Software Engineering
progetto "PEdestrian LIght CONtroller"

Alessio Tonioni

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
alessio.tonioni@studio.unibo.it

1 Introduction

Come caso di studio si analizzi un sistema di gestione di un semaforo per automobili e pedoni. Il sistema verrà modellato come un automa a stati finiti; per testare la validità dell'analisi si realizzi un prototipo del sistema nella maniera più semplice e veloce possibile.
Valutare quindi la semplicitá di realizzazione, il grado di riutilizzo e l'estendibilitá del prototipo.

2 Vision

Una volta realizzata correttamente la modellazione del sistema, dovremmo essere in grado di realizzare senza alcuno sforzo o perdita di tempo un sistema che concretizzi il comportamento descritto in fase di analisi e che possa essere portato rapidamente in ambienti eterogenei e distribuiti.

3 Goals

Realizzare la modellazione e un prototipo di un sistema di gestione di un incrocio caratterizzato da un semaforo per auto e uno pedonale munito di prenotazione. Cercare di identificare una metodologia per mappare i concetti espressi in fase di analisi in un'implementazione funzionante.

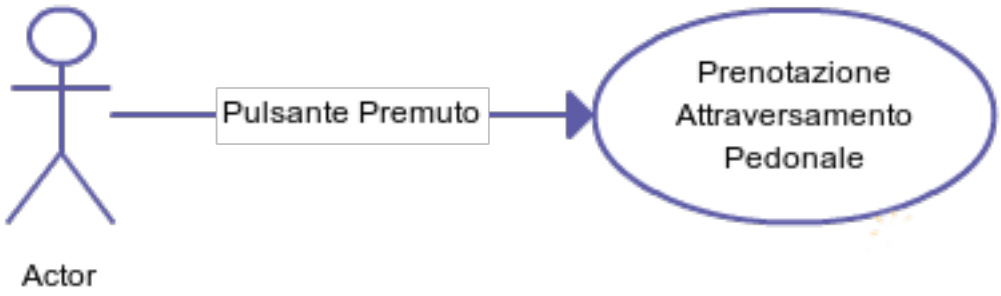
4 Requirements

Nella situazione di riposo il semaforo delle auto é verde, quello dei pedoni rosso. Un pedone potrá "prenotare" l'attraversamento attraverso la pressione di un apposito pulsante, in questo caso il semaforo delle autovetture diventerà giallo e dopo un certo intervallo di tempo rosso,mentre il semaforo pedonale potrá diventare verde autorizzando i pedoni all'attraversamento.
Dopo un certo intervalo di tempo il semaforo pedonale diventerà giallo ed infine rosso permettendo di nuovo al semaforo delle autovetture di tornare verde.
In ogni istante il pedone potrà prenotare il semaforo, ma il sistema garantirà un certo tempo minimo di semaforo verde per le autovetture tra due semafori verdi per i pedoni.

5 Requirement analysis

5.1 Use cases

Il sistema verrà utilizzato da due classi di utenti: veicoli e pedoni. Entrambe le entitá potranno attraversare l'incrocio solamente quando il rispettivo semaforo é nello stato verde, ovviamente mentre i pedoni stanno attraversando i veicoli dovranno attendere e viceversa. Nello stato di riposo il semaforo dei veicoli è "VERDE" permettendone il passaggio, quello pedonale é invece "ROSSO". L'unico caso d'uso significativo per la modellazione del sistema software é dato da un pedone che preme il pulsante per prenotare l'attraversamento pedonale. Questo fa o farà partire la serie di cambiamenti di stato del sistema semaforo che permettono ai pedoni di attraversare in sicurezza l'incrocio.



5.2 Scenarios

5.3 Glossary

Termine	**Significato**
Incrocio	Entitá composta da due semafori(uno per pedoni e uno per veicoli) e un pulsante.
Sistema Semaforo	Sinonimo di Incrocio.
Semaforo	Entitá che puó trovarsi in uno dei tre stati ROSSO, VERDE, GIALLO.
Pulsante	Dispositivo fisico usato da un pedone per prenotare l'attraversamento.
Veicolo	Entitá autonoma esterna al sistema che attraversa l'incrocio quando il Sem_Veicolo é VERDE.
Pedone	Entitá autonoma esterna al sistema che attraversa l'incrocio quando il Sem_Pedoni é VERDE.

5.4 (Domain)model

Il sistema incrocio può essere scomposto in 4 sottosistemi:

- Semaforo Pedonale
- Semaforo Veicoli

– Pulsante di prenotazione
– Controller

Semafori:
Il sistema è composto da due semafori esteriormente diversi, ma il cui comportamento è esattamente identico da un punto di vista di analisi del sistema.
Per questo all'interno del sistema sono modellabili come entitá passive che espongono un metodo per settare il colore della propria luce. Semaforo pedonale e Semaforo per veicoli possono essere quindi realizzati entrambi come implementazioni della seguente interfaccia:

```
1 package pelican;
2
3 public interface ISemaphore {
4   public void setLight(ColorType color);
5   public ColorType getCurrentColor();
6 }
```

/home/alessio/Scrivania/Natali/telecomandoRobot/sema/src/pelican/ISemaphore.java

```
1 package pelican;
2
3 public enum ColorType{
4   RED,GREEN,YELLOW;
5 }
```

/home/alessio/Scrivania/Natali/telecomandoRobot/sema/src/pelican/ColorType.java

Pulsante di prenotazione:
Il pulsante può a sua volta essere suddiviso in due sottositemi a seconda della prospettiva da cui lo si guarda: quella del pedone che lo andrà ad utilizare per effettuare la prenotazione o quella del progettista del sistema software.
Nel primo caso il bottone é modellabile come un entitá passiva che si limita ad attendere la pressione del pulsante fisico da parte dell'utente.
Nel secondo caso il bottone é un'entitá attiva modellabile secondo il pattern observer e l'inversione di controllo. Al momento della pressione il pulsante metterà in esecuzione gli handler associati a quell'azione.
Le due parti unite formano l'entitá pulsante, ma la parte che dovremo incorporare nel sistema é quella interna implementabile tramite pattern observer standard.

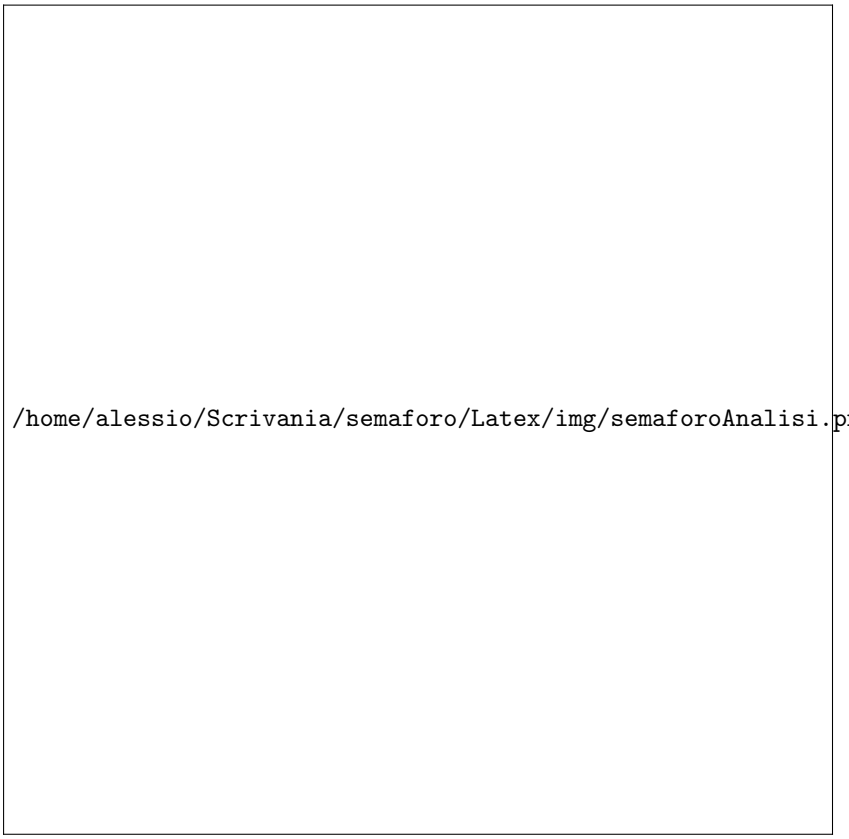
Controller:
Il controller è l'entitá che ha il compito di coordinare il funzionamento del resto dei componenti. É un'entitá passiva il cui compito é mantenere lo stato del sistema, ogni volta che nel sistema si verifica un evento di qualche genere dovrá essere invocato il metodo "updateState" che provvederá ad effettuare, se necessario, modifiche allo stato del sistema. Le interconnessioni tra il controller e le altre componenti del sistema sono realizzate al momento della configurazione iniziale del sistema da un componente addetto allo scopo.

```java
package pelican;

public interface IController {
  void setPedestrianLight(ISemaphore s);
  ISemaphore getPedestrianLight();
  void setVeichleLight(ISemaphore s);
  ISemaphore getVeichleLight();
  boolean hasReservation();
  void updateState(String newState);
}
```

/home/alessio/Scrivania/Natali/telecomandoRobot/sema/src/pelican/IController.java

Il comportamento del controller puó essere modellato con questo automa a stati finiti che indica su che valore verranno settati in ogni stato il semaforo pedonale e quello dei veicoli.
Gli eventi di transizione da uno stato al successivo in questo rappresentazione sono generati da entitá esterne al sistema: pedoni, tramite la pressione del pulsante, e timer.



5.5 Test plan

Il comportamento del semaforo deve soddisfare questo Test plan:

```
1  package pelican;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Before;
6  import org.junit.Test;
7
8  public class SemaphoreTest {
9    private ISemaphore s;
10
11   @Test
12   public void testSetLight() {
13     s.setLight(ColorType.GREEN);
14     assertEquals(ColorType.GREEN, s.getCurrentColor());
15     s.setLight(ColorType.RED);
16     assertEquals(ColorType.RED,s.getCurrentColor());
17     s.setLight(ColorType.YELLOW);
18     assertEquals(ColorType.YELLOW,s.getCurrentColor());
19   }
20
21 }
```

/home/alessio/Scrivania/Natali/telecomandoRobot/sema/src/pelican/SemaphoreTest.java

Il comportamento del controller deve invece soddisfare questo Test plan:

```
1  package pelican;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  public class ControllerTest {
8    private IController controller;
9    private ISemaphore pedestrianLight;
10   private ISemaphore vehicleLight;
11
12   @Test
13   public void testSetterGetter() {
14     controller.setPedestrianLight(pedestrianLight);
15
     assertEquals(pedestrianLight,controller.getPedestrianLight());
16     controller.setVeichleLight(vehicleLight);
17     assertEquals(vehicleLight,controller.getVeichleLight());
18   }
19
20   @Test
```

```
21    public void testStateTransition(){
22      controller.updateState("Verde-Auto-Stoppabile");
23
24      assertEquals(controller.getVeichleLight().getCurrentColor(),
            ColorType.GREEN);
25
26      assertEquals(controller.getPedestrianLight().getCurrentColor(),
            ColorType.RED);
27      assertEquals(controller.hasReservation(),false);
28
29      controller.updateState("Giallo-Auto");
30
31      assertEquals(controller.getVeichleLight().getCurrentColor(),
            ColorType.YELLOW);
32
33      assertEquals(controller.getPedestrianLight().getCurrentColor(),
            ColorType.RED);
34      assertEquals(controller.hasReservation(),true);
35
36      ....
37
38    }
39
40  }
```

/home/alessio/Scrivania/Natali/telecomandoRobot/sema/src/pelican/ControllerTest.java

Il test stateTransition andrà poi completato con gli stati mancanti. Per omogeneità si sono usati i nomi degli state presenti nel diagramma della macchina a stati finiti come argomento da passare alla funzione updateState;

6 Problem analysis

6.1 Logic architecture

Il sistema sarà composto da tre entità passive che devono essere in grado di comunicare tra di loro e reagire ad eventi esterni. In particolare l'entità controller dovrà essere in grado di comunicare ai semafori la luce corretta da accendere e dovrà essere attivata quando un evento che innesca una transizione di stato si verifica nel sistema.
Secondo questa concezione di sistema tutta la business logic è mantenuta dal controller, che è anche l'unico che sa in quale stato si trova il sitema in ogni istante. Qualsiasi evento si verifichi nel sistema dovrà quindi riflettersi in una chiamata ad "updateState" di controller con come parametro il nome dello stato in cui il sistema deve transitare. Così facendo abbiamo centralizzato tutta la parte critica di gestione del sistema in un unico punto, semplificandone la realizzazione, ma esponendo il sistema a possibili fault.
La natura intrinsecamente distribuita del progetto non cambia le precedenti considerazioni, l'unica cosa che viene modificata è la modalità di comunicazione: non

si useranno più delle invocazioni di funzioni semplici, ma un sistema a scambio di messaggi o ad eventi. L'unica modifica evidente è che in questa situazione tutte le entità dovranno avere un componente aggiuntivo che si occupi della parte di trasmissione, a meno che questo livello non sia implementato da un framework in maniera trasparente allo sviluppatore.

6.2 Abstraction gap

Per la corretta realizzazione del sistema avremmo bisogno di una metodologia o ancora meglio di un linguaggio/framework o libreria che ci permetta di descrivere e quindi generare automi a stati finiti.
Dovremmo quindi essere in grado di modellare in qualche modo il concetto di stato e di evento che causa una transizione di stato.
Sarebbe inoltre auspicabile che il sistema finale fosse in grado di funzionare senza problemi anche in ambienti distribuiti ed eterogenei, possibilmente modificando il meno possibile l'implementazione.
Date le caratteristiche dei requisiti é molto probabile che l'applicativo finale debba comunicare con dispositivi di controllo hardware, sarebbe quindi auspicabile l'utilizzo di tecnologie a basso costo ed ottima integrazione a basso livello che funzionino bene anche su questo genere di dispositivi.

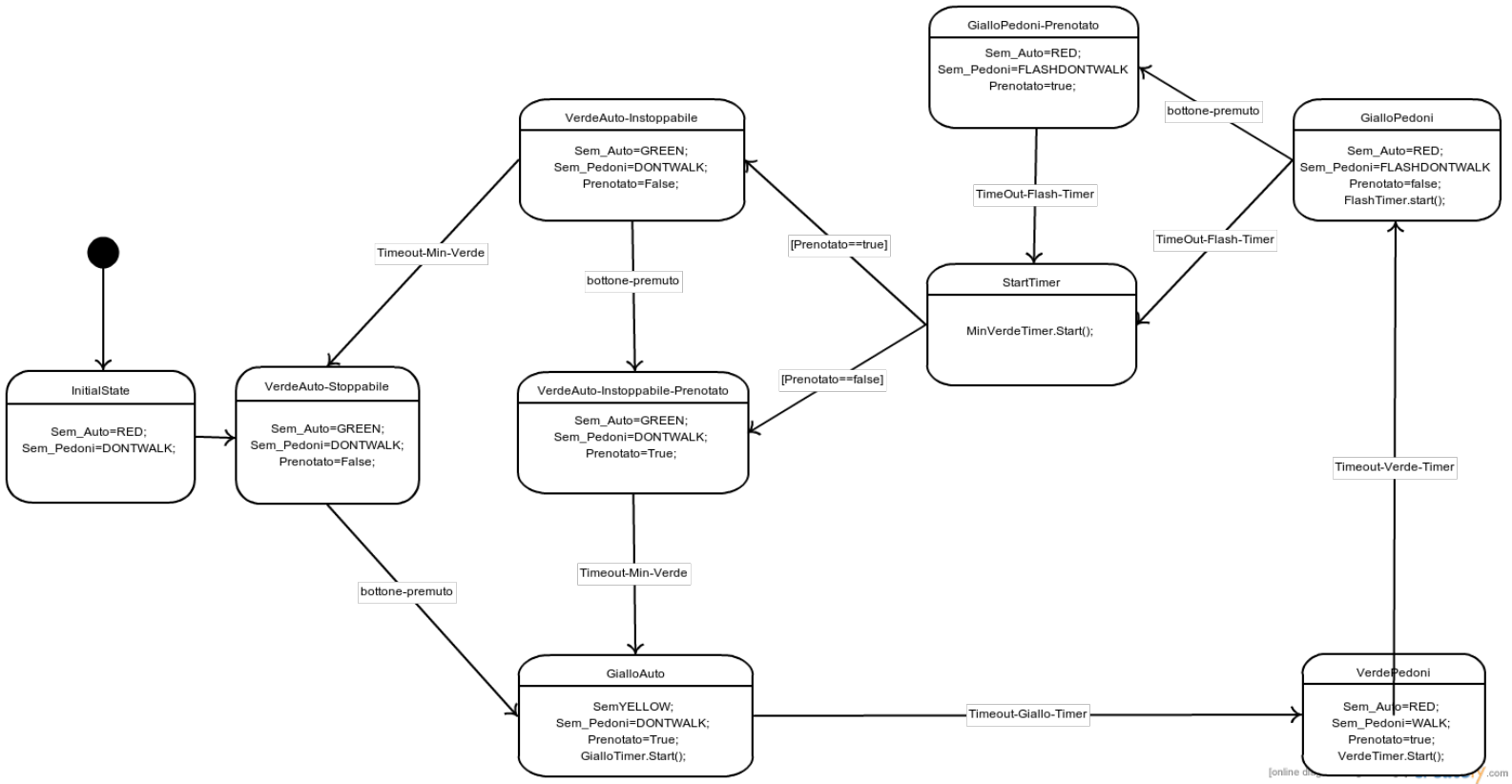
6.3 Risk analysis

7 Work plan

8 Project

8.1 Structure

Il prototipo é realizzato con la creazione di un'interfaccia grafica che rappresenti i due semafori e il pulsante di prenotazione e di un entitá controller in grado di comunicare con la GUI e il cui comportamento deve essere un'implementazione dell'ASF presentato in analisi. Le due entitá nel prototipo comunicheranno in locale in quanto questa prima implementazione sará concentrata e omogenea.
Affinche il prototipo sia testabile dovremo essere in grado di generare a comando gli eventi che determinano le transizioni di stato.
L'evento di prenotazione e quindi la pressione del bottone é stato delegata all'utente mediante interfaccia grafica, mentre si é deciso di far generare gli eventi di timeout dal sistema stesso mediante timer innescati negli stati del sistema.
Il sistema rivisto alla luce di queste considerazioni puó essere così rappresentato:


InitialState
Sem_Auto=RED;
Sem_Pedoni=DONTWALK;
VerdeAuto-Stoppabile
Sem_Auto=GREEN;
Sem_Pedoni=DONTWALK;
Prenotato=False;
VerdeAuto-Instoppabile
Sem_Auto=GREEN;
Sem_Pedoni=DONTWALK;
Prenotato=False;
VerdeAuto-Instoppabile-Prenotato
Sem_Auto=GREEN;
Sem_Pedoni=DONTWALK;
Prenotato=True;
GialloAuto
SemYELLOW;
Sem_Pedoni=DONTWALK;
Prenotato=True;
GialloTimer.Start();
VerdePedoni
Sem_Auto=RED;
Sem_Pedoni=WALK;
Prenotato=true;
VerdeTimer.Start();
GialloPedoni
Sem_Auto=RED;
Sem_Pedoni=FLASHDONTWALK
Prenotato=false;
FlashTimer.start();
GialloPedoni-Prenotato
Sem_Auto=RED;
Sem_Pedoni=FLASHDONTWALK
Prenotato=true;
StartTimer
MinVerdeTimer.Start();
Timeout-Min-Verde
bottone-premuto
[Prenotato==true]
[Prenotato==false]
bottone-premuto
Timeout-Min-Verde
Timeout-Giallo-Timer
TimeOut-Flash-Timer
bottone-premuto
TimeOut-Flash-Timer
Timeout-Verde-Timer

8.2 Interaction

Il prototipo realizzato sará concentrato e omogeneo, quindi tutte le interazioni avverranno in locale e saranno realizzate mediante la tecnologie di callback fornita da javascript e tramite variabili condivise per la comunicazione vera e propria.
Nell'implementazione finale del sistema le varie entitá saranno sicuramente distribuite, servirá quidni un metodo per permetterne l'interazione. Un approccio potrebbe essere quello di modificare le callback del prototipo in modo che anziche agire su variabili locali spediscano messaggi a entitá remote; ad esempio al momento della commutazione di stato il controller dovrá inviare al semaforo un messaggio comunicandogli quale luce deve accendere. Questa schema di interazione prevede peró che tutte le entitá del sistema siano dotate di componenti attive, altrimenti non sarebbero in grado di ricevere messaggi.

8.3 Behavior

9 Implementation

Al fine di testare la validità della modellazione del sistema é stato creato un semplice e rapido prototipo in javascript e html.
Per la parte di interfaccia grafica e interazione con l'utente é stata creata una semplice pagina html composta da due che simulano con il loro colore il comportamento dei semafori e da due pulsanti: "Start" che porta il sistema nello stato iniziale e "Prenotati" che simula il pulsante di prenotazione pedonale.
I due rappresentano una semplice implementazione dell'**entitá "Semaforo"** descritta nell'analisi, per i metodi set e get colore é stato usata l'integrazione tra javascript e il DOM HTML mediante il campo "BackgroundColor" dei .
Il pulsante "Prenotati" implementa l'**entitá "Pulsante di prenotazione"** dell'analisi, per realizzare il pattern observer si é sfruttata l'integrazione di javascript con il DOM HTML. L'aggiunta di un handler al pulsante é resa possibile semplicemente sovrascrivendo la proprietá "OnClik" del pulsante.
L'**entitá "Controller"** che ospita la logica di business del sistema é stata realizzata nel prototipo in javascript puro.
Sono state utilizzate variabili globali per tener traccia dello stato corrente del sistema (Sem_Auto, Sem_Pedoni e prenotazioni attive), la funzione "updateState" è stata per comoditá scomposta in una funzione per ogni stato dell'ASF; gli intervalli di timeout tra uno stato e il successivo del semaforo sono anch'essi stati modellati come variabili globali.
Ciascuna delle funzioni che modella uno stato si occupa di effettuare quando invocata le giuste modifiche allo stato dei due semafori e se necessario di far partire i timer che innescheranno la successiva transizione; il loro compito é quindi quello di portare il sistema in un nuovo stato.É stata usata una funzione per ogni stato per poter mappare direttamente le transizioni di stato sfruttando la tecnologie delle callback.

All'evento "OnClick" del pulsante "Prenotati" é collegato l'handler che porta l'applicazione nello stato Sem_Auto = GIALLO e Sem_Pedoni = VERDE, iniziando la serie di passaggi di stato temporizzati che alla fine riporteranno il sistema nello stato Sem_Auto = VERDE e Sem_Pedoni = ROSSO. La configurazione iniziale del sistema é stata delegata ad una appositá funzionione "init()" che si occupa di collegare correttamente il controller alle altre entitá del sistema. L'attivazione di eventi temporizzati e quindi la commutazione tra gli stati durante la fase di attraversamento temporale é stata ottenuta sfruttando la funzione built-in di javascript "setTimeout".

Limiti realizzativi:
Per la realizzazione del prototipo é stato utilizzato javascript senza l'ausilio di nessuna libreria, per questo si é dovuti scendere a compromessi in favore della velocitá di prototipazione come ad esempio la realizzazione dello stato del sistema come variabili globali o il mapping degli stati in callback invocate al verificarsi di determinati eventi. Un linguaggio di programmazione che offrisse built-in il concetto di stato come inteso in una macchina a stati finiti avrebbe facilitato lo sviluppo del prototipo e dell'applicazione finale.
Gli eventi che abilitano le trasizioni di stato sono stati implementati sfruttando la funzione "setTimeout" e l'integrazione standard con il DOM HTML permessa da javascript. Cosí facendo non abbiamo peró ottenuto vere e proprie entità indipendenti capaci volendo di portare informazioni aggiuntive, ma piuttosto dei puntatori a routine da eseguire all'avvenire di determinate azioni. Ció porta all'uso di funzioni accessorie non previste nella modellazione del sistema.
Il principale limite di questa rappresentazione é l'assenza di modellazione dei vincoli tra stati: niente vieta al sistema, se non il flusso di controllo, di passare dallo stato iniziale ad uno qualunque degli altri in caso di errori di implementazione. Questa modellazione semplificata non é quindi in grado di garantire che da uno stato si possa passare solo al o ai diretti successori. Un modo per ovviare a questo potrebbe essere controllare in ogni funzioni di modifica di stato la coerenza della situazione interna del controller rispetto al modello di ASF.
Un linguaggio o un framework che offra supporto alla modellazione di ASF avrebbe permesso una rappresentazione piú accurata di queste entitá.

10 Testing

Testando il prototipo cosí realizzato è stata verificata la coerenza della descrizione del sistema tramite ASF con quanto indicato nei requisiti.

11 Deployment

12 Maintenance

See [?] until page 11 (CMM) and pages 96-105.

13 Information about the author

Photo of the author