SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI INGEGNERIA

tesina del corso di

METODI FORMALI NELL'INGEGNERIA DEL SOFTWARE

ReDiA-VeriFInt *(State/Transition diagram)*

Realizzatore **D**iagrammi **A**utomatico con **Veri**fica **F**ormale **I**ntegrata

Supervisore

Toni Mancini

Autori

Andrea Frasca, Gianluca Ciuffa

Anno accademico 2007/2008

Indice

1 Introduzione **5**
1.1 Il progetto ReDiA-VeriFInt 5
1.2 Cenni al funzionamento di sistema 6
1.3 Struttura della tesina 7

2 Model **8**
2.1 Progettazione 9
2.1.1 Modello concettuale 9
2.1.2 Rappresentazione dei dati in XML 10
2.2 Realizzazione 13
2.2.1 Codifica del modello concettuale 13
2.2.2 Annotazioni JAXB 14

3 View **15**
3.1 Introduzione 15
3.2 I pannelli grafici 15
3.3 Il modello grafico 17
3.3.1 Ricapitolazione sull'architettura dell'engine 17
3.3.2 Uso dell'engine per i nostri scopi 18
3.4 La logica di controllo 19

4 Interazione con NuSMV **21**
4.1 Formato dei file .smv 21

4.2 Traduzione di un diagramma UML degli stati e transizioni in NuSMV . . . 22
4.3 Il foglio di stile XSLT . . . 24
4.4 Interfacciamento con il prover . . . 25
4.4.1 Aggiunta parametri nel file setup.ini . . . 25
4.4.2 Comunicazione con NuSMV . . . 26

5 Test su alcuni esempi **28**
5.1 Esempio 1 (Docente Universitario) . . . 28
5.1.1 Specifica . . . 28
5.1.2 Analisi della specifica . . . 28
5.1.3 Utilizzo dell'editor e del prover . . . 29
5.2 Esempio 2 (Cliente) . . . 30
5.2.1 Specifica . . . 30
5.2.2 Analisi della specifica . . . 30
5.2.3 Utilizzo dell'editor e del prover . . . 33

A Manuale d'uso **36**
A.1 L'interfaccia utente . . . 36
A.2 Sintassi LTL . . . 38
A.3 Linee guida per l'utilizzo . . . 38

Elenco delle figure

2.1 Class Diagram 9

3.1 ScreenShot dell'interfaccia 16
3.2 Class Diagram View Engine 18

4.1 Provers Diagram 27

5.1 State diagram esempio 1 29
5.2 Output specifica Professore (1) 31
5.3 Output specifica Professore(2) 32
5.4 State diagram esempio 2 33
5.5 Output specifica Cliente 35

A.1 Aree dell'intefaccia grafica 37

Capitolo 1

Introduzione

1.1 Il progetto ReDiA-VeriFInt

Progettare un'applicazione software abbastanza complessa come tutti sappiamo non è per niente facile. Spesso un diagramma UML (Class diagram, Use-case diagram, State/Transition diagram, ...) apparentemente corretto dopo una opportuna progettazione, risulta inconsistente con le specifiche richieste o magari contiene bachi nascosti derivanti da dimenticanze o quant'altro. L'uso di metodi formali (logica del prim'ordine, logica temporale, etc.) e annessi di tool di verifica dei primi che supportino la progettazione di un applicativo, è pertanto una buona strada da percorrere per realizzare software di una certa qualità e stabilità.

ReDia-VeriFInt è un progetto che ha come scopo quello di costruire un ambiente di sviluppo integrato, in grado di assistere il progettista/programmatore in tutte le fasi di progettazione e realizzazione delle proprie applicazioni, validando continuamente le sue scelte progettuali/implementative, fin dalla raccolta ed analisi dei requisiti. L'ambiente assiste l'utente tramite l'uso di opportuni strumenti di dimostrazione automatica come Prover9[1], NuSMV[2] etc. L'architettura portante del sistema compresa la parte relativa ai diagrammi UML delle classi è già stata realizzata da altri studenti del corso, e

[1] Dimostratore automatico per logica del prim'ordine
[2] Model Checker basato su logiche temporali

viene illustrata in modo esauriente in [DCDA08]. In questa tesina ci occuperemo di integrare nel sistema la parte relativa alla progettazione e verifica automatica dei diagrammi UML degli Stati e Transizioni sfruttando come tool di verifica formale NuSMV.

1.2 Cenni al funzionamento di sistema

Il progetto ReDia-VeriiFInt fa uso di diverse tecnologie eterogenee per riuscire a far interagire le diverse parti del sistema. In modo molto intuitivo passeremo adesso a presentarvi quali sono e come vengono utilizzate le stesse all'interno del nostro sotto-insieme di sistema.

Il funzionamento base è il seguente :

1. Il modello (o model) in grado di gestire le strutture dati del nostro diagramma e l'interazione tra di loro è codificato in codice Java.

2. All'interno delle classi Java che rappresentano gli elementi del nostro diagramma, vengono appuntate direttive JAXB[3] . Questi "tag" opportunamente collocati all'interno del codice permettono di generare in output, in un secondo momento tramite l'utilizzo di uno strumento (*XMLMarhsaller*), un file XML che rappresenta le istanze delle classi presenti nel sistema a runtime.

3. Una volta preso questo file XML, esso viene processato da un file XSLT[4], che sarebbe un foglio di stile per file XML. Tramite il file XSLT opportunamente strutturato possiamo generare in output qualsiasi tipo di file testo. Nel nostro caso, dato il file XML in input, noi vogliamo che l'XSLT ci generi il codice da dare in pasto al risolutore NuSMV, quindi che rispetti la sintassi del solutore da utilizzare.

4. Una volta generato il codice tramite il foglio di stile, esso viene inserito in un file generato dinamicamente.

[3]Java Architecture for XML Binding - tecnologia, studiata appositamente per la piattaforma Java, che consente di convertire in XML determinate istanze di oggetti presenti in memoria e viceversa.

[4]eXtensible Stylesheet Language Transformations

5. Viene invocato da Java quindi il solutore NuSMV sul file che abbiamo generato precedentemente.

6. Tramite lo standard output e lo standard error possiamo catturare la risposta del solutore per la presentazione all'utente.

Per una spiegazione più esauriente, consigliamo di leggere l'intero documento [DCDA08] che affronta in modo dettagliato queste tematiche.

1.3 Struttura della tesina

Capitolo2 Viene illustrato in dettaglio il modello concettuale del sistema, le scelte progettuali, realizzative e le tecnologie utilizzate.

Capitolo3 Viene illustrata in dettaglio la parte View del sistema e la GUI realizzata.

Capitolo4 Viene illustrata l'interconnessione tra le parti del sistema e NuSMV.

Capitolo5 Vengono esposti alcuni esempi di utilizzo del tool sviluppato.

AppendiceA Guida all'utilizzo dell'interfaccia grafica per realizzare diagrammi UML degli Stati e Transizioni.

Capitolo 2

Model

Come abbiamo detto in precedenza, quello che vogliamo realizzare è un tool che fornisca le basi per poter disegnare e validare diagrammi UML degli Stati e transizioni di modo da poter assistere l'utente finale durante la progettazione di un qualsivoglia sistema software.

Come in [DCDA08], per quanto riguarda la rappresentazione concettuale di un diagramma UML degli Stati e Transizioni siamo partiti prendendo spunto, sotto consiglio del docente, dal lavoro svolto da uno studente di Ingegneria Gestionale de La Sapienza, Michele Proni, il quale in [PRO07]ha portato avanti come studio per la sua tesi di laurea (relatore Prof. Mancini), proprio l'analisi approfondita della modellazione dei dati necessari alla rappresentazione di varie tipologie di diagramma UML tra cui anche quello che si faceva riferimento al nostro caso.

Il nostro lavoro è dunque partito con il prendere il lavoro di Proni, analizzarlo, e cercare di apportare meno modifiche possibili al suo elaborato in maniera tale da spendere poco tempo per quanto riguarda questa parte e incentrare i nostri sforzi sul lavoro di integrazione e realizzazione.

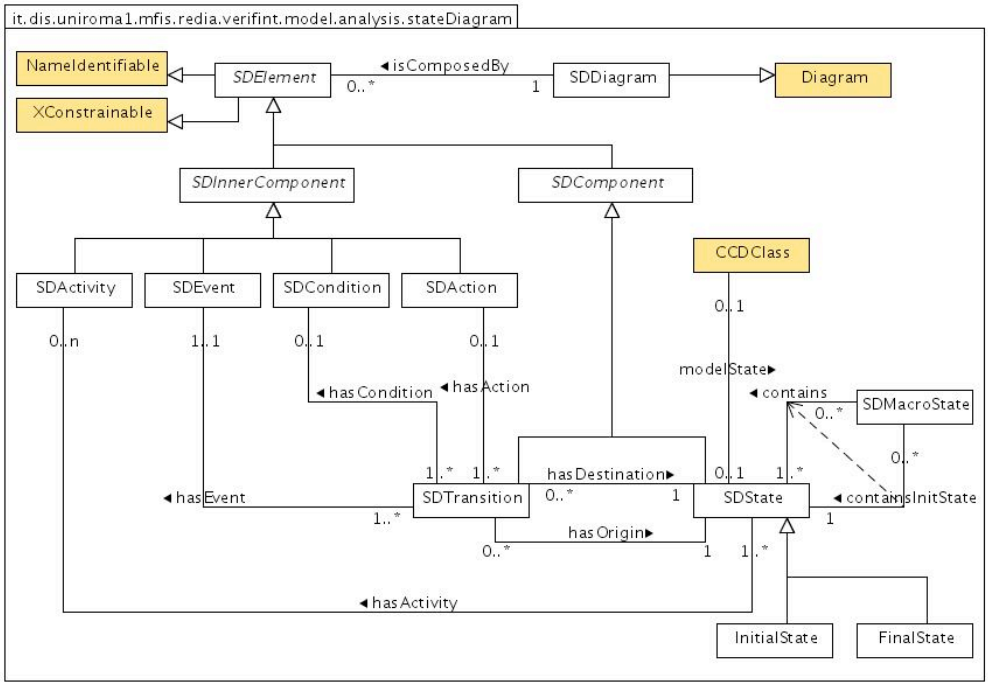

Figura 2.1: Class Diagram

2.1 Progettazione

2.1.1 Modello concettuale

Dall'analisi condotta da Michele Proni in [PRO07], emerge che il modello concettuale risultante di un diagramma degli stati e delle transizioni, è quello esposto in Fig2.1.

Vincoli esterni :

- Una istanza di *SDMacroState* non può contenere se stessa.
- Esiste solo una istanza di *InitialState* nel diagramma.

Con il colore arancio vengono indicate classi esterne al nostro sottosistema necessarie per l'integrazione con il resto dell'architettura portante.

Il suffisso SD, che sta a significare banalmente (State Diagram), è stato utilizzato per restare in linea con le regole di progettazione utilizzate in [DCDA08].

SDDiagram è la classe che rappresenta il diagramma, essa estende la classe padre *Diagram* (come tutti gli altri tipi di diagramma presenti nel sistema).

Ogni diagramma è composto da un insieme di elementi (*SDElement*). Questi vengono suddivisi in due classi fodamentali : *SDComponent* e *SDInnerComponent*; i primi rappresentano quelle entità auto-sufficienti (come uno stato) e i secondi quelli che devono necessariamente riferirsi ai primi per la loro identificazione (ad esempio, una azione).

Un ruolo di centrale interesse nel diagramma è ricoperto da una transizione (*SDTransition*). Essa è legata sempre, a un evento (*SDEvent*) e due stati (*SDState*) rispettivamente di origine e di destinazione, e in modo condizionale a Condizioni (*SDCondition*) e Azioni (*SDAction*).

Un altro elemento importante è lo stato (*SDState*) e le sue classi figlie. In particolare il macro stato (*SDMacroState*) è uno stato speciale che contiene all'interno un insieme di stati tra i quali ne esiste sempre uno iniziale relativo al proprio campo di applicazione.

2.1.2 Rappresentazione dei dati in XML

Seguendo la strada illustrata da [DCDA08], il secondo punto da affrontare è stato quello di progettare la stuttura di un opportuno file XML che descrivesse il nostro sistema. La prima domanda da farsi per strutturare un file XML è la seguente: quali sono le informazioni necessarie per descrivere completamente il sistema? Quali sono le dipendenze da mettere in risalto? Ovviamente ci possono essere più soluzioni equivalenti per descrivere tramite XML un dato sistema.

Tale struttura una volta decisa sarà univoca. Questo vuol dire che per ogni diagramma creato, l'XML finale dovrà essere strutturato seguendo le direttive progettate.

Nel nostro caso, la gerarchia dell'XML necessaria ai nostri scopi, è stato scelta secondo la forma :

```
<sdDiagram>
     <name>xxx</name>
     <components>
          <state identifierName="xxx" id="xxx">
               <name>xxx</name>
               <isInit>xxx</isInit>
               <isFinal>xxx</isFinal>
          </state>
          ...
          <transition identifierName="xxx" id="xxx">
               <name>xxx</name>
               <event identifierName="xxx" id="xxx">
                    <name>xxx</name>
               </event>
               <action identifierName="xxx" id="xxx">
                    <name>xxx</name>
               </action>
               <condition identifierName="xxx" id="xxx">
                    <name>xxx</name>
               </condition>
               <origin identifierName="xxx" id="xxx">
                    <name>xxx</name>
                    <isInit>xxx</isInit>
                    <isFinal>xxx</isFinal>
               </origin>
               <destination identifierName="xxx" id="xxx">
                    <name>xxx</name>
                    <isInit>xxx</isInit>
                    <isFinal>xxx</isFinal>
               </destination>
```

```
        </transition>
        ...
    ...
    </components>
    <innerComponents>
        <event identifierName="xxx" id="xxx">
            <name>xxx</name>
        </event>
        <action identifierName="xxx" id="xxx">
            <name>xxx</name>
        </action>
        <condition identifierName="xxx" id="xxx">
            <name>xxx</name>
        </condition>
    ...
    </innerComponents>
    <macroState>
        <name>xxx</name>
        <initialState>xxx</initialState>
        <states>
            <state identifierName="xxx" id="xxx">
                <name>xxx</name>
                <isInit>xxx</isInit>
                <isFinal>xxx</isFinal>
             </state>
        ...
        </states>
    ...
    </macroState>
```

La gerarchia è abbastanza intuitiva ed è stata scelta in questo modo per i seguenti motivi. Dato un diagramma riusciamo a ricavare tutti i suoi componenti (Components e InnerComponents) e i macrostati (se presenti) che

lo caratterizzano. Data ogni transizione riusciamo ad individuare l'evento, lo stato di origine, di destinazione e le eventuali azioni, condizioni associate. Da questo punto, se uno dei due stati (origine/destinazione) è un macrostato, o magari lo sono entrambi, usando l'attributo id (o identifierName) che identificano univocamente l'oggeto, possiamo navigare l'albero XML nella sezione *<macroState>* e processare le sue informazioni relative.

2.2 Realizzazione

2.2.1 Codifica del modello concettuale

Una volta progettato il nostro modello siamo dunque passati direttamente alla fase di realizzazione. Nel modellare la parte del diagramma realizzativo relativa al diagramma degli stati, è stata mantenuta praticamente inalterata la struttura prevista in sede di analisi ad eccezione della semplificazione fatta, come in [PRO07], riguardante la fusione in un'unica classe progettuale tra le classi di analisi "Stato", "StatoIniziale" e "StatoFinale". Il linguaggio di programmazione adottato è stato Java. Per ciò che riguarda la traduzione del diagramma è stata utilizzata la metodologia di realizzazione illustrata nel corso di Progettazione del Software I, illustrata esaurientemente in [MASCA08]. Per motivi del tutto pratici abbiamo assunto che tutte le responsabilità sulle associazioni fossero doppie. Il package di riferimento del codice implementato è *:*

it.dis.uniroma1.mfis.redia.verifint.model.analysis.stateDiagram.

In particolare per ciò che riguarda le classi relative alle associazioni, esse sono state inserite in un opportuno sottopackage: *_link.*

Seguendo lo stile di programmazione adottato in [DCDA08], abbiamo adottato anche per questa parte di progetto, per ovviare al problema di gestire associazioni con responsabilità multipla e vincolo di partecipazione, il design pattern Factory. Quindi per creare una istanza di ogni classe del diagramma bisogna passare per la classe Factory la quale occupa di creare gli oggetti e risolvere gli eventuali vincoli di partecipazione relativi all'istanza creata.

Un esempio di classe factory relativa alla classe *SDEvent* è il seguente :

```
public class SDEvent_Factory {
	public SDEvent event;

	public SDEvent createInstance(SDDiagram diagram,
				SDTransition transition,
				String name){
		this.event = new SDEvent(name);
		this.resolveLinkConstraints(diagram, transition);
		return this.event;
	}
	private void resolveLinkConstraints(SDDiagram diagram,
					SDTransition transition) {
		SDDiagram_isComposedBy_SDElement.addLink(diagram, this.event);
		SDTransition_hasEvent_SDEvent.addLink(transition, this.event);
	}
}
```

La funzione privata *resolveLinkConstraints* è quella che si occupa di risolvere i vincoli di partecipazione.

2.2.2 Annotazioni JAXB

In 1.1.2 abbiamo illustrato il DTO scelto per il nostro file XML di output. Per realizzarlo abbiamo dovuto apporre opportune annotazioni JAXB all'interno delle classi Java del nostro model. Per una spiegazione dettagliata su come apporre i "tag" JAXB si faccia riferimento all'Appendice A. di[DCDA08].

Il lavoro da una parte è stato abbastanza semplice, l'unica cosa a cui bisognava stare attenti era gestire il fatto di evitare la creazione di cicli nella produzione dell' XML risultante, incorrendo cosi in un errore da parte della funzione di *Marshall*[1].

[1]date delle classi java con annotazioni JAXB viene creato il file xml seguendo le etichettatura

Capitolo 3

View

3.1 Introduzione

Così come è stato fatto per il resto del progetto, anche la realizzazione dell'interfaccia grafica della nostra applicazione si appoggia al lavoro sviluppato in [DCDA08] , sia perché ne rappresenta il naturale proseguimento, sia perché parti delle classi e delle interfacce fornite si sono rivelate assai utili nell'ottica di una politica tesa al riuso del software. La filosofia di sviluppo seguita nella realizzazione della GUI è stata quella di implementare un interfacciamento con l'utente quanto più possibile semplice e funzionale, in modo che con pochi click si possa creare e gestire facilmente un diagramma degli stati e delle transizioni, sia al livello puramente grafico, sia al livello di verifica di consistenza.

3.2 I pannelli grafici

Le classi di supporto alla grafica sono state realizzate sfruttando le potenzialità messe a disposizione dalle librerie java: *java.awt* e *javax.swing*. Come è solito nelle applicazioni grafiche java la finestra principale è un' istanza della classe *JFrame* che contiene al suo interno un pannello di classe *JPanel*, la classe *ControlPanel*. Tale pannello ci permette di visualizzare sullo schermo

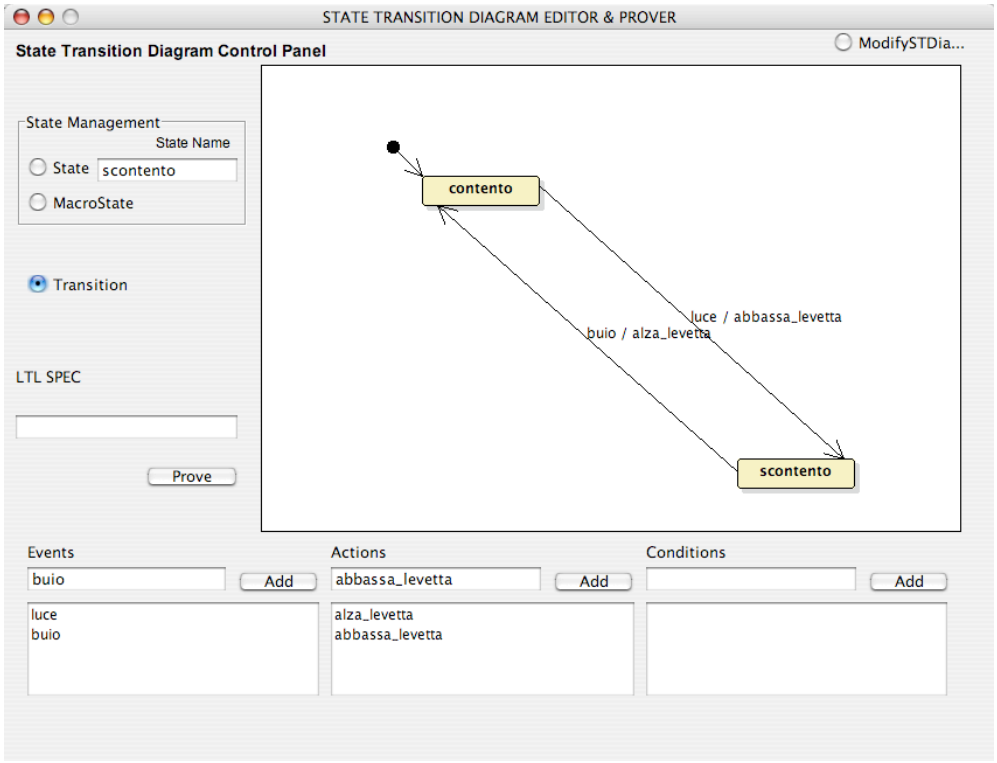

Figura 3.1: ScreenShot dell'interfaccia

tutti gli strumenti necessari a disegnare e verificare un diagramma degli stati e transizioni.

Uno screenshot dell'interfaccia utente dell'applicazione è visibile in Fig.3.1

Sulla colonna di sinistra tramite la selezione di 3 *radiobutton*, mutuamente esclusivi, si può scegliere tra la funzionalità di inserimento di stati, macrostati (quest'ultima non ancora completamente implementata) e transizioni, specificando nei primi due casi un nome nella apposita area testuale. In basso invece sono presenti 3 pannelli di classe *JList* che si riferiscono a nomi di eventi, azioni e condizioni fruibili nell'etichettatura di una transizione. In alto un ultimo radiobutton permette di interagire direttamente con il pannello centrale per modificare quanto già disegnato. Il pannello centrale permette di disegnare stati e transizioni semplicemente usando il mouse e

selezionando la funzionalità desiderata tramite i radio button di cui sopra. Tutto ciò che viene disegnato nel pannello è stato implementato a partire dal lavoro del progetto in [DCDA08].

3.3 Il modello grafico

Ciò che dobbiamo rappresentare sul pannello dedicato alla realizzazione del diagramma sono gli stati e le transizioni. Chiaramente il disegno risulta essere piuttosto semplice, poiché le funzioni messe a disposizione da java permettono di disegnare con una sola riga di codice sia rettangoli (per rappresentare gli stati), sia linee o frecce(per rappresentare le transizioni). Utilizzando tali funzionalità però si perde molto sia nella qualità grafica della rappresentazione, sia nell' interazione del pannello grafico con la logica di controllo dell'applicazione.

3.3.1 Ricapitolazione sull'architettura dell'engine

La struttura della componente grafica del progetto in [DCDA08] è composta da una interfaccia *Drawable* e da una serie di classi che la implementano vedi Fig.3.2.

Tale interfaccia offre un metodo *draw()*, che prende in input un oggetto grafico di classe *Graphics2D*. Tutte le classi del package in questione implementano tale interfaccia e di conseguenza implementano il metodo draw, che permette di disegnare, tramite le funzioni offerte da java, figure mediamente complesse. Ci sono due classi fondamentali in questo modello: la classe *Figure* e la classe *Edge* ed entrambi implementano l'interfaccia *Drawable*. La prima rappresenta tutti gli oggetti da disegnare sul pannello che hanno una bounding box, ossia un bordo, che risulta molto utile per l'interazione con il modello grafico, la seconda serve per disegnare le componenti grafiche di connessione come gli archi. Le classi *Label* e *Frame* estendono la classe *Figure*. La prima consente di visualizzare contenuti testuali, la seconda invece, può contenere al proprio interno altre istanze di classe Figure(come Label ad esempio) per poterle visualizzare raggruppate in un unico oggetto grafico.

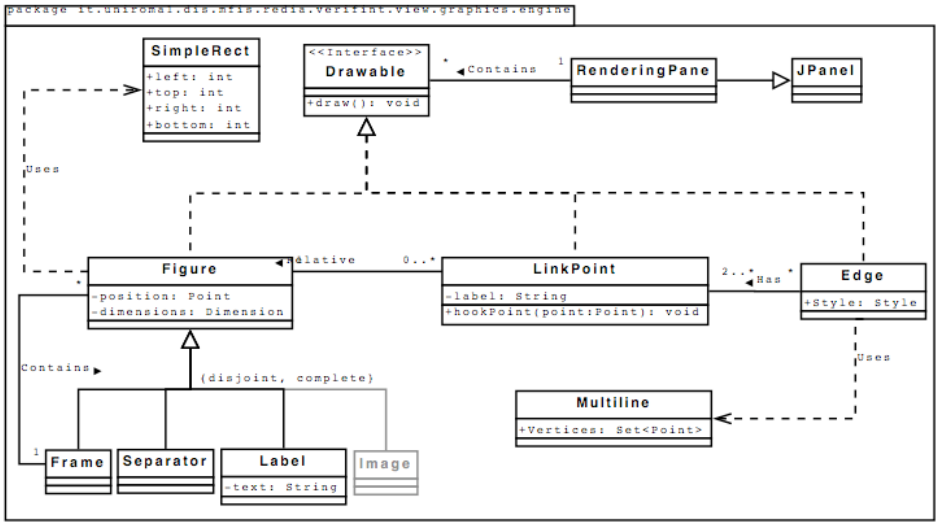

Figura 3.2: Class Diagram View Engine

Per una descrizione più completa delle classi che compongono il package di si rimanda alla discussione in [DCDA08].

3.3.2 Uso dell'engine per i nostri scopi

Partendo da quanto offerto dalle classi descritte in precedenza sono state realizzate le classi *StateFrame* e *MacroStateFrame* classi figlie della classe *Figure*, e *EdgeST* e *EdgeLoop* classi figlie di *Edge*. *StateFrame* rappresenta gli stati del nostro diagramma. La sua realizzazione è molto simile alla classe Frame, in quanto consente di disegnare un rettangolo e di specificare un oggetto di classe label da visualizzare al proprio interno. Il metodo *draw()* implementato disegna un rettangolo e tutte le label di cui si tiene traccia tramite la variabile di istanza di classe *List*. La classe ha anche due liste di oggetti di classe *EdgeST*, per tenere in memoria sia le transizioni che hanno tale stato come stato di destinazione, sia le transizioni che hanno tale stato come stato di origine. Tali strutture dati sono fondamentali per poter mantenere la coerenza di dati sia al livello grafico che concettuali quando si effettuano operazioni di cancellazione e spostamento. La classe offre metodi

per manipolare tali strutture dati. Inoltre un variabile booleana permette di stabilire se lo stato è iniziale o meno. *MacroStateFrame* rappresenta i macrostati nel diagramma ST. E' un'altra classe figlia della classe Figure e implementa il metodo *draw()* per raccogliere all'interno di un rettangolo gli stati selezionati sul pannello grafico. L'implementazione di tale classe tuttavia non è completa. *EdgeST* è la classe utilizzata per disegnare una transizione da uno stato ad un altro del diagramma. La classe ha delle variabili di istanza per memorizzare l'oggetto figure di destinazione e l'oggetto figure di origine della transizione. Il metodo *draw()* in questo caso disegna una linea, dal bordo di uno stato ad un altro, prendendo come riferimento le posizioni sulle quali il mouse ha cliccato. *EdgeLoop* è una classe figlia di *EdgeST* e serve per rappresentare le transizione che hanno lo stesso stato di origine e destinazione. Si è fatto inoltre largo uso della classe Label per etichettare stati e transizioni.

3.4 La logica di controllo

Per poter interagire graficamente con il pannello grafico è stato seguito il paradigma di programmazione Observer/Observable. Oltre ai listener posti in ascolto sui *JButton*, sono di particolare importanza i listener che controllano le azioni del pannello poiché proprio da lì nasce l'interazione dell'utente con l'applicazione. In particolare sul pannello di disegno è possibile specificare la posizione di uno stato quando viene creato o spostarlo. Si possono inoltre eliminare stati e transizioni. I listener si occupano dunque di gestire la veste grafica dell'applicazione, ma sono anche responsabili della coerenza tra quanto è rappresentato sul diagramma e quanto è tenuto in memoria dal modello concettuale. Il pannello su cui sono raccolti tutti gli oggetti che compongono la GUI, il *ControlPanel*, mantiene in memoria tramite alcune variabili d'istanza tutti gli oggetti che vanno disegnati sul pannello di disegno e un riferimento ad un oggetto di classe *StateDiagramBinder*, che estende l'interfaccia Binder (realizzata dal nostro gruppo in *it.mfis.dis.uniroma1.mfis.redia.verifint.control.binding*). Con il primo riferimento, una lista di oggetti Drawable, si realizza l'interazione grafica di

selezione, cancellazione e inserimento. Il secondo riferimento invece è quello che lega il livello di "view" al livello "model" dell' applicazione. La classe lega i nomi (scelti a livello grafico) delle componenti disegnate, agli oggetti creati al livello concettuale, memorizzandoli come coppia chiave valore in una HashMap. Grazie alla HashMap il "binder" può recuperare in qualsiasi momento l'id dell'oggetto su cui deve effettuare operazioni. La classe offre i metodi di inserimento e cancellazione di stati e transizioni, e di impostazioni delle proprietà degli oggetti del diagramma.

Capitolo 4

Interazione con NuSMV

Una volta strutturato il modello di sistema e avendo a disposizione un file XML che lo rappresenti in pieno, il problema fondamentale era interfacciarci con il model checker NuSMV per la verifica formali delle proprietà del sistema. NuSMV, disponibile con licenza open source alla pagina nusmv.irst.itc.it, è una reimplementazione ed estensione del sistema SMV (acronimo di symbolic model verifier), sviluppato originariamente presso la Carneige-Mellon University. Quest'ultimo permette la verifica di specifiche in CTL[1] per processi cooperanti che comunicano attraverso variabili condivise. NuSMV ha un più ampio linguaggio per le specifiche e nel suo sviluppo sono state introdotte numerose funzionalità nuove. Dal punto di vista linguistico, NuSMV fornisce la possibilità di specificare moduli, che rappresentano strutture temporali mediante variabili appartenenti a domini finiti (ad es., variabili booleane) e relazioni di transizione mediante opportuni costrutti. Inoltre è possibile dichiarare lo stato iniziale delle variabili. E' possibile specificare formule in CTL, LTL e LTL con operatori per il passato.

4.1 Formato dei file .smv

Per funzionare con il prover NuSMV un dato file deve avere estensione *.smv* e una opportuna sintassi. Per una analisi più approfondita sulla sintassi dei

[1] Computation Tree Logic.

file *.smv* si faccia riferimento a [NMAN08]. NuSMV può essere utilizzato sia come Model-Checker che come Model-Finder. Per quello che ci riguarda a noi interessano le sue capacità di uso come Model-Checker quindi lo utilizzeremo in questa modalità.

Di seguito a tipo puramente illustrativo è riportato il formato di file .smv che usano le direttive base necessarie per i nostri scopi.

```
MODULE main
VAR
...
ASSIGN
...
TRANS
    case
    ...
    esac;
--end MODULE
LTLSPEC
...
```

La direttiva MODULE indica il nome del modulo che vogliamo realizzare; main è un modulo speciale obbligatorio e necessario per far funzioare NuSMV.

La sezione VAR conterrà le variabili usate durante l'esecuzione. La sezione ASSIGN viene usata per dare una assegnazione o delle regole da applicare su alcune variabili. La funzione TRANS descrive le transizioni presenti nel nostro sistema sincrono. La sezione LTLSPEC conterrà la specifica che vogliamo verificare sul sistema descritto dalle sezioni precedenti.

4.2 Traduzione di un diagramma UML degli stati e transizioni in NuSMV

La metodologia adottata per la traduzione di un diagramma UML degli stati e transizioni si basa su quella presente in [CM07]. Un diagramma UML

degli stati e transizioni singolo, rappresenta bene un sistema sincrono, che reagisce ad eventi esterni mediante un cambiamento di stato e l'esecuzione di un'azione.

Dato un diagramma U, il file NuSMV avrà il seguente formato:

1. Viene dichiarato il modulo obbligatorio (main) per la descrizione del diagramma U che vogliamo realizzare
2. In questo modulo vengono dichiarate le variabili necessarie a descrivere gli stati, gli eventi, le azioni e le condizioni di U, che appartengono tutte a tipi enumerati opportunamente dichiarati; tali tipi comprendono null, che sta sia per l'evento sia per l'azione che per condizione nulla.
3. Se è contemplato uno stato iniziale nel diagramma esso viene indicato nella sezione ASSIGN
4. La funzione di transizione del primo modulo specifica le transizioni e le "non-transizioni" di U :
   (a) per ogni transizione di U va specificato un caso;
      i. nell'antecedente del caso (a sinistra di ":") vanno specificati :
         A. lo stato di partenza
         B. l'evento scatenante
         C. la condizione da verificare (se presente)
      ii. nel conseguente del caso (a destra di ":") vanno specificati :
         A. lo stato di arrivo
         B. l'azione compiuta (se presente),
   (b) le "non-transizioni" di U vengono catturate tutte insieme con un "caso di default" (in NuSMV viene eseguito solamente il primo caso il cui antecedente è vero); tale caso ("1", cioè "vero") specifica che, all'istante successivo; lo stato rimane lo stesso, e non viene compiuta alcuna azione (null);
5. la specifica LTL (unica) contiene la formula che si desidera verificare o confutare.

4.3 Il foglio di stile XSLT

Basandoci su quanto detto in 4.2 abbiamo dunque creato un foglio di stile che permettesse, dato un file XML contentente le informazioni descrittive del diagramma in esame, di generare in output codice per NuSMV che seguisse gli standard decisi preliminarmente.

Il file XSLT processa il file XML in input nel seguente modo (ricordiamo che la struttura del file XML da processare è riportato in 2.2.1) :

1. Scrive la direttiva : **MODULE main**
2. Scrive nella sezione **VAR** le variabili necessarie, processando opportunamente prima i nodi *<state>* e poi i figli di *<innerComponents>*.
3. Scrive la direttiva **ASSIGN** e l'eventuale stato iniziale del diagramma processando i nodi *<isInit>* figli di *<state>,* con la direttiva **init(state) := nomestato**
4. Scrive le transizioni in questo modo; per ogni nodo *<transition>* processato:
   (a) Se i suoi nodi figli *<origin>* e *<destination>* sono stati normali, viene scritta la transizione nel modo indicato in 4.2.
   (b) Se il suo nodo figlio *<origin>* è un macrostato e *<destination>* è uno stato normale, viene processato il nodo *<macrostate>* con lo stesso identificativo di *<origin>* e vengono aggiunte tante transizioni quanti sono gli elementi *<state>* contenuti nel *<macrostate>* con origine *<state>* e destinazione *<destination>*.
   (c) Se il suo nodo figlio *<origin>* è uno stato normale e *<destination>* è uno macrostato, viene processato il nodo *<macrostate>* con lo stesso identificativo di *<destination>* e viene inserita una transizione con stato di origine *<origin>* stato di destinazione il *<initState>* figlio di *<macrostate>*.

(d) Se il suoi nodi figli <*origin*> e <*destination*> sono entrambi dei macrostati, si procede come in (b) solo che come stato finale viene inserito l'elemento <*initState*> del macrostate riferito a <*destination*>.

La specifica LTLSPEC non viene scritta in questo momento ma aggiunta dal motore Java successivamente.

Per quanto riguarda la crezione di fogli di stile XSL(T) è stato redatto dal nostro gruppo un breve tutorial [FRA08], il quale illustra come sia semplice creare un file di output strutturato a partire da un XML e tramite l'uso di XSL.

4.4 Interfacciamento con il prover

Una volta avuto a disposizione il codice da processare l'ultima cosa da fare era realizzare la parte di controllo per la comunicazione con il Prover NuSMV.

4.4.1 Aggiunta parametri nel file setup.ini

Nella distribuzione del programma è presente un file di setup (setup.ini) da cui l'applicazione può leggere le direttive parametriche fondamentali, come il percorso interno al FileSystem locale in cui sono installati i verificatori automatici, i comandi da shell con il quale avviare questi ultimi, ecc. Il gestore di questo file, istanziato al lancio dell'applicazione, è denominato ExecutionProperties, e si appoggia alla classe del core Java java.util.Properties. Per quanto riguarda il nostro solver (NuSMV) la parte aggiunta è stata la seguente :

```
####################
# NUSMV parameters #
####################

### The "bin" folder into the NuSMV home directory
        nusmv_path = exe/
```

```
### The command used to launch NuSMV
### PLEASE correct this if and only if you are very sure of what you're doing
        nusmv_exeCommand = ./NuSMV

### The path of the XSLT file used to convert the local state transition diagram
### model to NuSMV input.
### PLEASE correct this if and only if you are very sure of what you're doing
        nusmv_sd_xsltFilePath = resources/xml/xslt/sd/nusmv.xslt

### The path of the .smv file generated at runtime
        nusmv_file_smv_path = exe/smv/SD.smv
```

che specificano essenzialmente dove si strova l'applicazione, il comando per eseguirla, dove si trova il foglio di stile XSLT e dove si trova il file .smv generato a runtime dall'applicazione da passare a NuSMV.

4.4.2 Comunicazione con NuSMV

E' statà realizzata dunque la classe *SDNuSMVProver* che estende la classe padre preesistente Prover ed è stata inserita rispettando l'architettura spiegata in [DCDA08] secondo lo schema in Figura 4.1.

La classe essenzialmente funziona nel seguente modo :

1. Prende in input un elemento XMLSerializable (il nostro diagramma di tipo *SDDiagram*)
2. Alla chiamata della funzione *prove(String goal)* :
   (a) Viene generato l'XML risultante tramite l'XMLMarshaller
   (b) Viene applicato il foglio di stile XSLT associato alla ripologia di diagramma passato in input (*nusmv.xlst*)
   (c) Viene aggiornato il file SD.smv contenente l'output
   (d) Viene aggiunto al file la specifica LTL da verificare (goal)

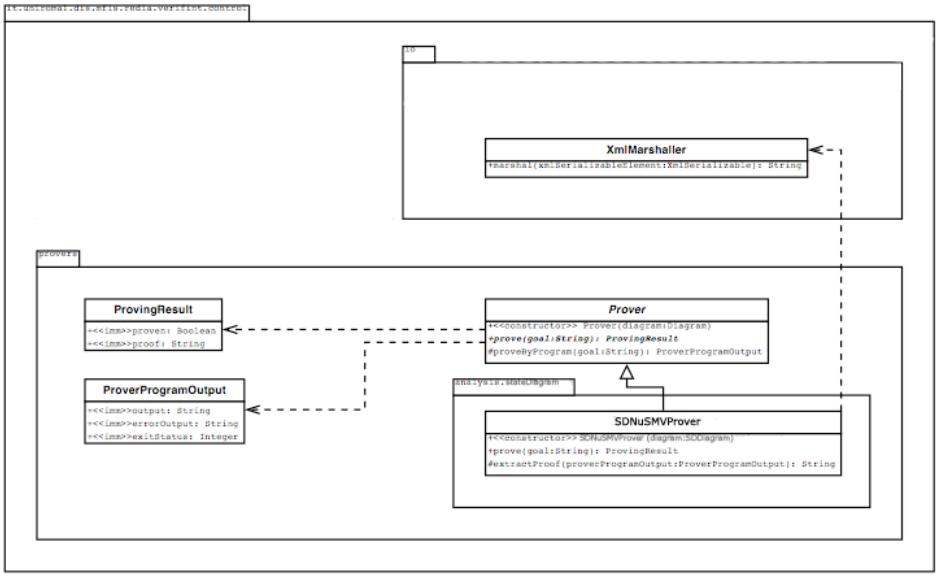

Figura 4.1: Provers Diagram

(e) Viene chiamato NuSMV sul file in questione

(f) Viene catturato e restituito l'output di NuSMV

Capitolo 5

Test su alcuni esempi

Andremo ora ad illustrare il funzionamento del tool realizzato in questa tesina applicandolo ad alcuni esempi presi da [CM07]. Per ciò che concerne la sintassi usata per le proprietà LTL da verificare, si faccia riferimento all'Appendice A di questo documento.

5.1 Esempio 1 (Docente Universitario)

5.1.1 Specifica

Un docente universitario può essere in servizio oppure assente, in particolare per ferie, malattia o anno sabbatico. Dall'anno sabbatico si può tornare solamente in servizio, mentre se se si è in ferie si può anche andare in malattia e se si è in malattia si può anche andare in anno sabbatico.

5.1.2 Analisi della specifica

- Un docente universitario può essere in servizio oppure assente, in particolare per ferie, malattia o anno sabbatico;
- Dall'anno sabbatico si può tornare solamente in servizio;
- Se se siè in ferie si può anche andare in malattia;
- Se si 'e in malattia si può anche andare in anno sabbatico;

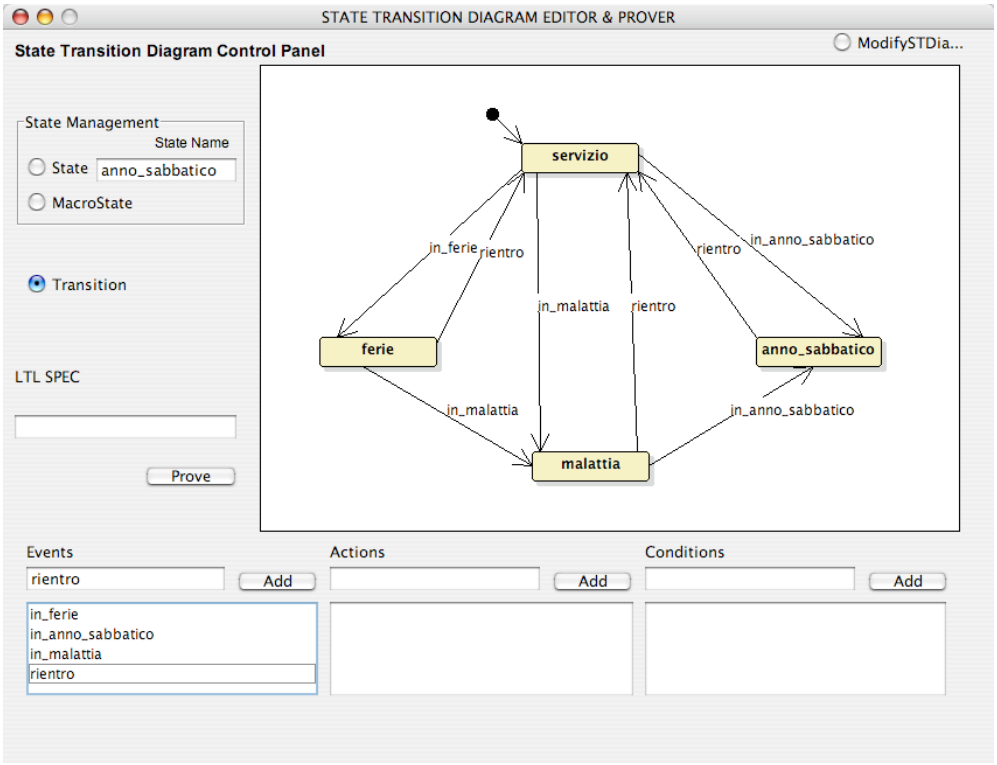

Figura 5.1: State diagram esempio 1

A titolo di esempio, aggiungiamo alcune proprietà che sono implicite nei requisiti, e che desideriamo siano verificate:

- Dalle ferie non si può andare direttamente in anno sabbatico;
- Dall'anno sabbatico non si può andare direttamente in ferie;
- Da ferie, anno sabbatico e malattia si può sempre tornare in servizio;

5.1.3 Utilizzo dell'editor e del prover

La rappresentazione del sistema tramite l'editor grafico risulta quella risultante in Fig 5.1.

Una volta disegnato il nostro sistema possiamo passare a testare alcune proprietà interessanti che dovrebbe soddisfare.

In particolare verificheremo le due seguenti proprietà :

1. Dall'anno sabbatico non si può andare direttamente in ferie

   - Specifica LTL : *G (state = anno_sabbatico -> X state != ferie)*

2. Dall'anno sabbatico si può tornare solamente in servizio

   - Specifica LTL : *G ((state = anno_sabbatico & event = rientro) -> – X (state = in_servizio))*

Inserendo le specifiche LTL da verificare nel campo riservato, e premendo successivamente il tasto "prove" dell'interfaccia verrà dato in output come risultato rispettivamente Fig 5.2 e Fig 5.3

5.2 Esempio 2 (Cliente)

5.2.1 Specifica

Un utente generico, appena diventa cliente di un istituto bancario viene catalogato come nuovo. Dopo il primo evento il cliente è affidabile o moroso a seconda del fatto che esso paghi oppure sia in ritardo con i pagamenti; Dopo il primo evento il cliente non sarà mai più nuovo. Un cliente che continua a pagare rimane affidabile. Un cliente in ritardo anche solo una volta non sarà mai più affidabile.

5.2.2 Analisi della specifica

- Inizialmente il cliente è nuovo ;
- Dopo il primo evento il cliente è affidabile o moroso;
  - Se un cliente nuovo paga diventa affidabile;
  - Se un cliente nuovo non paga diventa moroso;
- Dopo il primo evento il cliente non sarà mai più nuovo;

Figura 5.2: Output specifica Professore (1)

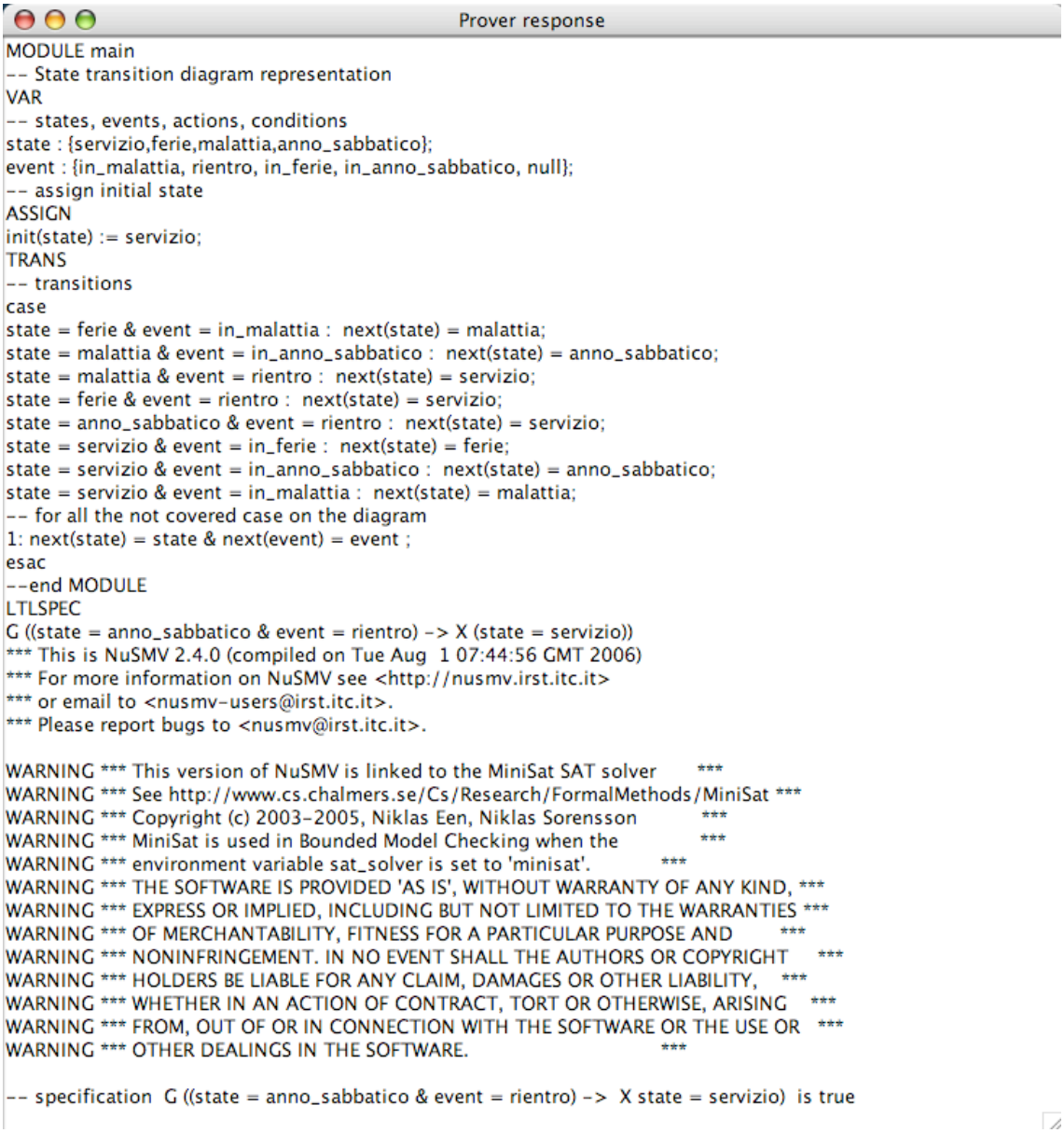

Figura 5.3: Output specifica Professore(2)

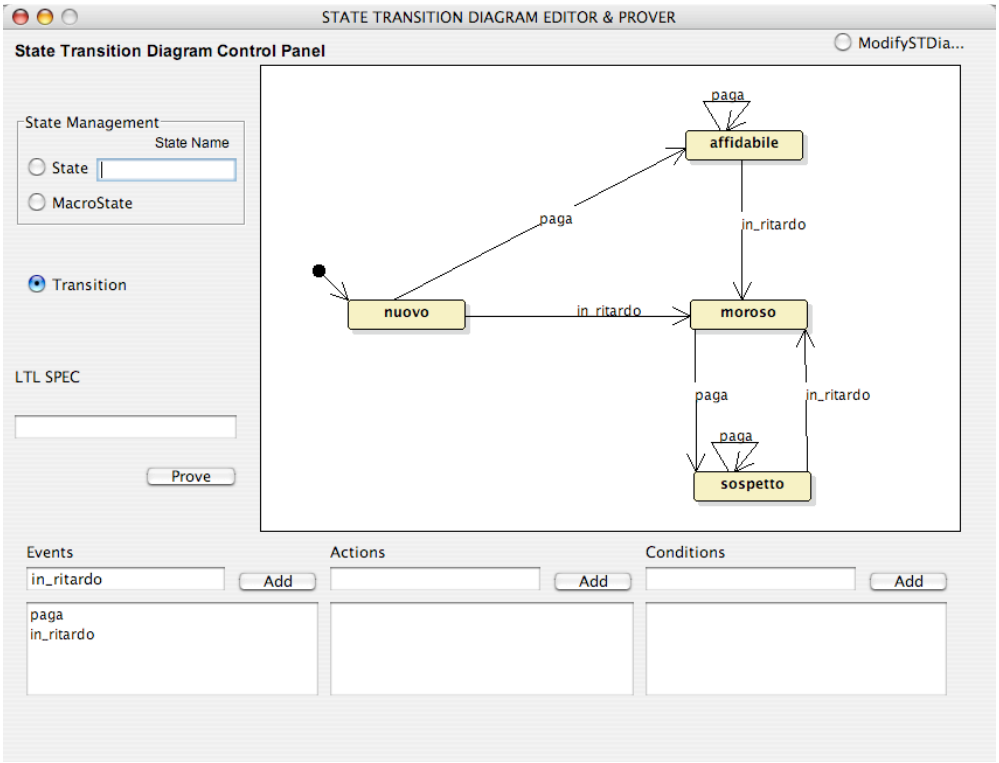

Figura 5.4: State diagram esempio 2

- Un cliente continua a pagare rimane affidabile o nuovo;
- Un cliente in ritardo anche solo una volta non sarà mai più affidabile.(necessità della creazione di un nuovo stato "sospetto").

5.2.3 Utilizzo dell'editor e del prover

La rappresentazione del sistema tramite l'editor grafico risulta quella risultate in Fig 5.4.

Una volta disegnato il nostro sistema possiamo passare a testare alcune proprietà interessanti che dovrebbe modellare.

In particolare verificheremo la seguente proprietà :

- Un cliente in ritardo anche solo una volta non sarà mai più affidabile :

- Specifica LTL :
  - *G(F event = in_ ritardo -> F G (state != affidabile))*
  - specifica migliore, usando W : *(event != ritardo U X G state != affidabile) | G event != in_ ritardo*

Inserendo la specifica LTL da verificare nel campo riservato, e premendo successivamente il tasto “prove” dell’interfaccia verrà dato in output come risultato vedi Fig 5.5

Figura 5.5: Output specifica Cliente

Appendice A

Manuale d'uso

Il seguente paragrafo costituisce una guida all'uso del sistema "State/Transition Diagram Editor & Prover ".

A.1 L'interfaccia utente

Graficamente possiamo dividere l'interfaccia grafica in tre aree visualizzabili graficamente in Fig. A.1:

1. Un'area posta a sinistra della schermata dove troviamo i comandi per inserire stati e transizioni e le specifiche LTL da verificare (Colore rosso Fig A.1).

2. Una fascia inferiore troviamo i pannelli per inserire nomi di eventi, azioni e condizioni (Colore blu in Fig A.1).

3. Una parte centrale dove è presente il pannello di disegno, con un radio button posto in alto, che permette di cambiare la modalità da inserimento a modifica di quanto già disegnato (Colore verde in Fig A.1).

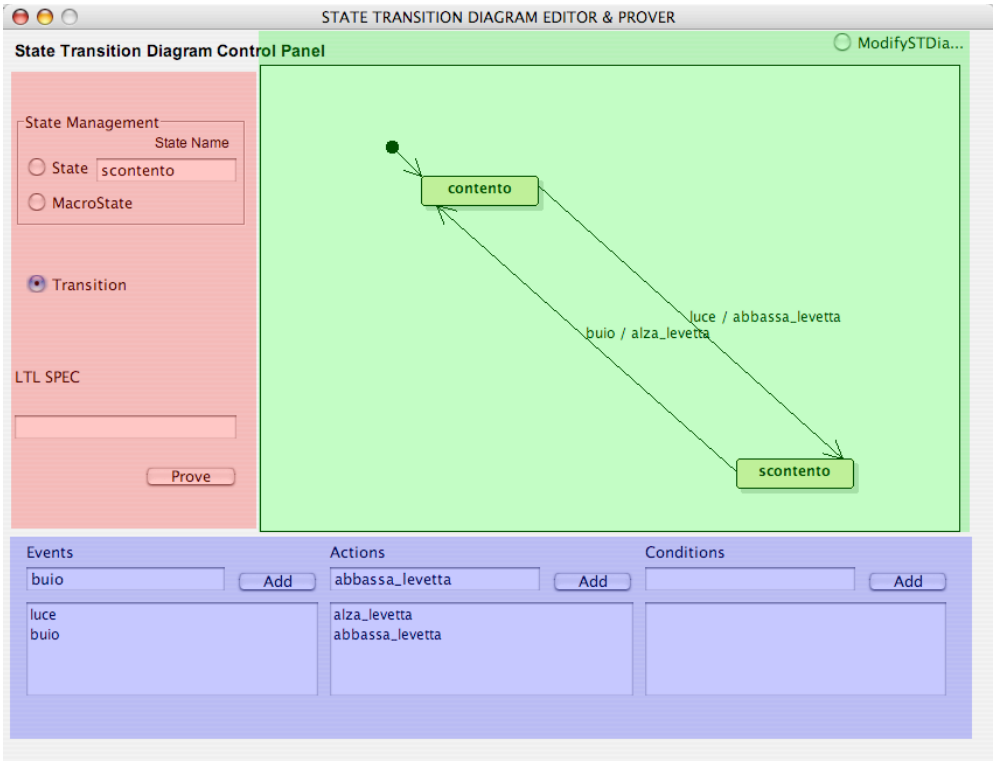

Figura A.1: Aree dell'intefaccia grafica

A.2 Sintassi LTL

Per ciò che concerne le proprietà LTL da inserire nella sezione LTLSPEC dell'interfaccia, la sintassi da utilizzare è quella standard di NuSMV (per approfondimenti al riguardo si faccia riferimento a [NMAN08]). In particolare gli stati, le azioni, le condizioni fanno riferimento alle seguenti variabili di default :

- La variabile che identifica gli stati è *state.*
- La variabile che identifica le azioni è *action.*
- La variabile che identifica le condizioni è *condition.*

I possibili valori assegnabili alle variabili sono esclusivamente i nomi degli stati, azioni e condizioni inseriti nel diagramma. Per qualche esempio di utilizzo si faccia riferimento agli esempi riportati in Cap.5 .

A.3 Linee guida per l'utilizzo

Passiamo adesso alla specifica delle azioni da compiere per disegnare uno *state & transition diagram:*

- **Inserire uno stato all'interno del diagramma**: selezionare il radio button *State* e specififcare all'interno dell'area testuale *State Name* il nome dello stato. Lo stato viene inserito facendo click sul pannello centrale. Il primo stato inserito viene impostato come stato iniziale del diagramma. Sarà successivamente possibile modificare questa impostazione.
- **Inserire una transizione**: selezionare il radio button *Transition* , selezionare nelle tre liste nella parte inferiore del pannello almeno un item che rappresenti il nome dell'evento (se non è presente nessun item è necessario aggiungerlo - vedi dopo). A questo punto fare click sul pannello all'interno di uno stato: appare una linea che segue il movimento del mouse fino a che il mouse non sarà clickato di nuovo.

La transizione verrà disegnata solamente i due click sono stati effettuati all'interno di due stati del diagramma.

- **Aggiungere un evento**: la prima delle tre liste serve a tenere traccia degli eventi che sono stati presi in considerazione. Per aggiungerne uno è sufficiente digitare il suo nome nell' area di testo posta sopra la lista, e premere il pulsante Add. Per aggiungere una condizione o una azione si segua la stessa procedura, utilizzando le aree di testo ad esse dedicate.

- **Modifcare il diagramma**: selezionare il radio button Modify State Diagram posto sopra il pannello di disegno. Facendo click su uno stato appare un menù che permette di eliminare, spostare e impostare come iniziale lo stato selezionato. Non si può eliminare uno stato iniziale. Per spostare uno stato selezionare Move e trascinare lo stato nella posizione desiderata. Per deselezionare uno stato, fare nuovamente click su di esso. Analogamente, è possibile eliminare le transizioni: cliccando vicino alla freccia apparira un menù che consente di effettuare la cancellazione.

- **Verifica formale delle proprietà del sistema**: inserire una specifica nella casella di testo LTLSPEC seguendo la sintassi menzionata in A.1.2 e premere il tasto Prove. Apparirà una nuova finestra che visualizza l'output del prover NuSMV.

  Attenzione: tutti gli elementi inseriti (stati, transizioni, eventi, condizioni e azioni) devono avere un nome diverso, in caso contrario l'interfaccia ne impedirà l'inserimento.

Bibliografia

[CM07] Marco Cadoli and Toni Mancini. Metodi Formali nel l'Ingegneria del Software, chapter 8 - Diagrammi UML degli stati e delle transizioni . SAPIENZA - Dipartimento di Informatica e Sistemistica, 2007.

[PRO07] Michele Proni. OOPS: un'applicazione per il supporto alla progettazione e alla realizzazione di software object oriented. Tesi di Laurea in Ingegneria Gestionale a.a. 2006-2007, presso SAPIENZA – Università di Roma, 2007.

[FRA08] Andrea Frasca. XML e XSLT per la generazione automatica di codice. SAPIENZA - Univesità di Roma, 2008.

[DCDA08] Claudio Di Ciccio and Fabio D'Aprano, ReDiA-VeriFInt Realizzatore Diagrammi Automatico conVerifica Formale Integrata. SAPIENZA – Università di Roma, 2008.

[MASCA08] Toni Mancini and Monica Scannapieco. Corso di Progettazione del Software per il Corso di Laurea in Ingegneria Gestionale, chapter S.R.1-S.R.5. SAPIENZA - Dipartimento di Informatica e Sistemistica, 2008. http://www.dis.uniroma1.it/~tmancini.

[NMAN08] NuSMV 2.4 User Manual. Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri and Andrei Tchaltsev ITC-irst - Via Sommarive 18, 38055 Povo (Trento) – Italy Email: *nusmv@irst.itc.it*