Relazione sul progetto di Sistemi Concorrenti e Distribuiti

Alberto De Bortoli

17 ottobre 2010

Sommario

Relazione finale sul progetto del corso di Sistemi Concorrenti e Distribuiti tenutosi l'anno accademico 2009/'10 dal Prof. Tullio Vardanega.
La relazione discute problematiche di concorrenza e distribuzione in relazione a un prodotto preesistente. Inizialmente viene proposta una soluzione personale alla specifica del progetto realizzato, successivamente viene descritto il prodotto esistente. Seguono le fasi di analisi e di test. Infine vengono proposti e in parte implementati alcuni miglioramenti.

Indice

Capitolo 1

Introduzione

Il progetto del corso di Sistemi Concorrenti e Distribuiti tenutosi l'A.A. 2009/'10 prevede l'analisi di un progetto svolto dagli studenti che hanno seguito il corso l'A.A. precedente. Lo svolgimento del progetto prevedeva la realizzazione di un simulatore concorrente e distribuito di una competizione sportiva di Formula 1. La specifica di seguito riportata è presente sul sito web del docente (*http://www.math.unipd.it/ tullio/SCD/2008/Progetto.html*).

Tale sistema avrebbe dovuto soddisfare i seguenti requisiti:

- un circuito, possibilmente selezionabile in fase di configurazione, dotato almeno della pista e della corsia di rifornimento, ciascuna delle quali soggetta a regole congruenti di accesso, condivisione, tempo di percorrenza, condizioni atmosferiche, ecc.
- un insieme configurabile di concorrenti, ciascuno con caratteristiche specifiche di prestazione, risorse, strategia di gara, ecc.
- un sistema di controllo capace di riportare costantemente, consistentemente e separatamente, lo stato della competizione, le migliori prestazioni (sul giro, per sezione di circuito) e anche la situazione di ciascun concorrente rispetto a specifici parametri tecnici
- una particolare competizione, con specifica configurabile della durata e controllo di terminazione dei concorrenti a fine gara.

Il progetto poteva essere svolto singolarmente o in gruppi di 2 persone, a seconda del livello di difficoltà. Il progetto che si è deciso di analizzare in questa relazione è quello di livello 2, che prevedeva la redazione di una relazione tecnica sulle problematiche di concorrenza e distribuzione con annesso prototipo implementato in linguaggio Ada. L'elaborato analizzato è

dei Dottori Nicolò Navarin e Tobia Zorzan.
La specifica del progetto 2009/'10 chiede di fare un'analisi critica del progetto nei seguenti requisiti [**Rx**] e fornire determinati prodotti in uscita [**Px**]:

- **Requisiti**

  R1 Verifica di copertura, nel progetto esaminato, dei requisiti di concorrenza e distribuzione espliciti e impliciti derivanti dalla specifica 2008/'09

  R2 Verifica di congruità semantica e architetturale delle soluzioni realizzative adottate nel progetto esaminato

  R3 Specifica di correzioni, estensioni, miglioramenti. Opzionalmente, realizzazione e dimostrazione delle specifiche proposte.

- **Prodotti**

  P1 Relazione tecnica articolata e completa.

  R2 Presentazione per discussione in sede d'esame.

  R3 Opzionalmente, prototipo dimostrativo, completo di specifica e istruzioni di prova.

Vengono inoltre rischiesti miglioramenti architetturali [**Ax**] e funzionali [**Fx**]:

- **Miglioramenti architetturali**

  A1 Gestione realistica dei box (accesso, congestione, rallentamento).

  A2 Completezza delle misurazioni intermedie (p.es., come istanza di distributed snapshot).

  A3 Analisi e controllo delle sorgente di non-determinismo indesiderabile.

- **Miglioramenti funzionali**

  F1 Cambio dinamico di condizioni meteorologiche.

  F2 Visualizzazione tracciato e posizioni.

  F3 Gestione classifica campionato.

Nel presente documento si farà riferimento ai nomi dati ai singoli punti per riferimento e tracciamento.

Capitolo 2

Soluzione proposta

Prima di affrontare il prodotto preesistente si espone la soluzione che si è pensata essere la più adatta. Oltre agli esempi di programmazione concorrente analizzati a lezione, è risultato molto utile l'esempio mostrato al paragrafo 8.4 del libro di testo *Concurrent and Real-Time Programming in Ada* come punto di partenza per affrontare la progettazione.

2.1 Entità fondamentali

Lo scenario reale che si vuole rappresentare nel sistema concorrente e distribuito prevede la presenza di determinate entità di seguito elencate. Per differenziare dalle entità del progetto preesistente si usano nomi in lingua italiana.

Auto: il concorrente è rappresentato con un'entità attiva;

Settore: una parte di pista è rappresentata con un'entità passiva, risorsa protetta;

Corsia: la corsia appartenente a un determinato settore è rappresentato con un'entità passiva, risorsa protetta.

Auto, *Settore* e *Corsia* sono le entità coinvolte degne di nota per la progettazione di un protocollo corretto per l'aspetto concorrente. Una *Corsia* è associata a un *Settore*, mentre lo stesso può avere più *Corsie* (molteplicità 1:n). Altre entità come monitor di gara o competizioni vengono tralasciate in quanto non principalmente coinvolte per la tematica affrontata.

Per giungere alla progettazione ritenuta corretta si mostrano 3 progettazioni denominate in serie **P1**, **P2**, **P3**. Ad ogni passo si esplicherà il perché la progettazione non sia adatta al problema, non sia implementabile o perché presenti degli errori non trascurabili.

2.2 Progettazione P1

2.2.1 Pseudocodice Ada per P1

Si mostra lo pseudocodice in Ada delle 3 entità elencate al paragrafo precedente. Nel paragrafo successivo si descriverà il protocollo progettato.

Listing 2.1: Entità attiva Auto

```
task type Auto;

task body Auto is
  featuresAuto         : <CustomType>;,
  tempoPercorrenza     : Duration;
  tempoIniziale        : Time;

  begin
    for G in 1 .. <numeroDiGiri> loop
      for S in 1 .. <numeroDiSettori> loop
        S.Entra(featuresAuto, tempoPercorrenza);
        -- attività di monitoring e tracciamento --
        tempoIniziale := Clock;
        delay until (tempoIniziale + tempoPercorrenza);
      end loop;
    end loop;
end Auto;
```

Listing 2.2: Risorsa protetta Settore con espressione di guardia con accesso a parametri in ingresso

```
protected type Settore(featuresSettore : <CustomType>) is
  entry Entra(featuresAuto : <CustomType>;
          tempoPercorrenza : out Duration);
  -- altre procedure --
  private
    arrayUscite : arrayUscite_T(1 .. <numeroAuto>);
    -- campi privati per le features --
end Settore;

protected body Settore is
  entry Entra(featuresAuto : <CustomType>;
          tempoPercorrenza : out Duration) when
```

```
13            (featuresAuto.ID = <SettorePrecedente>.
                 getArrayUscite(1)) is
14      begin
15        -- scelta della Corsia meno trafficata --
16        requeue Corsia.Enter;
17    end Entra;
18 end Settore;
```

Listing 2.3: Risorsa protetta Corsia

```
1  protected type Corsia(featuresCorsia : <CustomType>) is
2    entry Entra(featuresAuto : <CustomType>;
3                tempoPercorrenza : out Duration);
4    -- altre procedure --
5    private
6      -- campi privati per le features --
7  end Settore;
8
9  protected body Corsia is
10   entry Entra(featuresAuto : <CustomType>;
11               tempoPercorrenza : out Duration) when True
                    is
12     begin
13       tempoPercorrenza := CalcolaTempoDiPercorrenza;
14       -- rimuove da arrayUscite del Settore precedente
15       <SettorePrecedente>.popFromArrayUscite(featuresAuto
             .ID);
16       -- aggiunge in coda ad arrayUscite del Settore che
             contiene la Corsia
17       <SettorePadre>.addToArrayUscite(featuresAuto.ID);
18   end Entra;
19 end Corsia;
```

2.2.2 Descrizione del protocollo P1

Si descrivono le azioni intraprese o subite dalle entità.

Auto: astraendo dal problema di creazione e avvio di un task di questo tipo, *Auto* ha il compito di simulare una competizione di Formula1 percorrendo in sequenza tutti i settori del tracciato per il numero di giri richiesto (rappresentato dai due loop).

Ad ogni *Settore* verrà effettuata una chiamata alla entry *Entra*, a fine della quale verrà riempito il campo *tempoPercorrenza* (parametro in access mode **out**) che verrà utilizzato per la sospensione del task.

Settore: nella guardia (o barriera) della entry *Entra* è presente un parametro in ingresso alla richiesta, in questo modo sarebbe possibile sapere quale task sta invocando la entry e decide se farlo eseguire. Nella guardia viene controllato che il task sia effettivamente il primo ad essere uscito dal settore precedente.

L'array *arrayUscite* tiene conto dei tempi di uscita dal *Settore*, in questo modo si prevengono sorpassi indesiderati tra due settori successivi della pista: i sorpassi sarebbero causati dal risveglio non ordinato dei task dalla `delay until`.

L'azione intrapresa da *Entra* è quella di calcolare deterministicamente quale sia la corsia meno trafficata sulla quale effettuare il riaccodamento tramite `requeue` (trasferimento di coda asincrono).

Corsia: una volta che un task *Auto* viene riaccodato su *Entra* di *Corsia* la richiesta viene accettata (guardia sempre aperta) e il tempo di percorrenza dell'auto viene calcolato deterministicamente in base alle caratteristiche dell'invocante.

Il riferimento all'auto viene rimosso da *arrayUscite* del *Settore* precedente e viene aggiunto nella posizione appropriata (per tempi di uscita) in *arrayUscite* del *Settore* contenente la *Corsia*. Si tralasciano dettagli a livello di codice in quanto problemi prettamente di programmazione.

Il tempo di percorrenza verrà ritornato al task *Auto* tramite parametro in access mode **out**.

2.2.3 Problemi derivanti da P1

Un aspetto molto discusso nella progettazione di un linguaggio concorrente è se rendere possibile consentire all'espressione di guardia di una entry di avere accesso ai parametri in ingresso della richiesta. Nella progettazione **P1** si assume sia possibile fare ciò nel linguaggio Ada.

Il difetto principale di tale approccio è il suo eccessivo costo realizzativo. Legare la sincronizzazione al valore di un parametro della richiesta comporta che tutte le richieste accodate debbano essere rivalutate ogni volta che la condizione possa essere cambiata, con un eccessivo costo realizzativo.

L'alternativa che Ada mette a disposizione è trasferire la richiesta attualmente non soddisfacibile a fronte di un maggiore sforzo di progettazione

come vedremo in **P2** e **P3**. È inoltre sconsigliato fare riferimento a variabili condivise nelle espressioni di guardia.

2.3 Progettazione P2

2.3.1 Pseudocodice Ada per P2

Si tengano validi gli pseudocodici di *Auto* e *Corsia* rispettivamente ai punti 2.1 e 2.3, viene modificata solo l'entità *Settore*.

Listing 2.4: Risorsa protetta Settore

```
protected type Settore(featuresSettore : <CustomType>) is
  entry Entra(featuresAuto : <CustomType>;
              tempoPercorrenza : out Duration);
  -- altre procedure --
  private
    arrayUscite : arrayUscite_T(1 .. <numeroAuto>);
    Guardia     : Boolean      := True;
    N           : Integer      := 0;
    -- campi privati per le features --
end Settore;

protected body Settore is
  entry Entra(featuresAuto : <CustomType>;
              tempoPercorrenza : out Duration) when
              (Entra'Count = N + 1 or Guardia) is
    begin
      if (featuresAuto.ID = <SettorePrecedente>.
          getArrayUscite(1).ID)
      then
        -- scelta della Corsia meno trafficata --
        Guardia := True;
        requeue Corsia.Enter;
      else
        if (Guardia = False and then Entra'Count = 0)
            then
          N := 0;
        end if;
        N := N + 1;
        Guardia := False;
        requeue <SettoreAttuale>.Entra;
      end if;
  end Entra;
end Settore;
```

2.3.2 Descrizione protocollo P2

In questa progettazione dell'entità *Settore* viene utilizzato il meccanismo di `requeue` come alternativa alla valutazione dei parametri in ingresso alla richiesta dentro alla guardia.

Inizialmente l'espressione booleana di guardia (*Guardia*) è settata a `True`: quando un task richiede accesso, la guardia valuta positivamente e l'esecuzione avviene. Se il task è effettivamente il primo ad essere uscito dal *Settore* precedente viene ora verificato (a differenza della progettazione **P1**) e in caso positivo viene riaccodato sulla *Corsia* appropriata, scelta deterministicamente. Se i task uscenti dal *Settore* precedente si svegliassero dalla `delay until` in maniera congrua ai tempi previsti si eseguirebbe sempre il ramo `then`.

I casi di "risvegli imprevisti" da parte dei task vengono gestiti nel ramo `else` tramite `requeue` sulla stessa entry. Il ramo controlla se *Guardia* valuta a `False` e se la coda dei task accodati su *Entra* è vuota, in tal caso resetta il valore di `N`. `N` serve per tenere traccia di quanti sono i task che hanno ottenuto accesso alla risorsa superando la guardia sulla prima condizione (`Entra'Count = N + 1`) e che verranno riaccodati sulla stessa entry.

La prima condizione si verifica all'arrivo di un nuovo task su *Entra* che ne causerà la rivalutazione (modifica di `'Count`). All'apertura della guardia a tutti i task verrà data la possibilità di rieseguire per controllare se le condizioni per eseguire nel ramo `if` si sono avverate. Quando il task che deve eseguire il ramo `if` accede alla risorsa viene risettata a `True` il valore di `Guardia` per permettere ai task successivi di eseguire nuovamente accedendo a *Entra* su guardia verificata sulla seconda codizione.

L'uso di `requeue` crea sospensione; eventuali richieste a *Entra* durante il riaccodamento sono contemplate, la guardia verrà riaperta solo quando saranno presenti `N + 1` richieste sulla coda, valore che si raggiungerà quando tutti i task eseguenti `else` verranno riaccodati.

2.3.3 Problemi derivanti da P2

Purtroppo la soluzione è alquanto complessa e non sfrutta appieno le potenzialità della `requeue`. Infatti un riaccodamento corretto deve avvenire su un'altra coda d'attesa poiché al task è stata già data possibilità di eseguire sulla entry.

Vi sono inoltre altri problemi non trascurabili:

1. in uno scenario concorrente, task possono accodarsi nel periodo temporale compreso tra l'apertura della guardia con la prima condizione e

il controllo di `'Count` a riga 23. Lo stato di `N` non risulterà consistente non venendo settato a `0` alla riga 24, cosa che renderà impossibili i successivi tentativi di accedere alla risorsa protetta;

2. all'arrivo di un task su *Entra*, la barriera viene rivalutata e i task in attesa verranno spostati all'interno della risorsa protetta dove l'unico task potenzialmente primo nell'array `arrayUscite` eseguirà per ultimo, solo dopo che tutti gli altri task già valutati avranno eseguito. Infatti riaccodando sulla stessa entry si prevede l'accodamento in coda, che causa la successiva (e inutile) riesecuzione dei task già valutati prima dell'esecuzione del nuovo task accodato (unico ad avere possibilità di eseguire il ramo `if`).

2.4 Progettazione P3

2.4.1 Pseudocodice Ada per P3

Si tenga valido lo pseudocodice di *Auto* al punto 2.1, vengono modificate le entità *Settore* e *Corsia*.

Listing 2.5: Risorsa protetta Settore

```
protected type Settore(featuresSettore : <CustomType>) is
  entry Entra(featuresAuto : <CustomType>;
              tempoPercorrenza : out Duration);
  procedure Rilascia;
  -- altre procedure --
  private
    entry Riprova(featuresAuto : <CustomType>;
                  tempoPercorrenza : out Duration);
    arrayUscite : arrayUscite_T(1 .. <numeroAuto>);
    Guardia     : Boolean      := False;
    Rimanenti   : Integer      := 0;
    -- campi privati per le features --
end Settore;

protected body Settore is
  entry Entra(featuresAuto : <CustomType>;
              tempoPercorrenza : out Duration) when True is
    begin
      if (featuresAuto.ID = <SettorePrecedente>.
          getArrayUscite(1).ID)
      then
        -- scelta della Corsia meno trafficata --
        requeue Corsia.Enter;
      else
        requeue Riprova;
      end if;
  end Entra;

  procedure Rilascia is
    begin
      if Riprova'Count > 0 then
        Rimanenti := Riprova'Count;
        Guardia := True;
      end if;
```

```
34   end Rilascia;
35
36   entry Riprova (featuresAuto : <CustomType>;
37                  tempoPercorrenza : out Duration) when
                       Guardia = True is
38     begin
39       Rimanenti := Rimanenti - 1;
40       if Rimanenti = 0 then
41         Guardia := False;
42       end if;
43
44       if (featuresAuto.ID = <SettorePrecedente>.
            getArrayUscite(1).ID)
45       then
46         -- scelta della Corsia meno trafficata --
47         requeue Corsia.Enter;
48       else
49         requeue Riprova;
50       end if;
51
52   end Riprova;
53 end Settore;
```

Listing 2.6: Risorsa protetta Corsia

```
1  protected type Corsia(featuresCorsia : <CustomType>) is
2    entry Entra(featuresAuto : <CustomType>;
3                tempoPercorrenza : out Duration);
4    -- altre procedure --
5    private
6      -- campi privati per le features --
7  end Settore;
8
9  protected body Corsia is
10   entry Entra(featuresAuto : <CustomType>;
11               tempoPercorrenza : out Duration) when True
                     is
12     begin
13       tempoPercorrenza := CalcolaTempoDiPercorrenza;
14       -- rimuove da arrayUscite del Settore precedente
15       <SettorePrecedente>.popFromArrayUscite(featuresAuto
             .ID);
```

```
16          -- aggiunge in coda ad arrayUscite del Settore che
                contiene la Corsia
17          <SettorePadre>.addToArrayUscite(featuresAuto.ID);
18          <SettorePadre>.Rilascia;
19      end Entra;
20  end Corsia;
```

2.4.2 Descrizione protocollo P3

Si descrivono i comportamenti delle entità modificate.

Settore: la guardia di *Entra* è sempre aperta, se al task non è ancora consentito uscire viene riaccodato su la entry *Riprova*, in caso contrario viene scelta deterministicamente una *Corsia* sulla quale effettuare il riaccodamento.

L'array *arrayUscite* tiene conto dei tempi di uscita dal *Settore*, in questo modo si prevengono sorpassi indesiderati tra due settori successivi della pista: i sorpassi sarebbero causati dal risveglio non ordinato dei task dalla `delay until`.

L'azione intrapresa da *Entra* è quella di calcolare deterministicamente quale sia la corsia meno trafficata sulla quale effettuare il riaccodamento tramite `requeue` (trasferimento di coda asincrono).

Rilascia apre la guardia di *Riprova* se ci sono task in attesa e salva in `Rimanenti` il numero di task che entrano in risorsa protetta dopo la valutazione della guardia. Quando tutti i task trasferiti all'interno della risorsa protetta hanno eseguito, la guardia viene chiusa (azione effettuata dall'ultimo task eseguente in risorsa protetta).

Il restante codice di *Rilascia* ha lo stesso comportamento del corpo di *Entra* in quanto viene effettuato lo stesso test per controllare che sia giunto il turno del task invocante per uscire dal *Settore* (tramite `requeue` su *Corsia*.

Corsia: l'unica modifica necessaria è effettuare una richiesta a *Rilascia* del *Settore* a cui la *Corsia* fa riferimento.

L'uscita da un settore avviene in modalità FIFO. Ogni aspetto legato al determinismo è considerato ponendo le basi per ottenere esecuzioni identiche su stessi input.

Si mostra in figura 2.1 il diagramma di sequenza del protocollo progettato.

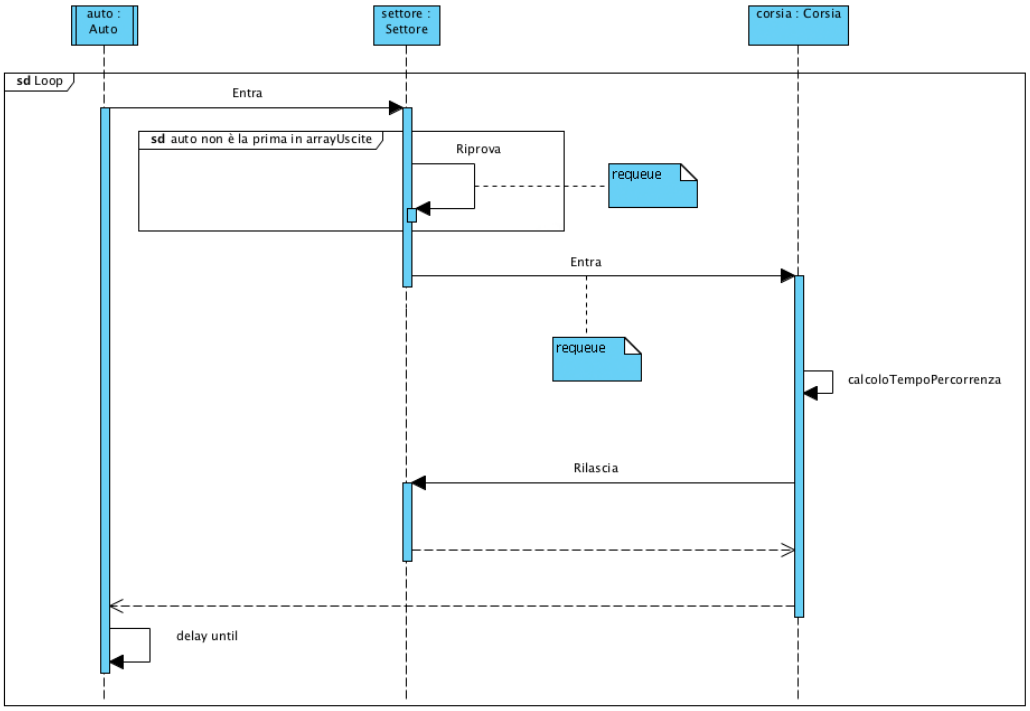

Figura 2.1: Diagramma di sequenza di percorrenza del circuito

2.4.3 Correttezza temporale

Sulla base delle seguenti considerazioni, si può assumere che il protocollo progettato sia corretto dal punto di vista temporale:

- L'ordine di ingresso in un *Settore* è controllato e i tempi di percorrenza (dipendenti dalle caratteristiche di *Auto*) si basano solo sui tempi calcolati, quindi il protocollo è corretto indipendentemente dall'ordine e dal momento in cui eseguono i task;
- Non vi sono problemi legati alla sincronizzazione degli orologi: dopo l'avvio di ogni task *Auto*, i tempi di percorrenza sono tutti relativi e quindi indipendenti dall'orologio di sistema;
- Il risveglio delle auto dalla `delay until` può avvenire in maniera non ordinata: è stato descritta la soluzione utilizzando un array che mantiene i tempi di uscita dal *Settore* precedente. Trattasi di una soluzione algoritmica ad un ordine di esecuzione dei processi non corretto da un punto di vista concettuale;

- I processi vengono accodati in una coda interna a *Settore*; i componenti *Auto* riescono ad entrare nel *Settore* successivo solo rispettando l'ordine di uscita dal precedente (calcolato sui tempi di percorrenza);
- Un task che esegue in un momento non previsto, ovvero quando non è il primo nell'array dei tempi di uscita, viene riaccodato sulla coda interna, attendendo l'esecuzione di altri task;
- Tra l'uscita da un *Settore* e l'entrata nel successivo, l'ordine di esecuzione dei task potrebbe cambiare a causa del prerilascio ma problemi in tal senso vengono gestiti dal protocollo.

La gestione dei tempi di gara sembra risultare corretta e coerente. Auto con le stesse caratteristiche hanno stessi tempi di percorrenza.

2.5 Considerazioni

2.5.1 Non determinismo causato da *delay* statements

L'utilizzo dello statement *delay* amplifica un problema insito nella progettazione, problema peraltro comune a progettazioni di sistemi concorrenti.

L'utilizzo di delay causa la sospensione del task Auto che ha come azione successiva l'accodamento sulla entry *Entra* del *Settore* successivo. La sospensione amplifica la probabilità di prerilascio già presente nel corpo del task Auto tra le invocazioni a *Entra* tra un *Settore* e il successivo.

Durante la sospensione del task *Auto* eseguente al tempo i-esimo è possibile che un task *Auto* eseguente al tempo j-esimo per $j > i$ sia risvegliato dalla *delay* e faccia richiesta a *Entra* del Settore successivo: ciò comporeterebbe un sorpasso dell'Auto j-esima.

È necessario mantenere l'ordine di uscita da un settore (uscita dalla risorsa protetta *Corsia*) per preservarlo nel *Settore* successivo, altrimenti potrebbero presentarsi sorpassi indesiderati.

Ciò viene implementato con una implementazione accorta della risorsa *Settore*, utilizzando una Guardia sulla entry, la funzionalità offerta dalla `requeue` e un array *arrayUscite* con l'ordine dei task *Auto* che hanno completato un settore per tempi d'uscita.

2.5.2 The *requeue* facility

Da quanto si può evincere dalle progettazioni descritte, l'utilizzo della `requeue` è l'unico modo per esprimere sincronizzazione basata sui parametri in ingresso della richiesta. Ada non è capace di discriminare le richieste sui parametri in ingresso, la risorsa protetta è ignara della richiesta fino a quando questa non viene accettata, e quando ciò succede essa è potenzialmente non soddisfacibile.

Riaccodare su un'altra coda è una soluzione elegante per aggirare la limitazione imposta dal linguaggio per evitare costi computazionali elevati; purtroppo è onere del progettista/programmatore utilizzare correttamente tale potere espressivo.

2.6 Distribuzione

Come spiegato a lezione e come illustrato nelle slide del Dott. Franco Gasperoni *Programming Distributed System*, la distribuzione di un progetto si può apportare dopo aver progettato una corretta soluzione concorrente concependola in uno scenario ignaro dell'aspetto distribuito.

Utilizzando *Ada 95 Distributed Systems Annex* si categorizzano i package contenenti le entità descritte. *Auto* è di sua natura una entità che deve poter eseguire remotamente ed è quindi categorizzabile con `pragma Remote_Types`. I nodi del sistema distribuito possono eseguire più entità *Auto*.

Tutte le entità *Settore* e *Corsia* devono essere presenti su un solo nodo del sistema. Esse possono essere racchiuse in un package che rappresenti il tracciato della pista di gara, tale package viene categorizzato con `pragma Remote_Call_Interface` che tramite procedure apposite permetterà l'interfacciamento delle entità *Auto* con la pista.

Capitolo 3

Descrizione prodotto preesistente

Il progetto preesistente è stato sviluppato con il linguaggio Ada che, come appreso dal corso, si presta bene alla modellazione di scenari concorrenti e distribuiti.

La progettazione ha previsto la presenza di determinati componenti tra risorse protette, entità passive, attive e server di seguito elencate con i seguenti acronimi per una migliore tracciabilità nel corso del documento:

Sector : il componente Sector rappresenta un tratto della pista, ovvero un determinato settore del percorso ed è realizzato come una risorsa protetta. La scelta risulta essere appropriata in quanto non deve essere presente un flusso di controllo per questo componente ed altre entità dovranno accedervi in mutua esclusione;

Lane : il componente Lane rappresenta una corsia all'interno di un determinato *Sector* ed è ragionevolmente realizzato anch'esso come risorsa protetta. Esso calcola, a partire dal tempo di invocazione e dalle caratteristiche dell'auto in questione, il tempo di percorrenza del tratto che è quindi calcolato concettualmente all'entrata in risorsa protetta;

Track : il componente Track rappresenta il circuito vero e proprio composto da *Sector*. Esso viene implementato come una entità passiva *Remote Call Interface* invocata da *Car*;

Race : il componente Race rappresenta la gara vera e propria ma ha anche il compito di gestione delle statistiche ricevendo informazioni da *Track* per poi renderle disponibili a *Monitor*;

Monitor : il componente Monitor rappresenta un generico monitor per la presentazione di statistiche, tempi e posizioni all'utente e per queste ragioni è di natura strettamente passiva e legato alla sua GUI;

Car : il componente Car rappresenta un concorrente con determinate caratteristiche per la competizione. Non è da confondersi con i *CarTask*;

CarTask : sono task anonimi e sono effettivamente i componenti che svolgono i compiti che mappano le azioni reali di una macchina in una competizione reale.

3.1 Concorrenza

La gestione della concorrenza è un compito in generale difficile; il progetto, da come risulta progettato, affronta in maniera chiara le problematiche che emergono.

3.1.1 Orologio

La gestione dei tempi di gara sono intrinsecamente soggetti ad un orologio. La scelta è stata quella di salvare il tempo del sistema (locale, al nodo ove avviene la competizione in un'unica istanza) all'avvio della competizione e calcolare tutti i tempi successivi di ogni concorrente (*CarTask* per compito di *Car*) in base a tale tempo. Questo sistema di calcolo per mezzo di offset risulta essere molto pulito per il problema in questione: esso non prevede l'utilizzo di chiamate al Clock per ottenere il tempo relativo dei concorrenti per i tempi intermedi. Chiamate al Clock alla conclusione di ogni giro o di ogni settore per ricavarne i tempi di gara risulterebbero troppo invasive e difficilmente gestibili in uno scenario concorrente.

Il meccanismo di somma di offset inoltre dovrebbe permettere in linea di principio di rendere una competizione riproducibile con risultati in output identici ponendo le basi per azzerare il non determinismo dell'esecuzione. Una tale situazione, desiderabile, non sarebbe facilmente raggiungibile con continue chiamate al Clock da parte di più entità attive nel sistema.

3.1.2 Riaccodamento per l'entrata in un settore

Essendo il circuito di gara suddiviso in settori (*Sector*), i task anonimi (*CarTask*) che rappresentano i concorrenti attivi in gara vengono riaccodati tramite il meccanismo di *requeue* dalla entry *Enter*, che ha il compito di effettuare la *requeue* sulla entry *Demand* di una specifica *Lane*. Tale meccanismo di *requeue* preserva l'ordine sulla entry *Enter* della risorsa protetta del settore. Era necessario preservare l'ordine di uscita da un settore per l'entrata nel settore successivo.

Track esegue `delay until` sui task anonimi *CarTask* rappresentanti le auto, ma il risveglio di questi non è assicurato entro un determinato tempo (non avendo `delay until` upper bound ma solo lower bound). Se i *CarTask* si svegliassero in un ordine non congruo a quello previsto delle `delay until`, l'entrata nel settore successivo risulterebbe errata. È stato dunque necessario *escogitare* un modo per aggirare il problema intrinseco creato da una progettazione tale.

La procedura esplicata nel paragrafo 3.1.3 della relazione del progetto appare elegante e coerente con ciò che il linguaggio Ada promette di fare. Per mantenere un ordine viene introdotto un algoritmo di riaccodamento dei *CarTask* svegliati non ordinatamente sulla entry *ExitLane* del *Sector* corrente e farli proseguire al settore successivo solo dopo che tutti i *CarTask* precedenti siano stati svegliati (ovvero che abbiano terminato la percorrenza del settore e possano quindi percorrere il settore successivo). In questo modo l'uscita dal settore avviene in modalità FIFO.

Si immagini che il *CarTask* con tempo di percorrenza minore (quindi minor tempo di `delay until`) vada in esecuzione dopo la `delay` per ultimo rispetto agli altri *CarTask* con tempi maggiori a causa dello scheduling. In uno scenario Murphyniano come questo, i *CarTask* svegliati per primi rimarranno riaccodati sulla entry ExitLane di *Sector* fino a quanto il *CarTask* con tempo minore non si sveglierà dalla `delay until`.

3.2 Distribuzione

Si discutono le caratteristiche di trasparenza per la parte di distribuzione del progetto.

La distinzione tra *Car* e *CarTask* fa sì che in uno scenario distribuito il primo componente possa essere remotizzato, mentre il secondo farà parte di *Track* ed eseguirà sullo stesso nodo dove è presente *Track*. *Track* esegue su un solo nodo ed è quindi giusto affermare che la simulazione avviene in pratica su un solo nodo.

Il sistema presenta buone qualità di trasparenza per la distribuzione. Le trasparenze d'accesso, di collocazione e di migrazione sono verificate su sistemi eterogenei, la qualità di spostamento di collocazione di una risorsa durante l'uso non è prevista per nessuna entità del simulatore. La trasparenza di spostamento è di difficile implementazione e troppo ambiziosa per un progetto di ambito accademico. Le qualità di trasparenza di replicazione sono valide per le entità *Car* e *Monitor*. Eventuali malfunzionamenti non sono preventivati e non vi sono procedure di recupero per stati imprevisti.

Una volta che tutti i nodi hanno reperito il file `ior.txt` generato dal name server, l'utente non necessita di ulteriori informazioni per l'esecuzione. Il reperimento di tale file non avviene automaticamente e tutti i nodi devono reperirlo con metodi manuali dal nodo eseguente il name server.

Capitolo 4

Analisi codice sorgente

Il questo capitolo viene analizzato il codice in maniera critica cercando di verificarne la congruità semantica e architetturale.

4.1 Analisi codice per funzionalità

4.1.1 Utilizzo di Ada.Calendar, delay

La gestione dei tempi è un punto focale. Il Clock viene concettualmente salvato una sola volta a inizio competizione nella funzione *PutOnPitLane*, chiaramente prima dell'inizio del ciclo while, il cui corpo esegue ad ogni settore. Il realtà il Clock viene salvato in altri 2 punti differenti del codice come si vedrà nel capitolo relativo ai miglioramenti. In *PutOnPitLane*, la entry *Enter* riceve tale parametro con access mode *in out*, effettua la *requeue* su *Demand* che a sua volta modificherà il parametro sommandoci il tempo di percorrenza del settore calcolato.

Nel frammento di codice sottostante si riassumono le istruzioni che coinvolgono i tempi in *PutOnPitLane* per tracciarne più facilmente i cambiamenti.

Listing 4.1: gestione del tempo in PutOnPitLane di Track

```
1 old_time := Clock;
2 my_time := old_time;
3 UpdateStatistics(my_time-Time_Of(2009,9,21), <other_pars
      >);
4 Sectors(Sectors'Last).Enter(my_time, <other_pars>);
5 Sectors(Sectors'Last).BookExit(old_time, <other_pars>);
6 my_time := old_time;
```

```
7  Sectors(Sectors'Last).Release(my_time, <other_pars>);
8  -- inizio while
9  Sectors(my_sect).BookExit(CarProperties, my_time);
10 UpdateProperties(my_time - old_time, <other_pars>);
11 old_time := my_time;
12 delay until my_time;
13 Sectors(my_sect).Release(my_time, <other_pars>);
14 BoxSector.Release(my_time, <other_pars>);
15 UpdateStatistics(old_time-Time_Of(2009,9,21), <other_pars
       >);
16 -- fine while
```

UpdateProperties e *UpdateStatistics* sono 2 procedure appartenenti rispettivamente a *Car* e *Race*, hanno il compito principale di aggiornare la GUI. L'estratto di codice rispecchia fedelmente quanto descritto nella relazione del progetto e le chiamate a entry sono conformi al modello *Release and Enter* descritto.

Nel frammento di codice sottostante si riassumono le istruzioni che coinvolgono i tempi in *Demand* per tracciarne i cambiamenti.

Listing 4.2: gestione del tempo in Demand di Lane

```
1 CalculateDriveTime(t, <other_pars>);
2 my_time := my_time + t;
```

CalculateDriveTime esegue conti per ottenere un valore indicativo della permanenza di un'auto in un settore. I calcoli (dipendenti dalle caratteristiche dell'auto e dal contesto della gara) vengono eseguiti su un tipo *Float* e successivamente trasformato in un tipo *Duration*.

L'utilizzo del Clock iniziale potrebbe inoltre apparire in un primo momento superfluo e fuorviante, in quanto tutti i calcoli nella competizione restituiscono valori che vengono sommati tra loro simulando un meccanismo di offset; in realtà esso è necessario, in quanto è previsto l'uso di *delay until* che richiede un'espressione di tipo *Time*. Il settaggio di un valore numerico di inizio avrebbe comportato l'utilizzo di *delay* che avrebbe richiesto un'espressione di tipo *Duration* e una sospensione relativa con i problemi che verranno descritti.

Listing 4.3: variante con utilizzo di delay

```
1 old_time := 0.0;
```

```
2 my_time := old_time;
3 -- other statements
4 delay my_time;
5 -- other statements
```

Tale codice esegue correttamente ma non ha i vantaggi di *delay until* rispetto a *delay* in presenza di prerilascio.

Delay until è *immune* al prerilascio a differenza della sospensione relativa di delay, intendendone con questa affermazione il comportamento semantico differente, non alludendo a fattori di altro genere (ad esempio la disabilitazione degli interrupt). Un'entità che esegue *delay* o *delay until* viene sospesa fino al verificarsi del trascorrimento del periodo di attesa (relativo o assoluto) richiesto che porterà l'entità in uno stato runnable. Gli svantaggi di *delay* contro *delay until* si possono apprezzare con un esempio.

Si supponga che un task calcoli un tempo di sospensione relativa t, ma che venga prerilasciato prima della successiva istruzione di *delay*. Al ritorno in esecuzione verrà invocata la *delay* con espressione t, ma effettivamente l'effetto della sospensione sarà t sommato al tempo in cui gli altri processi sono andati in esecuzione durante il prerilascio. Con *delay until* viene evitato tale inconveniente perché il calcolo del tempo t di sospensione piò avvenire in qualsiasi momento, precedente o successivo ad un eventuale prerilascio prima dell'invocazione della *delay until* che ha come espressione un tempo assoluto di risveglio.

Nel caso specifico del progetto esaminato ciò ha una implicazione nella gestione dei sorpassi degna di nota che verrà discussa nel paragrafo 4.1.3.

4.1.2 Decelerazione

La simulazione, per come progettata, non ammette un realismo tale da consentire decelerazioni alle auto. Le auto infatti decelerano istantaneamente quando sono precedute sulla stessa lane da auto con velocità inferiore. Di fatto, un comportamento tale assomiglia di più ad una corsa di cavalli dove i concorrenti non sono consapevoli dell'ambiente di gara e della posizione dei competitori.

Il cambio di velocità è istantaneo e non avviene durante il calcolo del tempo di percorrenza del settore da parte dell'auto, bensì durante la entry *Demand* per entrare in una lane. Nella entry viene controllato se il tempo d'uscita più alto tra le auto precedenti è maggiore del tempo d'uscita dell'auto che dovrebbe decelerare, e in tal caso i valori di velocità e tempo

di uscita vengono aggiornati agli stessi dell'auto con tempo d'uscita maggiore, come si può notare nel ramo `elseif` del codice riportato di seguito (`simulator-track.adb`).

Listing 4.4: decelerazione istantanea in Demand di Lane

```
1 if (my_time > exit_time) then
2         exit_time := my_time;
3         exit_speed := my_speed;
4 elsif (my_time < exit_time) then
5         my_time := exit_time;
6         if (my_speed > exit_speed) then
7                 my_speed := exit_speed;
8         end if;
9 end if;
```

In una prima analisi si può pensare che la modifica del tempo d'uscita comporti concettualmente una decelerazione della macchina durante il percorso, in realtà, quando viene eseguito *CalculateDriveTime*, viene tenuto conto della velocità potenziale di uscita per derivare i consumi di carburante. Tali consumi corrispondono ad una accelerazione dell'auto fino a fine settore, dove avverrà istantaneamente la decelerazione.

La presenza del solo moto uniformemente accelerato non contempla decelerazioni realistiche, ciò è voluto e dovuto ad una progettazione che semplifica tale aspetto. Con l'utilizzo di decelerazioni istantanee non sono inoltre contemplati eventuali possibili incidenti dovuti a tamponamenti.

4.1.3 Sorpassi

Da quanto già descritto, i sorpassi sono contemplati soltanto tra due settori, quando la entry *Demand* di **Sector** si occupa di scegliere la Lane meno trafficata come si mostra nel codice seguente (`simulator-track.adb`).

Listing 4.5: calcolo della corsia meno trafficata

```
1 n_lane := 1;
2 for Index in 1.. Multiplicity loop
3     if (LaneCounter(Index) < LaneCounter(n_lane)) then
4         n_lane := Index;
5     end if;
```

```
6 end loop;
```

Ancora una volta è evidente che *Track* è l'unico gestore attivo della gara essendo l'unico componente che prende decisioni. Non sono le auto che decidono spontaneamente se effettuare un sorpasso.

Analizzando il caso di uscita da un settore da parte di auto con stessi tempi di uscita, il meccanismo che prevede l'utilizzo di *delay* descritto nel paragrafo 4.1.1 risulta meno elegante. I tempi effettivi di attesa sarebbero potenzialmente soggetti a sospensioni prolungate pregiudicando l'ordine di uscita da un settore. Il workaround implementato (tramite *Release and Enter* e *BookExit*) verrà illustrato aggirare il problema di risvegli inattesi. Una soluzione in principio sarebbe possibile lato compilazione: un compilatore ottimizzato che cerca di evitare il prerilascio del task tra l'operazione di assegnamento del tempo t e l'esecuzione della *delay* potrebbe in certe situazioni triviali aggirare il problema..

4.1.4 Gestione dei box

Ulteriore semplificazione risiede nel settore dei box che ha molteplicità pari al numero di auto in gara. Da un punto di vista più realistico i box dovrebbero avere molteplicità 1 (una sola lane) e strategie di gara potrebbero avvenire in questa fase per favorire sorpassi all'interno del settore box. Impostando una molteplicità pari al numero di auto si crea una situazione che consente strategie di gara più realistiche rispetto a ciò che sarebbe consentito nel prevedere un box con una sola lane.

4.1.5 Gestione della competizione

Una scelta forte alla base della progettazione è stata la pianificazione delle entità che rappresentano i concorrenti come task anonimi all'interno di *Track*. Tale scelta ha come conseguenza il fatto che i concorrenti non sono autonomi nelle scelte di gara. Essi non hanno un flusso di controllo proprio e non intraprendono azioni sulla pista spontaneamente.

La scelta della lane da percorrere, eventuali sorpassi, accodamenti e cambi di velocità vengono infatti decisi dalla componente *Track* per mezzo delle entry di *Sector* e *Lane*, nelle quali risiede la logica di gara.

La progettazione di un sistema che prevede l'utilizzo di task con logica di gara interna alle componenti *Car* risulta molto complesso e forse troppo al di sopra degli scopi che può avere un progetto che risolve problematiche di concorrenza.

Dall'analisi del progetto si evince quindi che la progettazione ha cercato di semplificare il comportamento reale di una simulazione di Formula1 realistica. Talvolta ciò viene fatto per simularne le caratteristiche tramite stratagemmi (molteplicità lane nel settore box) e talvolta alterando la natura intrinseca delle entità presenti (*Track* decide i comportamenti delle auto).

4.2 Problematiche

Sebbene lo studio di esempi di codice Ada durante il corso sia stato adeguatamente approfondito, affrontare codice scritto da terzi è risultato particolarmente ostico.

La sintassi e il linguaggio hanno rappresentato in primis una difficoltà non indifferente, sicuramente non paragonabile allo sforzo (maggiore) di scrivere codice Ada ex novo al quale sono andati incontro i realizzatori del simulatore.

Il codice è risultato purtroppo poco commentato e manutenibile: modifiche minimali che a ragione non dovrebbero intaccare la logica del sistema, in realtà hanno spesso comportato errori dalla difficile interpretazione a livello di compilazione.

Inoltre, confrontare output tra differenti esecuzioni in simulazioni non banali ha richiesto l'analisi dei log stampati su terminale oltre che i risultati sulla componente Monitor. Tali stampe sono sicuramente state utili agli sviluppatori per testare il corretto funzionamento ma di scarso aiuto se non ben consapevoli della struttura del codice.

Capitolo 5

Test e comportamenti attesi

A seguito dello studio del codice e dopo aver appreso il funzionamento logico globale del sistema, si sono eseguiti test in maniera tale da testare il comportamento di più competizioni con lo stesso input e su input differenti.

5.1 Esecuzione locale

Le simulazioni sono state eseguite su 2 macchine con caratteristiche hardware differenti. Tale fattore però non rappresenta una vera discriminante per valutare output ottenuti. Onde evitare una eccessiva prolissità, si mostrano i risultati di un solo test ben articolato.

5.1.1 Input

Si mostrano di seguito gli input delle competizioni per il test.

	Value
PitLane Order	by ID
Laps	5
Cars	6
user requests during execution	none

	Pilot1	**Pilot2**	**Pilot3**	**Pilot4**	**Pilot5**	**Pilot6**
tyres	slick	slick	slick	slick	slick	slick
performance	75	60	90	50	50	100
fuel	50	45	55	30	30	60
fuel limit	10	10	15	20	5	5
tyres limit	0,90	0,90	0,95	0,85	0,70	0,70

Le auto si registrano con dei parametri in maniera da consentire una competizione né banale né troppo irrealistica.

5.1.2 Output

Le auto si iscrivono alla gara in ordine in base all'identificativo del pilota. Appare chiaro che la situazione creata dovrebbe servire output identici per esecuzioni diverse, in quanto vi dovrebbe essere assenza di non determinismo. Il risultato non conferma questa ipotesi. Si mostrano degli screenshot delle istanze dei monitor a fine competizione, dati sui quali si effettuano i confronti.

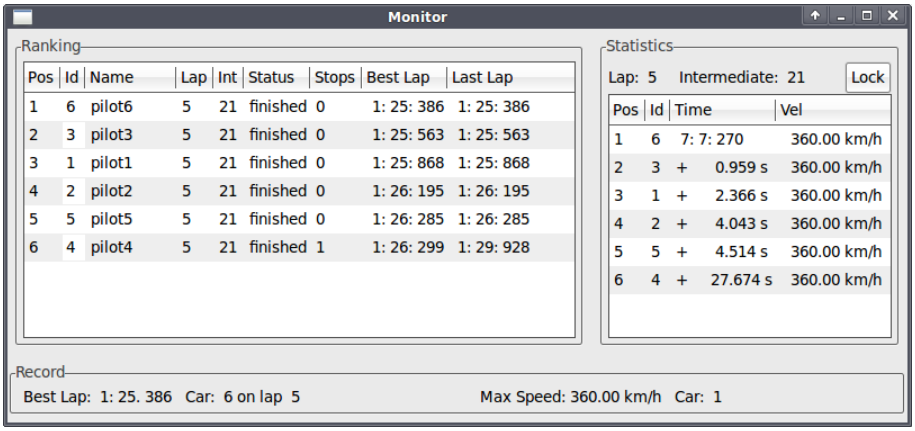

Figura 5.1: Iterazione 1

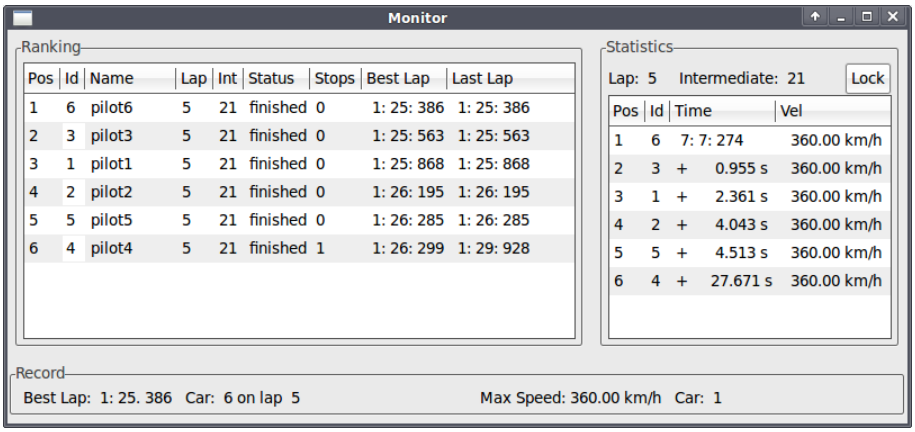

Figura 5.2: Iterazione 2

Si nota che ogni competizione differisce per alcuni millesimi di secondo. Nel capitolo di descrizione del progetto si è anticipato che la granularità dell'orologio (*Ada.Calendar*) è più lasca dell'orologio artificiale creato ed usato

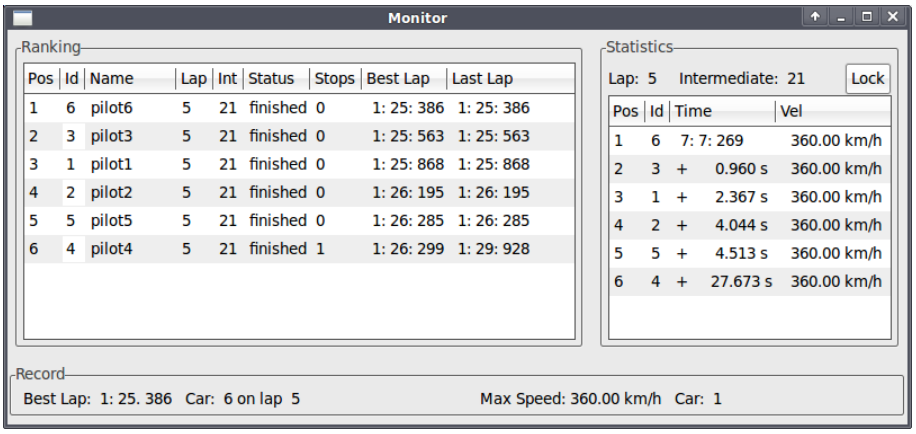

Figura 5.3: Iterazione 3

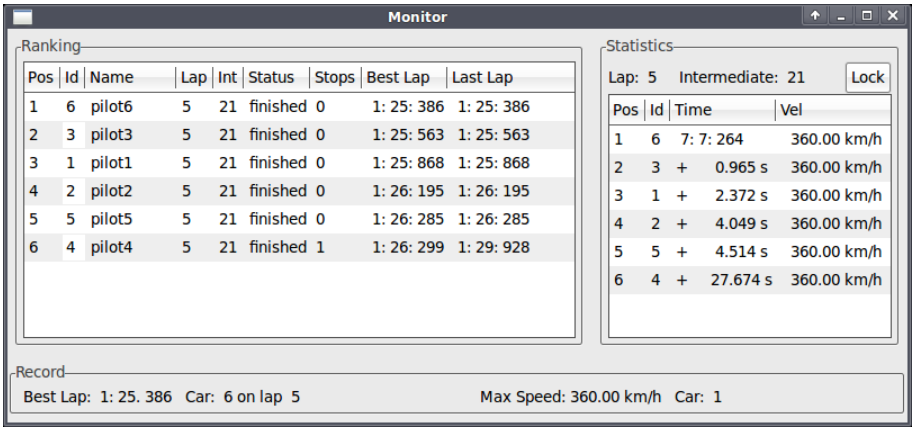

Figura 5.4: Iterazione 4

dalla simulazione. Esso infatti viene trasformato da un tipo *Float* a un tipo *Duration* e ciò consente di avere una granularità più fine rispetto a quella di *Ada.Calendar*[1]. Uno scenario del genere non è comunque tollerabile ed è indice della presenza di almeno una fonte di non determinismo.

Come descritto nella relazione del progetto e da quanto si può evincere studiando il codice, la *delay until* dovrebbe essere puramente un fattore di slow down che non intacca la vera esecuzione della simulazione. Le chiamate su entry e i riaccodamenti tramite requeue dovrebbero avvenire in maniera coerente e sempre preservando gli ordini di uscita dai settori. Annullando il fattore di slow down sarebbe inoltre possibile eseguire test più velocemente e confrontare un numero maggiore di output.

[1]La granularità dell'orologio varia da 100 μs a 20ms

Sono stati eseguiti inoltre iterazioni sul progetto compilato senza l'istruzione `delay until my_time;` presente nella procedura *PutOnPitLane* di *Track*. I risultati in output ottenuti da simulazioni distinte con stessi input sono risultati essere *totalmente* casuali rispetto a simulazioni che usano *delay until*. Si mostrano degli screenshot delle istanze dei monitor a fine competizione.

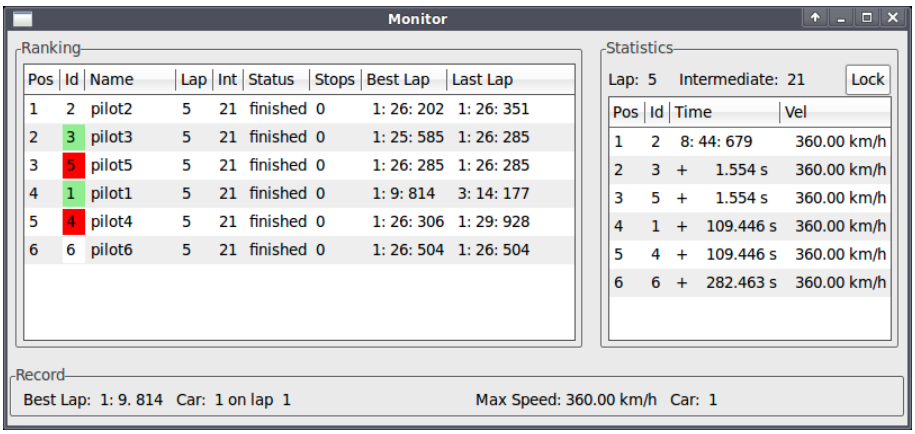

Figura 5.5: Iterazione 5 (senza delay until)

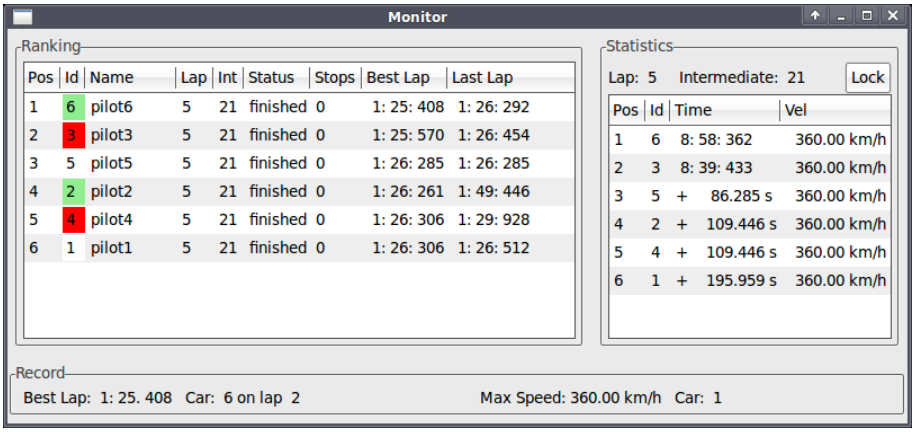

Figura 5.6: Iterazione 6 (senza delay until)

5.2 Esecuzione distribuita

Esecuzioni per testare l'effettivo funzionamento corretto in scenari distribuiti sono stati effettuati con la collaborazione di colleghi.

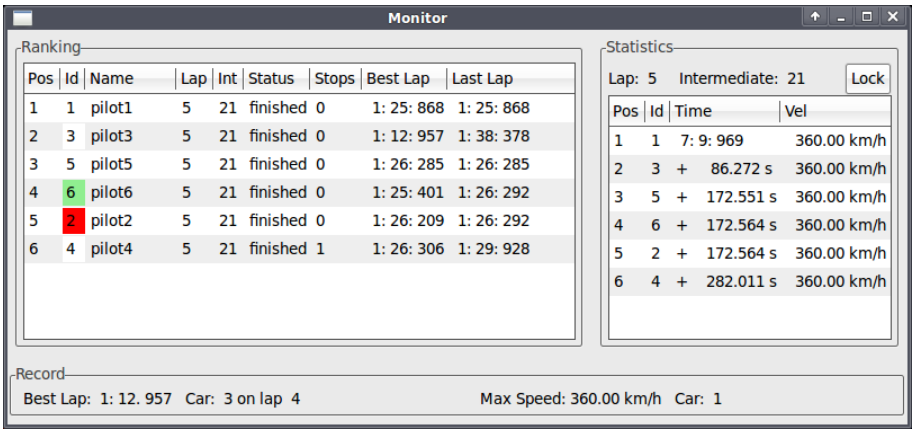

Figura 5.7: Iterazione 7 (senza delay until)

Sono stati eseguiti test in maniera tale da verificare che qualsiasi componente fosse capace di eseguire remotamente. Dopo ogni esecuzione è stato necessario riavviare il *name server* con il comando:

```
killall po_cos_naming
```

In tabella si mostrano le varie esecuzioni testate.

	Macchina 1	**Macchina 2**	**Risultato**
Test 1	Name Server Track Race 1 Monitor 1 Car	2 Car 2 Monitor	Esecuzione corretta
Test 2	Name Server 2 Monitor 1 Car	Track Race 2 Car 2 Monitor	Esecuzione corretta
Test 3	Name Server Race 2 Car 2 Monitor	Track 1 Car 2 Monitor	Esecuzione corretta
Test 4	Name Server Track 2 Car 2 Monitor	Race 1 Car 2 Monitor	Esecuzione corretta

Inoltre i risultati di una competizione distribuita sono risultati essere gli stessi della stessa competizione (stessi input) eseguita localmente.

5.3 Considerazioni

La rimozione della `delay until my_time` causa la rimozione della possibilità di sospensione dell'esecuzione. Simulazioni con `delay 0.0` e `delay 1.0` al posto di `delay until my_time` mostrano risultati intermedi ad esecuzioni con `delay until my_time` ed esecuzioni senza sospensione.

Si citano delle frasi tratte dalla relazione del progetto 2008/'09 (pag. 26, par 5.1.2):

> *Le procedure di delay sono quindi diventate un semplice fattore di slow down, ottenendo un protocollo corretto a prescindere dai tempi utilizzati.*
>
> *[...] la progettazione definitiva ha portato ad un protocollo solido e robusto, dipendente solo dai tempi calcolati e non dal tempo "reale" di esecuzione dei vari task.*

Da quanto si è potuto testare tali affermazioni sembrano essere confutate. Da tali risultati si evince quindi esservi la presenza di una falla nel sistema che compromette la correttezza temporale. Tale incoerenza crea in primis situazioni di non determinismo su simulazioni con stessi input, e in secundis la totale perdita del controllo della gara che di fatto diventa del tutto casuale.

Si giudica grave tale problema di insoddisfacimento di un requisito implicito, probabilmente dovuto ad una errata progettazione. Si cercherà di scoprirne le cause a monte, siano esse derivanti dalla progettazioni o dal linguaggio (ad esempio a causa della granularità dell'orologio).

Capitolo 6

Confronto soluzioni

Alla luce della progettazione proposta e di quella invece implementata nel prodotto preesistente si confrontato le soluzioni da un punto di vista critico sul piano teorico.

6.1 Concorrenza

Sviluppare una soluzione per la specifica data porta sicuramente a scontrarsi con delle problematiche di fondo inevitabili. Le due progettazioni oggetto di confronto, ovvero la progettazione **P3** e quella proposta dai Dottori Navarin e Zorzan nella relazione allegata al prodotto, hanno infatti molti aspetti in comune.

Il progetto preesistente soddisfa appieno i requisiti posti dalla specifica e tutte le entità tralasciate nella progettazione proposta **P3** vengono tenute conto per la creazione del sistema in tutte le sue parti.

Appare naturale identificare la presenza delle entità rappresentanti i concorrenti, i settori della pista e le varie corsie, entità ritenute interessanti per la corretta progettazione di fondo.

L'individuazione dell'entità rappresentante il concorrente come attiva è comune alle progettazioni, anche se con qualche differenza. Nel prodotto preesistente il task del concorrente esegue in remoto presso una RCI creando un task anonimo, quindi i flussi di controllo delle entità attive sono presenti su un solo nodo del sistema, quello esponente la RCI.

Nella progettazione **P3** ogni entità attiva risiede ed esegue su un proprio nodo che può essere differente da quello dove è presente la RCI.

La problematica principale affrontata nella relazione del prodotto è la soluzione trovata al problema dell'accodamento sul settore successivo preservando l'ordine di uscita dal settore precedente. Nel paragrafo 3.1.3 vengono illustrate le procedure di *Release and Enter* e *BookExit* progettate per ovviare al problema e il riaccodamento su coda interna. È evidente che tale soluzione è emersa a seguito di progettazioni non realizzabili simili a quelle descritte in **P1** e **P2**.

Molti dettagli di natura più o meno implementativa sono apprezzabili solo nel prodotto finale. Ad esempio, le entità rappresentanti i settori e le corsie sono concettualmente parte di una entità rappresentante il tracciato, e l'implementazione dei monitor per la presentazione dei tempi di gara viene considerata accessoria alle problematiche discusse nella progettazione **P3**.

6.2 Distribuzione

La scelta di categorizzare l'entità rappresentante i concorrenti con `pragma Remote_Types` risulta scontata come quella di categorizzare il tracciato di gara con `pragma Remote_Call_Interface`.

Il lavoro di implementazione della distribuzione va sicuramente al di là di quando esposto nella relazione. Lo si può apprezzare nella decorazione del codice come nel file di configurazione delle partizioni che però non si affronta nella presente.

6.3 Considerazioni

Seppur vi siano differenze tra la soluzione **P3** e quella del prodotto, non si possono non notare le stesse problematiche emerse. Il prodotto implementa un sistema complesso che per sua natura ha comportato la scelta di certe decisioni progettuali. Si pensi ad esempio alla problematica del passaggio tra due settori implementata con la procedura di *Release and Enter* in maniera leggermente diversa da quanto appare possibile fare con la progettazione **P3**; o come avviene l'esecuzione effettiva dei task del concorrenti.

Il progetto viene considerato molto valido e la sua implementazione supera le aspettative di un progetto per l'esame a cui fa riferimento. Esso aggiunge molta complessità ad una progettazione basilare mirata a presentare un protocollo corretto.

Capitolo 7

Miglioramenti architetturali

Il questo capitolo vengono illustrati alcuni miglioramenti architetturali e correzioni che si possono apportare al progetto. Si fa notare che il progetto, pur vertendo su problematiche di concorrenza e distribuzione, è migliorabile dal punto di vista del realismo della simulazione della gara.

7.1 Miglioramenti di tipo A1

La gestione dell'accesso ai box può essere migliorata per essere resa più realistica. Originariamente, se un'auto richiedeva un pitstop, questo avveniva al successivo raggiungimento del settore dei box, l'auto entrava quindi in tale settore al posto del corrispondente settore della pista. L'auto percorreva l'intero settore alla velocità massima di 80km/h, senza però fermarsi concettualmente per effettuare il pitstop. Inoltre per semplicità viene settato il numero di lane pari al numero di concorrenti al fine di evitare qualsiasi tipo di congestione.

7.1.1 Introduzione della piazzola di sosta

Il miglioramento che si propone prevede l'utilizzo di un'ulteriore risorsa protetta che rappresenta la piazzola di sosta del box, entità denominata *Piazzola*. L'auto viene accodata sulla entry *StopPiazzola* che ha il compito di aggiornare i parametri dell'auto richiesti con il pitstop tramite entry *CalculatePitStopTime*. Il tempo calcolato per il pitstop è lo stesso che veniva calcolato dalla procedura *CalculateDriveTime* in caso di esecuzione in un settore box, ma all'uscita dalla risorsa protetta la velocità dell'auto sarà pari a zero, ovvero l'auto dovrà tornare ad accelerare con moto uniforme, aumentandone quindi il realismo di gara.

La molteplicità delle corsie potrebbe venire ridotta a uno, come in un caso reale. Ogni auto possiede una piazzola di sosta, ma sviluppi futuri (ad esempio con l'introduzioni di scuderie alle quali possono appartenere più di una sola auto) sono tenuti conto, consentendo ad auto facenti riferimento ad un'unica piazzola di sosta di accodarsi come può avvenire realmente.

Si presentano 2 problemi per l'attuazione di questo miglioramento:

- per ottenere uno scenario che possa prevedere congestioni nei box, appare sensato pensare di settare la molteplicità delle lane a uno. Ciò comporta l'utilizzo di *requeue* su *StopPiazzola* trasferendo la chiamata dentro la nuova risorsa protetta *Piazzola*. Purtroppo, per come è progettato il simulatore non è possibile riaccodare l'auto sulla lane dei box per permettere la continuazione della corsa fino all'uscita del settore: si perdono infatti i dati del settore e le modifiche ai valori di *my_fuel* e *my_consumption* (rispettivamente carburante e consumi delle gomme). Pur essendo parametri con access mode `out` non vengono preservati dopo il riaccodamento su entry successivamente a modifiche. Inoltre la signature della entry di *Piazzola* deve necessariamente essere la stessa di quella della entry dalla quale viene eseguita la *requeue* (sono presenti parametri non concettualmente necessari in *Piazzola*) e semanticamente non ha senso riaccodare su un tipo di entità differente.

- senza utilizzare la procedura di *requeue*, si potrebbe accodare la richiesta direttamente su *StopPiazzola*, operazione bloccante, che non lascerebbe possibilità di esecuzione ad altri concorrenti dentro *Piazzola* durante l'esecuzione del task. La risorsa protetta *Lane* rimarrebbe bloccata, motivo che costringe all'utilizzo di un settore dei box con molteplicità pari alle auto (o in sviluppi futuri al numero di scuderie in gara). L'utilizzo di un meccanismo di trasferimento di chiamata asincrono tramite `select` statement e `then abort` part non risolverebbe il problema, dovuto ad una progettazione ignara di una possibile estensione in tal senso.

Il secondo punto è di più pulita implementazione e non causa modifiche profonde al sistema, richieste invece al primo punto. Si mostra il codice implementato e testato.

Listing 7.1: gestione delle Piazzole

```
1 type Piazzola_Array_T is array
2       (Positive range <>) of access Piazzola;
```

```
3 Piazzole : access Piazzola_Array_T;
4 Piazzole:= new Piazzola_Array_T (1 .. MaxId);
5
6 procedure AddPiazzola(n_piazzola : Natural;
7                       altraPiazzola : Boolean) is
8    begin
9       Piazzole(n_piazzola) := new Piazzola(n_piazzola);
10 end AddPiazzola;
```

Listing 7.2: procedura CalculatePitStopTime

```
1 procedure CalculatePitStopTime(<pars>; finish: out
     Duration)  is
2    t : Float := 0.0;
3    begin
4       <calcolo_tempi_di_pitstop>
5       [...]
6       t := t + <tempo_di_pitstop>;
7
8       <aggiornamento_dati_auto>
9       my_consumption := <ripristino_il_consumo_gomme>
10      my_fuel := <aggiungo_carburante>
11
12      finish := Duration(t);
13   end CalculatePitStopTime;
```

Listing 7.3: risorsa protetta Piazzola

```
1 protected type Piazzola(id : CarId_T) is
2    entry StopPiazzola(<pars>);
3    private
4       PiazzolaId : CarId_T := id;
5    end Piazzola;
6
7    protected body Piazzola is
8       entry StopPiazzola(<pars>)  when True is
9          t : Duration := 0.0;
10      begin
11         CalculatePitStopTime(<pars>, t);
12         my_time := my_time + t;
13         my_duration := t;
14         my_speed := 0.0;
```

```
15        end StopPiazzola;
16     end Piazzola;
```

Listing 7.4: modifiche a Demand in Lane

```
1  entry Demand(<pars>)  when True is
2        t : Duration := 0.0;
3     begin
4        exit_time := my_clock;
5        my_temp_fuel := my_fuel;
6        my_temp_consumption := my_consumption;
7        if (IsBox) then
8           Piazzole(CarId).StopPiazzola(<pars>);
9        end if;
10
11       CalculateDriveTime(<pars>; t);
12       my_time := my_time + t;
13       [...]
14       end Demand;
```

Listing 7.5: modifiche al task *The_Race*

```
1  accept CarRegister (id: out CarId_T; Success: out Boolean
       )  do
2     <car_registration_statements>
3     n_piazzole := n_piazzole + 1;
4     Track.AddPiazzola(n_piazzole, True);
5     <other_statements>
6  end CarRegister;
```

Test su vari input sono stati eseguiti e riportano risultati previsti e, in casi simili, congrui con quelli del progetto originale.

7.1.2 Approccio per la decelerazione nei box

Un miglioramento teorico non implementato potrebbe prevedere una parte di decelerazione nel settore dei box per permettere all'auto di arrivare a velocità nulla alla piazzola di sosta oltre a farla partire da ferma. La fase di decelerazione può essere vista come una estensione allo scenario presentato nel paragrafo precedente.

Il settore dei box viene diviso in 2 parti, le cui lunghezze sommano al corrispondente settore descritto nel file xml che descrive il tracciato. La prima parte, chiamata *Box1*, serve per la decelerazione fino all'entrata nella piazzola di sosta, successivamente la seconda parte, chiamata *Box2*, serve per l'accelerazione da ferma.

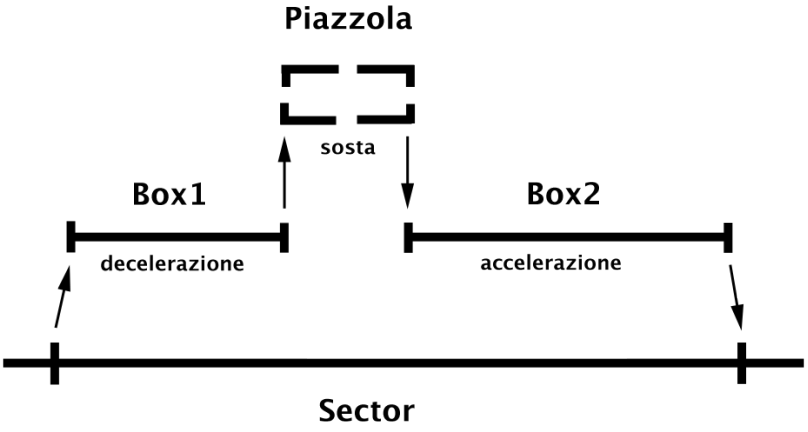

Figura 7.1: schematizzazione di utilizzo della piazzola di sosta

La configurazione corretta dei settori di box avviene nella funzione *ReadTrackConf* di *Track*.

Il protocollo prevede i seguenti passi:

1. l'auto accede al Box1 quando necessita di un pitstop;
2. viene calcolato un tempo di moto decelerato in base alle caratteristiche dell'auto;
3. l'auto viene riaccodata tramite *requeue* o invoca la entry *StopPiazzola* relativa alla sua scuderia[1];
4. vengono effettuati i calcoli del tempo di pitstop tramite la procedura *CalculatePitStopTime*;
5. l'auto viene sospesa per il tempo necessario alla decelerazione e al pitstop (il controllo non ritorna a *PutOnPitLane*, i tempi calcolati finora verrebbero persi al passo successivo);

[1]nel paragrafo precedente si è spiegato essere preferibile l'invocazione su entry (operazione potenzialmente bloccante), modulo il fatto di mantenere la molteplicità delle lane pari al numero delle auto (o delle scuderie)

6. l'auto viene riaccodata tramite *requeue* su Box2;
7. viene calcolato un tempo di moto uniformemente accelerato in base alle caratteristiche dell'auto tramite la procedura *CalculateDriveTime*;
8. il controllo ritorna a *PutOnPitLane* che eseguirà il delay del tempo calcolato nel Box2;
9. l'auto viene riaccodata sul settore successivo.

7.2 Miglioramenti di tipo A3

7.2.1 Utilizzo del Clock

Chiamate al Clock vengono utilizzate 3 volte, ognuna di esse ad inizio competizione per salvare il tempo di inizio.

1. per ogni auto all'inizio dell'esecuzione di *PutOnPitLane*;
2. per ogni esecuzione di *Demand* in *Lane*;
3. nell'inizializzazione degli oggetti di tipo *ExitRecord*.

Salvando il Clock in punti differenti, anche i corrispondenti valori risulteranno differenti.

Il Clock al punto 1 è necessario, come già visto nel capitolo di analisi. Ogni esecuzione di *PutOnPitLane* salverà un clock diverso a seconda di quando l'auto viene posizionata sulla griglia di partenza (chiamata sulla entry). Nella progettazione è stato implementato un protocollo di risveglio delle auto per l'inizio della competizione dipendente dal tempo in cui vengono posizionate sulla griglia di partenza e non in base al Clock. Ciò non appare quindi desiderabile. Anche con lo stesso tempo iniziale il protocollo risulterebbe robusto in quanto gli offset temporali sarebbero gli stessi e l'ordine dei vari accodamenti su entry e di percorrenza nella competizione preservati.

I Clock al punto 2 sono superflui e introducono un ritardo nel tempi finali osservabili sul Monitor. La chiamata al Clock comporta istruzioni macchina non necessarie, in quanto il corpo della *Demand* verifica una condizione sempre a vero per il primo task che esegue sulla risorsa protetta Lane. È consigliabile la rimozione di Clock non necessari.

Listing 7.6: utilizzo superfluo del Clock al punto 2

```
1 exit_time := Clock;
2 -- other statements
3 if (my_time > exit_time) then
4 -- other statements
```

Tale Clock deve rappresentare un valore di time *Time* che riferisce ad una data passata (in quanto *my_time* è sicuramente maggiore per il primo task che esegue su *Demand*).

Passare il Clock salvato al punto 1 consente una simulazione corretta. Per applicare tale modifica occorre modificare la signature delle entry *Enter*,

Release e *Demand* con l'aggiunta di un parametro con access mode in.

Listing 7.7: utilizzo errato di delay

```
1 entry <entry_name>(my_clock : in Time; <other_pars>)
```

I Clock salvati al punto 3 servono soltanto per avere un valore di tipo *Time* per decidere l'ordine di uscita nella entry *BookExit*. Il valore deve riferire ad una data passata e precedente a *my_time* per il corretto funzionamento.

7.2.2 Non determinismo causato dal Clock

Come descritto nel capitolo di Test, esecuzioni su input uguali mostrano output differenti osservabili sulla componente Monitor (nella sezione "Statistics"). Dopo un'attenta analisi e una moltitudine di test, si scopre la causa di tale non determinismo nel Clock listato al punto 1 nella sezione 7.2.1.

Tale Clock risulta differente per ogni esecuzione di *PutOnPitLane* ma, come spiegato precedentemente, ciò non è né necessario né desiderabile. Ogni esecuzione della entry avviene in tempi diversi e ciò pregiudica le statistiche mostrate su Monitor.

7.2.3 Miglioramento della gestione del tempo

Si vuole ottenere una situazione tale da annullare il non determinismo in tal senso ed ottenere output identici per simulazioni con input uguali. Il miglioramento che si propone è la modifica del momento in cui viene salvato l'unico Clock usato in *PutOnPitLane*, al quale vogliamo ora che vi accedano tutti i task che eseguono il corpo di tale entry. Si fa carico Track di salvare il Clock nella funzione *StartRace*, che sarà lo stesso per tutte le Car.

Listing 7.8: dichiarazione di *my_start_time*

```
1 package body Simulator.Track is
2     <data declarations>
3     my_start_time : Time;
4     <procedures, types, Sector and Lane declarations>
5 end Simulator.Track;
```

Listing 7.9: salvataggio del Clock in *my_start_time*

```
1 function StartRace(<pars>) return Boolean is
```

```
2 begin
3     <statements StartRace body>
4     my_start_time := Clock;
5     <other statements StartRace body>
6 end StartRace;
```

Listing 7.10: utilizzo del Clock a inizio gara

```
1 procedure PutOnPitLane(<pars>) is
2     <data declarations>
3 begin
4     old_time := my_start_time;
5     <PutOnPitLane body>
6 end PutOnPitLane;
```

Con questa modifica i tempi osservabili su Monitor risultano coerenti in diverse esecuzioni. Questo miglioramento corregge inoltre una falla che sembrava essere dovuta al protocollo di gestione della competizione. È ora possibile rimuovere lo statement di sospensione (assoluta o relativa che sia), facendo diventare l'utilizzo di *delay until* un vero e solo fattore di slow down.

Concettualmente le auto partono allo stesso tempo (aspetto desiderabile) pur continuando a venir *svegliate* tramite `Signal` (procedura di *Semaphore*) dall'auto che precede ognuna di esse.

La presenza nel progetto originale di Clock salvati in momenti diversi e la rimozione della sospensione pregiudicavano la competizione. Di fatto ciò era un non soddisfacimento di correttezza funzionale risolto con lo spostamento del salvataggio del Clock.

Capitolo 8

Miglioramenti funzionali

Si discutono di seguito eventuali miglioramenti funzionali apportabili al progetto per migliorarne l'usabilità, l'interfacciamento con l'utente ed estensioni minori.

8.1 Miglioramenti di tipo F1

Ciò che si vuole ottenere introducendo la gestione dinamica delle condizioni meteorologiche è la possibilità di modificare la proprietà `Weather` (che può assumere valori `Dry` e `Wet`) delle risorse protette *Lane*. Le condizioni del tracciato influiscono sul tempo di percorrenza calcolato da *CalculateDriveTime*.

Inizialmente si era pensato di creare un nuovo componente remotizzabile per la gestione del meteo, ma oltre ad aggiungere complessità non necessaria non era la soluzione migliore, in quanto solo il nodo eseguente *Track* deve avere la possibilità di modificare le condizioni climatiche.

Il miglioramento che si propone prevede la modifica della GUI di *Track* per permettere il cambiamento delle condizioni delle *Lane* nei diversi *Sector* dopo che il tracciato sia stato caricato, in un qualsiasi momento a runtime. La logica va modificata aggiungendo le seguenti procedure *UpdateWeather*:

Listing 8.1: UpdateWeather in risorsa protetta Lane_T

```
1 procedure UpdateWeather(my_weather : Weather_T) is
2    begin
3        Weather := my_weather;
4 end UpdateWeather;
```

Listing 8.2: UpdateWeather in Track

```
1  procedure UpdateWeather(Sector : in Natural;
2                          Lane : in Natural;
3                          Weather : in Boolean) is
4     begin
5
6       if ((Corsie(Sector) /= null) and then
7           (Corsie(Sector)(Lane) /= null)) then
8          if (Weather = True) then
9            Corsie(Sector)(Lane).UpdateWeather(Weather_T(
                Dry));
10         else
11           Corsie(Sector)(Lane).UpdateWeather(Weather_T(
                Wet));
12         end if;
13       else
14         Simulator.TrackGui.Print_NoLock (The selected
              Sector or Lane does not exist);
15       end if;
16 end UpdateWeather;
```

La prima procedura è presente all'interno di *Lane* e serve a settare il tempo scelto da GUI. La seconda procedura appartiene a *Track* ed è invocata direttamente dal pulsante "`Set Weather`" mostrato in figura 8.1.

Modifiche alla GUI risultato essere di scarso interesse in quanto si opera in linguaggio Ada con le librerie GtkAda. La GUI risultante viene mostrata in figura 8.1

Sono stati effettuati test per controllare la correttezza su più esecuzioni. Gli output hanno riportato i risultati positivi attesi.

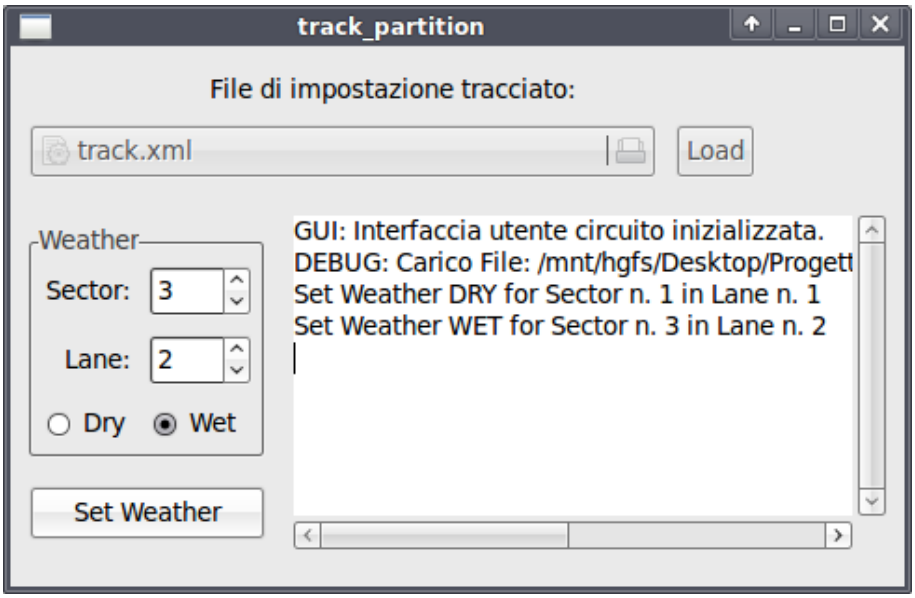

Figura 8.1: Track GUI modificata

8.2 Miglioramenti di tipo F2

8.2.1 Tecnologia

L'aggiunta della visualizzazione del tracciato e delle posizioni è una miglioria prettamente grafica partendo da una situazione dove la logica di aggiornamento delle GUI delle varie entità è già presente.

Non viene implementata una soluzione ma vengono consigliate le basi per lo sviluppo. L'aspetto principale da considerare è la scelta della tecnologia migliore per permettere la visualizzazione grafica di elementi non banali. Visualizzare il tracciato e le relative auto necessita in linea di principio l'uso di librerie grafiche esterne con potenzialità che GtkAda non offre.

Scegliendo l'implementazione dell'interfaccia in un secondo linguaggio (ad esempio Java), semplificherebbe la realizzazione della GUI magari usufruendo di tool grafici, l'aspetto negativo di tale approccio è il bisogno che si crea di far comunicare i due linguaggi tramite IDL e molto probabilmente questa complicazione è stata decisiva nella scelta di utilizzare le librerie GtkAda.

Le librerie grafiche Qt (*http://qt.nokia.com/*) a partire dalla metà del 2009 fortunatamente hanno ottenuto un riscontro sempre maggiore presso le comunità di sviluppatori per le più disparate piattaforme ed una versione ad hoc per in linguaggio Ada è stata sviluppata recentemente.

Il framework QtAda (*http://www.qtada.com/en/index.html*) si integra con Ada2005, non vi è bisogno di un secondo linguaggio e soprattutto non si

presenta il problema di far comunicare i due tramite interfacce IDL.

8.2.2 Problematiche di fondo

Si presenta un problema legato alla rappresentazione logica del tracciato. Lo schema XML usato per dichiarare i tracciati prevede l'attributo `level` obbligatorio nel tag `sector`, il valore di tale attributo indica la presenza di un rettilineo (valore pari a 0) o di un curva (valore positivo o negativo a seconda della curvatura). La descrizione non risulta adatta per una rappresentazione geometrica efficace della pista.

8.3 Miglioramenti di tipo F3

La gestione di un campionato rappresenta un'estensione che prevede una quantità non indifferente di modifiche al simulatore. Il progetto prevede un solo tracciato sul quale avviare una sola gara. Tutti i concorrenti non hanno quindi possibilità di scelta sulla competizione.

Lo scenario che si vorrebbe creare, oltre alla possibilità di eseguire una gara come già avviene, prevede:

1. Possibilità di avviare più gare (modalità campionato);
2. Possibilità di eseguire il giro di qualifica per ogni gara;
3. Possibilità di caricare diversi tracciati (realisticamente uno per ogni gara);
4. Possibilità di salvare lo stato del campionato inter-gara.

All'avvio del simulatore, il nodo eseguente *Track* permette la scelta della modalità campionato, e una serie di tracciati corrispondenti a tutte le piste del campionato vengono caricati. I concorrenti dovranno iscriversi ad ogni gara (non soltanto al campionato), poiché per ogni gara il concorrente può partecipare con prestazioni differenti. L'ordine di iscrizione non sarà importante in quanto vengono introdotti i giri di qualifica durante i quali viene selezionato il giro migliore (per tempo) di ogni concorrente, tempi utilizzati per il posizionamento sulla griglia di partenza della gara. Un punteggio viene attribuito ad ogni concorrente a competizione conclusa. Le gare vengono eseguite in serie e a fine campionato vengono salvati i risultati. Ogni gara viene avviata dall'utente del nodo sul quale esegue l'istanza di *Race*.

Il makefile del progetto viene esteso per permettere di avviare un numero di *Race* scelto dall'utente. Ogni *Race* si registra presso il name server.

La componente *Track* viene estesa per supportare tracciati multipli (il ché di fatto dovrebbe comportarne il cambiamento del nome in *Tracks*). La componente *Race* essendo un *Remote_ Type* può eseguire le diverse competizioni su nodi differenti (aspetto che peraltro permetterebbe esecuzioni in maniera parallela).

Alla fine di ogni gara il punteggio viene salvato in un file `xml` aggiornandolo incrementalmente. Nel caso in cui l'esecuzione del campionato venga terminata prima della fine, il file `xml` servirà per ripristinare lo stato raggiunto nell'esecuzione precedente.

8.3.1 Estensione punto 1

L'oggetto *RaceGroup* (file `simulator-controller.adb`) è già disponibile e ha un comportamento simile all'oggetto *Group* che si occupa di registrare i partecipanti. *RaceGroup* registra una sola gara, la modifica necessaria prevede la possibilità di registrare più gare. Ogni concorrente *Car* potrà quindi reperire distintamente le diverse gare inizializzate (aspetto non prettamente necessario nella progettazione che si sta esplicando).

La GUI di *Race* (file `simulator-race-gui.adb`) viene modificata disabilitando la possibilità di scelta di registrazione delle auto nel caso campionato. La GUI di *Car* (file `simulator-car-gui.adb`) viene modificata aggiungendo la possibilità di iscrizione alle diverse gare disponibili, per ognuna di esse con caratteristiche diverse potenzialmente. Ciò può essere fatto ripristinando la GUI di *Car* a termine gara per impostare i parametri per la competizione successiva.

8.3.2 Estensione punto 2

La GUI di *Race* (file `simulator-race-gui.adb`) viene modificata con l'inserimento di un pulsante per l'avvio del giro di qualifica per ottenere l'ordine di registrazione per la gara. Quando il giro di qualifica è terminato su richiesta del concorrente i tempi vengono memorizzati per derivarne l'ordine di inizio competizione. Un tempo limite viene utilizzato per garantire la terminazione di questa fase.

La GUI di *Car* (file `simulator-car-gui.adb`) viene modificata con l'inserimento di un pulsante per la terminazione del giro di qualifica. La scelta del giro migliore viene eseguita automaticamente e l'informazione viene inviata al componente *Race* che provvederà a creare l'ordine di partenza.

8.3.3 Estensione punto 3

L'estensione a *Track* prevede una modifica alla GUI per permettere all'utente di selezionare più tracciati (differenti file `xml`). La logica che occorre modificare risiede nella funzione *ReadTrackConf* che carica il tracciato per creare le strutture dati rappresentanti tutte le piste.

8.3.4 Estensione punto 4

Al termine di ogni competizione i risultati intermedi vengono aggiornati in automatico su un file `xml` assieme ai riferimenti ordinati ai file dei tracciati selezionati. Il file è presente sul nodo eseguente *Track*. Se un campionato

viene sospeso può essere ripristinato leggendo tale file. L'estensione a *Track* prevede la possibilità del caricamento del file con i punteggi parziali, se nessun file viene caricato un nuovo campionato viene inizializzato.

È necessaria anche l'implementazione di controlli per verificare che i concorrenti iscritti al campionato all'esecuzione precedente siano coerenti con quelli all'esecuzione successiva. Non vi dovrà essere un numero maggiore di concorrenti, un numero inferiore è consentito in quanto i concorrenti possono essersi ritirati dal campionato. L'assenza di un concorrente all'avvio di una gara del campionato è considerata come un ritiro (non vi è quindi la necessità di esplicitare tale funzione). Il nome del pilota viene utilizzato come discriminante, non vi possono essere quindi concorrenti diversi tra le varie competizioni del campionato.

Una simulazione interrotta durante l'esecuzione di una gara può essere ripresa dal salvataggio dello stato del sistema precedente, ovvero al termine dell'esecuzione della gara precedente. Tutti i risultati intermedi maturati con l'esecuzione dell'ultima gara non fanno parte del salvataggio dello stato nel file `xml`, uno scenario che vuole prevedere ciò risulta troppo ambizioso e occorrerebbe un algoritmo di distributed snapshots a supporto. Se la simulazione viene sospesa dopo l'iscrizione di alcuni concorrenti al giro di qualifica ma prima dell'inizio della gara, lo stato non viene salvato e all'esecuzione successiva i tempi del giro di qualifica saranno perduti. È comunque possibile estendere la progettazione per prevedere tale situazione (salvando lo stato anche dopo i giri di qualifica).

8.4 Miglioramenti minori

Si elencano miglioramenti minori che si sono notati durante l'analisi del codice e lo studio del progetto.

1. Per rendere trasparente il reperimento del file `ior.txt` si può copiare in un luogo pubblico tale file dal nodo eseguente il name server, luogo dal quale ogni nodo potrebbe reperirlo prima dell'esecuzione. Tale soluzione non prevederebbe trasparenza ai guasti nel nodo di residenza del file, ma semplificherebbe l'esecuzione distribuita che attualmente è possibile solo tramite scambio non automatico del file.

2. Se viene percorso un tratto con condizione climatica "Wet", la procedura *CalculateDriveTime* dovrebbe diminuire l'accelerazione di un terzo (come espresso nei commenti e come è realistico sia), in realtà essa viene ridotta a un terzo. Stessa cosa succede se l'auto monta gomme non adatte.

   Tale errore dovuto probabilmente a una svista in fase di programmazione non consente simulazioni realistiche e spesso pregiudica in partenza l'esito di una gara per concorrenti con determinate caratteristiche. La correzione è banale: è sufficiente modificare il denominatore nelle formule del calcolo dell'accelerazione.

Appendice A

Modifiche apportate

La specifica pone tra i requisiti opzionali la consegna del progetto comprendente le modifiche apportate. La relazione presenta già la descrizione delle modifiche che sono state implementate e il relativo codice; si allega comunque alla presente il progetto modificato.

Le modifiche comprendono:

- Correzione generale dell'utilizzo del Clock;
- Riprogettazione della gestione dei box;
- Cambio dinamico di condizioni meteorologiche;
- Pulitura codice.

I primi 3 punti vengono illustrati appropriatamente nel corso della relazione. Un side effect dello studio del progetto preesistente è stata la pulitura del codice che è risultata necessaria durante la fase di analisi per comprendere più facilmente il codice. Purtroppo i file di codice sorgente sono risultati a una prima lettura molto mal formattati e non indentati appropriatamente; si è ritenuto di buon senso applicare una formattazione che cerca di rispettare le GNAT Coding Style rules (*http://gcc.gnu.org/onlinedocs/gnat-style/*) con tab space uguale a 3. Ciò ha anche aiutato ad aumentare di gran lunga la leggibilità e l'*estetica* del codice.

Appendice B

Installazione del progetto

Nel capitolo 8 della relazione del progetto vengono brevemente descritte le dipendenze necessarie per eseguire il progetto sotto ambiente Linux, sottolineando il fatto che l'installazione su ambienti Windows e MacOS X non sono risultati fattibili a causa della mancanza di alcune librerie per determinate piattaforme. Riuscire ad installare un ambiente solido funzionante non è affatto stato triviale e merita spendere alcune righe a riguardo che sono sicuramente utili e integrative al capitolo della relazione sul progetto.

B.1 Ambiente per l'esecuzione

È indubbio che a diversi mesi di distanza dallo sviluppo del progetto le tecnologie sottostanti possano essere mutate. Nel mese di maggio 2010 l'installazione di un ambiente adatto alla compilazione e all'esecuzione del progetto si è ottenuta sulle seguenti piattaforme:

- Linux (distribuzione Xubuntu 9.04)
- Mac OS X (v. 10.6.4)

I componenti richiesti sono:

- compilatore Ada (ambiente GNAT 2005)
- GtkAda
- XMLAda
- PolyORB

Molteplici sono state le dipendenze di librerie riscontrate, complicando l'installazione dell'ambiente e rallentandolo di un certo numero di ore. Per eseguire il progetto era necessario il compilatore distribuito (`po_gnatdist`) che non risultava presente dopo l'installazione di PolyORB. Lo script di configure deve essere avviato con il seguente flag per la corretta installazione.

```
-with-appli-perso=‘‘dsa’’
```

Purtroppo le scarse documentazioni a riguardo, sia del progetto che del componente PolyORB, non avevano sottolineato tale necessità.

B.2 Problematiche riscontrate

B.2.1 Ambiente Linux

L'installazione su ambiente Linux è stata provata inizialmente su distribuzioni Ubuntu (versioni 9.04 e 10.04) ma l'installazione delle librerie GtkAda richiedeva l'installazione di altre librerie, tra le quali la libreria GTK+ che se installata su una piattaforma che utilizza l'ambiente grafico Gnome causa dei conflitti con le librerie preesistenti. Per tale motivo si è scelta la distribuzione Xubuntu con interfaccia grafica Xfce.
Essendo consci di questo problema apparentemente non risolvibile l'installazione si può eseguire facilmente scaricando i pacchetti necessari dal sito *http://libre.adacore.com* e installandoli nell'ordine indicato nell'elenco puntato al paragrafo B.1.

B.2.2 Ambiente MacOS X

Per quanto riguarda l'installazione su piattaforma Mac OS X, la procedura è risultata più complicata. L'installazione delle librerie GtkAda non andava a buon fine o le stesse non risultavano viste dal sistema in maniera corretta. Fortunatamente R-project fornisce librerie precompilate per Mac OS X (GTK+ v2.18.5) che hanno tutte le dipendenze soddisfatte.
Risolvendo questo unico punto critico, l'installazione delle componenti rimanenti termina. Con accorgimenti minori alle variabili PATH d'ambiente e al Makefile del progetto, il prototipo, così come l'IDE GPS, eseguono correttamente in un ambiente X11.

Appendice C

Tabella di versionamento

Versione	Data	Notes
1.0	13 settembre 2010	Prima stesura completa del documento.
2.0	17 ottobre 2010	Apportate correzioni richieste alla relazione. Presentata una soluzione alle problematiche di concorrenza riscontrate. Allegate modifiche al prodotto finale. Correzioni generali.