le configurazioni, senza dover riscrivere completamente l'istanziazione dei singoli oggetti. Infatti nell'estendere il modello base di Marco Paolieri [2], in cui mancava la stazione di Pisa, è stato necessario rivedere tutti gli array `prev` e `next` di ogni Circuito di Binario, operazione facilmente soggetta ad errori. Un Tool di generazione automatica sarebbe molto comodo.

A *UML on-the-fly Model Checker*

Lo strumento UMC permette di effettuare il *model checking* di formule μUCTL [5] a partire da:

- un insieme di *statecharts* [6] che descrivono il comportamento dinamico delle classi del sistema;
- un diagramma degli oggetti per lo stato iniziale del sistema;
- una serie di "criteri" che specificano quali attributi ed eventi osservare.

Una formula μUCTL è verificata *"on-the-fly"* [1], generando incrementalmente le parti della macchina a stati necessarie alla verifica corrente.

L'algoritmo di *model checking* impiegato è di tipo *bounded* ed effettua successive ricerche in profondità aumentando il limite fino a determinare se la formula è soddisfatta o meno. Questo, oltre a impedire l'esplosione dello spazio degli stati, permette di fornire controesempi di lunghezza ridotta nel caso di formule non soddisfatte.

A.1 Definizione del modello

La struttura dei modelli UMC [7] comprende tre sezioni:

Definizione delle classi Definisce le classi del sistema in termini di macchine a stati UML. Ogni definizione di classe permette di specificare:

- gli eventi asincroni (`Signals`) e le chiamate sincrone (`Operations`) accettate dalla classe;
- le variabili locali della classe (`Vars`);
- la struttura di stati e sottostati della statechart (`State`);
- le transizioni, con trigger, guardia ed azioni.

Dichiarazione degli oggetti Una volta definite le classi è possibile istanziarle ed inizializzare gli oggetti impostandone le variabili locali. I nomi associati alle istanze degli oggetti costituiscono variabili globali visibili all'interno di tutte le definizioni di classe.

Regole di astrazione Specificano quali eventi e proprietà del sistema osservare, e giocano un ruolo essenziale al momento della verifica, costituendo i predicati atomici su cui definire formule μUCTL.

A.2 Verifica di formule in logica temporale

UMC supporta numerose logiche temporali [8] che comprendono tutto il μ-Calcolo, oltre ad operatori di più alto livello simili a quelli delle logiche CTL ed *Action-based CTL* (ACTL) [9].

In particolare, sono presenti:

- operatore *diamond* `<action>`ϕ e *weak diamond* `<<action>>`ϕ ;
- operatore *box* `[action]`ϕ e *weak box* `[[action]]`ϕ;
- operatori di massimo e minimo punto fisso (`max`, `min`);
- operatori `EX{action}`ϕ, `AX{action}`ϕ, `ET`ϕ, `AT`ϕ;
- operatori `EF`ϕ, AFϕ, EGϕ, AGϕ;
- operatori `E[`ϕ_1`{action}`$\cup\phi_2$ `]` e `A[`ϕ_1`{action}`$\cup\phi_2$`]`
- operatori `E[`ϕ_1`{act1}`$\cup$`{act2}`ϕ_2`]` e `A[`ϕ_1`{act1}`$\cup$`{act2}`ϕ_2`]`

I predicati atomici di *state formulae* e *action formulae* sono definiti attraverso regole di astrazione, che specificano quali condizioni e quali azioni osservare nel sistema, associando ad esse un nome utilizzabile nelle formule in logica temporale.

Per la verifica è possibile utilizzare una versione a riga di comando di UMC oppure l'interfaccia web disponibile all'indirizzo `http://fmt.isti.cnr.it/umc` [10]. L'interfaccia web dispone di numerose funzionalità aggiuntive, fra cui la minimizzazione dell'automa e la generazione di rappresentazioni grafiche.

Per una guida all'uso si rimanda direttamente al sito. Un interessante esempio di utilizzo è descritto in [1].

B Codice del modello

Riportiamo di seguito l'intero codice del nostro modello.

Listati

Listato B.1: Modello della classe Train

```
Class Train is

  Signals:
    start, wait, aborted, retry;

  Vars:
    routes: int[];
    route_pos: int := 0;
    nodes: obj[];
    position: int := 0;
    stops: obj[];
    stop_pos: int := 0;
    node: obj;
    aborting: bool[];

  State Top = READY, WAITING_OK, MOVING, ARRIVED, STOPPED, ABORTING

  Transitions:
    READY -> WAITING_OK { - / nodes[position].req(self, nodes[position],
        routes[route_pos]); }

    WAITING_OK -> READY { wait }
    WAITING_OK -> MOVING { start }
    WAITING_OK -> ABORTING { - [aborting[position] = True] /
        nodes[position].aborting(self, nodes[position],
        routes[route_pos]); }

    MOVING -> ABORTING { - [aborting[position] = True AND position <
        nodes.length - 1 AND stops[stop_pos] /= nodes[position]] /
        nodes[position].aborting(self, nodes[position],
        routes[route_pos]); }

    MOVING -> MOVING { - [aborting[position] = False AND position <
        nodes.length - 1 AND stops[stop_pos] /= nodes[position]] /
                    node := null;
                    nodes[position + 1].sensorOn(self, nodes[position +
                        1], routes[route_pos]);
                    nodes[position].sensorOff(self, nodes[position],
                        routes[route_pos]);
                    position := position + 1;
                    node := nodes[position];
    }

    MOVING -> STOPPED { - [stops[stop_pos] = nodes[position] AND
        position < nodes.length - 1] /
        nodes[position].train_stopped(self); }
    MOVING -> ARRIVED  { - [position = nodes.length - 1] / node := null;
```

```
        node := nodes[position].sensorOff(self, nodes[position],
        routes[route_pos]); }

    STOPPED -> READY { - [position < nodes.length - 1] /
                    stop_pos := stop_pos + 1;
                    route_pos := route_pos + 1;
    }

    ABORTING -> READY { aborted / aborting[position] := False; }
    ABORTING -> MOVING { retry / aborting[position] := False;
        aborting[position + 1] := True; }

end Train
```

Listato B.2: Modello della classe TrackCircuit

```
Class TrackCircuit is

  Signals:
    req(sender: obj, dest: obj, route: int);
    ack(sender: obj, dest: obj, route: int);
    nack(sender: obj, dest: obj, route: int);
    commit(sender: obj, dest: obj, route: int);
    agree(sender: obj, dest: obj, route: int);
    disagree(sender: obj, dest: obj, route: int);
    train_stopped(sender: obj);
    aborting(sender: obj, dest: obj, route: int);
    cancel(sender: obj, dest: obj, route: int);

  Operations:
    sensorOn(sender: obj, dest: obj, route: int);
    sensorOff(sender: obj, dest: obj, route: int);

  Vars:
    next: obj[];
    prev: obj[];
    train: obj := null;
    outer_station: bool;

  State Top = FREE, WAITING_ACK, WAITING_COMMIT, WAITING_AGREE,
      RESERVED, TRAIN_PASSING, ABORTING

  Transitions:
    FREE -> WAITING_ACK { req(sender, dest, route) [(train=null OR
        sender=train) AND next[route] /= null] / next[route].req(self,
        next[route], route); }
```

```
FREE -> WAITING_COMMIT { req(sender, dest, route) [(train=null OR
    sender=train) AND next[route] = null] / prev[route].ack(self,
    prev[route], route); }
FREE -> FREE { req(sender, dest, route) [train/=null AND
    sender/=train] / sender.nack(self, sender, route); }

WAITING_ACK -> WAITING_COMMIT { ack(sender, dest, route)
    [prev[route] /= null AND train = null] / prev[route].ack(self,
    prev[route], route); }
WAITING_ACK -> WAITING_AGREE  { ack(sender, dest, route) [train /=
    null] / next[route].commit(self, next[route], route); }
WAITING_ACK -> FREE { nack(sender, dest, route_id) [(prev[route] =
    null) AND (train /= null)] / train.wait; }
WAITING_ACK -> FREE { nack(sender, dest, route_id) [prev[route] /=
    null] / prev[route].nack(self, prev[route], route); }
WAITING_ACK -> WAITING_ACK { req(sender, dest, route) /
    sender.nack(self, sender, route); }
WAITING_ACK -> ABORTING { aborting(sender, dest, route) /
    next[route].aborting(self, next[route], route); }

WAITING_COMMIT -> WAITING_AGREE  { commit(sender, dest, route)
    [next[route] /= null] /  next[route].commit(self, next[route],
    route); }
WAITING_COMMIT -> Top.RESERVED      { commit(sender, dest, route)
    [next[route] = null] / prev[route].agree(self, prev[route],
    route); }
WAITING_COMMIT -> WAITING_COMMIT { req(sender, dest, route) /
    sender.nack(self, sender, route); }
WAITING_COMMIT -> ABORTING { aborting(sender, dest, route)
    [next[route] /= null] / next[route].aborting(self, next[route],
    route); }
WAITING_COMMIT -> FREE { aborting(sender, dest, route) [next[route]
   = null] / prev[route].cancel(self, prev[route], route); }

WAITING_AGREE -> Top.RESERVED { agree(sender, dest, route)
    [prev[route] /= null AND train = null] / prev[route].agree(self,
    prev[route], route); }
WAITING_AGREE -> Top.RESERVED { agree(sender, dest, route) [train /=
    null] / train.start; }
WAITING_AGREE -> FREE { disagree(sender, dest, route_id)
    [(prev[route] = null) AND (train /= null)] / train.wait; }
WAITING_AGREE -> FREE { disagree(sender, dest, route_id)
    [prev[route] /= null] / prev[route].disagree(self, prev[route],
    route); }
WAITING_AGREE -> WAITING_AGREE { req(sender, dest, route) /
    sender.nack(self, sender, route); }
WAITING_AGREE -> ABORTING { aborting(sender, dest, route) /
    next[route].aborting(self, next[route], route); }
```

```
    Top.RESERVED -> TRAIN_PASSING { sensorOn(sender,dest,route) / train
        := sender; }
    Top.RESERVED -> FREE { disagree(sender, dest, route) / if
        next[route] /= null then { next[route].disagree(self,
        next[route], route) }; }
    Top.RESERVED -> FREE { sensorOff(sender,dest,route) / train := null;
        }
    Top.RESERVED -> Top.RESERVED { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    Top.RESERVED -> ABORTING { aborting(sender, dest, route)
        [next[route] /= null] / next[route].aborting(self, next[route],
        route); }
    Top.RESERVED -> FREE { aborting(sender, dest, route) [next[route] =
        null] / prev[route].cancel(self, prev[route], route); }

    TRAIN_PASSING -> FREE { sensorOff(sender,dest,route) [next[route] =
        null] / if(outer_station = True) { train := null; return null;}
        else { return self; } }
    TRAIN_PASSING -> FREE { sensorOff(sender,dest,route) [next[route] /=
        null] / train := null; }
    TRAIN_PASSING -> FREE { train_stopped(sender) [train = sender] }
    TRAIN_PASSING -> TRAIN_PASSING { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    TRAIN_PASSING -> ABORTING { aborting(sender, dest, route) /
        next[route].aborting(self, next[route], route); }

    ABORTING -> FREE { cancel(sender, dest, route) [train /= null] /
        train.aborted; }
    ABORTING -> FREE { cancel(sender, dest, route) [train = null AND
        prev[route] /= null] / prev[route].cancel(self, prev[route],
        route); }

end TrackCircuit
```

Listato B.3: Modello della classe Switch

```
Class Switch is

  Signals:
    req(sender: obj, dest: obj, route: int);
    ack(sender: obj, dest: obj, route: int);
    nack(sender: obj, dest: obj, route: int);
    commit(sender: obj, dest: obj, route: int);
    agree(sender: obj, dest: obj, route: int);
    disagree(sender: obj, dest: obj, route: int);
    aborting(sender: obj, dest: obj, route: int);
```

```
    cancel(sender: obj, dest: obj, route: int);

  Operations:
    sensorOn(sender: obj, dest: obj, route: int);
    sensorOff(sender: obj, dest: obj, route: int);

  Vars:
    next: obj[];
    prev: obj[];
    conf: bool[];
    reversed: bool := False;
    train: obj := null;
    requested_route: int;

  State Top = FREE, WAITING_ACK, WAITING_COMMIT, WAITING_AGREE,
      POSITIONING, RESERVED, TRAIN_PASSING, CHECK_POSITION, ABORTING
  State WAITING_ACK Defers req(sender: obj, dest: obj, route: int)

  Transitions:
    FREE -> WAITING_ACK { req(sender, dest, route) /
        next[route].req(self, next[route], route); }

    WAITING_ACK -> WAITING_COMMIT { ack(sender, dest, route) /
        prev[route].ack(self, prev[route], route); }
    WAITING_ACK -> FREE { nack(sender, dest, route) /
        prev[route].nack(self, prev[route], route); }
    WAITING_ACK -> WAITING_ACK { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    WAITING_ACK -> ABORTING { aborting(sender, dest, route) /
        next[route].aborting(self, next[route], route); }

    WAITING_COMMIT -> WAITING_AGREE { commit(sender, dest, route) /
        next[route].commit(self, next[route], route); }
    WAITING_COMMIT -> WAITING_COMMIT { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    WAITING_COMMIT -> ABORTING { aborting(sender, dest, route) /
        next[route].aborting(self, next[route], route); }

    WAITING_AGREE -> FREE { disagree(sender, dest, route) /
        prev[route].disagree(self, prev[route], route); }
    WAITING_AGREE -> WAITING_AGREE { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    WAITING_AGREE -> CHECK_POSITION { agree(sender, dest, route) /
        requested_route := route; }
    WAITING_AGREE -> ABORTING { aborting(sender, dest, route) /
        next[route].aborting(self, next[route], route); }
```

```
    CHECK_POSITION -> Top.RESERVED { - [reversed =
        conf[requested_route]] / prev[requested_route].agree(self,
        prev[requested_route], requested_route); }
    CHECK_POSITION -> POSITIONING { - [reversed /=
        conf[requested_route]] }
    CHECK_POSITION -> CHECK_POSITION { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    CHECK_POSITION -> ABORTING { aborting(sender, dest, route) /
        next[route].aborting(self, next[route], route); }

    POSITIONING -> Top.RESERVED { - / reversed := NOT reversed;
        prev[requested_route].agree(self, prev[requested_route],
        requested_route); }
    POSITIONING -> FREE  { - / prev[requested_route].disagree(self,
        prev[requested_route], requested_route);
        next[requested_route].disagree(self, next[requested_route],
        requested_route); }
    POSITIONING -> POSITIONING { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    POSITIONING -> ABORTING { aborting(sender, dest, route) /
        next[route].aborting(self, next[route], route); }

    Top.RESERVED -> TRAIN_PASSING { sensorOn(sender,dest,route) / train
        := sender; }
    Top.RESERVED -> FREE { disagree(sender, dest, route) /
        next[route].disagree(self, next[route], route); }
    Top.RESERVED -> Top.RESERVED { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    Top.RESERVED -> ABORTING { aborting(sender, dest, route) /
        next[route].aborting(self, next[route], route); }

    TRAIN_PASSING -> FREE { sensorOff(sender,dest,route) / train :=
        null; }
    TRAIN_PASSING -> TRAIN_PASSING { req(sender, dest, route) /
        sender.nack(self, sender, route); }
    TRAIN_PASSING -> TRAIN_PASSING { aborting(sender, dest, route)
        [train /= null AND sender = train] / train.retry; }

    ABORTING -> FREE { cancel(sender, dest, route) /
        prev[route].cancel(self, prev[route], route); }

end Switch
```

Listato B.4: Oggetti istanziati nella nostra simulazione

```
Objects
```

```
firenze: TrackCircuit (
  prev => [null, null, null, null, null, null, switchFi1, null,
      switchFi1, switchFi1, null, switchFi1, null, null, switchFi1,
      null, null, switchFi1],
  next => [switchFi1, null, switchFi1, switchFi1, null, switchFi1,
      null, null, null, null, null, null, switchFi1, switchFi1, null,
      null, null, null],
  train => meucci,
  outer_station => True);

switchFi1: Switch (
  prev => [firenze, null, firenze, firenze, null, firenze, switchFi2,
      null, switchFi2, switchFi2, null, switchFi2, firenze, firenze,
      pisa, null, null, pisa],
  next => [switchFi2, null, switchFi2, switchFi2, null, switchFi2,
      firenze, null, firenze, firenze, null, firenze, pisa, pisa,
      firenze, null, null, firenze],
  conf => [False, False, False, False, False, False, False, False,
      False, False, False, True, True, True, False, False, True],
  train => null);

switchFi2: Switch (
  prev => [switchFi1, null, switchFi1, switchFi1, null, switchFi1,
      prato, null, prato, porretta, null, porretta, null, null, null,
      null, null, null],
  next => [prato, null, prato, porretta, null, porretta, switchFi1,
      null, switchFi1, switchFi1, null, switchFi1, null, null, null,
      null, null, null],
  conf => [False, False, False, True, False, True, False, False,
      False, True, False, True, False, False, False, False, False,
      False],
  train => null);

porretta: TrackCircuit (
  prev => [null, null, null, switchFi2, null, switchFi2, null, null,
      null, null, switchBo2, switchBo2, null, null, null, null, null,
      null],
  next => [null, null, null, null, switchBo2, switchBo2, null, null,
      null, switchFi2, null, switchFi2, null, null, null, null, null,
      null],
  train => marconi,
  outer_station => False);

prato: TrackCircuit (
  prev => [switchFi2, null, switchFi2, null, null, null, null,
      switchBo2, switchBo2, null, null, null, null, null, null, null,
      null, null],
```

```
    next => [null, switchBo2, switchBo2, null, null, null, switchFi2,
        null, switchFi2, null, null, null, null, null, null, null,
        null],
    train => null,
    outer_station => False);

  switchBo2: Switch (
    prev => [null, prato, prato, null, porretta, porretta, null,
        switchBo1, switchBo1, null, switchBo1, switchBo1, null, null,
        null, null, null, null],
    next => [null, switchBo1, switchBo1, null, switchBo1, switchBo1,
        null, prato, prato, null, porretta, porretta, null, null, null,
        null, null, null],
    conf => [False, False, False, False, True, True, False, False,
        False, False, True, True, False, False, False, False, False,
        False],
    train => null);

  switchBo1: Switch (
    prev => [null, switchBo2, switchBo2, null, switchBo2, switchBo2,
        null, bologna, bologna, null, bologna, bologna, null, pisa,
        bologna, bologna, pisa, null],
    next => [null, bologna, bologna, null, bologna, bologna, null,
        switchBo2, switchBo2, null, switchBo2, switchBo2, null, bologna,
        pisa, pisa, bologna, null],
    conf => [False, False, False, False, False, False, False, False,
        False, False, False, False, False, True, True, True, True, False],
    train => null);

  bologna: TrackCircuit (
    prev => [null, switchBo1, switchBo1, null, switchBo1, switchBo1,
        null, null, null, null, null, null, null, switchBo1, null, null,
        switchBo1, null],
    next => [null, null, null, null, null, null, null, switchBo1,
        switchBo1, null, switchBo1, switchBo1, null, null, switchBo1,
        switchBo1, null, null],
    train => null,
    outer_station => True);

  pisa: TrackCircuit (
    prev => [null, null, null, null, null, null, null, null, null, null,
        null, null, switchFi1, switchFi1, switchBo1, switchBo1, null,
        null],
    next => [null, null, null, null, null, null, null, null, null, null,
        null, null, null, switchBo1, switchFi1, null, switchBo1,
        switchFi1],
    train => null,
    outer_station => False);
```

```
meucci Train (
  routes => [2,11,12,16],
  nodes => [firenze, switchFi1, switchFi2, prato, switchBo2,
      switchBo1, bologna, switchBo1, switchBo2, porretta, switchFi2,
      switchFi1, firenze, switchFi1, pisa, switchBo1, bologna],
  aborting => [False, False, False, False, False, False, True, False,
      False, False, False, False, False, False, False, False],
  stops => [bologna, firenze, pisa, bologna],
  node => firenze
);

marconi: Train (
  routes => [9,0,6,13],
  nodes => [porretta, switchFi2, switchFi1, firenze, switchFi1,
      switchFi2, prato, switchFi2, switchFi1, firenze, switchFi1, pisa,
      switchBo1, bologna],
  aborting => [False, True, False, False, False, False, False, False,
      False, False, False, False, False, False],
  stops => [firenze, prato, firenze, bologna],
  node => porretta
);
```

Listato B.5: Astrazioni per la verifica delle proprietà

```
Abstractions {
  Action $1($*) -> $1($*)

  State inState(meucci.ARRIVED) -> meucci_arrived
  State inState(marconi.ARRIVED) -> marconi_arrived

  State inState(meucci.ABORTING) -> meucci_aborting
  State inState(marconi.ABORTING) -> marconi_aborting

  State meucci.node = marconi.node AND meucci.node /= null -> crash

  State meucci.node = firenze -> meucci_in_firenze
  State meucci.node = prato -> meucci_in_prato
  State meucci.node = bologna -> meucci_in_bologna
  State meucci.node = porretta -> meucci_in_porretta

  State marconi.node = porretta -> marconi_in_porretta
  State marconi.node = firenze -> marconi_in_firenze
  State marconi.node = prato -> marconi_in_prato
  State marconi.node = bologna -> marconi_in_bologna

  State inState(switchFi2.POSITIONING) -> switchFi2_positioning
```

```
  State inState(switchBo2.POSITIONING) -> switchBo2_positioning
  State meucci.node = switchFi2 -> meucci_on_switchFi2
  State meucci.node = switchBo2 -> meucci_on_switchBo2
  State marconi.node = switchFi2 -> marconi_on_switchFi2
  State marconi.node = switchBo2 -> marconi_on_switchBo2
}
```

Listato B.6: Formule CTL di Safety

```
AG NOT crash

AG ((switchBo2_positioning IMPLIES NOT (marconi_on_switchBo2 OR
    meucci_on_switchBo2)) AND (switchFi2_positioning IMPLIES NOT
    (marconi_on_switchFi2 OR meucci_on_switchFi2)))

EF AG (marconi_arrived AND meucci_arrived)
```

Acronimi

2PC *Two Phase Commit Protocol*

UMC *UML on-the-fly Model Checker*

CTL *Computation Tree Logic*

ACTL *Action-based CTL*

Bibliografia

[1] S. Gnesi and F. Mazzanti, "On the fly model checking of communicating UML State Machines," in *Second ACIS International Conference on Software Engineering Research Management and Applications (SERA2004) (Los Angeles, USA, 5-7 May 2004)*, 2004.

[2] M. Paolieri, "Modellazione di un sistema di Interlocking Distribuito tramite lo strumento UMC," 2010.

[3] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '81. New York, NY, USA: ACM, 1981, pp. 164–176. [Online]. Available: http://doi.acm.org/10.1145/567532.567551

[4] U. Monile, "Appunti del corso di Informatica Industriale 2."

[5] S. Gnesi and F. Mazzanti, "A model checking verification environment for UML statechart," in *XLIII Congresso Annuale AICA*, ottobre 2005.

[6] O. M. Group, *OMG Unified Modeling Language Superstructure*, 2nd ed. Object Managment Group, 2009, pp. 525–586.

[7] F. Mazzanti, "Designing UML Models with UMC." [Online]. Available: http://fmtlab.isti.cnr.it/umc/V3.7/UMC-Models-v37.pdf

[8] ——, "UMC Logics." [Online]. Available: http://fmtlab.isti.cnr.it/umc/V3.7/UMC-Logics.pdf

[9] R. Meolic, T. Kapus, and Z. Brezocnik, "Ctl and actl patterns," in *EUROCON'2001, Trends in Communications, International Conference on.*, vol. 2, 2001, pp. 540 –543 vol.2.

[10] UML on-the-fly Model Checker. [Online]. Available: http://fmtlab.isti.cnr.it/umc/