Robots Cluster

Modellazione e verifica di un algoritmo 2PC-like per il comportamento cooperativo di un pool di robot

Angelo D'Amante, Marco Di Rienzo, Kevin Maggi

Marzo 2022

Sommario

In questo progetto ci siamo occupati della modellazione e della verifica di un algoritmo 2PC-like per la cooperazione di un pool di robot al fine di coordinarsi nell'esecuzione di un'azione comune. Successivamente abbiamo approfondito lo studio sull'algoritmo, prendendo in considerazione diversi possibili sviluppi, tra cui la fault tolerance, la scalabilità del sistema e la possibilità di pool multipli contemporanei.

Indice

1 Introduzione

Questo progetto affronta un caso di studio incentrato sul comportamento cooperativo tra robot, al fine di coordinarsi per muovere oggetti pesanti.

I robot costituiscono i nodi di un sistema totalmente distribuito senza elementi di centralizzazione.

Il sistema è stato modellato su Simulink usando il formalismo Stateflow, descrivendo i robot come *Extended Finite State Machine*.

Il progetto si è articolato su più fasi:

1. **modellazione** dei nodi robot e del sistema;
2. **verifica formale** delle proprietà di *validity*, *liveness* e *agreement* del sistema;
3. studio della **scalabilità**;
4. estensione alla possibilità di formazione di **pool multipli contemporanei**;
5. estensione alla possibilità di **tollerare fallimenti** di vario tipo.

1.1 Formulazione del problema

Un'entità esterna invia una *movement request* a un robot, che diventa così coordinatore e fa partire l'algoritmo. La richiesta di cooperazione viene inoltrata al successivo robot, secondo una topologia lineare, fino al raggiungimento di m robot partecipanti o alla conclusione che è impossibile formare un pool di m robot.

Nel caso di avvenuta formazione del pool un messaggio di *agreement* viene mandato indietro fino al coordinatore, il quale invia un *acknowledgement* all'entità esterna. In caso contrario invece il coordinatore riceverà un messaggio di *disagreement* e invia un *not-acknowledgement* all'entità esterna.

1.1.1 Assunzioni semplificative

Vengono fatte alcune assunzioni:

- non viene considerata nella modellazione l'azione comune, ma viene solo astratta da una busy wait;
- similmente non viene modellata la ragione per cui un robot decide di non partecipare a un pool (eccessiva distanza dal luogo dell'azione, batteria insufficiente, fallimento generico, etc.), riducendo tale scelta a un fattore non deterministico;
- il robot coordinatore (quello che comunica con l'entità esterna) è fissato.

2 Modellazione (obj1)

Il primo passo è stato quello di definire il protocollo in ogni suo aspetto e modellare sia i robot come singola unità sia l'intero sistema.

2.1 Protocollo

Il protocollo ricalca la struttura del *2PC protocol* con topologia lineare, in cui i robot sono virtualmente numerati da 0 a $n-1$ e il coordinatore è sempre lo 0.

- **Prepare phase**:
  Quando il coordinatore riceve la *movement request* dall'entità esterna, fa partire l'algoritmo inviando un messaggio "contatore" al robot successivo. Ogni robot quando riceve tale messaggio decide se unirsi al pool oppure no.
  - **Join**:
    Nel caso in cui il robot k si unisca, incrementa il contatore e, se questo è pari a m, parte la seconda fase con un messaggio di *agreement* in back flow; altrimenti inoltra il messaggio contatore incrementato al robot $k+1$.
  - **Not join**:
    Nel caso in cui decida di non unirsi al pool semplicemente inoltra il messaggio contatore invariato.
    La topologia lineare è chiusa ad anello, per cui quando il messaggio contatore torna al robot coordinatore, questo conclude l'impossibilità di formare il pool e fa partire la seconda fase con un messaggio di *disagreement* in back flow.

Nota bene

La chiusura ad anello non trasforma la topologia in circolare, perché in una topologia circolare il flusso informativo viaggia in una sola direzione, mentre in questo caso abbiamo due flussi informativi, quello delle richieste di coordinazione e quello del *(dis)agreement*, che viaggiano in sensi opposti. Questa soluzione consente ad ogni robot di essere unaware del suo numero, perché l'ultimo robot non ha necessità di sapere di esserlo: infatti è responsabilità del coordinatore (che già sa di esserlo) concludere dell'impossibilità di formazione del pool e far partire il back flow.

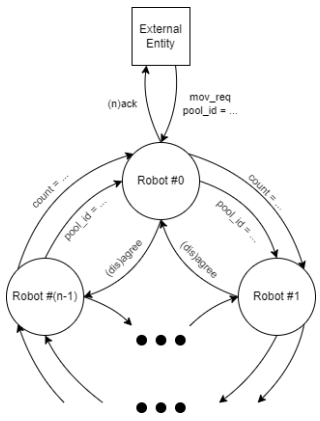

Figura 2.1: Topologia del sistema

In ogni istante il messaggio contatore in uscita dai robot contiene il numero di robot che fino a quel momento hanno espresso la propria disponibilità ad unirsi al pool. Inoltre abbinato al messaggio contatore c'è sempre un messaggio con l'id del pool da formare comunicato dall'entità esterna.

- **Commit phase**:
  - **Success**:
    Quando un robot completa il pool, come detto, fa partire un messaggio di *agreement* in back flow: quando i robot lo ricevono lo inoltrano al robot precedente e, se avevano dichiarato la disponibilità ad unirsi al pool, iniziano l'azione comune. Quando il coordinatore riceve l'*agreement* invia all'entità esterna il messaggio di *acknowledgement*.

  - **Failure**:
    Quando il robot coordinatore conclude l'impossibilità di formare un pool, come detto, fa partire un messaggio di *disagreement* in back flow: quando i robot lo ricevono lo inoltrano e, eventualmente, fanno *rollback*. Quando il coordinatore riceve il *disagree*, e quindi tutti sono stati informati, invia all'entità esterna il messaggio di *not acknowledgement*.

Esempio 2.1 → formazione di un pool

$$[n = 4, m = 2, join = \{\#0 : true, \#1 : false, \#2 : true, \#3 = false\}]$$

In figura 2.2 è illustrato un esempio in cui la formazione del pool avviene con successo: il `Robot #0` riceve la *movement request*, diventando coordinatore, e invia il contatore a `Robot #1`, il quale non si unisce e inoltra il messaggio a `Robot #2`; quest'ultimo accetta, completando il pool e iniziando il back flow dei messaggi *agree*: quando questo arriva al coordinatore, viene inviato l'*ack* all'entità esterna.

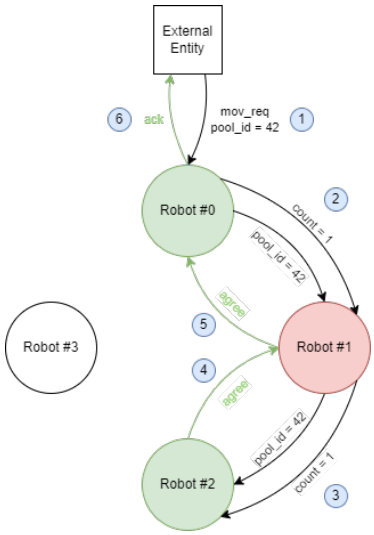

Figura 2.2: Sequenza dei messaggi in caso di formazione del pool

Esempio 2.2 → impossibilità di formazione di un pool

$$[n = 4, m = 2, join = \{\#0 : true, \#1 : false, \#2 : false, \#3 = false\}]$$

In figura 2.3 invece non è possibile formare il pool: infatti il contatore arriva, senza che si completi il pool, fino a `Robot #0`, il quale dà via al back flow di *disagree*; quando questo torna al coordinatore, viene inviato il *nack* all'entità esterna.

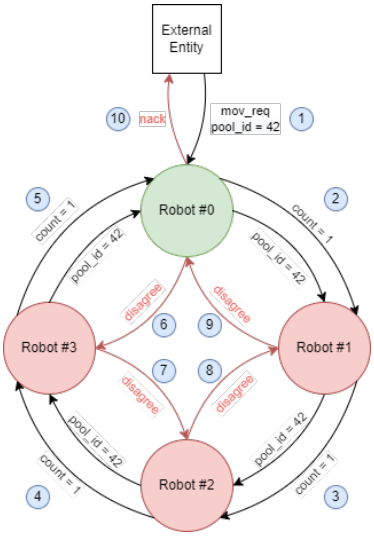

Figura 2.3: Sequenza dei messaggi in caso di impossibilità di formazione del pool

2.2 Modello del sistema

In figura 2.4 si vede il sistema così come lo vede l'entità esterna (nel caso di $n = 4$): soffermiamoci su alcuni aspetti.

Input

- *mov_request*: questo segnale corrisponde alla *movement request*, sta sempre a 0 e vale 1 per un tick temporale quando l'entità esterna invia la richiesta;
- *pool_id*: corrisponde all'id della richiesta comunicato dall'entità esterna. Viene considerato solamente in corrispondenza del *mov_request* (grazie allo switch).

Output

- *ack*: è il segnale di *(not)acknowledgement* inviato dal robot coordinatore all'entità esterna. Vale:

$$\begin{cases} +1 & ack \\ -1 & nack \\ 0 & \text{altrimenti} \end{cases}$$

- *pool_id_out* per ogni robot: normalmente vale 0, assume il valore del pool id quando un robot si unisce al pool o inoltra una richiesta al robot successivo;

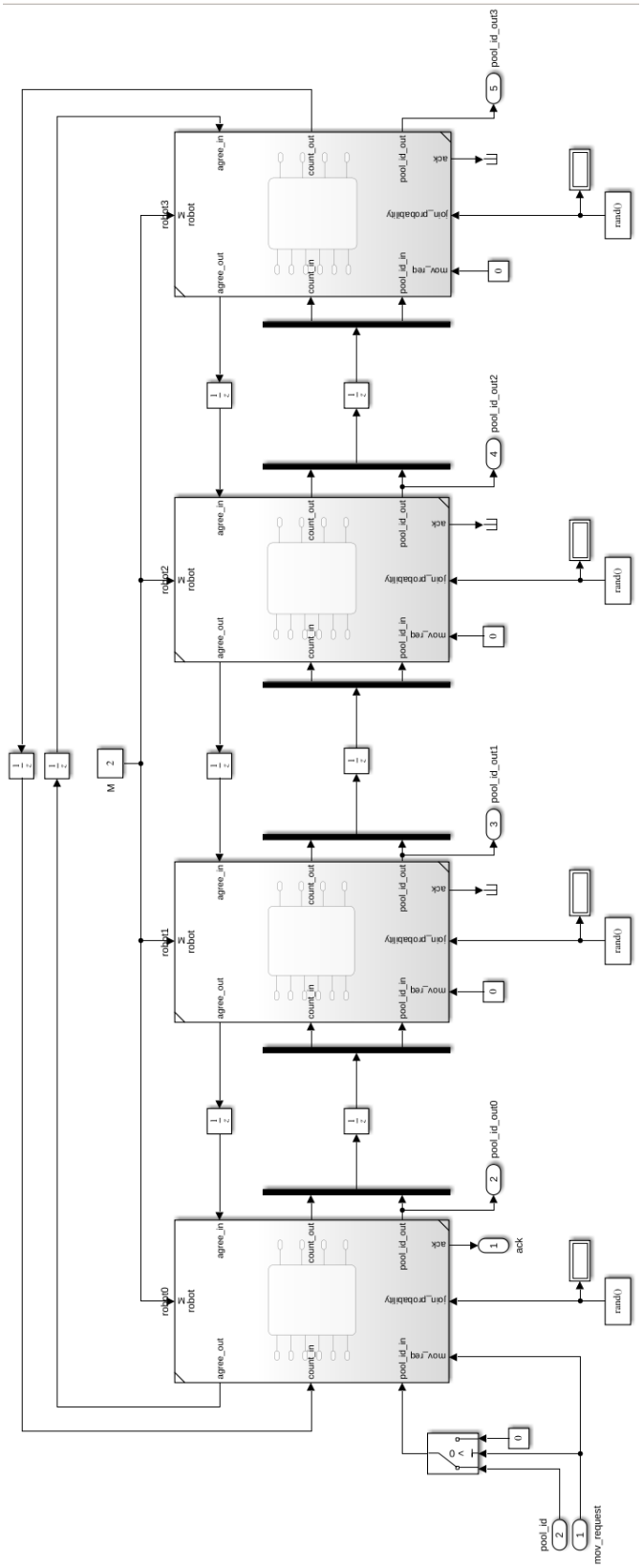

Figura 2.4: Modello del sistema visto dell'entità esterna

Parametri dei robot

- il numero m di unità necessarie a compiere un'azione coordinata;
- `rand()` per simulare la scelta non deterministica di join.

> **Nota bene**
>
> Come vedremo più avanti i robot sono tutti istanze dello stesso modello, che siano coordinatore o no, quindi tutti hanno l'input per la *mov_request* e l'output per l'*ack*; ovviamente avendo assunto che il coordinatore sia fissato, i robot diversi dal primo, hanno l'output a terminazione e l'input costante a 0.

Come anticipato i segnali *count* e *pool_id* viaggiano insieme tra un robot e l'altro e questo è evidenziato con l'uso di multiplexer e demultiplexer.

Possiamo vedere anche che il flusso dei messaggi di *agree* viaggia in senso inverso rispetto agli altri, questo perché la topologia rimane lineare e quindi si tratta di un back flow.

Notiamo poi l'inserimento di blocchi di ritardo tra le uscite di un robot e le entrate corrispondenti del successivo. Questo accorgimento evita anche il formarsi di loop algebrici.

2.3 Modello del robot

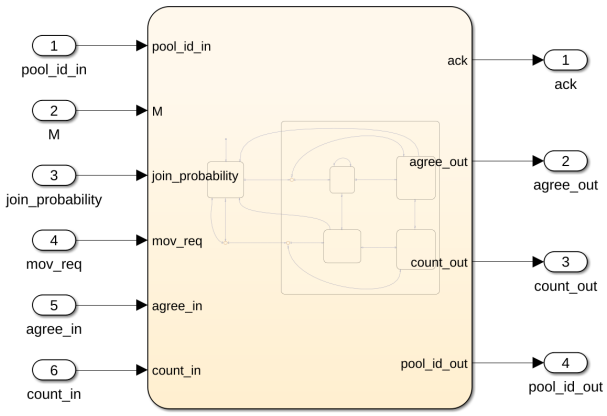

Figura 2.5: Input/output del robot

2.3.1 Input/output

Input

- M: il numero di unità da cui deve essere formato il pool;
- *join_probability*: la probabilità che un robot si unisca al pool;
- *mov_req*: il segnale di *movement request* proveniente dall'entità esterna;
- *count_in*: il messaggio "contatore" che ogni robot riceve dal precedente fino a che il pool non è completo;

- *pool_id_in*: l'id del pool relativo alla richiesta corrente;
- *agree_in*: segnali di *(dis)agreement* sul back flow ricevuti dal robot successivo.

Output

- *count_out*: il messaggio "contatore" che ogni robot invia al successivo se il pool non è completo;
- *pool_id_out*: l'id del pool relativo alla richiesta corrente;
- *agree_out*: segnali di *(dis)agreement* sul back flow inviati al robot precedente;
- *ack*: segnale di *(not) acknowledgement* inviato all'entità esterna.

2.3.2 Statechart

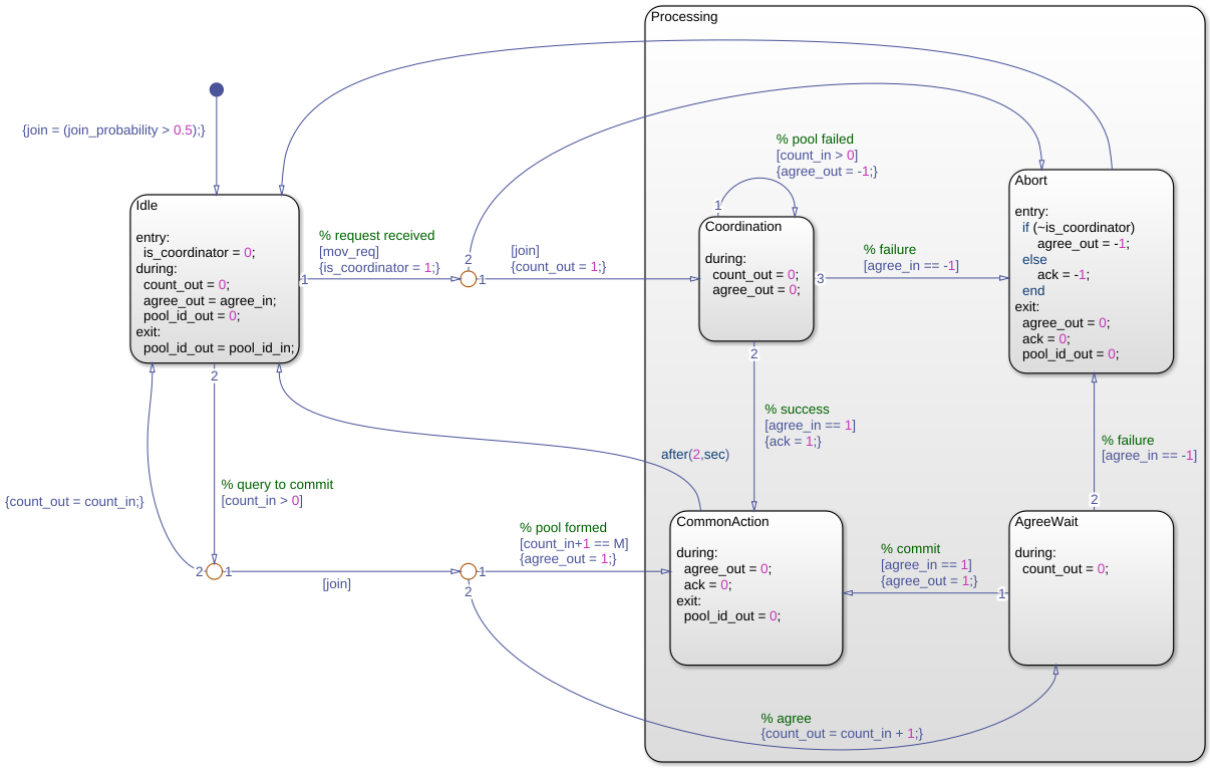

Figura 2.6: Statechart che modella il comportamento dei robot

All'inizio dell'esecuzione ogni robot decide, sulla base dell'input *join_probability* (che ricordiamo è random), se è disponibile ad entrare in un pool o meno. Per semplicità abbiamo assunto che questa decisione venga tenuta per tutta la durata dell'esecuzione.

- **Idle**:

  In questo stato il robot è in attesa di un segnale *movement request* dall'entità esterna o di un segnale *count* dal robot precedente.

  → **request received**:

Quando riceve una *movement request*, il robot diventa coordinatore e, se è disponibile, entra nello stato **Coordination** inviando il segnale *count* al robot successivo, altrimenti entra nello **Abort**.

→ **query to commit**:
Quando riceve un messaggio *count*, se non è disponibile lo inoltra invariato e torna in **Idle**, altrimenti fa join.

→ **pool formed**:
Se, incrementando il contatore, il robot completa il pool, inizia il back flow di messaggi *agree* e l'azione comune, andando nello stato **CommonAction**.

→ **agree**:
Se invece il pool non è ancora completo, inoltra il messaggio contatore incrementato e va in uno stato **AgreeWait**.

- **Coordination**:
  In questo stato il robot è coordinatore e quindi sta in attesa che arrivi un segnale di *(dis)agreement* sul back flow o il messaggio contatore che lo informa dell'impossibilità di formare il pool.

  → **success**:
  Se arriva un segnale *agree* positivo, inizia l'azione comune andando nello stato **CommonAction** e conclude il protocollo inviando il segnale *ack* positivo in output.

  → **failure**:
  Se arriva un segnale *agree* negativo, va nello stato **Abort**.

  → **pool failed**:
  Se gli ritorna, dall'ultimo robot, il messaggio contatore, conclude dell'impossibilità di formare il pool, quindi inizia il back flow di segnali *agree* negativi.

- **AgreeWait**:
  In questo stato un robot ha dichiarato la sua disponibilità a partecipare a un pool e aspetta la seconda fase del protocollo, ovvero un segnale sul back flow.

  → **commit**:
  Se arriva un segnale di *agree* positivo, lo inoltra al robot precedente e inizia l'azione comune.

  → **failure**:
  Se arriva un segnale di *agree* negativo va nello stato **Abort**.

- **Abort**:
  In questo stato se il robot è il coordinatore si occupa di inviare il segnale *ack* negativo all'entità esterna, altrimenti propaga il segnale *agree* negativo al robot precedente. Dopodiché torna subito allo stato **Idle**.

- **CommonAction**:

In questo stato il robot simula l'esecuzione dell'azione comune (per semplicità abbiamo assunto una durata fissata di 2 secondi), prima di tornare allo stato **Idle**.

> **Nota bene**
>
> Una volta terminata l'azione comune, grazie al fatto che i robot tornano nello stato di idle, il sistema è pronto a ricevere una nuova *movement request* dall'entità esterna, essendo di fatto tornato alla configurazione iniziale.

Il superstato **Processing** in realtà non ha alcuna funzione specifica, lo abbiamo introdotto solamente per chiarezza espositiva, differenziando i momenti in cui un robot sta in idle da quelli in cui sta elaborando una richiesta di formazione di un pool.

2.4 Simulazione

Con Simulink abbiamo potuto simulare il comportamento del sistema, emulando l'entità esterna con un segnale di *movement request* e *pool_id*. Inoltre abbiamo sostituito nel sistema i `rand()` con degli input dall'esterno per poter forzare la disponibilità dei robot a fare join, come si vede in figura 2.7.

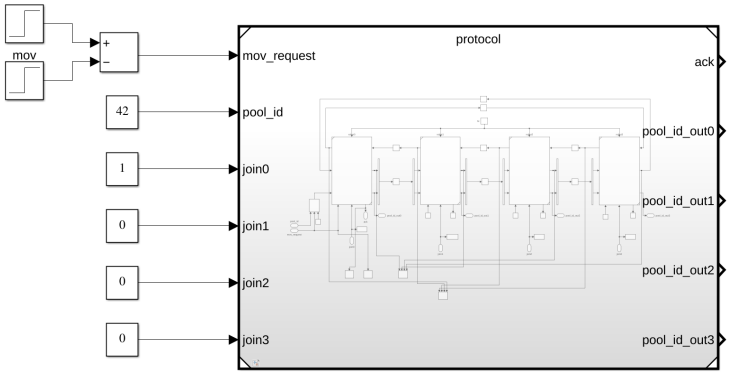

Figura 2.7: Interfaccia del sistema con l'entità esterna

Esempio 2.3 → formazione di un pool

$[n = 4, m = 2, join = \{\#0 : true, \#1 : false, \#2 : true, \#3 = false\}]$

Facendo riferimento all'esempio 2.1, simulando il modello si possono osservare i segnali in figura 2.8.

Ricordiamo che in questo caso `Robot #0` e `Robot #2` sono disponibili a fare join, mentre `Robot #1` e `Robot #3` no. `Robot #0` invia il messaggio contatore, che viene solamente inoltrato da `Robot #1` e ricevuto da `Robot #2`; quest'ultimo fa join e completa il pool, dando il via al segnale *agree* in back flow, che quando raggiunge `Robot #0` gli fa inviare il segnale *ack*.

Figura 2.8: Segnali in caso di formazione del pool

Esempio 2.4 → impossibilità di formazione di un pool

$[n = 4, m = 2, join = \{\#0 : true, \#1 : false, \#2 : false, \#3 = false\}]$

Facendo riferimento all'esempio 2.2, simulando il modello si possono osservare i segnali in figura 2.9.

Ricordiamo che in questo caso solo `Robot #0` è disponibile a fare join. `Robot #0` invia il messaggio contatore, che viene solamente inoltrato da `Robot #1`, `Robot #2` e `Robot #3`; quando torna al coordinatore, questo dà il via al segnale *disagree* in back flow, che quando torna a `Robot #0` gli fa inviare il segnale *nack*.

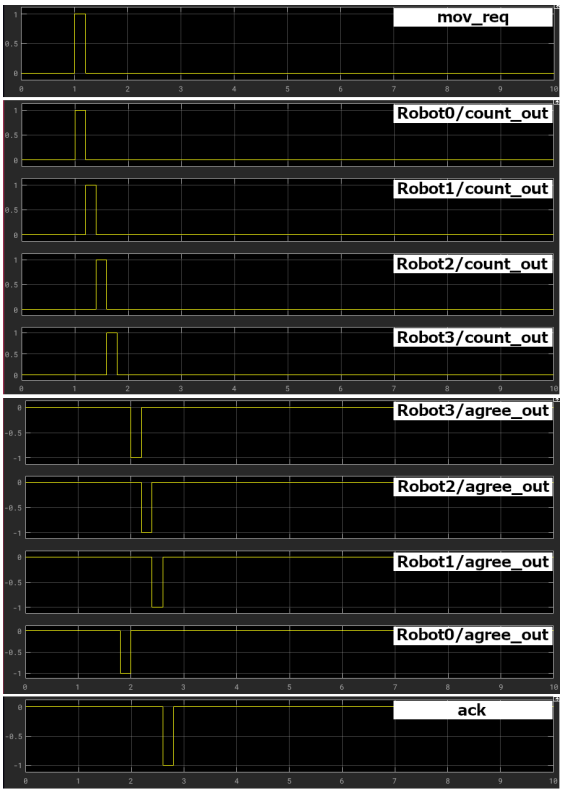

Figura 2.9: Segnali in caso di impossibilità di formazione del pool

3 Verifica formale (obj2)

Terminata la fase di modellazione, abbiamo provveduto alla verifica formale del sistema. Questa è avvenuta grazie allo strumento Design Verifier di Simulink. Il Design Verifier mette a disposizione alcuni blocchi per consentire la verifica delle proprietà:

- Verification Subsystem: dove modellare la proprietà da verificare;
- Proof Assumption: che permette di fare delle assunzioni sugli input del sistema;
- Proof Objective: che permette di verificare il valore di un output del sistema (eventualmente elaborato).

Il segnale *mov_req*, dato che deve essere un rect di durata un tick temporale il cui momento di arrivo è ininfluente, lo abbiamo fissato manualmente. Il pool id e i join invece non hanno nessun vincolo e quindi è stato possibile usare i blocchi Proof Assumption.

Nota bene

I blocchi Proof Assumption hanno un comportamento diverso da quello che ci serviva per vincolare i join (che devono essere 1 per forzare l'unione al pool o 0 per forzare la non disponibilità): infatti è possibile restringere il dominio di valori che può assumere il segnale, ma questo cambia valore (all'interno del dominio) ad ogni tick temporale. Nel nostro caso invece c'era la necessità che il valore fosse mantenuto costante per tutta la durata dell'esecuzione. Per questo motivo abbiamo settato i blocchi Proof Assumption a un solo valore (costringendo così il segnale ad essere costante) e abbiamo eseguito la verifica per ogni combinazione di valori.

Le proprietà che abbiamo verificato sono:

- **validity**: se un pool di robot si accorda, allora tutti i robot del pool hanno deciso di unirsi al pool;
- **liveness**: se un coordinatore riceve una *movement request*, allora prima o poi riceverà un acknowledgement (positivo o negativo che sia);
- **agreement**: non è possibile che si formino due pool separati per due diverse azioni.

3.1 Validity

Anzitutto abbiamo espresso la proprietà in formula logica:

$$ack \Rightarrow \forall i : (pool\_id_i \neq 0 \Rightarrow join_i) \tag{3.1}$$

che è equivalente a:

$$ack \Rightarrow \forall i : (\neg(pool\_id_i \neq 0) \vee join_i) \tag{3.2}$$

Dove la precondizione (ack) traduce l'accordo del pool e la postcondizione indica che se un robot si è unito ad un pool ($pool\_id_i \neq 0$), allora doveva essere disponibile ad entrarci ($join_i$).

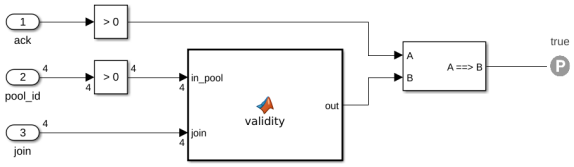

Figura 3.1: Verification Subsystem per la verifica della proprietà validity

Questo si è tradotto nel Verification Subsystem in figura 3.1, dove il comparatore con 0 è giustificato dal fatto che per ipotesi (che verrà formalmente verificata nella proprietà di agreement) si ha al più un solo pool, dunque non interessa quale sia il pool id di un robot, ma solamente che prenda parte ad un pool.

La funzione MATLAB è quella riportata di seguito e traduce la seconda parte dall'implicazione in formula 3.2:

```
function out = validity(in_pool, join)
    out = all((in_pool & join) == in_pool);
end
```

Nota bene

Nella funzione MATLAB non sarebbe stato sufficiente il controllo `join == in_pool`, perché potrebbero esserci dei robot disponibili a fare join, ma a cui non arriva mai la richiesta perché il pool si completa prima. Per questo motivo la condizione è diventata `(in_pool & join) == in_pool`.

Il modello per la verifica è quindi quello riportato in figura 3.2.

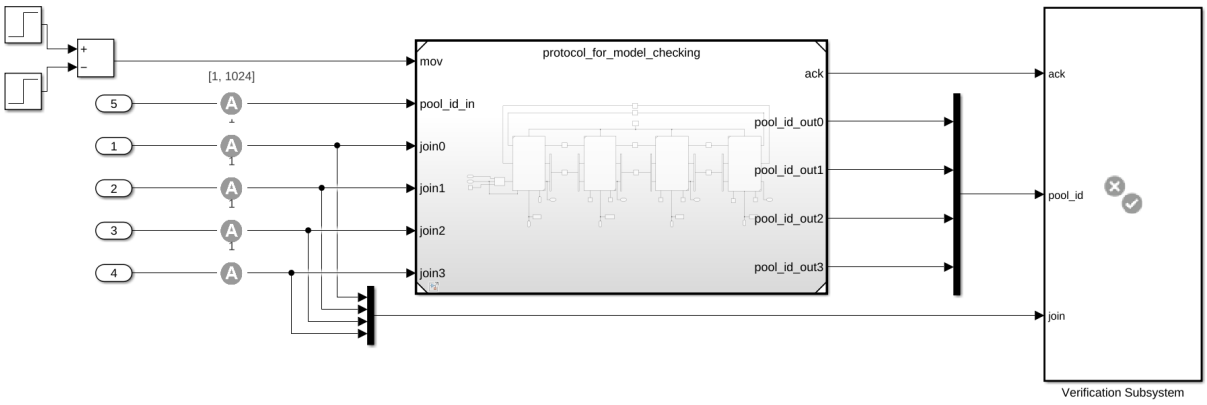

Figura 3.2: Modello per la verifica della proprietà validity

3.1.1 Risultati

Questo modello è stato eseguito dal Design Verifier in modalità Prove Properties per ogni combinazione di input dei join. Ogni verifica ha dato esito positivo, confermando la validità della proprietà.

3.2 Liveness

Anzitutto abbiamo espresso la proprietà in formula logica:

$$mov \Rightarrow \Diamond(ack \vee nack) \tag{3.3}$$

Nota bene → LTL in Simulink

Qua compare un frammento di logica LTL (in particolare l'operatore ***Eventually***), ma il Design Verifier di Simulink non supporta la logica LTL, se non quella traducibile in assertion [3]. Questo perché il Design Verifier verifica che una proprietà sia soddisfatta sempre, in ogni istante temporale; questo rende semplice verificare una proprietà con l'operatore ***Always***, ma non con l'operatore ***Eventually***.

Quello che abbiamo fatto è stato negare la proprietà, in modo da trasformare l'Eventually in Always, e verificare che questa fosse sempre, prima o poi, falsificata.
Confutando la proprietà negata per ogni input, dimostriamo così la proprietà originale.

La proprietà 3.3 diventa:

$$mov \Rightarrow \neg(\Box(\neg(ack \vee nack))) \tag{3.4}$$

Non considerando l'operatore Always e cercando un controesempio, la formula da tradurre in Simulink diventa $mov \Rightarrow \neg(ack \vee nack)$, come si vede in figura 3.3, dove il blocco extender serve a identificare tutti gli istanti successivi al segnale *mov* (dato che questo dura un solo tick) e la condizione $\neg(ack \vee nack)$ si traduce in `ack == 0`.

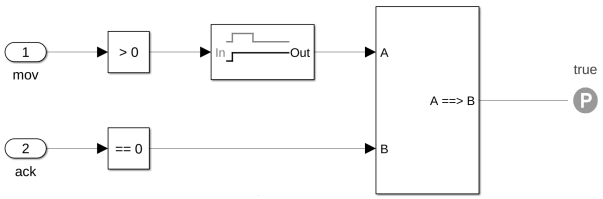

Figura 3.3: Verification Subsystem per la verifica della proprietà liveness

In altre parole dimostriamo essere falsa, quindi impossibile, la situazione in cui dopo il segnale *mov*, non arriva nessun segnale *(n)ack*, il che rende vera la proprietà.

Il modello per la verifica è quindi quello riportato in figura 3.4.

3.2.1 Risultati

Questo modello è stato eseguito dal Design Verifier in modalità Prove Properties per ogni combinazione di input dei join. Ogni verifica ha dato esito negativo, confermando la validità della proprietà originale.

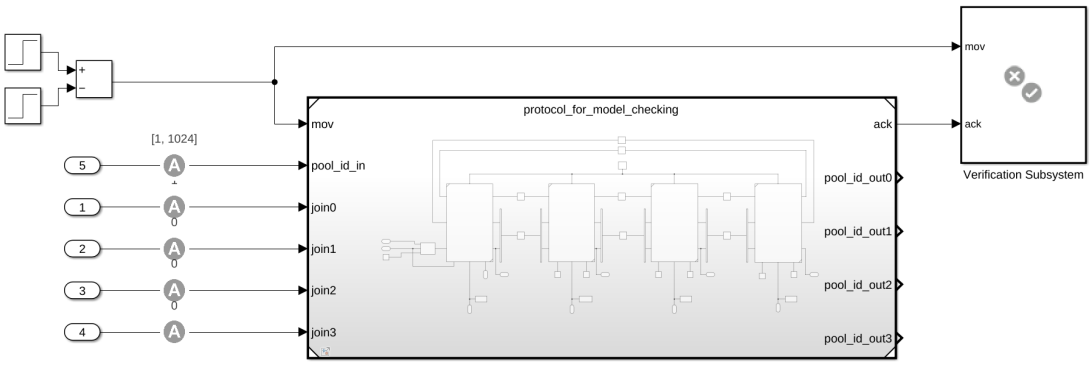

Figura 3.4: Modello per la verifica della proprietà liveness

3.3 Agreement

Anzitutto abbiamo espresso la proprietà in formula logica:

$$\forall i, j (pool\_id_i == 0 \vee pool\_id_j == 0 \vee pool\_id_i == pool\_id_j) \tag{3.5}$$

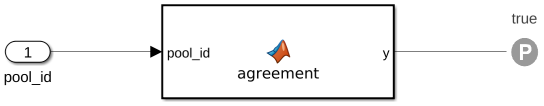

Figura 3.5: Verification Subsystem per la verifica della proprietà agreement

Questo si è tradotto nel Verification Subsystem in figura 3.5, dove la funzione MATLAB che traduce la formula 3.5 è la seguente:

```
function y = agreement(pool_id)
    y = all((pool_id == max(pool_id)) | (pool_id == 0));
end
```

Da notare che, per semplicità, anziché confrontare ogni coppia di pool id, confrontiamo ognuno di essi con il massimo: il risultato è il medesimo.

Il modello per la verifica è quindi quello riportato in figura 3.6.

3.3.1 Risultati

Questo modello è stato eseguito dal Design Verifier in modalità Prove Properties per ogni combinazione di input dei join. Ogni verifica ha dato esito positivo, confermando la validità della proprietà.

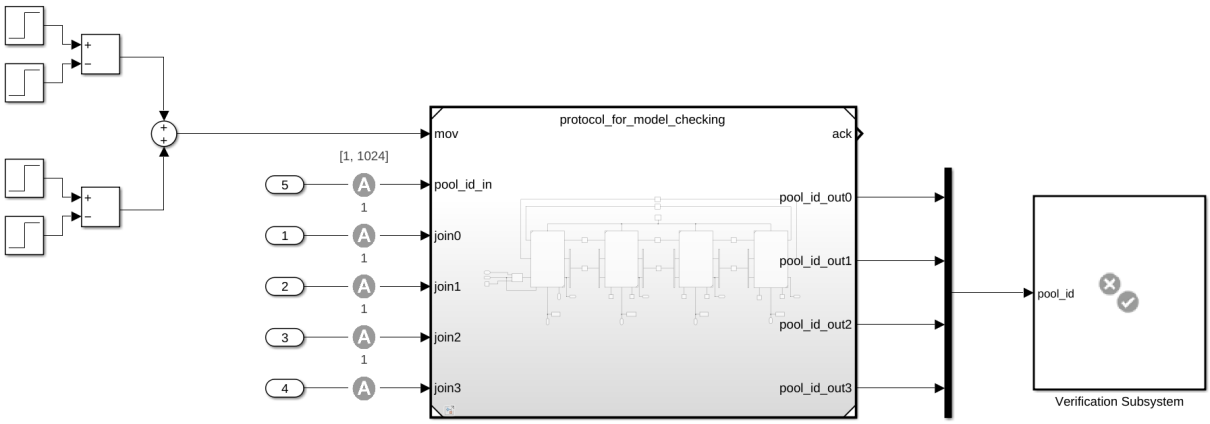

Figura 3.6: Modello per la verifica della proprietà agreement

Riepilogo

Proprietà	**Esito**	**Tempo di computazione** (best case/worst case)
Validity	Positivo	$\sim 15s / \sim 2500s$
Liveness	Negativo[1]	$\sim 15s / \sim 15s$
Agreement	Positivo	$\sim 2700s / \sim 2700s$

Tabella 3.1: Riepilogo degli esiti delle verifiche formali

Dove per "best case" si intende la configurazione di join che rende minimo il tempo di verifica e per "worst case" quello che lo rende massimo.

[1] Falsificazione della negazione

4 Scalabilità (obj3)

Siamo poi passati a studiare la scalabilità del sistema, prendendo in considerazione sia la facilità di definizione e di uso del modello, sia la possibilità di eseguire la verifica formale.

4.1 Modello

Il modello risulta particolarmente semplice da estendere in quanto aggiungere dei robot al sistema richiede solamente la modifica di pochi collegamenti.

Di contro, però, risulta presto di difficile utilizzo perché la complessità grafica cresce molto rapidamente. Infatti non solo nel modello crescono il numero di collegamenti, elementi di ritardo e altri componenti accessori, ma aumentano anche il numero di output (come si vede in figura 4.1), complicando anche l'interfaccia con l'entità esterna (nel caso della verifica formale anche i join in input).

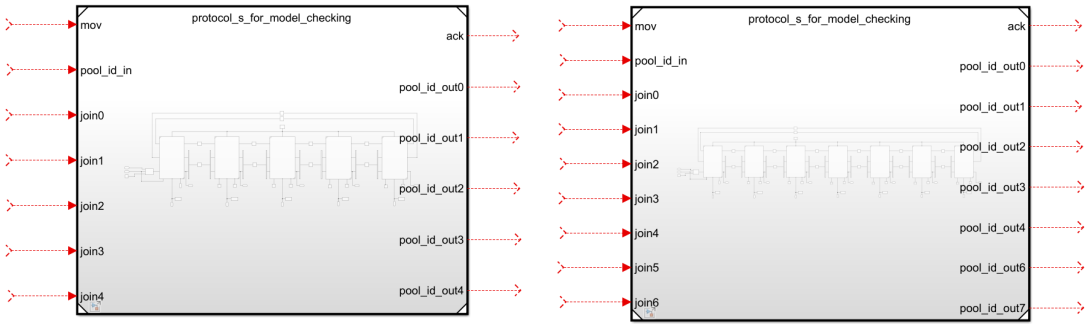

Figura 4.1: Esempio di interfaccia esterna con, rispettivamente, 5 e 7 robot

4.2 Verifica formale

Per quanto riguarda la verifica formale dobbiamo considerare due aspetti: il numero di verifiche e il tempo di elaborazione.

Infatti, come abbiamo spiegato nel capitolo 3, per verificare formalmente una proprietà dobbiamo lanciare il Design Verifier per ogni combinazione dei join (che deve essere fissato a 0/1 per forzare il join/not join di ogni robot). Ovviamente aggiungere un robot fa crescere esponenzialmente il numero di combinazioni da provare e già questo è un aspetto altamente non scalabile.

Poi dobbiamo inesorabilmente considerare il tempo necessario al Design Verifier per eseguire la verifica. Aggiungendo un solo robot abbiamo registrato incrementi non trascurabili, sia in termini relativi che in termini assoluti. Questo vuol dire che aggiungendo più robot, come ci si aspetta in un caso reale, la possibilità di eseguire la verifica formale svanisce velocemente. Anche questo aspetto dunque è non scalabile.

In tabella 4.1 è possibile confrontare i tempi necessari alla verifica (di una sola combinazione dei join) per i sistemi di 4 e 5 robot.

	4 robot ($m = 2$)		**5 robot** ($m = 3$)	
	best case	worst case	best case	worst case
validity	$\sim 15s$	$\sim 2500s$	$\sim 20s$	$\sim 3900s$
liveness	$\sim 15s$	$\sim 15s$	$\sim 17s$	$\sim 26s$
agreement	$\sim 2600s$	$\sim 2700s$	$\sim 5400s$	$\sim 6000s$

Tabella 4.1: Confronto tra i tempi necessari alla verifica

Conclusioni

In conclusione il modello è abbastanza scalabile se si tratta di usare il modello in Simulink, ma è altamente non scalabile quando si tratta di eseguire la verifica formale delle proprietà (sebbene dipenda dalle proprietà, perché per alcune i tempi assoluti non sono elevati e dunque anche se non scala bene rimane comunque fattibile eseguire la verifica).

5 Coordinatori multipli (obj4)

Inizialmente era stata fatta l'assunzione di coordinatore singolo e fissato. Per questo capitolo rilassiamo questa assunzione permettendo ad ogni robot di agire come coordinatore facendo partire un *agreement run* quando riceve una *movement request*, permettendo quindi pool simultanei.

5.1 Assunzioni

Nel progettare il protocollo abbiamo fatto due assunzioni:

- l'entità esterna invia una *movement request* solamente dopo aver ricevuto il *(not) acknowledgement* relativo alla richiesta precedente. Nel caso in cui venga inviata un richiesta senza rispettare il vincolo temporale, questa viene ignorata dal sistema;
- è l'entità esterna a decidere a quale robot inviare una *movement request*, il quale può dichiararsi disponibile, caso in cui avvia l'*agreement run*, oppure no, caso in cui ritorna subito *not acknowledgement*. Non è quindi previsto un **leader election protocol**.

5.2 Protocollo

Rispetto al protocollo descritto nel capitolo 2 servono delle modifiche:

- il robot k comunica con il robot precedente $(k-1)\mathrm{mod}n$ e col successivo $(k+1)\mathrm{mod}n$;
- ogni robot può ricevere *movement request* dall'entità esterna, momento in cui (se accetta) diventa coordinatore, e a questa restituirà al termine del protocollo il messaggio di *(n)ack*;
- ogni robot che è già impegnato in un'azione comune, per via della topologia lineare, deve partecipare ad un eventuale altro *agreement run* inoltrando i messaggi tra il robot precedente a lui e il successivo, altrimenti si interromperebbe la "catena" e non sarebbe possibile formare un nuovo pool.

Nota bene

Da notare che, siccome ogni robot comunica con l'entità esterna, il segnale *pool_id_in* può arrivare sia da questa che dal robot precedente. La gestione di questo aspetto è stata modellata a livello di sistema con uno switch: il robot legge in ingresso il valore comunicato dall'entità esterna solamente quando è attivo il segnale *mov*, altrimenti legge quello dal robot precedente

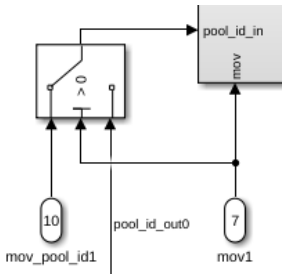

Figura 5.1: Input *pool_id_in*

Esempio 5.1 → formazione di due pool simultanei

$$mov0 : [n = 4, m = 2, join = \{\#0 : true, \#1 : false, \#2 : false, \#3 : true\}]$$
$$mov1 : [n = 4, m = 2, join = \{\#0 : busy, \#1 : true, \#2 : true, \#3 : busy\}]$$

In questo caso l'entità esterna invia una *movement request* a `Robot#0` che, in virtù del non join di `Robot#1` e `Robot#2`, forma il pool con `Robot#3`; successivamente l'entità esterna invia un'altra *movement request* a `Robot#1`, che forma un pool con `Robot#2`, essendo entrambi di nuovo disponibili al join.

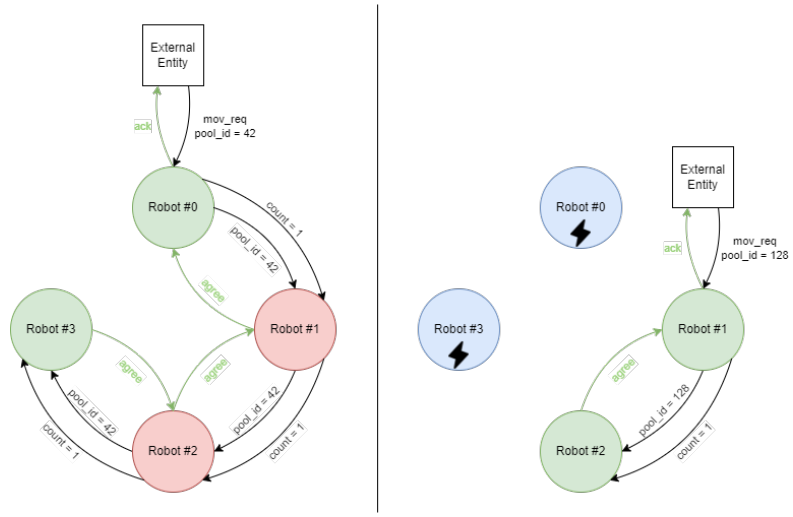

Figura 5.2: Sequenza degli *agreement run* in caso di due pool simultanei

5.3 Modello robot

Il modello dei robot diventa quello rappresentato dallo statechart in figura 5.3.

Una prima differenza da evidenziare è il nuovo output *current_pool_id* e il nuovo ruolo dell'output *pool_id_out*. Nel modello precedente l'output *pool_id_out* svolgeva due funzioni: comunicava al robot successivo l'id del pool da formare nella prima fase del protocollo e comunicava verso l'esterno l'id del pool a cui il robot partecipava durante l'esecuzione dell'azione comune. Tuttavia, essendoci adesso la necessità di partecipare allo scambio di messaggi anche durante l'esecuzione di un'azione comune, queste due funzioni vengono separate in due output: a *pool_id_out* resta la funzione di comunicazione durante la prima fase del protocollo (anche durante l'azione comune) e a *current_pool_id* il compito di segnalare a quale pool appartiene il robot.

Dunque la differenza principale nello statechart sta proprio nella gestione di questi due output; infatti *pool_id_out* viene settato solamente durante la ricezione di una richiesta e tenuto a 0 nel superstato **Processing**. *current_pool_id* invece viene settato in entrata nello stato **CommonAction** e resettato in uscita da esso.

L'ultima differenza, come già accennato, è nello stato **CommonAction**, dove deve essere fatto l'inoltro dei messaggi *count*, *agree* e *pool_id* utili alla comunicazione.

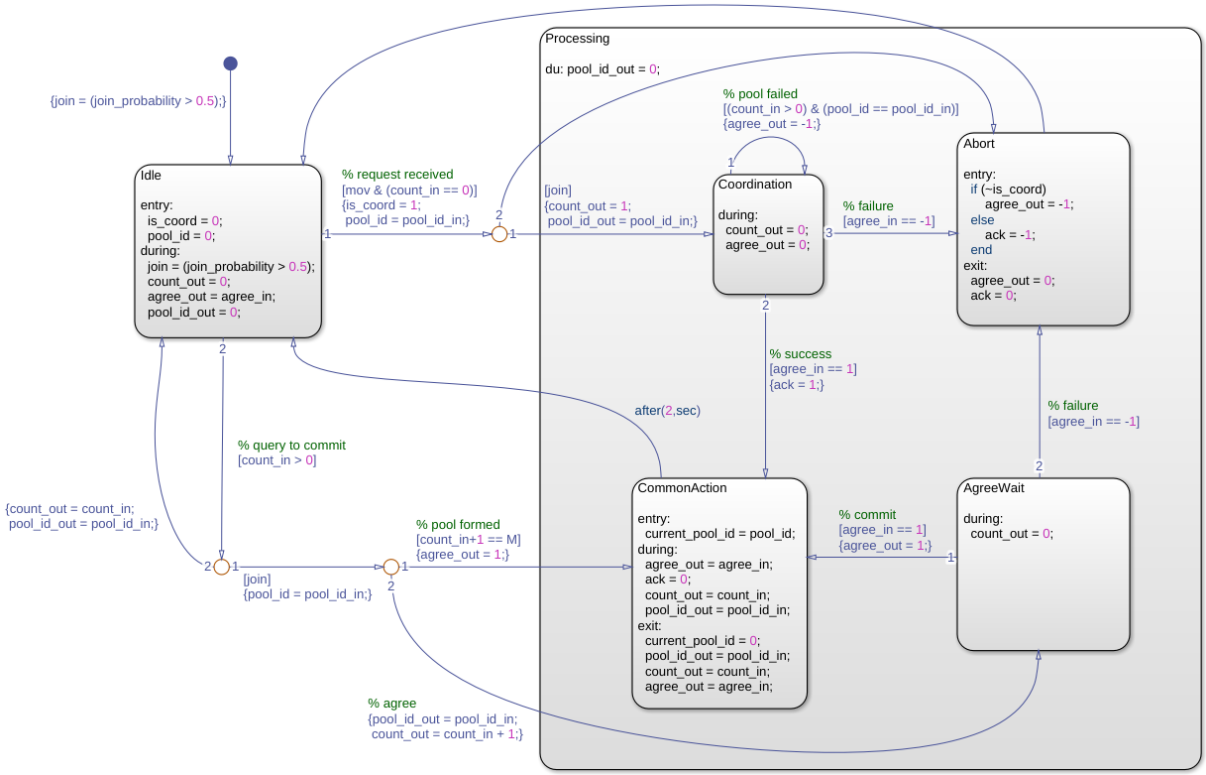

Figura 5.3: Statechart che modella il comportamento dei robot

Esempio 5.2 → formazione di due pool simultanei

$$mov0 : [n = 4, m = 2, join = \{\#0 : true, \#1 : false, \#2 : false, \#3 : true\}]$$
$$mov1 : [n = 4, m = 2, join = \{\#0 : busy, \#1 : true, \#2 : true, \#3 : busy\}]$$

Facendo riferimento all'esempio 5.1, simulando possiamo vedere, in figura 5.4, che il *current_pool_id* dei due pool è diverso.

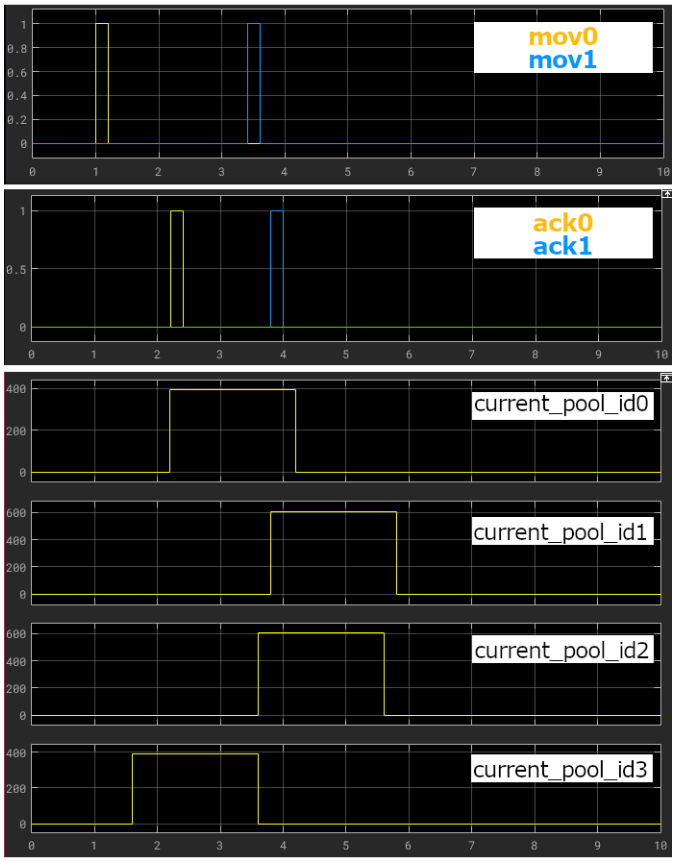

Figura 5.4: Segnali in caso di formazione di due pool simultanei

5.4 Verifica formale

Infine abbiamo verificato che le proprietà di interesse continuassero a valere anche in questo nuovo modello. Ovviamente in questo caso non è richiesta la proprietà di *agreement* perché, per ipotesi, è consentita la formazione di più pool simultanei.

I risultati sono stati positivi: è stato dimostrato che entrambe le proprietà continuano a valere.

6 Fault tolerance (obj5)

La descrizione del problema vista finora non considera la possibilità che un robot non risponda o che la comunicazione non avvenga con successo. Abbiamo dunque investigato su cosa succede nel modello progettato (così come è presentato nel capitolo 2) qualora sia ammesso un fallimento di questo tipo e successivamente abbiamo progettato un nuovo modello che cerca di mitigare questi effetti.

6.1 Effetti di un fallimento

Il modello attuale fa totale affidamento al fatto che i robot non sperimentino nessun tipo di fallimento, così come la comunicazione. Un eventuale fallimento può comportare conseguenze diverse in base al momento in cui si verifica (inteso come prima o seconda fase del protocollo) e al successo o meno di formazione del pool.

La mancata propagazione del segnale di richiesta (esempio 6.1) fa sì che un pool non ancora formato non riesca mai a formarsi, ma questo non è l'unico effetto; infatti i robot che fino a quel momento hanno dato disponibilità a unirsi al pool e si sono messi in attesa di un segnale *(dis)agree* in back flow rimarranno bloccati in questo stato indefinitamente.

La mancata propagazione in back flow del segnale di *(dis)agreement* invece ha effetti potenzialmente peggiori. Infatti in caso di *agreement* (esempio 6.2) i robot che nella catena si collocano topologicamente tra il robot che ha completato il pool (esso stesso compreso) e il fallimento, nel caso in cui abbiano accettato di unirsi al pool inizieranno l'azione comune come da protocollo, ma potrebbero ritrovarsi a compierla in sottonumero (nel caso in cui il fallimento tagliasse fuori alcuni robot che avevano dato disponibilità ad unirsi). Questo in linea di principio potrebbe causare effetti dannosi per gli oggetti e/o i robot stessi, pur rimanendo verosimilmente nel caso di **fallimento benigno**, ovvero con conseguenze dello stesso ordine di grandezza in termini economici del beneficio prodotto dal sistema in mancanza del fallimento, ma questo potrebbe anche non essere vero (un'eventuale classificazione del fallimento come catastrofico dipende comunque dal contesto).

In caso di *disagreement* invece (esempio 6.3), si ha un effetto simile al precedente, dove alcuni robot si troveranno permanentemente bloccati in uno stato di attesa di un messaggio *(dis)agree*.

In tutti i casi poi l'effetto più evidente è quello di bloccare l'intero sistema: infatti in nessun caso il robot coordinatore riuscirà a dare un segnale di *(not) acknowledgement* all'entità esterna e questo, per l'assunzione di singolo coordinatore e per come è stato progettato il modello, impedisce e quest'ultima di inviare altre *movement request*.
In realtà però questo comportamento, in mancanza di sistemi di fault tolerance a livello del protocollo, è desiderato. Infatti così facendo qualsiasi fallimento del sistema è, dal punto di vista dell'entità esterna, un **fallimento per omissione** (altresì detto **crash**), rendendo così l'intero sistema **fail silent**. L'entità esterna può dunque implementare meccanismi per la rilevazione di un fallimento (per esempio un timeout) e la segnalazione del possibile avvento di un fallimento, andando in una sorta di stato **fail safe** e/o inviando a tutti i robot un segnale di abort immediato.

Esempio 6.1 → Fallimento nella propagazione della richiesta

In questo caso il fallimento di `Robot #3` o della comunicazione tra `Robot #2` e `Robot #3` comporta che `Robot #2` e `Robot #0`, che si erano dichiarati disponibili, attendano indefinitamente una risposta che non arriverà mai.

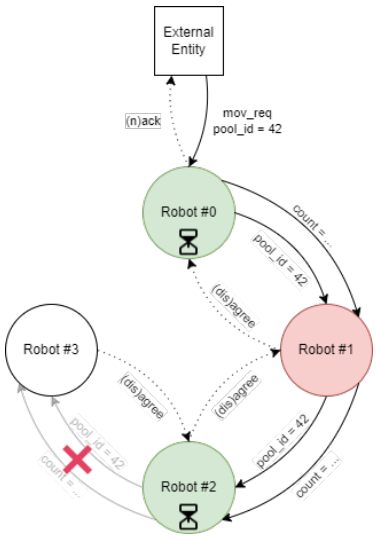

Figura 6.1: Fallimento nella propagazione del segnale di richiesta

Esempio 6.2 → Fallimento nella propagazione dell'*agree*

In questo caso il fallimento di `Robot #3` (dopo che abbia accettato di unirsi al pool) o della comunicazione tra `Robot #3` e `Robot #2` comporta che `Robot #1` e `Robot #0`, che si erano dichiarati disponibili, attendano indefinitamente una risposta che non arriverà mai, mentre `Robot #3` se non è fallito eseguirà l'azione comune in solitudine.

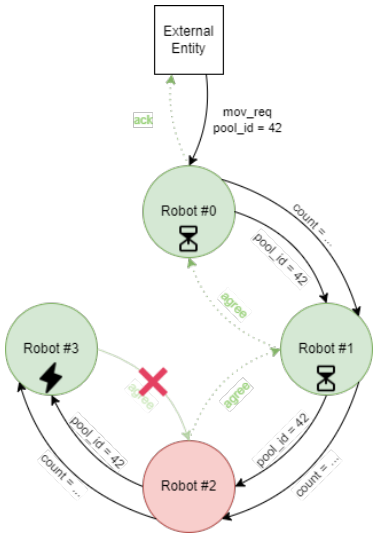

Figura 6.2: Fallimento nella propagazione dell'*agree*

Esempio 6.3 → Fallimento nella propagazione del *disagree*

In questo caso il fallimento di `Robot #3` (dopo aver inoltrato la richiesta al successivo) o della comunicazione tra `Robot #3` e `Robot #2` comporta che `Robot #1` e `Robot #0`, che si erano dichiarati disponibili, attendano indefinitamente una risposta che non arriverà mai.

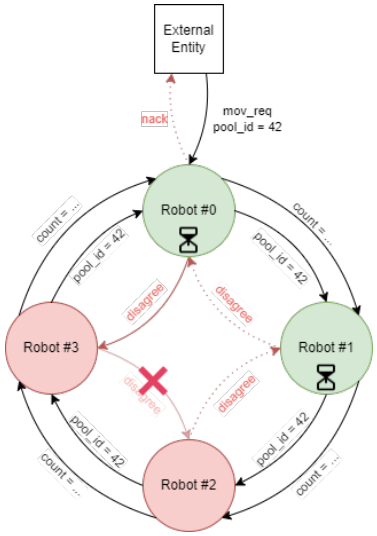

Figura 6.3: Fallimento nella propagazione del *disagree*

6.2 Modello fault tolerant

Abbiamo poi riprogettato il modello dei robot, in maniera da introdurre della fault tolerance nel protocollo.

Come si nota dall'analisi degli effetti dei fallimenti il problema principale è che i robot rimangono bloccati negli stati di attesa del segnale *(dis)agree* in back flow. La nostra modifica quindi ha avuto come obiettivo proprio questo aspetto. Il nuovo modello (statechart in figura 6.4) introduce due **timeout** che permettono dopo un certo periodo di tempo senza ricezione di una risposta di abortire l'elaborazione di una richiesta.

Le due nuove transizioni quindi partono degli stati **AgreeWait** e **Coordination** per finire in **Abort** dopo un certo periodo senza una risposta. Chiaramente questo tempo dipende dal numero di robot presenti e dalla posizione di ciascun robot nella "catena": infatti più robot ci sono e più impiegherà il segnale di richiesta (nel caso peggiore) a propagarsi a tutti i robot e quello di *(dis)agree* a tornare indietro. Per questo motivo il tempo di attesa, espresso in secondi, è pari a `2*(N-id)`, in modo che il primo a rilevare il fallimento e iniziare il back flow di *disagree* sia sempre quello più prossimo ad esso.

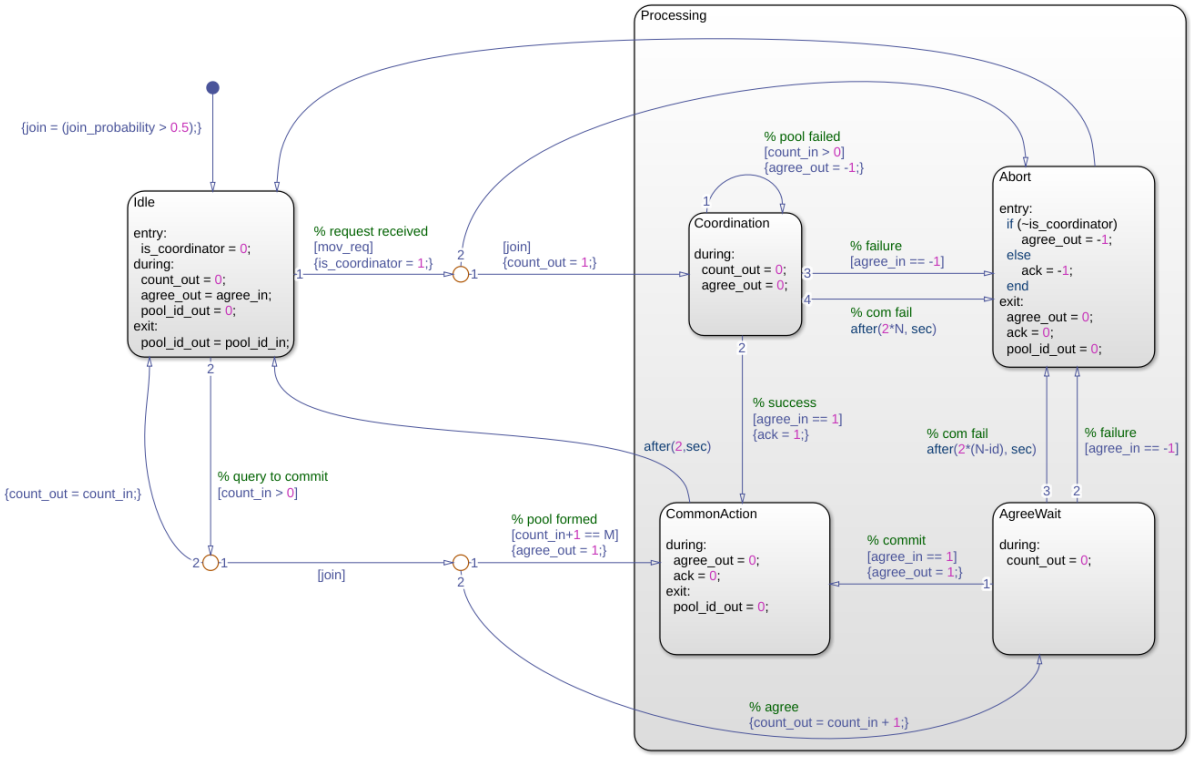

Figura 6.4: Statechart che modella il comportamento dei robot fault tolerant

Nota bene → Fault tolerance solo parziale

La fault tolerance ottenuta così non è totale. Infatti si ha vera e propria fault tolerance solamente se il fallimento avviene nella prima fase del protocollo; infatti in questo caso non si propaga più il segnale di richiesta e dopo un certo lasso di tempo i robot che hanno processato la richiesta vanno in **Abort**.

Quando invece si ha un fallimento sul segnale *(dis)agree* in back flow, la fault tolerance è solo parziale: viene evitata l'attesa indeterminata di alcuni robot, ma può ancora accadere che dei robot si trovino a compiere l'azione comune in sottonumero. Inoltre all'entità esterna viene sempre inviato un *not acknowledgement*.

Si può quindi considerare di aver raggiunto fault tolerance totale solamente nel caso in cui compiere l'azione comune in sottonumero non comporti nessuna conseguenza per i robot e nessuna modifica sull'ambiente. Infatti se tutto resta invariato non c'è discrepanza con ciò di cui è a conoscenza l'entità esterna, ovvero che l'azione non è stata eseguita per impossibilità di formazione del pool.

6.2.1 Verifica formale

Infine abbiamo verificato che le proprietà di interesse, *validity*, *liveness* e *agreement*, continuassero a valere anche in questo nuovo modello.

Per simulare il fallimento di un robot, o della comunicazione, abbiamo staccato le uscite di un robot verso il precedente e il successivo, come mostrato in figura 6.5.

I risultati sono stati positivi: è stato dimostrato che tutte le proprietà continuano a valere.

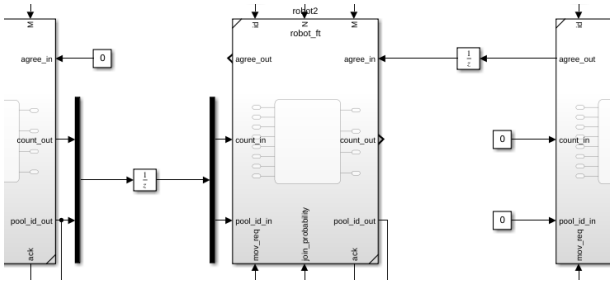

Figura 6.5: Simulazione di un fallimento nel modello

Esempio 6.4 → fallimento

$$[n = 4, m = 3, join = \{\#0 : true, \#1 : true, \#2 : true, \#3 : true\}, fault : \#2]$$

In questo caso `Robot#2` fallisce, quindi `Robot#1` dopo 6 secondi $(2 * (N - id) = 2 * (4 - 1) = 6)$ va in **Abort** e invia il segnale di *disagree*, di conseguenza poi `Robot#0` invia il *nack*.

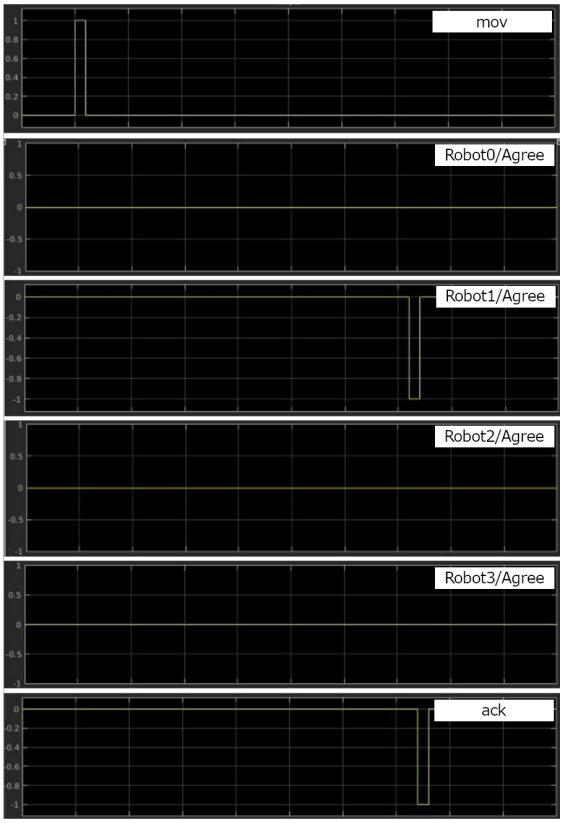

Figura 6.6: Segnali in caso di fallimento

7 Conclusioni

Ricapitoliamo gli obiettivi raggiunti in questo progetto:

- **Modellazione (obj1)**: abbiamo definito il protocollo con topologia lineare chiusa ad anello, ispirato al 2PC protocol lineare; abbiamo modellato i robot con uno statechart che seguisse il comportamento definito dal protocollo; infine abbiamo modellato il sistema come un insieme di robot opportunamente comunicanti.
- **Verifica formale (obj2)**: attraverso il Design Verifier di Simulink abbiamo verificato le proprietà di *validity*, *liveness* (trasformando l'operatore Eventually per poterlo verificare in Simulink) e *agreement*; le proprietà sono state tutte verificate con successo.
- **Studio sulla scalabilità (obj3)**: abbiamo studiato la scalabilità del modello giungendo alla conclusione che è abbastanza scalabile per quanto riguarda l'espansione del modello, ma non è assolutamente scalabile riguardo alla possibilità di eseguire la verifica formale in tempi ragionevoli.
- **Coordinatori multipli (obj4)**: abbiamo rilassato l'assunzione di singolo coordinatore adattando il modello alla possibilità che si potessero formare più pool operativi simultaneamente e abbiamo verificato formalmente (con successo) le proprietà di *validity* e *liveness* per il nuovo modello.
- **Fault tolerance (obj5)**: abbiamo studiato cosa succede in caso di fallimento a un robot o sulla comunicazione nel modello progettato e abbiamo proposto una possibile mitigazione che sfrutta un timeout; anche per questo nuovo modello sono state verificate formalmente (con successo) le proprietà di *validity*, *liveness* e *agreement*.

Per questo progetto sono state impiegate circa 80 ore di lavoro, sempre svolto in gruppo, quindi tutti i contributi presentati sono da intendersi comuni a tutti e 3 gli autori.

Bibliografia

[1] Alessandro Fantechi. *Informatica industriale*. Città Studi.

[2] *Github repository*. URL: https://github.com/marcodiri/robot_cluster.

[3] F. Leitner e S. Leue. ≪Simulink Design Verifier vs. SPIN – a Comparative Case Study≫. In: gen. 2008. URL: https://www.uni-konstanz.de/soft/research/publications/pdf/FMICS2008_FINAL.pdf.

[4] *Simulink Design Verifier*. URL: https://it.mathworks.com/help/sldv/.

[5] *What Is Property Proving?* URL: https://it.mathworks.com/help/sldv/ug/what-is-property-proving.html.