ReDiA-VeriFInt

Realizzatore Diagrammi Automatico conVerifica Formale Integrata

tesina svolta per il corso di:

Metodi Formali nell'Ingegneria del Software

Facolta' di Ingegneria Informatica
SAPIENZA – Universita' di Roma

Autori:
Giovanni Michele Toglia
Carlo Tassi

Supervisore: Prof. Toni Mancini

Indice relazione

1. Obiettivo e contesto

Il lavoro da noi svolto ha come obiettivo l'estensione del CASE tool in via di sviluppo per la verifica automatica di diagrammi UML delle classi, ReDiA-VeriFInt.

è stata progettata e sviluppata una prima versione di ReDiA-VeriFInt (Realizzatore Diagrammi Automatico con Verifica Formale Integrata), un progetto pilota di CASE, un IDE che permette di assistere il lavoro di analisti, quindi aiuta, progettisti e programmatori nello sviluppo di applicazioni, tramite la generazione e la gestione di diagrammi UML e codice lungo tutte le fasi, con annessa la verifica automatica di opportune proprietà formali, sulla base di quanto realizzato.

In particolare, il programma permette l'invocazione (per ora solo su richiesta dell'utente) di un dimostratore di teoremi del prim'ordine per verificare proprietà dei diagrammi UML delle classi. L'idea del nostro lavoro è quella di estendere il sistema per permettere:

1. Il supporto alla verifica automatica mediante Alloy, compito reso semplice dall'architettura del sistema;
2. Il completamento dell'interfaccia grafica per il disegno di diagrammi UML concettuali delle classi e dell'implementazione del livello di collegamento (Control) tra l'interfaccia (View) e la logica applicativa (Model).
3. Il supporto all'invocazione automatica, basata su eventi, dei diversi dimostratori di teoremi (ora il sistema permette l'invocazione da parte dell'utente di Prover9, il successore di Otter) su un portafoglio predefinito di tesi da dimostrare (ad es., l'invocazione automatica per dimostrare un'inconsistenza del diagramma dopo averne aggiunto un componente).

è in fase di sviluppo, da parte di più gruppi, un sistema CASE (la cui analisi è stata effettuata nell'ambito di un paio di tesi di laurea) per consente il supporto alle seguenti fasi del ciclo di sviluppo, fornendo servizi di verifica automatica:

A. Fase di Analisi:

A.1 Disegno di Diagrammi Use Case con attori, use-case, include, extend, is-a tra attori e use-case

A.2 Disegno di Diagrammi delle classi concettuali con classi (attributi e operaz.), assoc (anche con attibuti), is-a tra classi e gerarchie disjoint/complete (il supporto di questo costrutto dovrebbe essere gia' presente nel codice ma commentato, perche' Ottaviani ha poi proceduto a supportare solo la fase di progetto), subset di assoc., vincoli esterni.

A.3 Traduzione automatica del diagramma delle classi in Otter e Alloy, e verifica automatica di sue proprietà.

A.4 Disegno di diagrammi degli stati e transizioni (per le singole classi).

A.5 Traduzione automatica dei diagrammi degli stati e transizioni in SMV e altri linguaggi per la verifica di loro proprietà.

A.6 Definizione di Specifiche concettuali di tipi di dato, use-case, classi (ovviamente con legami agli elementi dei diversi diagrammi).

A.7 Attività di verifica su specifiche concettuali.

P. Fase di progetto:

P.1 Sintesi semi-automatica del Diagramma delle classi realizzativo con tipi java, ristrutturazione delle gerarchie, eliminazione subset in favore di vincoli esterni, visibilità

P.2 Traduzione automatica del diagramma delle classi realizzativo in Otter e Alloy, verifica automatica di sue proprietà, e verifica della sua equivalenza rispetto al diagramma

concettuale

P.2 Definizione di Specifiche realizzative di use-case, tipi (strutture dati), classi con specifica di algoritmi

P.3 Verifica della correttezza delle specifiche realizzative (algoritmi) rispetto alle specifiche concettuali

R. Fase di realizzazione:

R.1 Generazione automatica di codice per tutti i moduli definiti nella fase di progetto.

R.2 Possibilita' di scrivere codice Java dei metodi definiti in fase di progetto.

R.3 Verifica di correttezza del codice Java relativo ai metodi rispetto alle specifiche concettuali.

R.4 Generazione automatica di casi di test white-box per tutti i metodi.

2. Introduzione al lavoro svolto

Di seguito vengono riportate in un quadro molto generale, che verrà poi dettagliato in seguito, le caratteristiche del lavoro da noi svolto.

Questo lavoro è stato prodotto in team di due persone per cui i tempi di lavorazione si sono allungati notevolmente anche per via dell'apprendimento di nuove tecnologie e linguaggi.

Gran parte del tempo totale speso per la realizzazione dei suddetti punti di estensione è stato impiegato per la realizzazione della GUI e del file *xsl* per la traduzione del diagramma da linguaggio xml al linguaggio per il prover Alloy4. Il resto del tempo invece è stato impiegato per realizzare l'integrazione del tool di verifica formale Alloy4 e per realizzare lo strato di collegamento (control) tra la GUI (view) e la logica applicativa (model) quindi tra il prover e la logica applicativa.

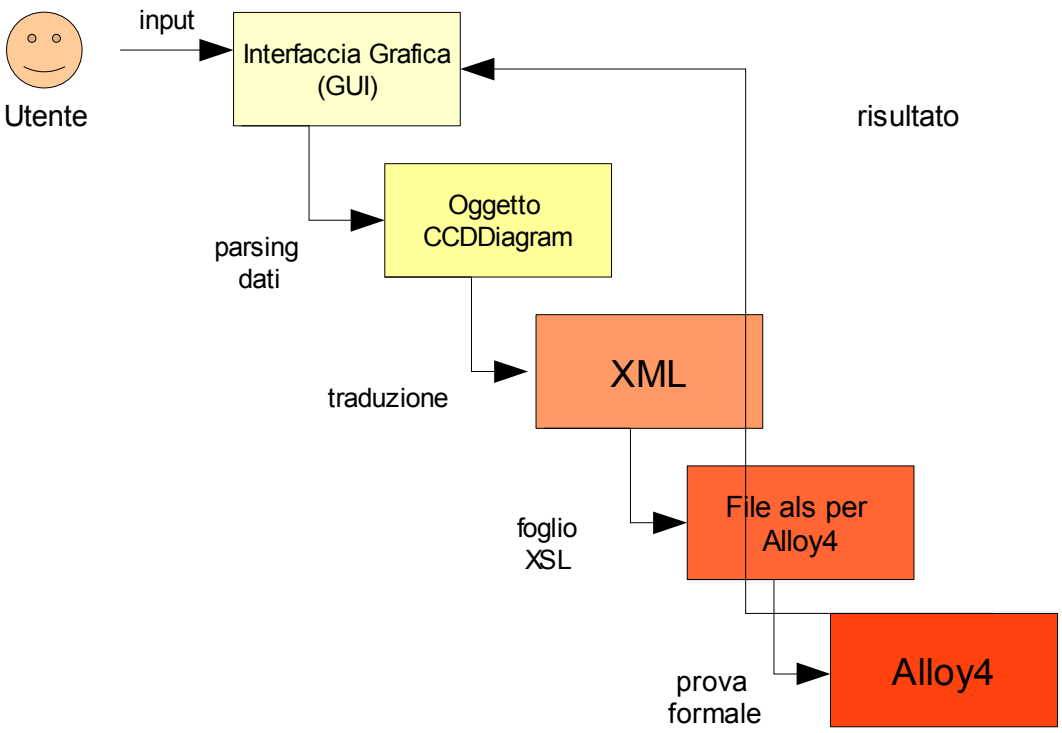
Per quanto riguarda la GUI siamo partiti dalla base sviluppata dai precedenti colleghi che permetteva tramite un motore grafico che opera a basso livello il disegno di tutti i componenti necessari ad un diagramma delle classi.

In realtà vi era anche una serie di classi di utilità per poter presentare e rendere modificabili a livello grafico le proprietà degli elementi dei diagrammi, ma si è preferito riprogettare tale livello in quanto si tratta di una caratteristica non sperimentata e non integrata col resto del sistema.

Il prover Alloy4 è stato integrato sfruttando la sua caratteristica di essere open source, infatti il package di Alloy4 è stato inglobato in quello del Rediaverifint per poi collegarlo tramite la creazione di una classe apposita resa accessibile dal tool di progettazione per la verifica formale dei diagrammi creati.

Infine le classi di collegamento (control) sono state realizzate modificando in parte il lavoro svolto precedentemente ma non ancora completato e in parte realizzando da zero un Class Diagram Controller in grado di trasformare l'input dell'utente fatto di "disegni" in oggetti che vanno a formare l'oggetto diagramma. Questo poi viene tradotto in xml (funzionalità già realizzata), a sua volta tradotto dal file xsl in linguaggio per il prover.

Qui sotto è riportato un semplice schema che descrive il funzionamento globale dell'applicazione:



3. Il lavoro svolto

3.1. Interfaccia grafica (GUI)

L'interfaccia grafica è stata studiata per permettere all'utente di poter gestire contemporaneamente più di un diagramma (per ora solo delle classi) e inoltre rendere semplice e guidato il processo di creazione del diagramma stesso cercando di minimizzare il numero di possibili errori che può inavvertitamente commettere l'utente grazie anche all'uso di chiari messaggi popup e al fatto che alcuni *MenuItem* siano disabilitati e abilitati a seconda dello stato in cui si trova il diagramma.

Il codice delle classi dell'iterfaccia è visibile nel package:

`it.uniroma1.dis.mfis.redia.verifint.view.graphics.gui.frames.ClassDiagram;`

La finestra principale (*PrincipaleFrame.java*) presenta in alto la classica barra dei menù da cui è possibile creare il proprio progetto inserendo un numero arbitrario di diagrammi, per ora solo delle classi, ma in futuro potrà essere esteso a qualsiasi tipo di diagramma grazie al fatto che il programma gestisce un *ArrayList* di oggetti di tipo *Diagram* (superclasse per ogni tipo di diagramma).

Ogni tab è associato ad uno ed un solo diagramma e presenta l'area divisa in tre zone principali:

- la parte centrale in cui viene disegnato il diagramma. Si tratta di un oggetto di tipo *RenderingPane* (che estende *JPanel*). Questo oggetto è stato "ereditato" dai precedenti sviluppatori per cui è stato semplice integrarlo e sfruttarne le capacità.
- la parte a destra in cui vengono indicate in forma testuale le classi, le associazioni e le generalizzazioni facenti parte del diagramma e aggiornate dinamicamente ad ogni modifica. Si tratta di semplici oggetti di tipo *JTextArea* non editabili.
- la parte inferiore dedicata alla verifica formale che a sua volta è divisa in due parti, una in cui viene mostrata la traduzione del diagramma in linguaggio Alloy4 e l'altra in cui viene

stampato il risultato della prova formale ogni volta che al diagramma aggiungo o sottraggo un componente. In ogni modo, in caso di errore formale viene mostrato un messaggio che avvisa dell'insoddisfacibilità del diagramma così come è stato costruito. Anche in questo caso si tratta di semplici oggetti di tipo *JTextArea.*

A questo punto è possibile iniziare ad editare il diagramma e a questo scopo gli strumenti necessari sono nel menù Edit o nel menù popup invocato cliccando col pulsante destro del mouse su un componente del diagramma.

Una volta editato uno o più diagrammi è possibile effettuare il salvataggio del progetto (costituito da tutti i tabs aperti e i relativi diagrammi) su filesystem. Il salvataggio avviene con le classiche modalità, sfruttando i componenti *javax.swing* già pronti per l'uso. I file salvati hanno estensione .mfis. Analogamente è possibile caricare i file salvati precedentemente, tuttavia la funzione di caricamento non è completa, manca solo la funzione di costruzione di ogni diagramma sia graficamente che come oggetto che non è stata realizzata per mancanza di tempo.

Non ci dilungheremo sulla descrizione "estetica" del programma visto che essa è visibile avviando il tool, piuttosto descriveremo come avviene il manipolamento dei dati.

3.2. Parsing dei dati e manipolazione

Per modellare il diagramma si è scelto di usare un'interfaccia grafica realizzata tramite swing di Java. In ogni frame vi sono dei campi di testo il cui valore verra' poi passato ai listener associati.

I vari listeners (anch'essi nello stesso package dei frames), hanno il compito di prelevare i dati immessi dall'utente e realizzare il disegno sull'interfaccia e chiamare le funzioni di controllo che creano gli oggetti di dominio per poi verificarne la correttezza.

Al momento del parsing dei dati immessi, il sistema esegue dei controlli appropriati quali l'accertamento che una classe con lo stesso nome non esista, o che i dati forniti non provengano da campi lasciati vuoti, ecc. Il tutto è supportato dall'uso di eccezioni che interrompono l'attività non permessa.

Tale eccezione si trova nel package di utilità dell'interfaccia grafica:

`it.uniroma1.dis.mfis.redia.verifint.view.graphics.gui._guiutils.UnspecifiedParameterException;`

3.3. Rappresentazione grafica del diagramma

Il disegno del componente e la cancellazione di esso dal diagramma avviene manipolando il *Set* di oggetti *Drawable* del *RenderingPane*. Questi due oggetti si trovano nel package:

`it.uniroma1.dis.mfis.redia.verifint.view.graphics.engine;`

Per l'approfondimento di questo argomanto rimandiamo all'appendice A in cui è riportato una parte della relazione dei nostri colleghi che hanno realizzato questo package.

Inoltre vanno aggiornati anche le aree di testo dei componenti e del prover formale.

Di seguito riportiamo a titolo di esempio il codice di realizzazione del disegno di una nuova classe che si trova in:

`it.uniroma1.dis.mfis.redia.verifint.view.graphics.gui.frames.classDiagram.ClassListener`

```
private void disegnaClasse(String nome, ArrayList attributi,
          ArrayList operazioni, String alloyIn, String alloyOut)  {

     JTabbedPane jtp = PrincipaleFrame.getJTP();
```

```
        ContentTabPane tabSelected =
             (ContentTabPane)jtp.getSelectedComponent();
        HashSet<Drawable> drawableObjects =
             tabSelected.getDrawableObj();
        RenderingPane renderingPane =
             tabSelected.getRenderingPane();

        Frame classe = new Frame(new Point(20,20));
        Label nomeClasse = new Label(nome);
        nomeClasse.setFont(RenderingPane.BOLD_FONT);
        nomeClasse.setTextAlignment(Label.ALIGNMENT_CENTER);
        classe.getFigure().add(nomeClasse);
        classe.getFigure().add(new Figure());
        Iterator itAttr = attributi.iterator();
        while(itAttr.hasNext()) {
             String[] rowAttributo = (String[])itAttr.next();
             String attributo = rowAttributo[3] + " " +
                  rowAttributo[0] + ": " + rowAttributo[1] + " (" +
                  rowAttributo[2] + ")";
             classe.getFigure().add(new Label(attributo));
        }
        classe.getFigure().add(new Figure());
        Iterator itOper = operazioni.iterator();
        while(itOper.hasNext()) {
             String operazione = (String)itOper.next();
             if(!operazione.equals(""))
                  classe.getFigure().add(new Label(operazione));
        }
        drawableObjects.add(classe);
        renderingPane.setDrawableObjects(drawableObjects);
        tabSelected.setRenderingPane(renderingPane);
        tabSelected.repaint();
        tabSelected.updateComponentsArea(
             ContentTabPane.CLASS_TYPE,nome);
        tabSelected.updateAlloyOutArea(alloyOut);
        tabSelected.updateAlloyInArea(alloyIn);
        frame.dispose();
}
```

3.4. L'oggetto ccdDiagram

La creazione/eliminazione dell'oggetto diagramma (*CCDDiagram*) e degli oggetti componenti (*CCDComponent*), per esempio una classe *CCDClass*, e l'aggiornamento del diagramma corrispondente è invece affidata alle funzioni della classe di controllo *ClassDiagramController* e *Archive* presente nel package:

`it.uniroma1.dis.mfis.redia.verifint.control;`

Queste funzioni sfruttano le classi di controllo e le classi di dominio già sviluppate per realizzare gli oggetti. In particolare fanno uso del pattern Factory per manipolare gli oggetti di dominio. Il compito di queste funzioni è creare gli oggetti che rappresentano le moltiplicità (di classe *Multiplicity*) e i tipi (di classe *CTBaseType*) per poi passarle ai factory e realizzare operazioni, attributi, classi, associaizoni, ecc.

Anche in questo caso riporto la funzione che crea un'oggetto *CCDClass* e lo aggiunge al diagramma *CCDDiagram.*

```
public static void costruisciClasse(CCDDiagram diagram, String
               nome, ArrayList attributi, ArrayList operazioni){

     CCDClass_Factory class_factory = new CCDClass_Factory();
     CCDOperation_Factory op_factory = new
                                   CCDOperation_Factory();
     CCDAttribute_Factory attribute_factory = new
                                   CCDAttribute_Factory();
     CCDAssociation_Factory association_factory = new
                                   CCDAssociation_Factory();
     CCDParameter_Factory parameter_factory = new
                                   CCDParameter_Factory();
     CCDSpecialization_Factory specialization_factory = new
                                   CCDSpecialization_Factory();

     CCDClass classe = class_factory.createInstance(diagram, nome);
     HashMap types = CTBaseType.getBaseTypes();
     Iterator attIt = attributi.iterator();
     while(attIt.hasNext()){//si cercano e creano gli attributi
          String[] row = (String[])attIt.next();
          String tipoStr = row[1];
          CTBaseType type = (CTBaseType)types.get(tipoStr);
          String[] multipRow = row[2].split(",");
          String max = multipRow[1];
          if(max.equals("*")) max = "" + Multiplicity.UNBOUNDED;
          Multiplicity multiplicity = new
                    Multiplicity(Short.parseShort(multipRow[0]),
                    Short.parseShort(max));
          CCDAttribute name_attribute =
                    attribute_factory.createInstance(diagram,
                    classe, type, row[0], multiplicity);
     }
     Iterator opIt = operazioni.iterator();
     while(opIt.hasNext()){//si cercano e creano le operazioni
          String op = (String)opIt.next();
          String params =
               op.substring(op.indexOf("("+1,op.indexOf(")"));

          String nomeOper = op.substring(0,op.indexOf("("));
          String valRit = op.substring(op.indexOf(")")+ 3);
          CTBaseType retType = (CTBaseType)types.get(valRit);
          CCDOperation oper = op_factory.createInstance(diagram,
                              nomeOper, classe);
          if(params.contains(",")){//operazioni con piu' paramet
               Scanner sc = new Scanner(params);
               sc.useDelimiter(", ");
               while(sc.hasNext()){
                    String parametro = sc.next();
                    String nomeparam =  parametro.substring(0,
                                        parametro.indexOf(":"));
                    String tipoparam = parametro.substring(
```

```
                                        parametro.indexOf(":")+2);
                    CTBaseType type =
                              (CTBaseType)types.get(tipoparam);
                    CCDParameter operparameter =
                              parameter_factory.createInstance(
                              diagram, oper, type, nomeparam);
               }
          }
          else {
               String parametro = params;
               if(params.contains(":")){// operazioni con 1 param
                    String nomeparam = parametro.substring(0,
                                        parametro.indexOf(":"));
                    String tipoparam =
                    parametro.substring(parametro.indexOf(":")+2);
                    CTBaseType type =
                         (CTBaseType)types.get(tipoparam);
                    CCDParameter operparameter =
                              parameter_factory.createInstance(
                              diagram, oper, type, nomeparam);
               }
          }
          if(retType!= null)
               CTType_isTypeOf_CTTypeDefined.addLink(retType,oper);
     }
}
```

3.5. Diagramma in XML

L'oggetto che rappresenta il diagramma viene ora tradotto in linguaggio xml per rendere il più possibile disaccoppiato e modulare il lavoro di integrazione di diversi provers per la verifica formale senza dover riprogettare tutto lo strato intermedio di traduzione.

Anche questa funzione era già stata sviluppata e a noi è spettato il compito di realizzare il foglio xsl per la traduzione in linguaggio alloy e l'integrazione di questo prover col resto dell'applicazione.

3.6. Realizzazione alloy4.xsl

Prima di descrivere il modo in cui è stato creato il nostro foglio di stile (da cui nasce l'input per il prover), viene esposta una breve introduzione a Alloy e XSL (ossia i protagonisti di questa fase).

Alloy è un tool per la verifica formale che si basa sulla creazione di modelli al fine di determinare la soddisfacibilita' di formule scritte in FOL (first order logic).

Vista la sua sintassi flessibile molto vicina ai linguaggi OO Alloy risulta molto efficiente per la verifica formale di un diagramma delle classi.Una volta “tradotto” da Uml a linguaggio Alloy, si fa con il model checking delle formule risultanti.Questo model checking viene realizzato mediante la verifica del modello, soddisfacendo una specifica formale.

Xsl (eXtensible Stylesheet Language)è il linguaggio di descrizione dei fogli di stile per i documenti in formato XML. Come è noto, lo standard XML prevede che i contenuti di un documento siano separati dalla formattazione della pagina in cui verranno pubblicati. D'altra parte è proprio questa distinzione a costituire uno dei punti di forza dell'XML come metalinguaggio, in quanto massimizza la possibilità di associare molti e diversi linguaggi di marcatura agli elementi del documento, arricchendone le proprietà semantiche.

Sfruttando queste proprietà di XML e XSL siamo riusciti a “tradurre” un diagramma delle classi in una formula in ligguaggio corretto da dare in input ad Alloy4.
Come gia' detto, il passaggio da “diagramma” a formula Alloy non è immediato, ma vi è appunto uno stato intermedio in cui dal disegno del diagramma, si ottiene un XML che lo rappresenta. Questo capitolo discute su come è stato realizzato il foglio di stile alloy4.xsl per la “traduzione” dal formato xml in cui è rappresentato il diagramma in linguaggio alloy da passare allo stesso prover. Xsl è nato proprio per dare uno stile a file XML. Infatti per la gestione dei vari tag di XML, XSL fa uso di template che vengono richiamati ogni qualvolta si trova il tag desiderato (ad esempio ogni volta che trovo un tag <classe>...</classe> dico al foglio di stile di gestirlo in un dato modo). Inoltre si possono creare template ad hoc per la creazione di funzioni che non sono discriminate soltanto da un tag.

Di seguito espongo come sono stati trattati i singoli elementi del diagramma.

a) Classi

Le classi sono rappresentate come predicati unari in FOL (detti anche atomi), in cui per la realizzazione delle stesse si usa la notazione :

sig <nome-classe> <eventuale estensione>

{

<attributi>

<operazioni>

}

Attributi e operazioni sono stati gestiti in modo banale, associando un template ad ogni tag corrispondente. Un problema è stato riscontrato nella rappresentazione dei tipi (degli attributi, dei parametri e dei valori di ritorno delle operazioni).I tipi non hanno attributi e operazioni quindi hanno una rappresentazione standard (ossia “sig” dal body vuoto).

Purtroppo ci sono stati dei problemi dovuti alla molteplice comparizione dei tipi, il che creava errori di sintassi in Alloy dovuti al fatto che le classi rappresentati i tipi venivano ripetute piu' volte (ossia ogni volta che un attributo o un'operazione usavano un tipo già usato). Ho risolto usando un controllo grazie al quale un tipo (cioe' la sig <nometipo> {}) viene inserito una sola volta nella nostra formula quando ha almeno una occorrenze nel diagramma.

b) Associazioni

Le associazioni in Alloy vengono gestite in modo molto simile a quello degli attributi, con la differenza che invece dei tipi viene associata la classe con cui la “sig” di riferimento è associata. Inoltre per ogni associazione c'è bisogno di un “fact” (ossia un vincolo globale del modello, che deve essere sempre rispettato) in cui viene formalizzata la relazione tra le classi coinvolte mediante una formula del tipo:

$$\forall XY\ class(X) \wedge class(Y) \rightarrow association(X, Y)$$

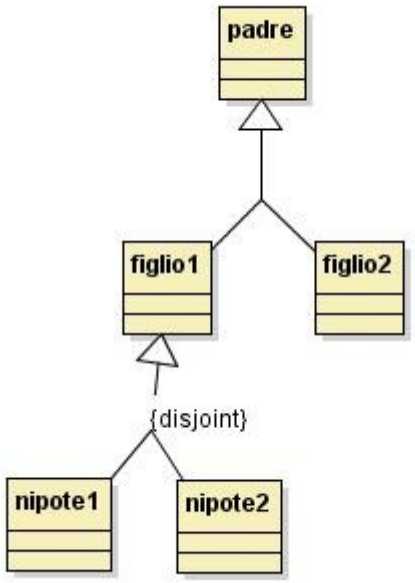
percio' una associazione viene tradotta in alloy secondo questa specifica:

sig classe1{
associazione : [cardinalita'] classe2
}

```
fact vincoliassociazione{
all c1:classe1 ,c2:classe2 |
(c1 in c2.associazione) && (c2 in c1.associazione)
}
```

c) Isa

La gestione degli Isa è stata la parte più problematica in quanto alloy non gestisce i casi di eredita` multipla (cioè quando una classe è figlia di più padri).
Abbiamo ovviato a ciò introducento un fact in cui si vincola che la classe figlia è un sottoinsieme della classe padre (per tutte le classi padri tranne la prima la quale è estesa come una semplice isa.
Un altro problema è stato riscontrato quando si ha una gerarchia del tipo in figura:



Per notazione quando due classi sono istanza di una superclasse e non sono disgiunte, viene scritto in questo modo:

```
sig figlio1 in padre{}
sig figlio2 in padre{}
```

Quando invece le classi sono disgiunte invece di “in” si usa “extends”. In una situazione come quella rappresentata in figura Alloy darebbe errore di sintassi in quando la notazione “extends” può essere usata solo per segnature “top level”.
Per ovviare a questa situazione, ogni qualvolta vi è un isa disjoint in cui la classe padre non è “top level” si gestisce come se fosse disjoint con le sorelle, (usando quindi la notazione “in” invece di “extends”) quindi aggiunti dei fact in cui viene vincolata l'intersezione vuota tra le suddette classi figlie di classi non di primo livello:

```
sig nipote in figlio1{}
fact disjointfigli{
all ni:nipote1 | (ni in nipote1 ) <=> ( (ni not in nipote2) )
}
```

d) Prova finale

Il nostro prover verifica l'inconsistenza delle classi inserite nel nostro diagramma. Ogni qualvolta viene inserita una classe insoddisfacibile appare un messaggio di errore. Per effettuare questa verifica Alloy usa una asserzione cioe' la dichiarazione di un vincolo che dovrebbe essere ridondante poiché implicato dai vincoli del modello; l'asserzione è una "sfida" al tool analizzatore a trovare un controesempio, entro uno scope finito, che violi il vincolo.

Per ogni classe viene quindi richiamata la seguente asserzione:

assert showclasse{

no classe} check showclasse for <numerodefinito>

3.7. Il prover Alloy4

L'ultima fase del lavoro consiste nell'integrazione del tool di verifica formale Alloy4.

Dopo alcuni tentativi di mantenere il file jar di Alloy4 esterno a Rediaverifint quindi richiamarlo dalla nostra applicazione come un qualunque programma esterno con cui cooperare, ci siamo accorti che il modo in cui veniva gestito il processo di esecuzione, passaggio di input e prelievo dell'output realizzato precedentemente non era adeguato alle nostre esigenze anche per via del diverso funzionamento del prover, e così è stato deciso di sfruttare la caratteristica open source di Alloy4.

In questo modo è stato semplice integrarlo, e renderlo più adatto al nostro scopo.

Infatti alloy4 presenta un'interfaccia grafica già di per sè e così si è scelto di realizzare un ponte ad hoc per far sì che sia privato di interfaccia grafica e restituisca il risultato della prova formale sottoforma di stringa e allegato all'oggetto *ProvingResult* di restituzione.

La classe ponte è *Alloy4External* e si trova nel package:

`edu.mit.csail.sdg.alloy4whole;`

La funzione che lancia il programma prende come parametro il nome del file con estensione .als che contiene la rappresentazione del diagramma in linguaggio alloy e le asserzioni da dimostrare.

Questa funzione e la creazione del file .als viene gestita completamente della classe *CCDAlloy4Prover* che si trova in:

`it.uniroma1.dis.mfis.redia.verifint.control.provers.analysis.classDiagram;`

ed estende *Prover*. Di questo però ridefinisce la funzione astratta *prove(String goal)* come una funzione che non fa nulla in quanto non è adeguata al nostro scopo. Il goal infatti, ossia le asserzioni da dimostrare, vengono fornite in input al prover Alloy4 all'interno del file .als, rendendo inutile così tale funzionalità.

La funzione seguente, tramite il metodo *marshal(CCDDiagram)*, trasforma il diagramma del tipo "ccdDiagram" in una stringa formato XML. Successivamente scrive sul file chiamato "nomediagramma.als", localizzato nella directory "**`tmp\`**", il risultato della traduzione della stringa XML in linguaggio alloy.

Questa conversione viene realizzata dalla funzione *createProverProgramInput(...)* che a sua volta usa il foglio di stile alloy4.xsl che si trova nella directory:**`resources\xml\xslt\ccd\`**

Alla fine viene lanciato il prover richiamando la funzione locata nella classe ponte di cui abbiamo parlato prima.

Le funzioni *marshal* e *createProverProgramInput* erano gia' stata implementate precedentemente per fare una verifica formale mediante "Prover9".

Di seguito riporto il codice che esegue la prova formale e restituisce l'oggetto *ProvingResult* con il risultato.

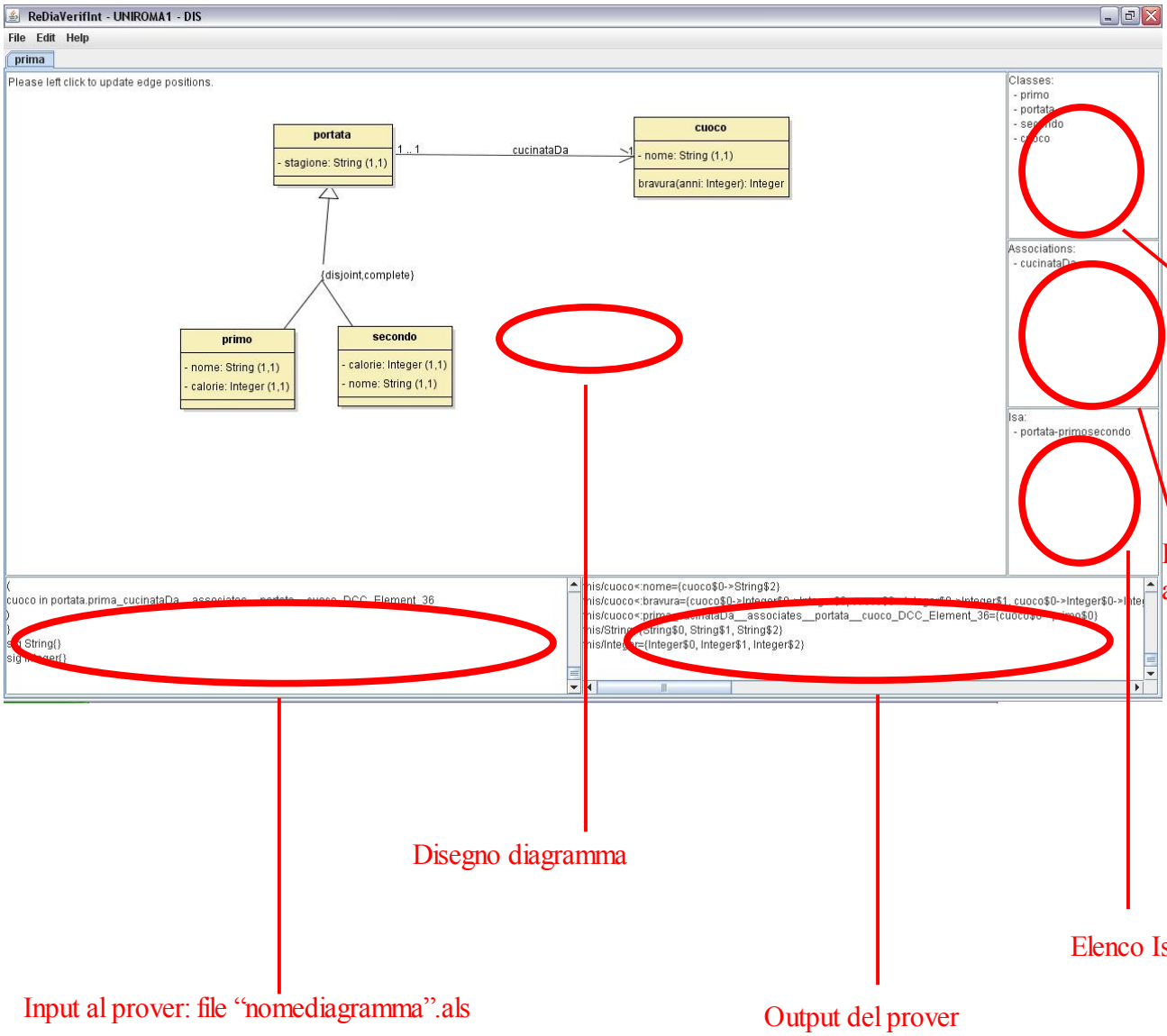
```
public ProvingResult proveAlloy4(String nomeDiagramma) throws Err{
     if (this.getXsltPath().equals(EMPTY_XSLT_PATH)) {
          throw new IllegalStateException(
               EXECUTION_PROPERTIES_NOT_SET_EXCEPTION_MESSAGE);
     }

     String diagramXml = new
               XmlMarshaller().marshal((CCDDiagram)this.diagram);
     String nomeFile = nomeDiagramma + ".als";
     PrintWriter pw = null;
     String nome_file_tmp =
               ExecutionProperties.getInstance().getProperty(
               Alloy4Constants.ALLOY_4_TMP_PATH_KEY) + nomeFile;

     try {
          FileWriter fw = new FileWriter(nome_file_tmp);
          pw = new PrintWriter(fw);

          pw.println(super.createProverProgramInput(diagramXml));
          pw.close();
          fw.close();
     }
     catch(Exception e) {
          throw new ApplicationException(e.getMessage());
     }

     String proverInput = nome_file_tmp;
     return Alloy4External.startAlloy4(proverInput);
}
```

Il risultato viene ora riportato nei relativi pannelli del tab del diagramma corrispondente così che possa essere consultato dall'utente sia nel caso in cui il diagramma risulti soddisfacibile sia nel caso non risulti soddisfacibile. In quest'ultimo caso, tuttavia, viene avvertito l'utente tramite un messaggio popup.

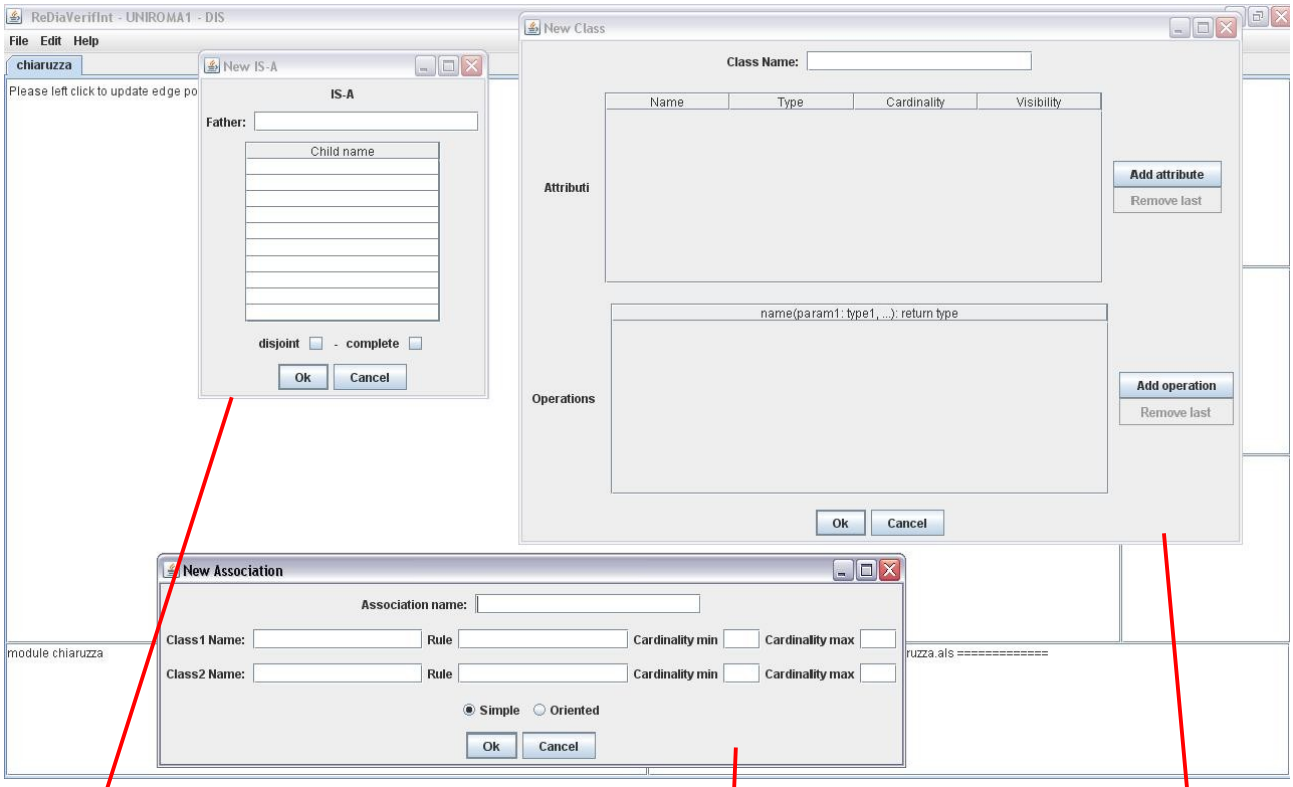
4. Screenshoot dell'applicazione

Di seguito sono riportati degli screenshot di Rediaverifint.

- Visione generale dell'interfaccia grafica:



- Inserimento entita':



Frame per l'inserimento di un ISA

Frame per la creazione di un'associazione

Frame per costruire una classe

5. Conclusioni

Per la realizzazione di questa tesina sono stati spesi all'incirca 5 mesi uomo. La parte che ha richiesto piu' tempo e' stata quella riguardante la realizzazione dell'interfaccia grafica. Per quanto riguarda il file “alloy4.xsl” abbiamo riscontrato difficolta' iniziali essendo la prima volta che ci approcciavamo a questo tipo di linguaggio.

Grazie a questo lavoro abbiamo aumentato le nostre capacità di programmatori, abbiamo imparato come si estende un CASE in via di sviluppo e abbiamo cercato di rendere il prodotto il piu' trasparente possibile per un futuro ampliamento/miglioramento.

In fine abbiamo capito a nostre spese che la fase di test deve avere molta piu' importanza di quanta ne eravamo abituati a dare.