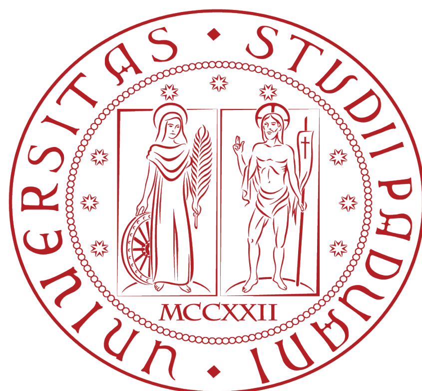
Università degli Studi di Padova

Corso di Laurea triennale in Statistica e Tecnologie Informatiche



Sistema configurabile per la generazione di allarmi real-time nell'ambito dell'autotrasporto di carburanti: modello di funzionamento e dimostratore.

Relatore: Prof. Nicola Zingirian

Dipartimento di Ingegneria dell'Informazione

Laureando: Francesco Masutti

Anno Accademico 2011/2012

Indice

1 Introduzione..1

1.1 Il sistema Click & Find 1

1.2 Requisiti tenuti presenti da Click & Find nella progettazione della piattaforma.................3

1.3 Architettura del sistema..3

2 Modello dell'applicazione...5

2.1 Obiettivo...5

2.2 Modellazione del sistema di allarmi...5

2.2.1 Esempi di automa 9

3 Sviluppo dell'applicazione..12

3.1 Schema generale di funzionamento..12

3.2 Configurazione...14

3.3 Evoluzione...20

4 Appendice..25

1 Introduzione

Il lavoro di tesi è stato svolto in collaborazione con l'azienda Click & Find operante nel campo dei Sistemi Informativi e delle Reti.

Click & Find, fondata nel 2000 da docenti dell'Università di Genova e dell'Università di Padova, ha come scopo quello di trasformare ricerche effettuate nel campo dei Sistemi Informativi e delle Reti in risultati industriali.

1.1 Il sistema Click & Find

Click & Find s.r.l è un'azienda che sviluppa sistemi e servizi per il controllo a distanza delle autocisterne per la distribuzione di carburanti. I prodotti e i servizi da essa erogati permettono di controllare i viaggi prevenendo possibili incidenti e/o furti.

L'impegno dell'azienda è rivolto alla progettazione e alla produzione di sistemi elettronici di telecontrollo di bordo, nonché alla erogazione di servizi telematici basati sulle informazioni provenienti in tempo reale sia dal trattore/motrice che dal semirimorchio/rimorchio (fig.1).

Alcune delle più comuni strumentazioni presenti sulle motrici sono: il tachigrafo digitale, i terminali operatori, i rilevatori di identità, le sonde di livello e il can-bus.

Sulle autobotti possono essere installati misuratori di prodotto, rilevatori di temperatura, di livelli, di pressione, di inclinazione, di azionamenti elettrici e pneumatici e altri sensori.

Il Centro Servizi Click & Find rende disponibili al cliente, per mezzo di diverse interfacce telematiche, tutti i dati raccolti.

La piattaforma Click & Find è studiata appositamente per fornire dati utili ad applicazioni avanzate per il controllo e l'osservazione di mezzi a distanza (RVMC Remote Vehicle Monitoring and Control Applications).

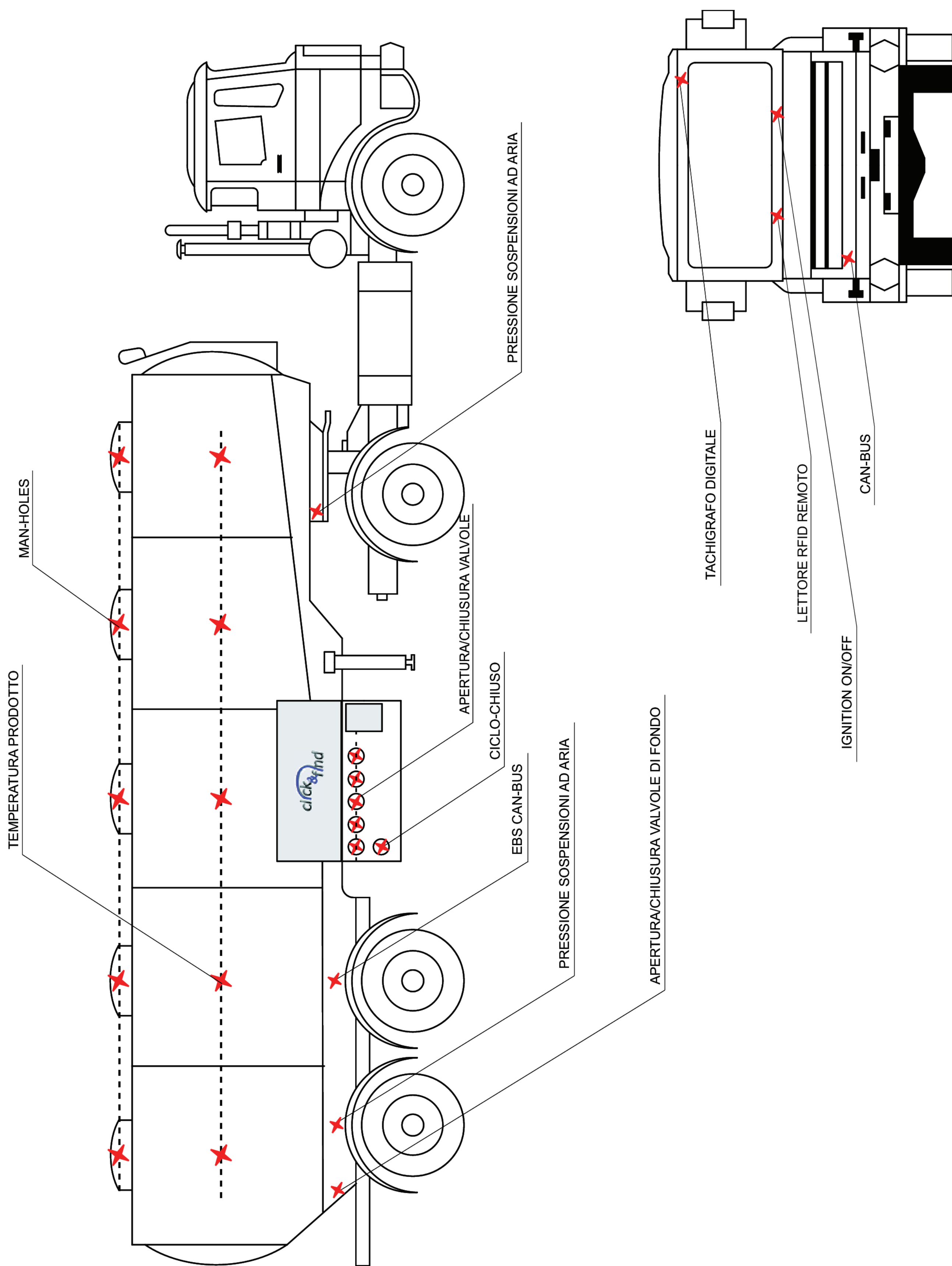


Figura 1

1.2 Requisiti tenuti presenti da Click & Find nella progettazione della piattaforma

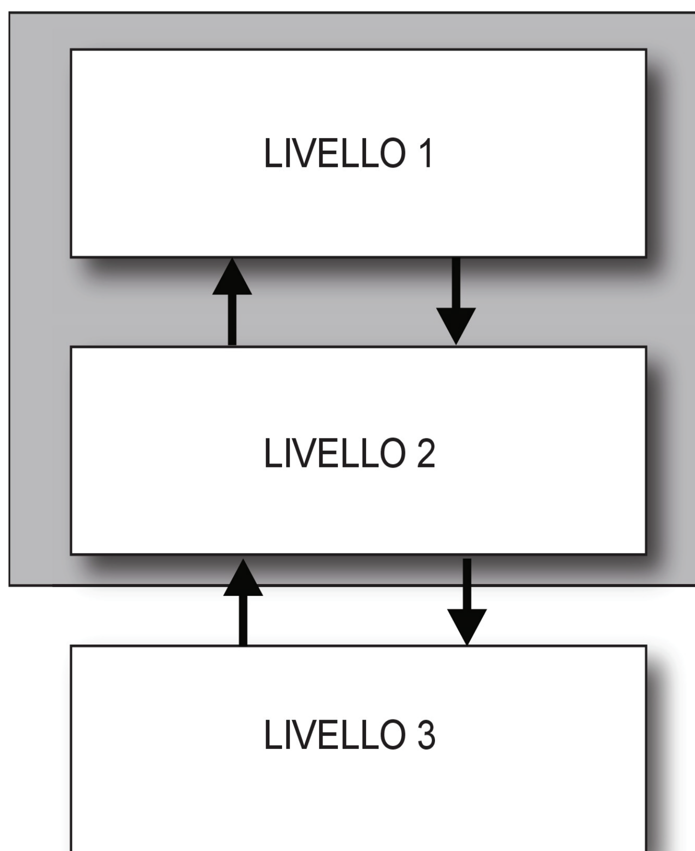
I requisiti che hanno guidato la realizzazione della piattaforma Click & Find sono:

- primo fra tutti la possibilità di offrire più applicazioni per lo stesso gruppo di mezzi, mettendone a disposizione almeno una per ogni categoria di utenti del settore (operatori logistici, imprese di trasporto e/o subvettori, assicurazioni, broker assicurativi, organi di sorveglianza, manager di infrastruttura …);
- ogni applicazione per il controllo e il monitoraggio a distanza è in grado di ottenere differenti insiemi di dati da un singolo mezzo (solo motrice o solo autobotte) o da mezzi integrati (motrice e autobotte) quali ad esempio: l'identità del conducente, la velocità del veicolo, le ore di guida, la quantità di prodotto trasportato, il peso, l'attivazione dell' EBS (Electronic Brake System), … ;
- i dati raccolti sui mezzi sono quasi tutti completamente in forma standard, il che facilita la loro trattazione nei livelli successivi;
- sul mezzo la sorgente dei dati è pensata per produrre molti tipi di dati e in grande quantità, vantaggio che ovviamente incide sulla significatività.
  Al livello applicativo, infatti, dove l'impegno maggiore è rivolto alla trasformazione dei dati in informazione, si privilegiano pochi tipi di dato e a bassa frequenza. Il sistema inoltre fornisce i dati alle applicazioni in tempo reale;
- il sistema operativo e il firmware che gestisce lo stack sull'unità di bordo del mezzo garantiscono affidabilità e la possibilità di essere gestiti a distanza, tenuto conto degli alti costi che si sosterrebbero dovendo tenere fermo il veicolo.

1.3 Architettura del sistema

Ogni RVMC (Remote Vehicle Monitoring and Control) può essere rappresentato su tre livelli.
Il primo di essi è composto dall'hardware e dal firmware montati sul mezzo per la raccolta dei dati e lo scambio di messaggi con il livello seguente.
Il secondo livello riguarda lo smistamento dei messaggi, l'immagazzinamento dei dati, la gestione dei dispositivi, la generazione e il filtraggio del flusso di dati diretto al livello successivo.
Al terzo livello vi sono le interfacce utente e le applicazioni per la configurazione del flusso logistico dell'impresa .

L'RVMC progettato e realizzato da Click & Find si distingue dagli altri sistemi in commercio, per la stretta interazione esistente tra il primo e il secondo livello.
Il primo dei due livelli può essere visto come una estensione (di tipo) hardware del secondo livello. Quest'ultimo infatti è stato progettato per poter eseguire frammenti di codice o addirittura sostituire l'intero firmware da remoto, nonché accedere alla memoria di sistema e al sistema di I/O (Input/Output).
La distinzione tra secondo e terzo livello ha dato origine ad un'interfaccia aperta e ben definita tra i livelli in questione, che permette a più applicazioni di condividere risorse messe a disposizione dall'hardware potendo impostare la logica di accesso ai dati senza dover tenere conto delle componenti fisiche del sistema.



2 Modello dell'applicazione

2.1 Obiettivo

Sulla base di esigenze manifestate dalla clientela e degli interessi operativi dell'azienda, ci si è posto come obbiettivo, posizionandosi al terzo livello dell'architettura Click & Find, la realizzazione di un sistema di allarmi configurabile il cui funzionamento si basa sul cambiamento di stato di alcuni dispositivi montati a bordo dei veicoli controllati a distanza.
Il sistema è definito configurabile perché le regole di funzionamento devono poter essere definite dall'utente.
La realizzazione del sistema di allarmi è stata raggiunto attraversando due momenti: nel primo lo studio è stato rivolto alla modellazione teorica; nel secondo l'attenzione si è concentrata nello sviluppo di un applicazione (software) il cui funzionamento si basa sui risultati teorici ottenuti nella prima fase.

2.2 Modellazione del sistema di allarmi

Il sistema di allarmi definisce, per il suo funzionamento, delle macchine a stati che si evolvono per mezzo di eventi accaduti e trasmessi dai veicoli verso i server Click & Find.
Con il termine eventi si indicano i cambiamenti di stato dei dispositivi che equipaggiano il veicolo (accensione/spegnimento del quadro di alimentazione, apertura/chiusura portellone, cambiamento pressione delle sospensioni, apertura/chiusura delle valvole di fondo, …). Un concentratore di segnali, montato sul veicolo, provvede a trasmettere gli eventi appena descritti ai server Click & Find.
Le macchine a stati previste dal sistema di allarmi sono programmabili dall'utente per mezzo di stringhe di configurazione, qui di seguito definite mediante notazione BNF (Backus-Naur Form):

```
<evento> ::= < cifra > < cifra > < cifra > < cifra >

<cifra> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<parentesi quadra aperta con apice> ::= [^

<parentesi quadra chiusa> ::=]

<eventi non ammessi> ::= < parentesi quadra aperta con apice > { < evento > } < parentesi quadra chiusa >

<stringa di configurazione> ::= <evento> { <evento> | <eventi non ammessi> <evento> }

La stringa di configurazione, appena descritta, definisce una sequenza di eventi che devono verificarsi secondo l'ordine stabilito dall'utente.
Il sistema di allarmi prevede che tra un evento e l'altro l'utente possa definire una lista senza alcun ordine prestabilito di eventi che egli desidera non si verifichino; l'elemento <eventi non ammessi>, qui sopra descritto utilizzando la notazione BNF, definisce la modalità di scrittura di tale lista all'interno della stringa di configurazione.

Evento	Sistema di allarmi
Inserimento alimentazione	0001
Disinserimento alimentazione	0002
Inserimento ciclo-chiuso	0003
Disinserimento ciclo-chiuso	0004
Apertura portellone	0005
Chiusura portellone	0006
Accensione quadro	0007
Spegnimento quadro	0008
Apertura valvola 1	0009
Chiusura valvola 1	0010
Apertura valvola 2	0011
Chiusura valvola 2	0012
Apertura valvola 3	0013
Chiusura valvola 3	0014

Tabella delle corrispondenze tra eventi

Esempi di stringhe di configurazione

es. 1) La stringa di configurazione

`000500030009001000040006`

esprime la seguente sequenza di eventi:

" da quando il portellone si apre (0005) a quando il portellone si chiude (0006) devono verificarsi i seguenti eventi nell'ordine: inserimento del ciclo chiuso (0003), apertura della valvola 1 (0009), chiusura della valvola (0010), disinserimento del ciclo chiuso (0004) ".

es. 2) La stringa di configurazione

`0005[^00010007]0006`

esprime la seguente sequenza di eventi:

"da quando il portellone viene aperto (0005) a quando il portellone viene chiuso (0006) si possono verificare nessuno, uno o più di uno degli eventi previsti dal sistema Click & Find tranne l'inserimento dell'alimentazione (0001) e l'accensione del quadro (0007)".

Il sistema di allarme crea un automa per controllare, in tempo reale, la corrispondenza (match) tra gli eventi che accadono sul veicolo e quelli descritti dalla stringa di configurazione. Tutte le volte che non vi è corrispondenza tra gli eventi trasmessi dal veicolo e la stringa di configurazione, il sistema provvede a generare un allarme.
Ogni macchina a stati è costituita da uno stato di attesa (idle) nel quale essa attende di ricevere il primo evento della stringa di configurazione per passare allo stato iniziale; se la corrispondenza tra eventi viene rispettata, l'automa giunge allo stato finale e non viene generato nessun allarme.
E' possibile definire lineare il funzionamento di questi automi, per il fatto che ognuno di essi raggiunge lo stato finale solo nel caso in cui gli eventi si verifichino nell'ordine descritto dalla stringa di configurazione.
In generale è possibile descrivere un'automa, generato dal sistema di allarmi, indicando con:

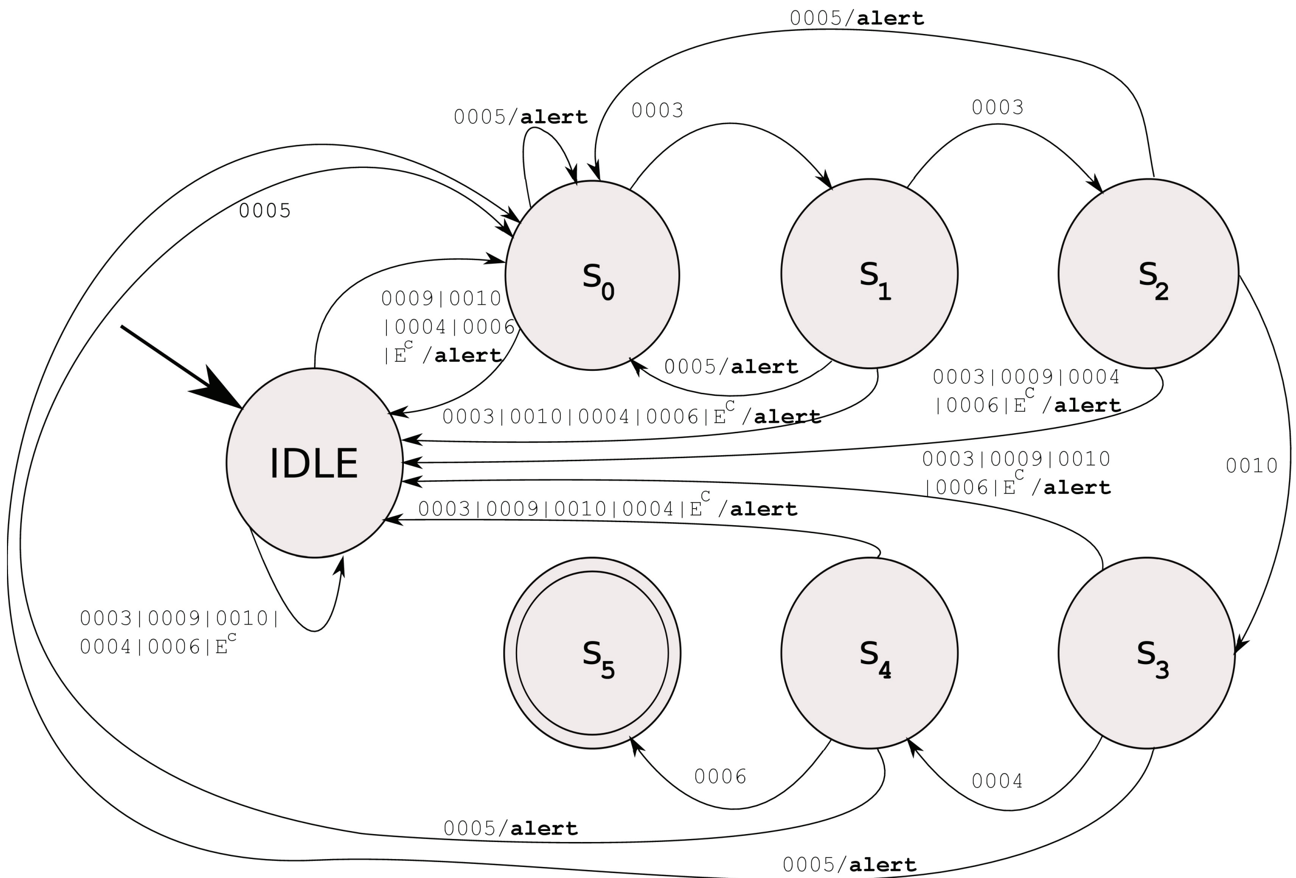
- **I,** l'insieme finito dei possibili input in ingresso;
- **O**, l'insieme finito dei possibili output;
- **S,** l'insieme finito degli stati;
- ***f***, la funzione di transizione di stato;
- ***g,*** la funzione di uscita.

L'insieme **I** è formato da tutti gli eventi trasmessi dal concentratore di segnali montato sul veicolo; l'insieme **O** è costituito dagli allarmi (in seguito definiti con **alert**) che vengono generati nel caso in cui non si verifichi la corrispondenza tra gli eventi provenienti dai veicoli e quelli descritti nella stringa di configurazione.
La funzione ***f***, permette di definire lo stato successivo a cui perviene l'automa in seguito all'evento in ingresso; essa permette di costruire una tabella delle transizioni di stato utile per descrivere in maniera intuitiva l'evolversi della macchina a stati.
La funzione di trasformazione ***g*** consente di definire gli output (i comportamenti) dell'automa in seguito all'evento in ingresso; essa permette di costruire una tabella della trasformazione degli output utile per mostrare il comportamento intrapreso dalla macchina a stati in seguito all'evento in ingresso.

2.2.1 Esempi di automa

es. 3) La seguente figura rappresenta il grafo dell'automa che deve verificare la corrispondenza tra gli eventi descritti nella stringa di configurazione dell'esempio 1 e quelli che si verificano sul veicolo.

Stringa di configurazione: `00050003000900100004 0006`



Nel grafo qui sopra esposto, con il simbolo E^C sono da intendersi tutti gli altri possibili eventi che si verificano sul mezzo oltre a quelli esplicitamente riportati.

Le due seguenti tabelle mostrano rispettivamente le funzioni di transizione di stato e le funzioni di uscita dell'automa.

	0005	0003	0009	0010	0004	0006	E^c[1]
IDLE	**s_0**	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE
s_0	s_0	**s_1**	IDLE	IDLE	IDLE	IDLE	IDLE
s_1	s_0	IDLE	**s_2**	IDLE	IDLE	IDLE	IDLE
s_2	s_0	IDLE	IDLE	**s_3**	IDLE	IDLE	IDLE
s_3	s_0	IDLE	IDLE	IDLE	**s_4**	IDLE	IDLE
s_4	s_0	IDLE	IDLE	IDLE	IDLE	**s_5**	IDLE

Tabella delle funzioni di transizioni di stato

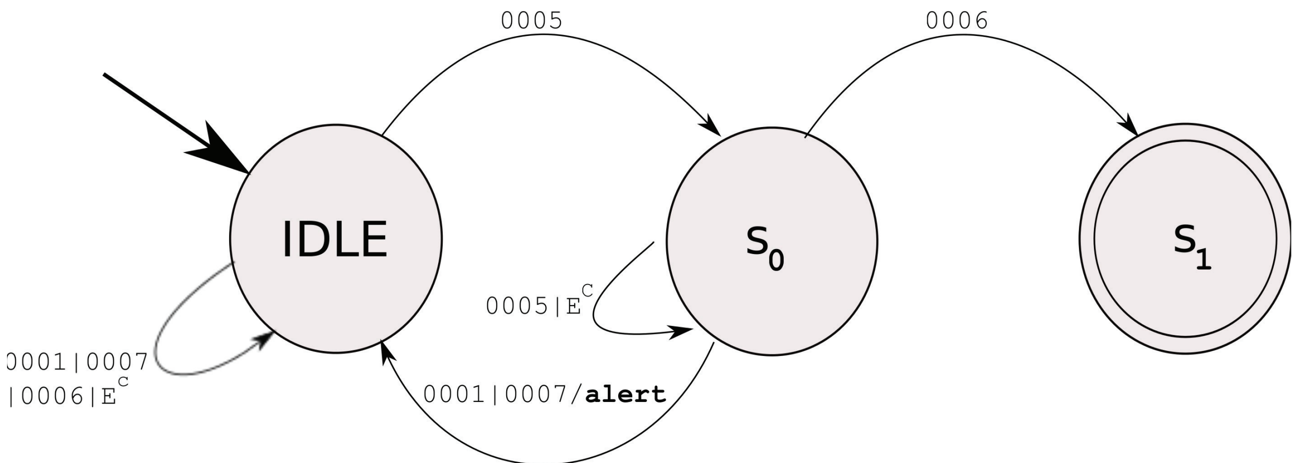
	0005	0003	0009	0010	0004	0006	E^c
IDLE							
s_0	alert		alert	alert	alert	alert	alert
s_1	alert	alert		alert	alert	alert	alert
s_2	alert	alert	alert		alert	alert	alert
s_3	alert	alert	alert	alert		alert	alert
s_4	alert	alert	alert	alert	alert		alert

Tabella delle funzioni di uscita

1 E^c : rappresenta l'insieme di tutti gli eventi che si possono verificare ad esclusione di quelli già presenti nella tabella.

es. 4) La seguente figura rappresenta il grafo dell'automa che deve verificare la corrispondenza tra gli eventi descritti nella stringa di configurazione dell'esempio 2 e quelli che si verificano sul veicolo.

Stringa di configurazione: `0005[^00010007]0006`



Nel grafo qui sopra esposto, con il simbolo E^C sono da intendersi tutti gli altri possibili eventi che si verificano sul mezzo oltre a quelli esplicitamente riportati.

Le due seguenti tabelle mostrano rispettivamente le funzioni di transizione di stato e le funzioni di uscita dell'automa.

	0005	**0001**	**0007**	**0006**	**E^C**[2]
IDLE	S_0	IDLE	IDLE	IDLE	IDLE
S_0	S_0	IDLE	IDLE	S_1	S_0

Tabella delle funzioni di transizioni di stato

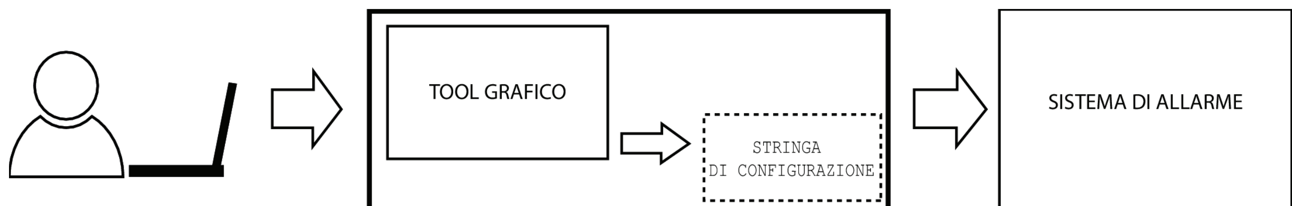
	0005	**0001**	**0007**	**0006**	**E^C**
IDLE					
S_0		alert	alert		

Tabella delle funzioni di uscita

2 E^C: rappresenta l'insieme di tutti gli eventi che si possono verificare ad esclusione di quelli già presenti nella tabella.

3 Sviluppo dell'applicazione

3.1 Schema generale di funzionamento



Nello schema, qui sopra riportato, un tool grafico, guida l'utente nella generazione di una stringa di configurazione (v. “Modellazione del sistema”) per il funzionamento del sistema di allarmi.
Il capitolo rivolge l'attenzione allo sviluppo dell'applicazione per il sistema di allarmi che è stato pensato in due fasi: configurazione ed evoluzione.
La configurazione dell'applicazione consiste nel predisporre alcune risorse; è durante la prima fase che:

- vengono definiti i file di log destinati a raccogliere dati e comportamenti provenienti dai vari processi avviati durante il funzionamento dell'applicazione;
- vengono definiti i percorsi (path) e le directory entro cui l'applicazione può eseguire letture e salvataggi di file necessari al suo funzionamento come ad esempio le stringhe di configurazione definite dall'utente;
- viene creata una macchina a stati (v. “Modellazione del sistema”) per ogni file contenente la stringa di configurazione definita dall'utente.

Nella seconda fase vengono fatte evolvere le macchine a stati configurate durante la prima fase per mezzo degli eventi contenuti all'interno dei messaggi in arrivo dai veicoli.
Il seguente schema rappresenta la suddivisione in due fasi del funzionamento dell'applicazione.

APPLICAZIONE PER LA GENERAZIONE DI ALLARMI REAL-TIME	
CONFIGURAZIONE	**EVOLUZIONE**
Lettura da file della stringa di configurazione	**Ricezione** dei messaggi in arrivo dal veicolo che contengono gli eventi
Riconoscimento della stringa di configurazione	**Evoluzione** degli automi creati, nella fase di configurazione, secondo gli eventi ricevuti
Creazione dell'automa per il matching tra la stringa di configurazione e gli eventi trasmessi dal veicolo	**Generazione** **allarmi**

L'utilizzo di Java come linguaggio di programmazione ha permesso, sfruttando le proprietà dei linguaggi di programmazione orientati agli oggetti, di definire con chiarezza i vari processi che costituiscono il funzionamento dell'applicazione.

Prima di addentrarsi nella descrizione del funzionamento dell'applicazione, è importante tenere presente la diversità di notazione adottata dalla piattaforma Click & Find e quella utilizzata nella stringa di configurazione per la descrizione degli eventi.

I protocolli di comunicazione Click & Find prevedono che un evento sia descritto mediante due stringhe: la prima descrive il dispositivo (portellone, valvola 1, ciclo-chiuso,...); la seconda, il cambiamento di stato dello stesso (apertura/chiusura, inserimento/disinserimento, accensione/spegnimento).

La notazione adottata per descrivere gli eventi nella stringa di configurazione è stata descritta nel paragrafo "Modellazione del sistema".

All'interno della classe `Server` (app. A) è contenuto il punto di ingresso del programma: l'istanziamento di un oggetto appartenente a questa classe costituisce l'avvio dei processi di configurazione ed evoluzione dell'applicazione.

3.2 Configurazione

Il seguente frammento di codice rappresenta una struttura di controllo for contenuta all'interno del metodo `run` della classe `SerializzatoreFsm` (app. B).

```
for(int i=0; i<rulefiles.length; i++){
      BufferedReader br;
      try {
            br = new BufferedReader(new FileReader(rulefiles[i]));
            String nomefile = rulefiles[i].getName();
            String linea = br.readLine();
            RuleParser ruleparser = new RuleParser(logger, linea);
            FsmCreator fsmcreator = newFsmCreator(logger,props,ruleparser.run(),
            nomefile);
            fsmcreator.run();
            br.close();
      }
      catch (IOException e) {
            logger.debug("errore nella lettura delle configurazioni utente");
            e.printStackTrace();
      }
}
```

Il ciclo `for`, per ogni file contenente una stringa di configurazione, svolge le seguenti operazioni:

1. lettura da file della stringa di configurazione definita dall'utente;
2. suddivisione della stringa di configurazione in un vettore di stringhe;
3. creazione dell'automa per il matching tra gli eventi provenienti dal veicolo e quelli descritti all'interno della stringa di configurazione.

I nomi dei file, che contengono la stringa di configurazione, cominciano con l'identificativo del veicolo per cui è stata creata la sequenza di allarme e sono seguiti da una breve descrizione.

Esempio di file contenente una stringa di configurazione

Il file `855-erogazione_carburante` contiene, per il veicolo 855 , una stringa di configurazione per l'erogazione di carburante.

La stringa di configurazione letta da file viene suddivisa in un vettore di stringhe dal metodo `run` della classe `RuleParser` (app. C), qui di seguito riportato.

```
public Vector<String> run(){
	Vector<String> vettoreregola = new Vector<String>(10,10);
	for(int t=0; t<regola.length(); t++){
		if(Character.isDigit(regola.charAt(t))== true){
			vettoreregola.add(regola.substring(t,t+4));
			t=t+3;
		}
		else if(regola.charAt(t) == '['){
			int from = t;
			while(regola.charAt(t) != ']'){
				t = t+1;
			}
			vettoreregola.add(regola.substring(from,t+1));
		}
	}
	return vettoreregola;
}
```

Il metodo scorre i caratteri che costituiscono la stringa di configurazione e aggiunge un elemento al `vettoreregola` di stringhe corrispondente all'`evento` o alla sequenza di `eventi_non_ammessi` riconosciuta al suo interno.
Di seguito, viene riportata la definizione di `evento` e di `eventi_non_ammessi`, utilizzando la metasintassi BNF già utilizzata nei paragrafi precedenti.

<evento> ::= < cifra > < cifra > < cifra > < cifra >

<cifra> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<parentesi quadra aperta con apice> ::= [^

<parentesi quadra chiusa> ::=]

<eventi non ammessi> ::= < parentesi quadra aperta con apice > { < evento > } < parentesi quadra chiusa >

Esempi di "vettoreregola"

es.5) Per la stringa di configurazione `0005000300090010000040006` dell'esempio 3, il vettore di stringhe elaborato dal metodo `run` della classe `RuleParser` (app. C) è il seguente:

Posizione	**Evento / eventi_non_ammessi**
0	0005
1	0003
2	0009
3	0010
4	0004
5	0

es.6) Per la stringa di configurazione *`0005[^00010007]0006`* dell'esempio 4, il vettore di stringhe elaborato dal metodo `run` della classe `RuleParser` (app. C) è il seguente:

Posizione	**Evento / eventi_non_ammessi**
0	0005
1	[^00010007]
2	0006

La macchina a stati per il matching tra gli eventi descritti nella stringa di configurazione e quelli che vengono trasmessi dal veicolo è stato realizzato in termini di programmazione mediante un vettore. Lo stato e le transizioni di stato coincidono rispettivamente con l'indice e con il contenuto di ogni elemento del vettore.

La classe `FsmCreator` (app. D) configura gli stati dell'automa contenuti all'interno del vettore in base al vettoreregola descritto precedentemente.

Il vettore che implementa la macchina è costituito da oggetti appartenenti alla classe `Stato` (app. E) che hanno le seguenti proprietà:

```
EventMessage inputatteso ;
boolean proibiti ;
boolean match ;
Vector<EventMessage> eventi_proibiti ;
```

L' `inputatteso` contiene l'evento che permette all'automa di passare allo stato successivo; quando la variabile booleana `proibiti` viene impostata a `true`, l'automa non si evolve per qualsiasi degli eventi previsti dal sistema all'infuori di quelli descritti nel vettore `eventi_proibiti`, per i quali è previsto, nel caso in cui si verifichino, che venga generato un allarme.

La variabile booleana `match` assume valore `true` qualora lo stato in questione sia quello finale: nel caso in cui l'automa raggiunga questo stato è da considerare rispettata la sequenza di eventi prestabilita dall'utente descritta all'interno della stringa di configurazione.

Ogni Stato (app. E) viene definito, a partire dal `vettoreregola` precedentemente descritto, all'interno del struttura di controllo `while` implementato nel metodo run della classe `FsmCreator` (app. D) e qui di seguito riportato.

```
Stato stato_iniziale = new Stato();
stato_iniziale.setInputatteso(TraduttoreRegole.parse(regola.firstElement()));
stati.add(stato_iniziale);
while( i < regola.size()-1){
	Stato stato = new Stato();
	if(regola.get(i+1).contains("[") == true ){
		stato.setProibiti(true);
		stato.setEventi_proibiti(this.estrattoreEventi(regola.get(i+1)));
		if( i+2 != regola.size()-1){
```

```
			stato.setInputatteso(TraduttoreRegole.parse(regola.get(i+2)));
		}
		else if(i+2 == regola.size()-1){
			stato.setMatch(true);
			stato.setInputatteso(TraduttoreRegole.parse(regola.get(i+2)));
		}
		i = i+2;
	}
	else if(regola.get(i+1).contains("[") == faStato stato_iniziale = new
	Stato();
		stato_iniziale.setInputatteso(
		TraduttoreRegole.parse(regola.firstElement()));
		stati.add(stato_iniziale){
			if(i+1 != regola.size()-1){
			stato.setInputatteso(TraduttoreRegole.parse(regola.get(i+1)));
			}
			else if(i+1 == regola.size()-1){
				stato.setMatch(true);
			stato.setInputatteso(TraduttoreRegole.parse
			(regola.get(i+1)));
		}
		i = i+1;
	}
	stati.add(stato);
	}
```

Il primo degli stati che viene configurato è quello di idle dal quale è possibile passare al successivo per mezzo dell'evento contenuto nel primo elemento del `vettoreregola`.

Le strutture condizionali di controllo, definite all'interno del ciclo `while`, permettono di creare differenti configurazioni di oggetti appartenenti alla classe Stato (app. E).

Per ogni elemento contenuto all'interno del `vettoreregola`, a partire dal secondo, vengono verificate le seguenti condizioni:

1) se l'elemento non è l'ultimo del vettore e contiene una sequenza di eventi non_ammessi;
2) se l'elemento è l'ultimo del vettore e contiene una sequenza di eventi non_ammessi;
3) se l'elemento non è l'ultimo del vettore e contiene un singolo evento;
4) se l'elemento è l'ultimo e contiene un singolo evento.

Se si verifica la prima condizione, viene creato uno oggetto della classe `Stato` (app. E) inizializzando le variabili che lo definiscono nel seguente modo: l' `inputatteso` viene posto uguale all'evento contenuto nell'elemento del `vettoreregola` che lo segue, le variabili booleane `proibiti` e `match` vengono impostate rispettivamente a `true` e a `false`, mentre il vettore `eventi_proibiti` è riempito con gli eventi descritti nel sequenza `eventi_non_ammessi`.

Se si verifica la seconda condizione, viene creato un oggetto della classe `Stato` (app. E) inizializzando le variabili che lo definiscono nel seguente modo: l' `inputatteso` viene posto uguale all'evento contenuto nell'elemento del `vettoreregola` che lo segue, entrambe le variabili booleane `proibiti` e `match` vengono impostate a `true`, mentre il vettore `eventi_proibiti` è riempito con gli eventi descritti nel sequenza `eventi_non_ammessi`.

Se si verifica la terza condizione, viene creato un oggetto della classe `Stato` (app. E) inizializzando le variabili che lo definiscono nel seguente modo: l' `inputatteso` viene posto uguale all'evento contenuto nell'elemento del `vettoreregola` che lo segue, le variabili booleane `proibiti` e `match` vengono impostate rispettivamente a `true` e a `false`, mentre il vettore `eventi_proibiti` rimane vuoto.

Se si verifica la quarta condizione, viene creato un oggetto della classe `Stato` (app. E) inizializzando le variabili che lo definiscono nel seguente modo: l' `inputatteso` viene posto uguale all'evento contenuto nell'elemento del `vettoreregola` che lo segue, entrambe le variabili booleane proibiti e match vengono impostate a `true`, mentre il `vettore` `eventi_proibiti` rimane vuoto.

Non appena tutti gli stati dell'automa sono configurati viene istanziato e serializzato un oggetto `Fsm` (app. F) costituito dalle seguenti proprietà:

```
public Vector<Stato> stati;
public int stato_attuale;
```

Il vettore `stati` contiene tutti gli oggetti di tipo `Stato` (app. E) configurati; la variabile intera `stato_attuale` indica lo stato in cui si trova l'automa: al momento dell'istanziamento dell'oggetto essa viene posta uguale a zero.
La fase di configurazione dell'applicazione termina non appena tutte le stringhe di configurazione sono state tradotte in oggetti `Fsm` (app. F).

3.3 Evoluzione

Gli eventi che si verificano sui veicoli telecontrollati determinano l'evoluzione degli oggetti `Fsm` (app. F) serializzati nell'ultimo processo della fase di configurazione.
Ogni evento in arrivo all'applicazione è contenuto all'interno di un file il cui nome è composto da due parti: la prima identifica il veicolo; la seconda parte, separata dalla prima da un "-", rappresenta la data e l'orario di ricezione dell'evento dai server Click & Find.
All'interno del file vi è una stringa di caratteri che un costruttore della classe `EventMessage` (app. G) è in grado di riconoscere e suddividere utilizzando come separatore di campi il simbolo di punteggiatura ";".

```
public EventMessage(String messaggio) {
      String[] partiMsg = messaggio.split(";");
      setCodtrasp(partiMsg[0]);
      setTempo_evento(partiMsg[1]);
      setLat(Double.parseDouble(partiMsg[2]));
      setLng(Double.parseDouble(partiMsg[3]));
      setEventCode(partiMsg[4]);
      if (partiMsg.length > 5) { setEventData(partiMsg[5]); }
      else
      { setEventData("");}}
```

Vengono, così, estratte dalla stringa di caratteri le seguenti informazioni trasmesse:

- il codice identificativo del veicolo sul quale si è verificato l'evento;
- la data (anno-mese-giorno) e l'orario (ore-minuti-secondi) in cui si è verificato l'evento;
- posizione del veicolo (latitudine e longitudine);
- il codice che identifica il dispositivo montato a bordo del veicolo che ha subito il cambiamento di stato;
- il cambiamento di stato del dispositivo indicato dal codice precedente.

Una struttura di controllo `for` nella classe `Server` (app. A) scorre, uno ad uno, tutti i file di evento disponibili in una directory del server.

```
for(int i=0; i<messaggifiles.length; i++){
	BufferedReader br ;
	try {
		br = new BufferedReader(new
		FileReader(messaggifiles[i]));
		String messaggio = br.readLine();
		logger.info("Messaggio in arrivo:" +messaggio);
		String codtrasp =
CodeTrasp.codTraspExtractor(messaggifiles[i].getName());
		EventMessage eventmessage = new EventMessage(messaggio);
		EvolverFsm evolverfsm = new EvolverFsm(logger, props, codtrasp,
eventmessage);
		evolverfsm.evolver();
		br.close();
	}
	catch (IOException) {
		logger.debug("errore nella lettura dei messaggi");
		e.printStackTrace();
	}
}
```

Questo processo iterativo provvede, per ogni evento in arrivo, a decifrare il tipo di messaggio servendosi della classe `EventMessage` (app. G) e successivamente a far evolvere l'oggetto corrispondente grazie alla classe `EvolverFsm` (app. H).

La classe `Server` (app. A) inizializza un oggetto della classe `EvolverFsm` (app. H) fornendo al costruttore, oltre ai vari parametri che lo compongono, l'oggetto `EventMessage` (app. G) che rappresenta l'evento in arrivo dal veicolo.

A questo punto viene chiamata la funzione evolver dell'oggetto `EvolverFsm` (app. H) appena istanziato: questo metodo procede a verificare che vi siano oggetti `Fsm` (app. F) serializzati ai quali corrisponda il codice del mezzo (codetrasp) contenuto nel messaggio in arrivo e, nel caso in cui ciò avvenga, l'oggetto viene deserializzato e si procede alla sua evoluzione.
La deserializzazione dell'oggetto `Fsm` (app. F) avviene per opera di un metodo della classe `Fsm` (app. F) denominato deserializzaFsm e programmato nel seguente modo:

```
public Fsm deserializzaFsm(File f) throws Exception{
          Fsm fsm = null;public Fsm deserializzaFsm(File f) throws
Exception{
          Fsm fsm = null;
          FileInputStream fisp = new FileInputStream(f);
          ObjectInputStream inp = new ObjectInputStream(fisp);
          fsm = (Fsm) inp.readObject();
          inp.close();
          return fsm;
     }

          FileInputStream fisp = new FileInputStream(f);
          ObjectInputStream inp = new ObjectInputStream(fisp);
          fsm = (Fsm) inp.readObject();
          inp.close();
          return fsm;
     }
```

Il metodo ritorna l'istanza della macchina a stati inizializzata in fase di configurazione dell'applicazione.
La funzione `evolver`, in base allo stato in cui si trova attualmente l'automa deserializzato e al tipo di evento decifrato dal messaggio in arrivo, fa progredire la macchina a stati secondo le sue regole di funzionamento per mezzo di istruzioni condizionali che ne vincolano il comportamento.
Il primo blocco condizionale provvede a verificare che l'oggetto `Fsm` (app. F) abbia già ricevuto il primo degli eventi della sequenza che rappresenta il primo stato.

```
if(stato_attuale == 0 &&
eventmessage.getEventCode().equals(eventcode) &&
eventmessage.getEventData().equals(eventdata)){

	if(stati.size() == 1 && match == true){
		out.write(this.regolaMecciata());
		fsm.resetFsm();
	}
	else{
		fsm.setStato_attuale(1);}}
```

Come si evince dalle righe di codice qui sopra riportate, l'applicazione esegue le istruzioni contenute all'interno del primo `if` solo nel caso in cui l'automa non si trovi già nello stato iniziale: questo significa andare a valutare che il primo degli eventi della sequenza non si sia ancora verificato e che l'evento in arrivo coincida con il campo stato_attuale dell'oggetto `Fsm` (app. F).

Nel caso in cui l'espressione sia verificata, il programma controlla se la macchina a stati è composta da uno o più stati e in base a ciò provvede a riportare l'automa in attesa o farlo evolvere allo stato iniziale.

Il secondo blocco condizionale riguarda tutti gli oggetti `Fsm` (app. F) serializzati che sono in uno stato iniziale o in uno successivo.

```
if(stato_attuale != 0){
	if(match == false && proibiti == false){
		if(eventmessage.getEventCode().equals(eventcode) &&
				eventmessage.getEventData().equals(eventdata)){
			fsm.setStato_attuale(stato_attuale+1);
		}
		else {
			out.write(this.scattaAllarme());

	if(eventmessage.getEventCode().equals(eventcodeiniziale) &&

	eventmessage.getEventData().equals(eventdatainiziale)){
			fsm.setStato_attuale(1);
			}
			else{fsm.resetFsm();}
		}
	}

	if(match == false && proibiti == true){
		if(eventmessage.getEventCode().equals(eventcode) &&
```

```
    eventmessage.getEventData().equals(eventdata) == true){
         fsm.setStato_attuale(stato_attuale+1);
         }
         else {
              for(int t=0; t<vettoreproibiti.size(); t++){

    if(vettoreproibiti.get(t).getEventCode().equals(eventmessage.
                   GetEventCode()) &&
vettoreproibiti.get(t).getEventData().
                   equals(eventmessage.getEventData())){

                        out.write(this.scattaAllarme());


    if(eventmessage.getEventCode().equals(eventcodeiniziale) &&

    eventmessage.getEventData().equals(eventdatainiziale)){
                   fsm.setStato_attuale(1);
                   }
                   else{
                   fsm.resetFsm();}}}}}
```

L'applicazione provvede a far evolvere l'automa a seconda delle impostazione delle variabili booleane `match` e `proibiti`, caratterizzanti il tipo di stato in cui si trova l'automa e descritte nel paragrafo precedente.

I comportamenti previsti sono:

1. lo stato è configurato per passare al successivo per mezzo di un evento prestabilito e qualora questo non si verifichi viene generato un allarme ;
2. lo stato è configurato per passare al successivo per mezzo di evento prestabilito e qualora questo non si verifichi, viene generato un allarme oppure la macchina non si evolve a genera un allarme nel caso si verifichino uno o più eventi presenti nel vettore denominato `eventi_proibiti`.

Per entrambi i casi sopra descritti, ogni qual volta viene fatto scattare l'allarme, la variabile `stato_attuale` appartenente all'oggetto `Fsm` (app. F) viene posta uguale a zero: l'automa ritorna in attesa di ricevere un input corrispondente al primo elemento della sequenza configurata.

Si è ovviamente considerato che l'evento scatenante l'allarme può coincidere con l'evento che riconduce il sistema allo stato iniziale, caso in cui la variabile stato_attuale viene quindi impostata a uno.

4 Appendice

A) La classe Server

```
package it.clickandfind.fsmevolver;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;
import java.util.Properties;
import org.apache.commons.io.comparator.NameFileComparator;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class Server {

	public Properties props;

	public static void main(String[] args) throws Exception{
		String iniFileName = "/home/francesco/fsm_rev2.2/conf/server.ini";

		new Server(iniFileName);

	}

	public Server(String iniFileName){
		props = new Properties();

		try {
			FileInputStream is;
			is = new FileInputStream(iniFileName);
			props.load(is);
			is.close();

		} catch (IOException e) {
		System.out.println("Errore nella lettura del file di
		configurazione");
		e.printStackTrace();
		System.exit(0);
		}
		if(props.containsKey("LOG4J_INIFILE")){

		 PropertyConfigurator.configure(props.getProperty("LOG4J_INIFILE",
		 "/home/clickandfind/fsm_rev2.2/conf/logger.ini"));
		                  }
		Logger logger = Logger.getLogger("FsmServer");
		logger.info("°°°°°°°°°°°°°°°°°°°°°°");
		logger.info("FsmServer starting ...");
		logger.info("°°°°°°°°°°°°°°°°°°°°°°");
		SerializzatoreFsm serializzatorefsm = new
```

```
            SerializzatoreFsm(logger,props);
            serializzatorefsm.run();
            logger.info("Fsm CONFIGURATE e in attesa di eventi !");
            String pathmessaggi = props.getProperty("FSM_EVENTI",
            "/home/francesco/fsm_rev2.2/messaggi/");
            File messaggidir = new File(pathmessaggi);
            File[] messaggifiles = messaggidir.listFiles();
            logger.info("E' in corso l'ordinamento di " + messaggifiles.length +
            " messaggi...");
            Arrays.sort(messaggifiles, NameFileComparator.NAME_COMPARATOR);

            for(int i=0; i<messaggifiles.length; i++){
                  BufferedReader br ;

                  try {
                        br = new BufferedReader(new
                        FileReader(messaggifiles[i]));
                        String messaggio = br.readLine();
                        logger.info("Messaggio in arrivo:" +messaggio);
                        String codtrasp=
                        CodeTrasp.codTraspExtractor(messaggifiles[i].getName());
                        EventMessage eventmessage = new EventMessage(messaggio);
                        EvolverFsm evolverfsm = new EvolverFsm(logger, props,
                        codtrasp, eventmessage);
                        evolverfsm.evolver();
                        br.close();
                  }

                  catch (IOException e) {
                        logger.debug("errore nella lettura dei messaggi");
                        e.printStackTrace();
                  }

            }


      }

}

class CodeTrasp{
      public static String codTraspExtractor(String s){
            String[] nomefile = s.split("-");
            String codTrasp = nomefile[0];
            return codTrasp;
            }
}

class NomeFile{
      public static String nomeFileExtractor(String s){
            String[] nomefileconestensione = s.split("\\.");
            String nomefile = nomefileconestensione[0];
            return nomefile;}
}
```

B) La classe SerializzatoreFsm

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;
import org.apache.log4j.Logger;

public class SerializzatoreFsm {

	public Logger logger;
	public Properties props;

	public SerializzatoreFsm(Logger l, Properties p){
		logger = l;
		props = p;
	}

	public void run(){
		String tmp = props.getProperty("FSM_RULE","/home/francesco/fsm-
		rev2.2/rule/");
		File pathrule = new File(tmp);
		File[] rulefiles = pathrule.listFiles();
		logger.info("Sono presenti : " +rulefiles.length +" file di
		configurazione.");
		logger.info("Lettura file configurazione scritti dall'utente in
		corso ...");

		for(int i=0; i<rulefiles.length; i++){
			BufferedReader br;
			try {
				br = new BufferedReader(new FileReader(rulefiles[i]));
				String nomefile = rulefiles[i].getName();
				String linea = br.readLine();
				RuleParser ruleparser = new RuleParser(logger, linea);
				FsmCreator fsmcreator = new
				FsmCreator(logger,props,ruleparser.run(), nomefile);
				fsmcreator.run();
				br.close();

			}
			catch (IOException e) {
				logger.debug("errore nella lettura delle configurazioni
				utente");
				e.printStackTrace();
			}
		}
	}
}
```

C) La classe RuleParser

```
import java.util.Vector;

import org.apache.log4j.Logger;

public class RuleParser {

	public String regola;
	public Logger logger;

	public RuleParser(Logger l, String r){
		logger = l;
		regola = r;
	}

	public Vector<String> run(){

		logger.info("Rule parsing...");
		Vector<String> vettoreregola = new Vector<String>(10,10);
		for(int t=0; t<regola.length(); t++){
			if(Character.isDigit(regola.charAt(t))== true){
				vettoreregola.add(regola.substring(t,t+4));
				t=t+3;
			}
			else if(regola.charAt(t) == '['){
				int from = t;
				while(regola.charAt(t) != ']'){
					t = t+1;
				}
				vettoreregola.add(regola.substring(from,t+1));
			}
		}

		logger.info("Il vettore regola contiene: " + vettoreregola.size() +
		" elementi");
		return vettoreregola;
	}


}
```

D) La classe RuleParser

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
import java.util.Vector;
import org.apache.log4j.Logger;

public class FsmCreator {

	public Logger logger;
	public Properties props;
	public Vector<String> regola;
	public String nome_file;

	public FsmCreator(Logger l,Properties p,Vector<String> v, String nf){
		logger = l;
		props = p;
		regola = v;
		nome_file = nf;
	}

	public void run(){

		Vector<Stato> stati = new Vector<Stato>(10,10);

		logger.info("E' in corso la creazione della fsm corrispondente");
		if(regola.size() > 1){
		Stato stato_iniziale = new Stato();
		stato_iniziale.setInputatteso(TraduttoreRegole.parse
		(regola.firstElement()));
		stati.add(stato_iniziale);
		int i= 0;

	while( i < regola.size()-1){
		Stato stato = new Stato();
		if(regola.get(i+1).contains("[") == true ){
			stato.setProibiti(true);
			stato.setEventi_proibiti
			(this.estrattoreEventi (regola.get(i+1)));

			if( i+2 != regola.size()-1){
				stato.setInputatteso(TraduttoreRegole.parse
				(regola.get(i+2)));
			}
			else if(i+2 == regola.size()-1){
				stato.setMatch(true);
				stato.setInputatteso(TraduttoreRegole.parse
				(regola.get(i+2)));
					}
			i = i+2;
			}

		else if(regola.get(i+1).contains("[") == false ){
			if(i+1 != regola.size()-1){

				stato.setInputatteso
				(TraduttoreRegole.parse(regola.get(i+1)));
			}
```

```
      else if(i+1 == regola.size()-1){
            stato.setMatch(true);
            stato.setInputatteso(TraduttoreRegole.parse(regola.get(i+1)));
                              }
i = i+1;
                        }
stati.add(stato);
            }
      }
      else if(regola.size() == 1){
            Stato stato = new Stato();
            stato.setMatch(true);
            stato.setInputatteso(TraduttoreRegole.parse(regola.get(0)));
            stati.add(stato);
      }

      logger.info("E' in corso la serializzazione della fsm !");
      Fsm fsm = new Fsm(stati);
      try {
            fsm.serializzaFsm(fsm,nome_file);
      }
      catch (Exception e) {
            logger.debug("Errore durante la prima serializzazione della
            fsm");
            e.printStackTrace();
      }

      //creazione log per fsm appena creata

      logger.info("Creazione file di log per fsm: " + nome_file + " appena
      creata!");
      String pathlogfsm =
      props.getProperty("FSM_LOG",
      "/home/francesco/fsm_rev2.2/log/fsm_log/");
      File filefsmlog = new File(pathlogfsm+nome_file+".log");
      BufferedWriter out;

      try {
            out = new BufferedWriter(new FileWriter(filefsmlog));
            out.write("##########################################\n");
            out.write("        FSM: "+nome_file+"    \n");
            out.write("##########################################\n");
            out.write("\nFSM in attesa di evolvere...");
            out.close();
      }
       catch (IOException e) {
            logger.debug("errore nella creazione del file di log della
            fsm" + nome_file);
            e.printStackTrace();
      }



}


public Vector<EventMessage> estrattoreEventi(String s){
      logger.info("Estrazione eventi proibiti ...");
      Vector<EventMessage> eventi = new Vector<EventMessage>(10,10);

      int i=2;// dopo aver letto [^
```

```
		//fino a che non leggi una ]
		while(s.charAt(i)!=']'){
			eventi.add(TraduttoreRegole.parse(s.substring(i,i+4)));
			i=i+4;
		}

		return eventi;
	}


}
```

E) La classe Stato

```
import java.io.Serializable;
import java.util.Vector;


public class Stato implements Serializable{

	private static final long serialVersionUID = 498840391;
	EventMessage inputatteso = null;
	boolean proibiti = false;
	boolean match = false;
	String stringa_allarme = null;
	Vector<EventMessage> eventi_proibiti = new Vector<EventMessage>(10,10);

	public Stato(){
		super();
	}


	public EventMessage getInputatteso() {
		return inputatteso;
	}

	public void setInputatteso(EventMessage inputatteso) {
		this.inputatteso = inputatteso;
	}

	public Vector<EventMessage> getEventi_proibiti() {
		return eventi_proibiti;
	}

	public void setEventi_proibiti(Vector<EventMessage> eventi_proibiti) {
		this.eventi_proibiti = eventi_proibiti;
	}

	public boolean isProibiti() {
		return proibiti;
	}

	public void setProibiti(boolean proibiti) {
		this.proibiti = proibiti;
	}

	public boolean isMatch() {
		return match;
	}

	public void setMatch(boolean allarme) {
		this.match = allarme;
	}

	public String getStringa_allarme() {
		return stringa_allarme;
	}



	public void setStringa_allarme(String stringa_allarme) {
		this.stringa_allarme = stringa_allarme;
	}
```

```
	public void stampa(int stato){
		System.out.println("###########################");
		System.out.println("          Stato " + stato);
		System.out.println("###########################");
		System.out.println("Variabile allarme :" + match);
		System.out.println("-------------------------------------");
		System.out.println("Inputatteso: " + inputatteso.toString());
		System.out.println("-------------------------------------");
		System.out.println("Variabile proibiti: " + proibiti);
		System.out.println("-------------------------------------");
		System.out.println("Eventi proibiti: ");

		for(int i=0; i<eventi_proibiti.size();i++){
			System.out.println(eventi_proibiti.get(i).toString());
		}

		System.out.println("-------------------------------------");
		System.out.println("Stringa allarme: " + stringa_allarme);
		System.out.println("-------------------------------------");
	}


}
```

F) La classe Fsm

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Vector;

public class Fsm implements Serializable{

	private static final long serialVersionUID = 49713039;
	public Vector<Stato> stati = new Vector<Stato>(10,10);
	public int stato_attuale = 0;

	public Fsm(){}

	public Fsm(Vector<Stato> f){
		super();
		stati = f;
	}

	public Vector<Stato> getStati() {
		return stati;
	}

	public void setStati(Vector<Stato> fsm) {
		this.stati = fsm;
	}

	public int getStato_attuale() {
		return stato_attuale;
	}

	public void setStato_attuale(int stato_attuale) {
		this.stato_attuale = stato_attuale;
	}


	public void serializzaFsm(Fsm fsm, String nf) throws Exception{
		File fsmFile = new
		File("/home/francesco/fsm_rev2.2/fsm_serializzate/"
		+nf+".ser");
		FileOutputStream fos = new FileOutputStream(fsmFile);
		ObjectOutputStream outp = new ObjectOutputStream(fos);
		outp.writeObject(fsm);
		outp.close();
		fos.close();
	}

	public Fsm deserializzaFsm(File f) throws Exception{
		Fsm fsm = null;
		FileInputStream fisp = new FileInputStream(f);
```

```
        ObjectInputStream inp = new ObjectInputStream(fisp);
        fsm = (Fsm) inp.readObject();
        inp.close();
        return fsm;
    }


    public void riserializzaFsm(Fsm fsm,File f) throws Exception{
        FileOutputStream fos = new FileOutputStream(f);
        ObjectOutputStream outp = new ObjectOutputStream(fos);
        outp.writeObject(fsm);
        outp.close();
        fos.close();
    }



    public void resetFsm(){
        stato_attuale = 0;
    }
}
```

G) La classe EventMessage

```
import java.io.Serializable;

public class EventMessage implements Serializable {

	public static final long serialVersionUID = 34567889;
	public String codtrasp = null;
	public String tempo_evento = null ;
	public Double lat = null;
	public Double lng = null;
	public String eventCode = null;
	public String eventData = null;

	public EventMessage() {
		super();
	}

	public EventMessage(String ec, String ed) {
		super();
		setEventCode(ec);
		setEventData(ed);
	}

	public EventMessage(String messaggio) {
		String[] partiMsg = messaggio.split(";");
		setCodtrasp(partiMsg[0]);
		setTempo_evento(partiMsg[1]);
		setLat(Double.parseDouble(partiMsg[2]));
		setLng(Double.parseDouble(partiMsg[3]));
		setEventCode(partiMsg[4]);
		if (partiMsg.length > 5) { setEventData(partiMsg[5]); } else
		{ setEventData(""); }
	}

	public void setTempo_evento(String tempo_evento) {
		this.tempo_evento = tempo_evento;
	}
	public String getTempo_evento() {
		return tempo_evento;
	}
	public void setCodtrasp(String codtrasp) {
		this.codtrasp = codtrasp;
	}
	public String getCodtrasp() {
		return codtrasp;
	}
	public void setLng(Double lng) {
		this.lng = lng;
	}
	public Double getLng() {
		return lng;
	}
	public void setLat(Double lat) {
		this.lat = lat;
	}
	public Double getLat() {
		return lat;}
	public void setEventCode(String eventCode) {
		this.eventCode = eventCode;
	}
```

```
	public String getEventCode() {
		return eventCode;
	}
	public void setEventData(String eventData) {
		this.eventData = eventData;
	}
	public String getEventData() {
		return eventData;
	}


	public String toString() {
		return "EventMessage [codtrasp=" + codtrasp + ", eventCode="
		+ eventCode + ", eventData=" + eventData + ", lat=" + lat
		+ ", lng=" + lng + ", tempo_evento=" + tempo_evento + "]";
	}

	public String toStringlogger() {
		return "EventMessage [codtrasp=" + codtrasp + ", eventCode="
		+ eventCode + ", eventData=" + eventData + ", tempo_evento=" +
		tempo_evento + "]";
	}

}
```

H) EvolverFsm

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
import java.util.Vector;

import org.apache.log4j.Logger;



public class EvolverFsm {
	public Logger logger;
	public Properties props;
	public String codtrasp;
	public EventMessage eventmessage;

	public EvolverFsm(Logger l, Properties p, String ct, EventMessage em){
		logger = l;
		props = p;
		codtrasp = ct;
		eventmessage = em;
	}

	public void evolver() throws IOException{
		String pathfsmserializzate
		props.getProperty ("FSM_SERIALIZZATE",
		"/home/francesco/fsm_rev2.2/fsm_serializzate/");
		File fsmserializzatedir = new File(pathfsmserializzate);
		File[] fsmserializzatefiles = fsmserializzatedir.listFiles();
		logger.info("Seleziono fsm aventi codtrasp: " + codtrasp);

		for(int i=0; i < fsmserializzatefiles.length; i++){
		if(CodeTrasp.codTraspExtractor
		(fsmserializzatefiles[i].getName()).equals(codtrasp) == true){
			Fsm fsm = new Fsm();
			try {
				fsm = fsm.deserializzaFsm(fsmserializzatefiles[i]);
			}
			catch (Exception e) {
				logger.debug("errore nella deserializzazione della
				fsm");
				e.printStackTrace();
			}
			String pathlogfsm = props.getProperty("FSM_LOG",
			"/home/francesco/fsm_rev2.2/log/fsm_log/");
			File file_log_fsm = new File(pathlogfsm+
			NomeFile.nomeFileExtractor(fsmserializzatefiles[i].getName())
			+".log");
			FileWriter fw = null;
			try {
				fw = new FileWriter(file_log_fsm,true);
			}
			catch (IOException e1) {
				logger.debug("errore nella lettura del log della fsm" +
				file_log_fsm.getName());
				e1.printStackTrace();
			}
```

```
            BufferedWriter out = new BufferedWriter(fw);
            out.write("Messaggio in arrivo:" + eventmessage.toString()
            +" \n");


            String eventcode =
            fsm.stati.get(fsm.getStato_attuale())
            .getInputatteso().getEventCode();

            String eventdata =
            fsm.stati.get(fsm.getStato_attuale())
            .getInputatteso().getEventData();

            String eventcodeiniziale =
            fsm.stati.get(0).getInputatteso()
            .GetEventCode();

            String eventdatainiziale = fsm.stati.get(0).getInputatteso()
            .getEventData();

            int stato_attuale= fsm.getStato_attuale();
            Vector<Stato> stati = new Vector<Stato>(10,10);
            stati = fsm.getStati();

            boolean match = fsm.stati.get(fsm.getStato_attuale())
            .isMatch();

            boolean proibiti = fsm.stati.get(fsm.getStato_attuale())
            .isProibiti();

            Vector<EventMessage> vettoreproibiti =
            fsm.stati.get(fsm.getStato_attuale())
            .getEventi_proibiti();

if(stato_attuale == 0 && eventmessage.getEventCode().equals(eventcode) &&
eventmessage.getEventData().equals(eventdata)){
      if(stati.size() == 1 && match == true){
            out.write(this.regolaMecciata());
            fsm.resetFsm();
      }
      else{
            out.write("**********************\n");
            out.write("   FSM inizializzata   \n");
            out.write("**********************\n");
            fsm.setStato_attuale(1);
      }

}






if(stato_attuale != 0){
      if(match == false && proibiti == false){
            if(eventmessage.getEventCode().equals(eventcode) &&
            eventmessage.getEventData().equals(eventdata)){
```

```
			fsm.setStato_attuale(stato_attuale+1);
		}

		else {
			out.write(this.scattaAllarme());
			if(eventmessage.getEventCode().equals(eventcodeiniziale)
			& &eventmessage.getEventData()
			.equals(eventdatainiziale)){

			fsm.setStato_attuale(1);
			}
			else{
				fsm.resetFsm();}
			}
		}


	if(match == false && proibiti == true){
		if(eventmessage.getEventCode().equals(eventcode) &&
		eventmessage.getEventData().equals(eventdata) == true){
		fsm.setStato_attuale(stato_attuale+1);
	}
		else {
		for(int t=0; t<vettoreproibiti.size(); t++){
			if(vettoreproibiti.get(t).getEventCode()
			.equals(eventmessage.getEventCode()) &&
			vettoreproibiti.get(t).getEventData()
			.equals(eventmessage.getEventData())){

				out.write(this.scattaAllarme());
				if(eventmessage.getEventCode()
				.equals(eventcodeiniziale)&&eventmessage
				.getEventData().equals(eventdatainiziale)){

					fsm.setStato_attuale(1);

				}

				else{fsm.resetFsm();}
			}
		}}}}

	if(match == true && proibiti == true){
		if(eventmessage.getEventCode().equals(eventcode)
		&& eventmessage.getEventData().equals(eventdata)){
			out.write(this.regolaMecciata());
			fsm.resetFsm();
		}




		else {
			for(int t=0; t<vettoreproibiti.size(); t++){
				if(vettoreproibiti.get(t).getEventCode()
				.equals(eventmessage.getEventCode())
				&& vettoreproibiti.get(t).getEventData()
				.equals(eventmessage.getEventData())){
```

```
                        out.write(this.scattaAllarme());
                        fsm.resetFsm();
                    }
                }
            }}


        if(match == true && proibiti == false){
            if(eventmessage.getEventCode().equals(eventcode) &&
            eventmessage.getEventData().equals(eventdata)){
                out.write(this.regolaMecciata());
                fsm.resetFsm();
            }
            else{
                out.write(this.scattaAllarme());
                if(eventmessage.getEventCode().equals(eventcodeiniziale)
                && eventmessage.getEventData()
                .equals(eventdatainiziale)){

                    fsm.setStato_attuale(1);
            }

            else{fsm.resetFsm();}
            }}

            out.close();
            try {
                fsm.riserializzaFsm(fsm, fsmserializzatefiles[i]);
                } catch (Exception e) {
                    logger.debug("errore nella riserializzazione della
                    fsm");
                    e.printStackTrace();
                }
            }}}

    public String scattaAllarme(){
        String scattaAllarme = "!!!!!!!!!!!!!!!!!!!\n" + " ALLARME SCATTATO!
        \n" + "!!!!!!!!!!!!!!!!!!!\n";
        System.out.println("!!!!!!!!!!!!!!!!!!!");
        System.out.println(" ALLARME SCATTATO! ");
        System.out.println("!!!!!!!!!!!!!!!!!!!");
        return scattaAllarme;
    }

    public String regolaMecciata(){
        String regolaRispettata = "OKOKOKOKOKOKOKOKOKOK\n" + "REGOLA
        RISPETTATA\n" +"OKOKOKOKOKOKOKOKOKOK\n";
        System.out.println("OKOKOKOKOKOKOKOKOKOK");
        System.out.println(" REGOLA RISPETTATA ");
        System.out.println("OKOKOKOKOKOKOKOKOKOK");
        return regolaRispettata;
    }
}
```