Università di Roma "La Sapienza"

Tesina per il Corso di Metodi Formali nell'Ingegneria del Software
Prof. Toni Mancini

Anno 2006-2007

Alloy e l'Alloy Analyzer vers. 4.0

Autore:
Alessandro Pagliaro

Al Prof. Marco Cadoli,
che con i suoi insegnamenti ha contribuito alla crescita,
tanto professionale quanto umana.
Il Suo ricordo sarà in me sempre vivo.

Sommario

Introduzione..5

Parte I: Nozioni Generali su Alloy e
l'Alloy Analyzer 7

Parte II: Sintassi e Semantica di Alloy 9
Sezione 2.1 - Espressioni...9
Sezione 2.2 - Formule...10
Sezione 2.3 - Quantificatori...10
Sezione 2.4 - Moduli...11
Sezione 2.5 - Segnature...12
Sezione 2.6 - Fatti...13
Sezione 2.7 - Predicati...13
Sezione 2.8 - Funzioni...14
Sezione 2.9 - Asserzioni..14
Sezione 2.10 - Comandi...14

Parte III: Uso dell'Alloy Analyzer....................................16
Sezione 3.1 - La Schermata Principale...16
Sezione 3.2 - Le 3 Modalità di Visualizzazione dei Risultati.........17
Sezione 3.3 - Opzioni dell'Alloy Analyzer....................................19

Parte IV: Modellazione in Alloy di elementi
degli UML Class Diagrams......................................21
Sezione 4.1 - Classi UML...21
Sezione 4.2 - Associazioni e loro Gerarchie...................................26
Sezione 4.3 - Specializzazione di Attributi e Associazioni.............32
Sezione 4.4 - Cenni sulla Traduzione di Vincoli Esterni................36

Parte V: Analisi di Confronto tra Alloy ed Otter/Mace
nel Contesto della Verifica di Proprietà degli
UML Diagrams...38

Sezione 5.1 - Overview delle Tipologie di Verifica.........................38
Sezione 5.2 - Verification Case Studies: Otter/Mace vs Alloy........39
Sezione 5.3 - Beyond Otter: la [Reflexive] Transitive Closure........46

Conclusioni..50

Bibliografia..51

Introduzione

L'introduzione di formalismi e metodologie come UML nelle "fasi alte" nello sviluppo di prodotti software ha contribuito enormemente alla realizzazione di software di qualità, in quanto formalizzare i requisiti significa esplorarli a fondo non lasciando nulla al caso, in modo da trovare e correggere errori il prima possibile evitando almeno in parte gli ingenti costi delle modifiche caratteristici delle fasi avanzate nel ciclo di vita del prodotto.

Il Corso Universitario di Metodi Formali nell'Ingegneria del Software si prefigge in questo scenario di fornire un ulteriore ausilio per mirare alla correttezza delle formalizzazioni dei requisiti nella fase di Analisi, ausilio che si concretizza nel fornire agli studenti un background nei settori della Logica Proposizionale, Logica del Primo Ordine e Logica Temporale Lineare, e quindi istruirli nell'uso di vari tools con lo scopo di tradurre specifiche software in logica ed esaminare tali formalizzazioni con gli strumenti informatici a disposizione, il tutto per dare un ulteriore contributo alla correttezza e completezza finale dei programmi.

Questa relazione è uno dei frutti di tale corso, e mira ad illustrare le funzionalità e le potenzialità del linguaggio Alloy, sviluppato nel 2000 dall'M.I.T. per trattare Logica del Primo Ordine (a cui d'ora in avanti ci si riferirà per comodità come FOL, acronimo di First Order Logic), nonché del tool associato, chiamato Alloy Analyzer, nell'ambito della parte del corso dedicata alla modellazione di aspetti statici dei programmi tramite UML Class Diagrams.

La versione di Alloy utilizzata al momento della redazione di questa relazione è la 4.0.

L'argomento verrà affrontato in cinque sezioni, con i seguenti obiettivi:

La sezione I fornirà una concisa e sintetica overview generale delle potenzialità e caratteristiche del linguaggio Alloy e dei possibili usi permessi dal tool associato, l'Alloy Analyzer.

La sezione II affronterà in modo dettagliato la sintassi e la semantica di Alloy, esplorando tutte le più importanti strutture del linguaggio illustrando per ognuna sintassi e semantica;

La parte III consisterà in un breve excursus riguardo all'uso e alle caratteristiche del tool Alloy Analyzer;

La sezione IV ospiterà una reinterpretazione delle metodologie viste durante il corso per la formalizzazione in FOL degli UML Class Diagrams, illustrando il passaggio da UML ad Alloy con particoare attenzione alle metodologie di traduzione in FOL viste al corso di Metodi Formali;

La sezione V passerà in rassegna gli scenari di verifica di proprietà degli elementi discussi nella sezione IV, analizzando il confronto tra l'uso di Alloy e dei tool esaminati a lezione, Otter e Mace, con riferimento ad espressività ed efficacia dei rispettivi linguaggi, nonché alle performances in termini di tempo d'esecuzione.

Seguirà infine una breve sezione riepilogativa e conclusiva.

Parte I: Nozioni generali su Alloy e l'Alloy Analyzer

Alloy è un linguaggio, piuttosto semplice, basato sulla Logica del Primo Ordine e creato allo scopo di assistere la progettazione di una grande varietà di sistemi software.
Si tratta di un linguaggio *dichiarativo*, pertanto particolarmente adatto per modellare le proprietà statiche di componenti nelle specifiche di Software Design come ad esempio i Diagrammi UML.
Per l'analisi dei modelli Alloy trasforma il codice relativo in un'istanza di SAT e usa uno tra vari SAT-solvers disponibili per effettuare l'analisi.

In Alloy tutto si fonda sui concetti di Atomo e Relazione: sebbene il linguaggio sia dotato di una certa ricchezza di operatori e le sue espressioni possono sembrare vicine sia ai linguaggi Object-Oriented sia alla teoria degli insiemi, e persino alle queries delle basi di dati, tutto si riconduce, behind the scenes, ad atomi in relazione tra loro.
Ogni *Atomo* in Alloy è un'entità primaria indivisibile e immutabile, in quanto non può essere diviso in parti e le sue proprietà non variano nel tempo.
Una *Relazione* consiste in un insieme di tuple ordinate di una arità maggiore o uguale ad uno che stabiliscono un legame tra Atomi. Le tuple non possono mettere in relazione tra loro altre relazioni poiché Alloy non tratta la Logica del Secondo Ordine.
Il linguaggio non prevede inoltre la nozione esplicita di valore scalare o di insieme (anche chiamato *set*): un set è semplicemente una relazione unaria, mentre uno scalare è una relazione unaria con una sola istanza.

Come già accennato, sebbene la vera semantica del linguaggio si fondi sui concetti base di Atomo e Relazione, la sintassi di Alloy è stata ingegnosamente resa molto vicina alla teoria degli insiemi (Set Theory) ed ai linguaggi OO: ad esempio la dichiarazione di un atomo può essere vista come la dichiarazione di una classe, il linguaggio possiede built-in il concetto di ereditarietà, e le relazioni alle quali un certo atomo partecipa possono essere facilmente viste come gli "attributi della classe"; si possono comporre relazioni in modo da simulare l'accesso ai campi della classe; è possibile inoltre utilizzare tutti i tradizionali operatori FOL (quantificatori, and, or, not, implies, iff) e gli operatori insiemistici (intersezione, differenza, unione), nonché una serie di altri operatori di vario uso (costrutto if-then-else, join, chiusura transitiva e riflessivo-transitiva...).
Alloy fornisce anche supporto per l'espressione di valori interi, il che rende particolarmente agevoli alcune comuni operazioni che risultano piuttosto articolate in FOL.
Per la trattazione dettagliata e corredata di esempi dei vari aspetti della sintassi e semantica di Alloy si rimanda alla sezione seguente.

Come possiamo usare Alloy per i nostri scopi di Ingegneri del Software?
L'idea è di creare, utilizzando il tool Alloy Analyzer, un modello di un sistema a partire da determinate specifiche, definendo gli atomi del modello e tutte le loro proprietà (cioè le loro relazioni), esprimendo inoltre tutti i vincoli globali che devono essere sempre rispettati nel modello.
A seconda della struttura che abbiamo creato sarà possibile l'esistenza di varie istanze che soddisfano tutti i vincoli del nostro modello; a questo punto possiamo sfruttare l'Alloy Analyzer per cercare istanze del nostro modello oppure sfidarlo a trovare un controesempio che smentisce determinate nostre asserzioni riguardo al modello creato, nei limiti di uno scope che caratterizza

il massimo numero di atomi di tipo top-level (non subsets di altri insiemi di atomi) che vogliamo utilizzare.
Una volta conclusa l'analisi, l'interfaccia grafica del programma mette a disposizione dell'utente diverse modalità di visualizzazione dei risultati, le quali possono anche essere customizzate. Per maggiori dettagli sull'Analyzer si rimanda alla sezione III.

Parte II: Sintassi e Semantica di Alloy

In questa sezione affronteremo uno per uno i costrutti più importanti che formano il linguaggio Alloy, dando indicazioni sul loro significato (anche alla luce delle brevi considerazioni fatte nella sezione I) e sul loro uso; questa parte può anche essere vista come una sintetica guida all'uso del linguaggio per chi volesse imparare ad utilizzarlo per i propri scopi.
Gli aspetti di Alloy che affronteremo sono:

- Espressioni (Expressions);
- Formule (Formulas);
- Quantificatori (Quantifiers);
- Moduli (Modules);
- Segnature (Signatures);
- Fatti (Facts);
- Predicati (Predicates);
- Funzioni (Functions);
- Asserzioni (Assertions);
- Comandi (Commands): Check e Run.

Sezione 2.1 - Espressioni (Expressions)

Esistono in Alloy due tipi di espressioni: Espressioni Relazionali (che denotano una relazione) ed Espressioni Intere. Le espressioni sono i blocchi primitivi fondamentali utilizzati da tutti gli altri costrutti.
Nel seguito verranno esaminati gli operatori e i tipi di espressione più comuni e importanti.

Un'espressione può essere semplicemente un nome di variabile, un riferimento ad una Signature (a cui per ora possiamo intuitivamente pensare come ad un predicato unario che individua un insieme di atomi), una costante relazionale come *none* (l'insieme vuoto) o *univ* (l'insieme di tutti gli elementi), oppure la composizione di espressioni mediante operatori unari o binari.
Gli operatori unari comprendono il "transpose" (**~expr**) che individua la relazione inversa di una relazione binaria, la chiusura transitiva (**^expr**) e la chiusura riflessivo-transitiva (***expr**), delle quali si parlerà più in dettaglio nella sezione V.
Tra gli operatori binari sono presenti l'unione insiemistica (**expr1 + expr2**), la differenza (**expr1 - expr2**) e l'intersezione (**expr1 & expr2**).

Di uso molto comune è anche il join di due espressioni relazionali (**expr1.expr2**) per ottenere la relazione che contiene tutte le combinazioni delle tuple di expr1 ed expr2, e, se l'ultimo elemento della prima corrisponde al primo elemento della seconda, omettendo gli elementi corrispondenti.
Un esempio può essere di notevole aiuto per chiarire l'uso dell'operatore: supponiamo di avere un insieme di atomi di tipo Persona (ovvero appartenenti ad una Signature Persona, per maggiori dettagli si veda la sezione 2.5), ognuno dei quali è in relazione (il nome della relazione è Cognome) con un altro atomo, di tipo Stringa (appartenente alla signature Stringa).
Sia p una variabile denotante un qualsiasi atomo Persona; l'espressione *p.Cognome* individua l'insieme degli atomi Stringa che condividono con tutti gli atomi denotati da p la relazione Cognome.

Il prodotto di due relazioni (**expr1 -> expr2**), che esprime la relazione ottenuta prendendo una qualsiasi combinazione di tuple tra prima e seconda relazione ed includendo la loro concatenazione.
Un'espressione può inoltre essere di tipo if-then-else (**formula => expr1 else expr2**), cioè l'espressione valore expr1 se formula è vera, expr2 altrimenti.

Alloy permette anche l'uso di valori interi e pertanto permette la nozione intuitiva di espressione intera, appplicando un overloading di operatori (+ e -) a seconda che l'espressione in questione sia intera o relazionale. Gli operatori interi comprendono la somma (**i + j**), la differenza (**i - j**), nonché operatori di confronto come minore (**i < j**), maggiore (**i > j**), uguale (**i = j**), minore o uguale (**i =< j**) e maggiore o uguale (**i >= j**).

Le espressioni possono essere liberamente parentetizzate; data la vastità dell'insieme di operatori disponibili, Alloy impiega una precisa policy di precedenze di cui non parleremo perché non molto rilevante nell'ambito di questo lavoro. Ogni esempio di espressione in Alloy fornito in futuro sarà doviziosamente corredato di parentesi.

Sezione 2.2 - Formule (Formulas)

Per Formula si intende l'applicazione di quantificatori ad espressioni relazionali, o la comparazione di espressioni relazionali o intere, in modo che la formula abbia valore true o false.
I quantificatori disponibili in Alloy verranno esaminati approfonditamente nella prossima sezione (la 2.3); il significato di una formula quantificata coincide con la semantica tradizionale in FOL.
Operatori di comparazione tra formule molto importanti sono l'uguaglianza (**exprRel1 = exprRel2**), intesa in tal caso come uguaglianza estensionale di relazioni, ovvero il contenere le stesse tuple, nonché l'operatore di subset (**exprRel1 in exprRel2**), che corrisponde alla nozione di sottoinsieme.
Le formule possono essere negazioni di altre formule (**!formula** oppure **not formula**), oppure combinazioni di formule tramite i seguenti operatori: l'and (**f1 && f2** oppure **f1 and f2**), l'or (**f1 || f2** oppure **f1 or f2**), il se e solo se (**f1 iff f2** oppure **f1 <=> f2**) e l'implicazione (**f1 => f2** oppure **f1 implies f2**); la semantica è ovviamente quella tradizionale degli opratori in FOL.
Formule divise da spazi equivalgono ad un and delle formule (**f1 f2 f3** significa **f1 && f2 && f3**).
Anche per le formule vale il discorso sulle parentesi fatto nella sezione precedente: gli esempi forniti saranno adeguatamente corredati di parentesi.

Sezione 2.3 - Quantificatori (Quantifiers)

L'uso di quantificatori abbinati a formule nel linguaggio in esame ha questa sintassi:
quantifier varName: sigName | formula
con il seguente significato intuitivo: relativamente ad un certo numero di atomi di tipo sigName, numero influenzato dal quantificatore, vale questa formula, nella quale ci riferiamo ad uno qualunque di tali atomi con il nome varName.
Alloy offre, oltre ai tradizionali quantificatori "per ogni" (denotato come **all**) ed "esiste" (denotato come **some**), una serie di quantificatori "utility" che spesso compaiono nell'analisi di software:

no significa "per nessun atomo vale tale formula"; **one** significa "tale formula vale per uno ed un solo atomo"; **lone** significa "tale formula vale per al più un atomo".

Molto interessante è il parallelismo tra questi quantificatori e le tradizionali molteplicità di attributi ed associazioni UML: ad esempio 'one' denota una molteplicità 1..1, 'some' denota l' 1..*, 'lone' è 0..1; la keyword **set**, applicabile in molteplici situazioni, ci permette di denotare la molteplicità 0..*.

I quantificatori di Alloy possono essere applicati in una grande varietà di casi; ad esempio l'operatore per il prodotto di due relazioni (->) ammette quantificatori sia a destra che a sinistra con la seguente sintassi e semantica:
exprRel1 m -> n exprRel2
Ciò significa che esiste una relazione tra i due insiemi exprRel1 ed exprRel2, regolata dalle molteplicità m ed n; ad esempio Studente one -> one Professore è una biiezione tra atomi Studente e atomi Professore, uno studente per ogni singolo professore, mentre Studente -> some Professore significa "ogni studente ha uno o più professori, sebbene possano esistere professori senza alcuno studente".
In assenza di molteplicità l'operatore '->' equivale a ' set->set ', ovvero non c'è restrizione tra le molteplicità della relazione prodotto.
Nel caso si volesse realizzare il prodotto cartesiano, la relazione mediante l'operatore prodotto deve essere accompagnata da opportuni vincoli di tipo Fact (cfr sez. 2.6).

Sezione 2.4 - Moduli (Modules)

Ogni file di Alloy ospita esattamente un modulo. Possiamo pensare ad un modulo come una sorta di contenitore per le nostre istruzioni dichiarative. In un modulo possiamo importare altri moduli, in modo che il nostro modello Alloy sia formato da più unità distinte, di cui però solo una è la principale o main, e solo essa può essere analizzata tramite i comandi Run e Check (di cui si parlerà in seguito). Tuttavia per i nostri scopi è sufficiente disporre di un singolo modulo, quindi non ci soffermeremo sull'aspetto dell'importazione.

Un modulo Alloy consiste in un *header*, zero o più *dichiarazioni di importazione*, e quindi zero o più *paragrafi*, i quali costituiscono il cuore del modulo.
Un header è semplicemente la dichiarazione del path del file Alloy (.als) e del nome del modulo.
Le dichiarazioni di importazioni sono simili agli import di Java o gli include di C, e come accennato servono ad usare costrutti definiti in altri moduli.
I paragrafi rappresentano il vero corpo del modulo; paragrafo è un nome generico che denota i costrutti che affronteremo, uno per uno, nella parte rimanente della sezione II; normalmente un modulo Alloy contiene:

- alcune *dichiarazioni di segnatura* (Signature Declarations), che denotano gli insiemi di atomi su cui lavoreremo;
- alcune *dichiarazioni di fatti* (Fact Declarations), che impongono dei constraints globali che devono essere sempre rispettati nel modulo;
- *dichiarazioni di predicati* (Predicate Declarations), ovvero dei blocchi di constraints che possono assumere un valore true o false a seconda dei parametri dati in input al predicato;
- *dichiarazioni di funzioni* (Function Declarations), simili ai predicati ma denotano un valore generico e non un valore di verità;
- *asserzioni* (Assertions), cioè formule ridondanti intese come "sfida" al tool analizzatore, che deve cercare di smentirle restituendoci un controesempio;

- *comandi* (Check and Run Commands), che avviano l'analisi del nostro modello da parte dell'Alloy Analyzer sulla base di predicati, funzioni e assserzioni.

Sezione 2.5 - Segnature (Signatures)

Per *Segnatura* (o Signature, d'ora in avanti anche abbreviata come *sig*) si intende un set di Atomi riuniti sotto un nome comune e partecipanti a determinate relazioni e constraints, in altre parole un predicato unario soddisfatto da un insieme di atomi.
Tale costrutto è in Alloy di fondamentale importanza; può essere visto come un insieme in Set Theory oppure una Classe Java nei moderni linguaggi di programmazione; già ora, mentre descriveremo sintassi e semantica delle sigs, vedremo come questo concetto si mappa in modo ideale alle classi UML.
Per illustrare in modo semplice ed intuitivo le sigs, si ricorrerà a degli esempi:

- sig Persona {} ; questa dichiarazione ci dice che esiste un insieme di atomi di tipo Persona; *sig* è la keyword per dichiarare le sigs, le parantesi graffe, come vedremo tra pochissimo, sono destinate ad ospitare relazioni a cui le Persone fanno parte; anche solo con questa dichiarazione primitiva potrei affermare "all p: Persona | formula", e cio significherebbe: "per ogni atomo p di tipo Persona vale questa formula (dipendente da p)".

- **one sig Persona {}** ; come l'esempio precendente, ma posso applicare delle molteplicità alla segnatura per affermare che l'insieme Persona contiene uno ed un solo atomo; similmente some sig Persona constringerebbe il nostro modello ad avere almeno una Persona. L'omissione di molteplicità corrisponde a set sig Persona, ovvero nessun constraint (0..*).

- **sig Persona { nome: Stringa, tel: set Stringa }**
sig Stringa {} ; in questo esempio abbiamo definito due sigs: una semplice sig Stringa, che possiamo vedere come un tipo base, ed una Persona; le dichiarazioni di relazioni all'interno di Persona ci dicono che ogni persona ha un nome, il quale è una stringa, ed ogni persona può avere zero o più stringhe che rappresentano numeri di telefono; in modo più vicino ad Alloy, possiamo affermare che ogni atomo di tipo Persona partecipa ad una relazione chiamata nome con un atomo di tipo Stringa, ed anche ad un numero arbitrario (zero o più) di relazioni di tipo tel con atomi di tipo Stringa.

- **sig superClass {}**
sig subClassA, subClassB extends superClass {} ; questo esempio ci illustra come Alloy tratta il concetto di ereditarietà e gerarchie; abbiamo tre sigs, super, subA e subB (dichiarate insieme per comodità, visto che il loro body è identico), delle quali subA e subB sono anche parte dell'insieme di atomi di super, ed erediterebbero relazioni di super, se questa ne avesse.
Notare che Alloy con 'extends' *impone la disjointness delle sottoclassi, ma non la completeness*, che deve essere definita a parte.
Per evitare anche la disjointness possiamo dichiarare **sig super {} sig subA, subB in super {}**.

- **sig Persona { tel: set Stringa }**
sig Manager extends Persona {} { #tel >= 2 }
sig Stringa {} ; in quest'ultimo esempio copriamo l'argomento degli *appended facts*. Per appended facts si intende un insieme di constraints (formule) all'interno di un body che segue il campo per le relazioni della sig; nell'esempio viene usato l'operatore # (**#exprRel)**, che denota il numero di tuple all'interno della relazione exprRel. Stiamo cioè dicendo che esistono Stringhe

e Persone, le quali hanno zero o più telefoni, ed esistono Managers, che sono Persone ed hanno dei numeri di telefono come le persone, ma *inoltre* devono avere almeno due numeri di telefono (cioè devono partecipare alla relazione tel con almeno due oggetti di tipo stringa).
Notare che un appended fact equivale ad un *fact constraint* che vale unicamente per tutti i membri della propria sig (cfr. sez 2.6).

Sezione 2.6 - Fatti (Facts)

In Alloy i *Fatti* sono delle formule che impongono dei constraints globali su tutto il modello: quelle formule saranno *sempre* verificate in ogni istanza del modello restituita dall'Alloy Analyzer.
Se ad esempio volessimo imporre che le Persone si partizionano in Operosi e Nullafacenti, potremmo dire:

sig Persona {}
sig Operoso extends Persona {}
sig Nullafacente extends Persona {} //nota: Operosi e Nullafacenti sono già disjoint

fact diligencePartition { Operoso + Nullafacente = Persona }

Abbiamo cioè affermato che esistono persone, nullafacenti e operosi (che sono tra loro disjoint poiché stiamo usando *extends* e non *in*), che nullafacenti e operosi sono persone, e che l'insieme degli operosi unito con l'insieme dei nullafacenti è uguale all'insieme delle persone. Questo è un efficace esempio per constatare come Alloy sia anche vicino alla Set Theory.
Il nostro fact proibisce pertanto all'Alloy Analyzer di considerare un modello in cui ci sia una persona che non sia né operosa, né nullafacente.
Come ultima osservazione sui facts, notiamo che essendo dei dogmi che devono sempre essere rispettati sono le principali fonti per il problema dell'*Overconstraining*, cioè imporre troppi vincoli che eliminano istanze del modello che desideremmo essere possibili. In genere può essere difficile individuare la fonte di un caso di overconstraining, soprattutto in modelli con molti facts e appended facts concorrenti.

Sezione 2.7 - Predicati (Predicates)

I Predicati sono costrutti costituiti da un nome per il predicato, una serie di parametri di input e delle formule (dei constraints) che mirano a fornire al predicato un valore booleano, quindi true o false, a seconda del valore dei parametri di input; i predicati possono pertanto essere visti anche come dei "template constraints", dipendenti dai valori immessi in input da chi li invoca.
Un predicato restituisce true se i valori passati al predicato soddisfano i vincoli dichiarati.
- pred predName [param1, ... paramN] { list of constraints on param1 ... paramN }

Qual è l'utilità di tale struttura in Alloy? In genere i predicati sono utilizzati nella risoluzione di particolari istanze di problemi: ad esempio parte del tutorial online su Alloy è dedicato ad un modello che definisce il noto problema del River Crossing e sfrutta un predicato per definire le transizioni di stato legali nel problema, ovvero il predicato accetta come input lo stato corrente e lo stato next di ognuna delle due sponde e stabilisce cosa può succedere durante un generico crossing; un apposito fact stabilisce le transizioni di stato legali invocando il predicato e

costringendo il modello a rispettare il predicato. Tale uso dei predicati, purché interessante, esula dai nostri scopi di modellazione statica dei programmi e pertanto non ce ne occuperemo.

Tuttavia può essere utile considerare un predicato "statico", senza parametri in input e con dei constraints non invasivi, per chiedere all'Alloy Analyzer di trovare un'istanza legale del nostro modello che soddisfi i nostri constraint aggiuntivi; un esempio è d'obbligo per chiarire il punto: supponiamo di avere un modello con due sigs, Persona e Banca, legati da un complesso insieme di vincoli, e vogliamo verificare se il nostro modello accetta un numero di istanze di Persona pari a 0 insieme ad una o più banche. Possiamo allora scrivere:
pred noPersoneMaBanche [] { #Persona = 0 && #Banca = 2 //nota: '#' indica la cardinalità }
e quindi usare un comando run (che tratteremo in seguito) per chiedere un'istanza del modello che soddisfa il vincolo, entro un certo scope finito: **run noPersoneMaBanche for 5** e quindi analizzare i risultati ottenuti.

Sezione 2.8 - Funzioni (Functions)

In Alloy le funzioni sono simili ai predicati, sia come forma che come utilizzo e significato; l'unica differenza è che non restituiscono un valore di verità ma un valore generico, ad esempio un insieme di tuple. La sintassi per le funzioni è di questo genere (da notare il tipo del valore di ritorno):
fun functionName [param1, param2 ... paramN]: returnDomain
{ body che individua un valore in Domain }

Come per i predicati, non abbiamo uno stretto bisogno delle funzioni per i nostri scopi di modellazione software. Anche le funzioni, come i predicati, possono essere invocati da altre formule e possono essere mandati in esecuzione tramite run commands.

Sezione 2.9 - Asserzioni (Assertions)

Le Asserzioni sono costituite da un nome di asserzione e da un body, che contiene una lista di formule di constraint; attenzione, però, a non confondere le asserzioni con i facts: le asserzioni sono intese come vincoli ridondanti che il nostro modello dovrebbe già rispettare in base a come l'abbiamo definito; scrivendo un'Assertion noi stiamo "sfidando" l'Alloy Analyzer a dimostrare che i nostri vincoli hanno una falla da qualche parte e la nostra asserzione non è vera, in altre parole l'analizzatore cerca un controesempio, entro uno scope, per smentire le nostre affermazioni.
Se l'Analyzer non trova un controesempio entro lo scope specificato allora *non esiste* un controesempio entro quello scope, e ci viene riferito che l'assertion potrebbe essere vera per ogni scope, altrimenti ci viene fornito un controesempio che possiamo analizzare per scovare problemi di over- o underconstraining.
La sintassi è molto semplice:
assert assertionName { list of "challenge constraints" }
A differenza di predicati e funzioni, le assertions usano il comando check anziché run.
check assertionName for X

Sezione 2.10 - Comandi (Check and Run Commands)

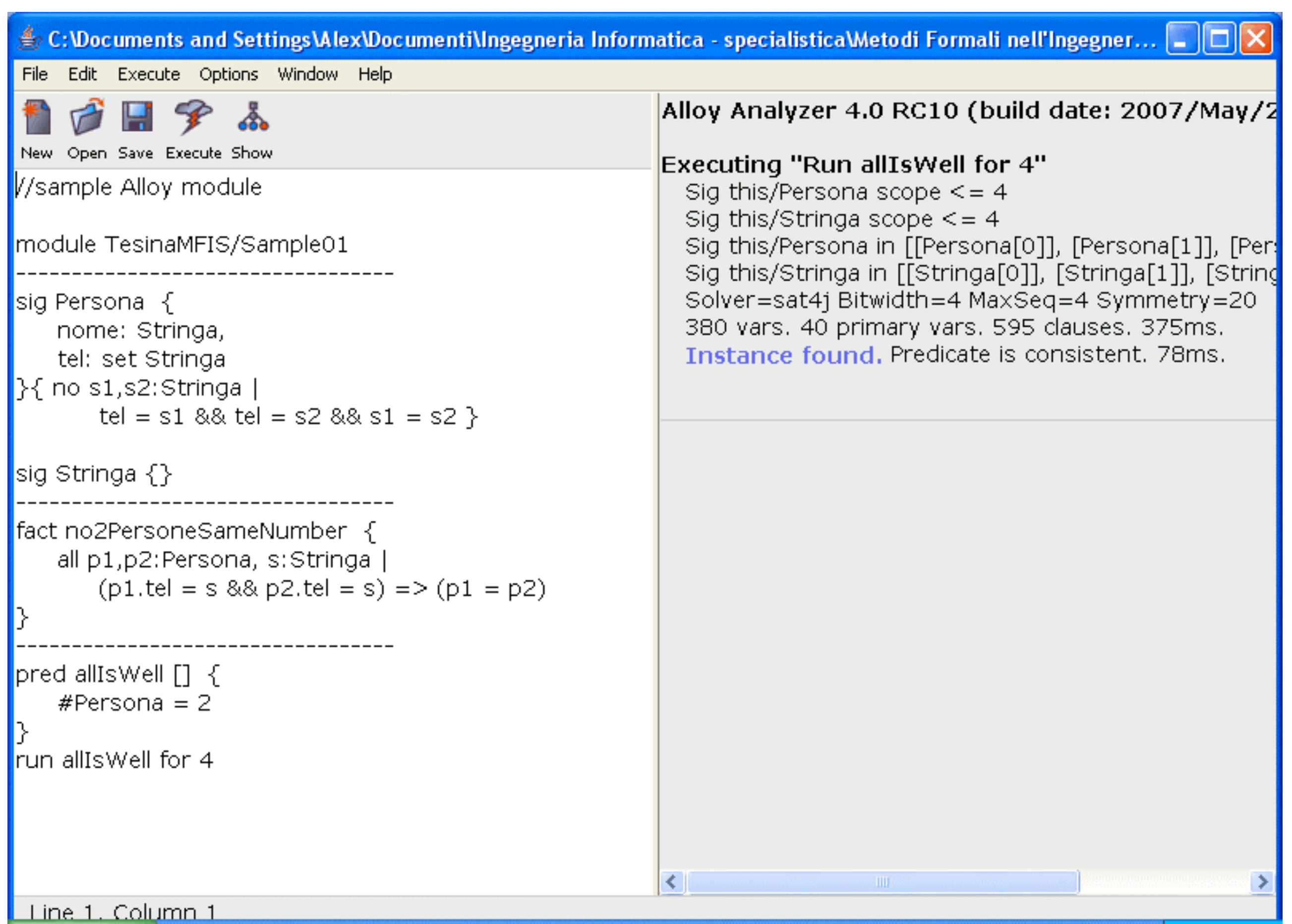
I comandi *Check* e *Run* sono direttive all'Alloy Analyzer per eseguire un'analisi di verifica del nostro modello. Il comando Check è usato per verificare Assertions (ovvero trovare un controesempio alle nostre affermazioni sul modello), mentre il comando Run è usato per invocare l'esecuzione di Predicati e/o Funzioni; l'uso di Run con predicati e funzioni è solitamente usato per risolvere problemi: ad esempio il già citato problema del River Crossing veniva mandato in esecuzione alla ricerca di una soluzione (sequenza di sigs State legali) tramite la Run di un predicato corrispondente allo stato finale del sistema (tutte le entità devono essere sulla riva opposta ripetto allo stato iniziale). Noi ci limiteremo ad un uso del run command simile a quello accennato nella sezione 2.7 sui Prediates.
La sintassi è simile nei due casi:
check assertionName for X but Y sigA, Z sigB
run predicateOrFunName for W sigC, sigD but R sigE, exactly S sigF
Ciò significa: "trova un controesempio all'Assertion assertionName su uno scope di X elementi (massimo numero di top-level sigs, non estensioni o subsets) tranne che per le sigs sigA e sigB, che devono essere al massimo di Y elementi" e "manda in esecuzione il predicato o funzione in questione con uno scope di W per sigC e sigD, uno scope di R per sigE ed esattamente S elementi per sigF".

Parte III: Uso dell'Alloy Analyzer

Questa sezione vuole delineare in breve l'aspetto e il modo d'uso del tool Alloy Analyzer, creato per la verifica automatica di modelli Alloy (file .als) e pensato per una agevole ispezione umana delle istanze fornite. Procederemo ad esplorare le nostre possibilità tramite un esempio.

Sezione 3.1 - La Schermata Principale

La schermata principale dell'analizzatore si presenta in questo modo:



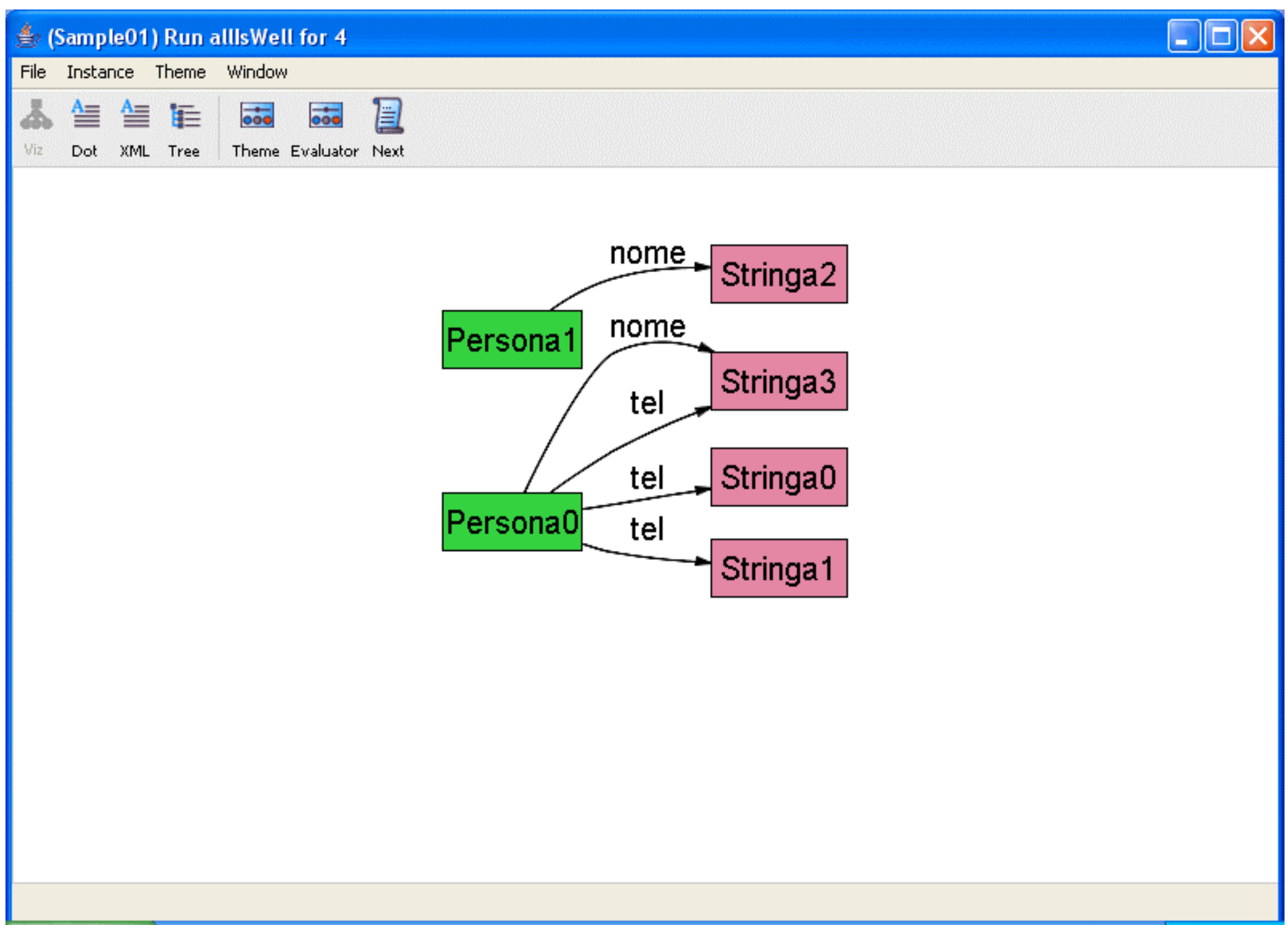
La schermata è normalmente divisa in due sezioni, liberamente dimensionabili:

- La sezione di sinistra ospita un normale text editor, da usare per creare i propri moduli Alloy da salvare come files .als; l'editor non presenta features degne di nota, ed è possibile utilizzare qualunque altro genere di programma per creare files .txt.
- La sezione di destra è invece la message box, contenente le informazioni che l'Alloy Analyzer riferisce in seguito ad un'analisi del modello tramite comandi Check e Run. Oltre a informazioni tecniche come il numero di variabili utilizzate o il tempo impiegato per l'analisi, notiamo che eventuali istanze fornite dall'Analyzer sono selezionabili come link dall'utente per essere esplorate a fondo (le parole "Instance found" scritte in blu).

Nell'immagine di esempio è stato definito un elementare module Alloy, chiamato Sample01, contenente Stringhe e Persone; queste ultime hanno un nome (Stringa) e zero o più numeri di telefono (Stringhe), con un constraint aggiuntivo (di tipo "Appended Fact", quindi valido solo per le Persone) per dichiarare che nessuna persona ha due numeri di telefono uguali; è inoltre presente un esempio di vincolo globale (un Fact), che impone che due persone diverse non possono condividere un numero di telefono, e un predicato "vuoto" (cfr. sez. 2.7 sui Predicati) accompagnato da un Run command con il quale chiediamo all'Analyzer di trovare un'istanza positiva su scope 4 (ovvero con al massimo 4 atomi Persona e 4 Stringhe) in cui ci sono esattamente due persone (operatore cardinalità '#').
NOTA: in Alloy sono possibili tre modi di denotare un commento, due identici a Java (//comment e /* comment */) e uno come per il tool NuSMV (-- comment).

Sezione 3.2 - Le 3 Modalità di Visualizzazione dei Risultati

Cosa otteniamo cliccando sul link "Instance found"? Dopo aver cliccato e applicato qualche customizzazione al colore ad al layout, possiamo esaminare l'istanza trovata nella cosiddetta "VIZ View", una delle possibili modalità di visualizzazione che l'Analyzer offre:



Il risultato trovato è un'istanza che, come richiesto, possiede esattamente due atomi Persona, Persona0 e Persona1, denotati in verde, e al massimo quattro (in questo caso proprio quattro) atomi Stringa: P1 ha come nome S2, P0 ha come telefoni S0, S1 ed S3, ed ha un nome uguale al suo telefono S3 (del resto il nostro modello non lo vietava!).

Prima della stesura del modello, avevamo previsto la possibilità che una persona potesse avere nome uguale al suo numero di telefono? Se si trattasse di un risultato indesiderato, ci saremmo imbattuti in un caso di Underconstraining e dovremmo aggiungere un vincolo globale che vieti tale eventualità.

Passiamo in rassegna le altre modalità di visualizzazione dei risultati offerte dall'Analyzer: La "Tree View" e la "XML View".

Il seguente è un esempio di XML View, che rappresenta appunto l'istanza come un XML document:

(Sample01) Run allIsWell for 4

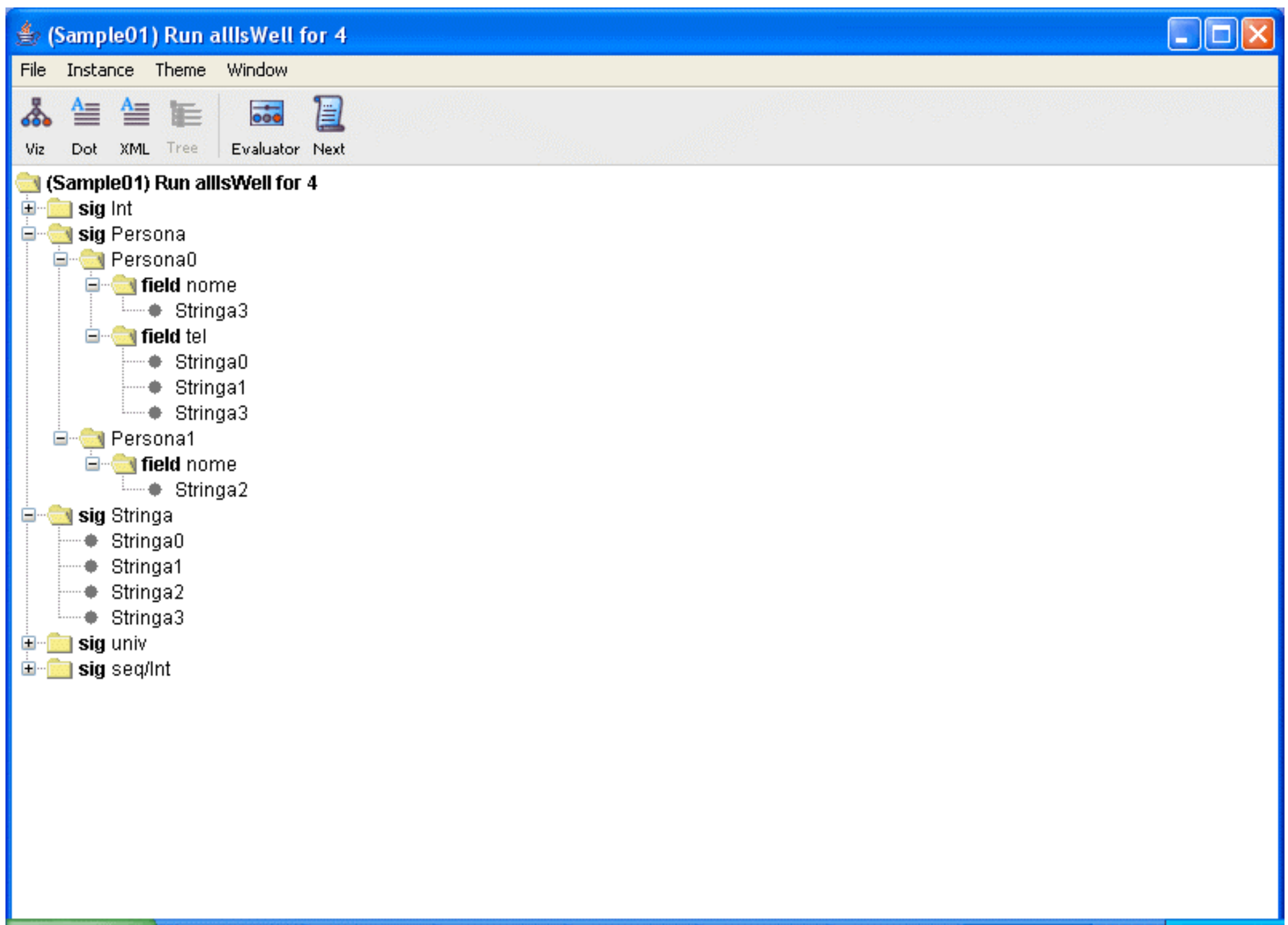
File Instance Theme Window

Viz Dot XML Tree Evaluator Next


```
<sig name="Persona">
  <atom name="Persona[0]"/>
  <atom name="Persona[1]"/>
</sig>

<field name="nome">
    <type> <sig name="Persona"/> <sig name="Stringa"/> </type>
    <tuple> <atom name="Persona[0]"/> <atom name="Stringa[3]"/> </tuple>
    <tuple> <atom name="Persona[1]"/> <atom name="Stringa[2]"/> </tuple>
</field>

<field name="tel">
    <type> <sig name="Persona"/> <sig name="Stringa"/> </type>
    <tuple> <atom name="Persona[0]"/> <atom name="Stringa[0]"/> </tuple>
    <tuple> <atom name="Persona[0]"/> <atom name="Stringa[1]"/> </tuple>
    <tuple> <atom name="Persona[0]"/> <atom name="Stringa[3]"/> </tuple>
</field>

<sig name="Stringa">
  <atom name="Stringa[0]"/>
  <atom name="Stringa[1]"/>
  <atom name="Stringa[2]"/>
  <atom name="Stringa[3]"/>
</sig>
```


Personalmente non ho trovato la XML View molto interessante; nel resto della trattazione non ci saranno ulteriori esempi di uso della XML View.

Per i nostri scopi, in caso di istanze grandi e un po' difficili da analizzare visivamente con il grafo della VIZ View, può essere vantaggioso ispezionare l'istanza con l'albero espandibile della Tree View. La possibilità di espandere i singoli atomi individualmente e di esplorare tutte le loro relazioni può risultare lento, ma offre la possibilità di una verifica sequenziale e sistematica della correttezza dell'istanza.



Nelle sezioni successive utilizzeremo principalmente VIZ e Tree Views per verificare la correttezza e completezza dei nostri modelli.

Sezione 3.3 - Opzioni dell'Alloy Analyzer

Nella parte rimanante della sezione III daremo una rapida occhiata alle opzioni che l'Analyzer offre tramite le menu bars nella parte alta dello schermo, ed eventuali altre "miscellaneous options".

Opzioni della Schermata Principale

File: le usuali funzionalità di apertura/chiusura/salvataggio et similia. Da notare la voce "Open Sample Modules" che permette di aprire moduli di esempio o anche "Utility Modules" da importare per operazioni complesse (Ordering, Booleans, Naturals, Integers, Binary and Ternary Relations' Operations).

Edit: le normali funzioni utiliy per copy/paste/find ecc.

Execute: questo menu offre una visione d'insieme di tutti i comandi presenti nel modulo; l'Alloy Analyzer può lavorare su un comando alla volta, e in caso di comandi multipli è necessario sceglierne uno da qui. Si possono anche scegliere comandi come "Show Latest Instance" (autoesplicativo), "Show Metamodel", che mostra una sorta di Class Diagram completo di relazioni ed estensioni del nostro modello (è appunto un "modello del

modello", visualizzabile con tutte le tre Views e potenzialmente molto utile, ad esempio nei casi di Reverse Engineering, o per avere una rappresentazione grafica e immediatamente comprensibile di un file Alloy che vediamo per la prima volta), e "Open Evaluator", utilizzato per la valutazione on the fly di funzioni data un'istanza risultato, e ciò ci interessa relativamente poiché faremo scarso uso delle funzioni.

Options: una lunga lista di voci, alcune piuttosto interessanti, tra cui la scelta del SAT Solver da utilizzare tra sette disponibili (default: SAT4J), tra i quali ZChaff, la scelta della quantità di memoria massima da allocare all'Analyzer, e la scelta della "Skolem Depth" da utilizzare tra 0, 1 e 2 (dafault: 0), cioè il massimo numero di alternanze all/exists permesse in una formula nella generazione di una funzione di Skolem.

Window, Help: non presentano nulla di interessante.

Icone: le icone presenti per quick access sono New, Open, Save, Execute (esegue il command eseguito per ultimo, o altrimenti il primo command del file), Show (cioè Show Latest Instance).

Opzioni della Schermata di Analisi

File, Window: nulla di interessante.

Instance: un unico comando, "Show Next Solution", autoesplicativo.

Theme: comandi per la customizzazione di colori, aspetto del grafo et similia per miglirare la visibilità. Esiste la possibilità di salvare/caricare temi.

Icone: shortcuts per VIZ, XML e Tree Views, la Dot View (poco interessante), nonché per Themes, Next Instance ed Evaluator, il già citato valutatore di funzione nell'istanza corrente.

Parte IV: Modellazione in Alloy di elementi degli UML Class Diagrams

Nel corso di Metodi Formali nell'Ingegneria del Software abbiamo affrontato una metodologia per formalizzare gli aspetti più importanti degli UML Class Diagrams in FOL, in modo da poter "interfacciare" il nostro modello di sistema, preso da un punto di vista concettuale e statico, con dei tools informatici al fine di eseguire una serie di controlli e verifiche per ridurre al minimo il numero di errori originati in questa delicata fase del ciclo di vita di un prodotto software.
Nel corso pertanto, oltre a definire il procedimento di trasformazione in FOL, abbiamo imparato ad usare i tools Otter e Mace per i nostri scopi, fornendo un input essenzialmente sotto forma di formule in FOL.
Lungo questa importante parte dell'argomento affrontato passeremo in rassegna ad uno ad uno tutti gli elementi dei diagrammi UML considerati: forniremo un chiaro e focalizzato esempio per ogni caso, accenneremo alla sua espressione in FOL come insegnata nel corso di Metodi Formali e vedremo una possibile modellazione nel linguaggio Alloy di tale formalizzazione, evidenziando dove opportuno vantaggi e svantaggi della versione in Alloy.
Gli elementi affrontati saranno, nell'ordine:

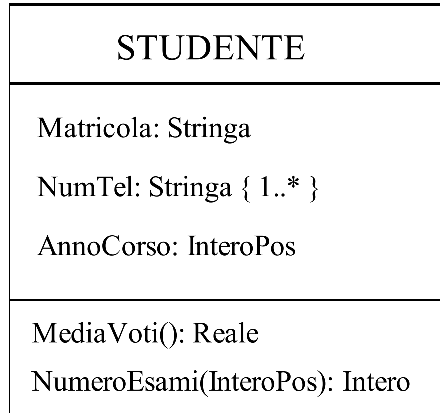
- Classi UML, con Attributi di classe, loro tipi e molteplicità, nonché Operazioni;
- Associazioni e Gerarchie di associazioni;
- Specializzazioni di Attributi e di Associazioni;
- Alcuni esempi di Vincoli Esterni.

Per favorire la comprensione e non introdurre troppi esempi nuovi, ricalcherò sia l'ordine che il tipo di frammenti UML d'esempio come compaiono nelle Dispense del corso dedicate ai metodi formali per gli UML Class Diagrams.

Qualche commento preliminare sulle "performances" di Alloy in questo frangente può aiutare il lettore ad avere un "feel" generale su cosa aspettarsi. Normalmente la modellazione in Alloy rispetto a quella in FOL risulta più breve e concisa, grazie alla potenza delle features di Alloy come le sigs e le "tre dimensioni": Relazioni-Insiemi-OO Programming; come downside, però, la formalizzazione risulta chiara e agevole solo per chi ha un po' di esperienza con Alloy, e dovrebbe risultare piuttosto ovvio che il tempo di apprendimento per usare le funzionalità base di Otter e Mace è in media notevolmente inferiore rispetto al training da dedicare ad Alloy.
Cionondimeno la "convenienza" e l'agevolezza con cui gli elementi di Alloy possono essere fatti interagire ed essere esaminati lasciano aperte molte possibilità applicative, che rendono sicuramente ben speso il tempo dedicato all'apprendimento. Di tali interessanti argomenti applicativi si tratterà nella sezione V, che parlerà del confronto tra Alloy Analyzer ed Otter/Mace.

Sezione 4.1 - Classi UML

A titolo di esempio considereremo la seguente classe UML Studente, contenente attributi di varia molteplicità e di diversi tipi, nonché due operazioni.



Il significato degli elementi della classe è autoesplicativo. Ora tradurremo, passo per passo, ogni elemento della classe in Alloy in maniera incrementale, e tenendo sempre presente la sematica della classe UML e la formalizzazione in FOL insegnata nel corso.

La Classe UML
Una classe è, in FOL, semplicemente un simbolo di predicato unario C/1 che un atomo deve soddisfare per "appartenere" alla classe; in Alloy è immediato pensare ad una UML Class come ad una Signature, poiché le sigs sono proprio predicati unari che denotano insiemi di atomi omogenei, in altre parole elementi di un insieme o istanze di una classe. Pertanto scriveremo in Alloy:

sig Studente {

}

Notare che l'apparteneza di un atomo ad una sig semplificherà notevolmente la formalizzazione di molti tra i concetti che vedremo, poiché il "controllo sui tipi", che si incontra spesso, non sarà praticamente mai presente, o meglio sarà implicito e automatico.

Gli Attributi di Classe e loro Tipi
La parte dedicata ad Attributi e Tipi è stata accorpata in quanto in Alloy, come vedremo, tali elementi vanno in tandem.
Nel corso è stato insegnato che un Tipo è un predicato unario T/1; ma allora possiamo benissimo rappresentare un Tipo in Alloy con un'altra Signature, poiché denota un insieme di atomi proprio come una classe; solitamente i tipi hanno dei bodies vuoti, poiché sono elementi primitivi che non dobbiamo modellare ulteriormente:

sig Stringa {}
sig Intero {}
sig InteroPos extends Intero {}

La sig InteroPos è stata dichiarata estensione di Intero, senza specificare in cosa differisca poiché come accennato non è per noi interessante.

Dobbiamo ora mettere in relazione gli Attributi con la Classe; in FOL è necessario un predicato binario che denoti l'appartenenza dell'Attributo alla Classe, nonché una formula che obblighi

quell'attributo di classe ad essere di un certo tipo; se seguenti formule in pseudo-FOL dovrebbero fornire un'idea abbastanza chiara:
- hasAttrib(classAtom, attribAtom)
- for all X,Y   isClass(X) && hasAttrib(X,Y) -> isType(Y)

In Alloy tale mapping è banalissimo: il concetto di "relazione di sig", o "campo di sig", una relazione con atomi di altre sigs, calza a pennello:

```
sig Studente {
        Matricola: Stringa,
        NumTel: Stringa,   //a tra poco le molteplicità
        AnnoCorso: InteroPos
}
```

Si ricordi che la semantica in questione, ad esempio per Matricola, è: per ogni atomo Studente esiste una relazione di nome Matricola con un solo altro atomo, di tipo Stringa.

Molteplicità degli Attributi
La facilità di traduzione in Alloy delle molteplicità varia da caso a caso, ma il passaggio è banale nella stragrande maggioranza dei casi. In questo frangente ci risultano notevolmente comode le varie keywords di molteplicità "embedded" in Alloy: *one*, *some*, *set*, *lone*.
Ricordiamo il mapping: one = 1..1; some = 1..*; set = 0..*; lone = 0..1.
Essendo tali molteplicità estrememente comuni in informatica, il constraint aggiuntivo è spesso banale; nel nostro esempio:

```
sig Studente {
        Matricola: Stringa,
        NumTel: some Stringa,
        AnnoCorso: InteroPos
}
```

La modifica dice essenzialmente: ogni Studente deve condividere la relazione NumTel con almeno un atomo di tipo Stringa (ma potrebbe essere più di uno).

Nel caso più generale, se abbiamo una molteplicità da X a Y, numeri finiti arbitrari, possiamo lavorare con l'operatore cardinalità (#) negli appended facts:

```
sig Example {
        someAttrib: someType //di tipo X..Y
} { #someAttrib >= X && #someAttrib =< Y }
```

Ciò impone, per tutti e soli gli atomi di tipo Example, che la relazione someAttrib abbia un numero di tuple pari almeno ad X e al massimo ad Y; in questo caso è risultato particolarmente conveniente la "comprensione" di Alloy di espressioni intere.

Possiamo facilmente notare come la caduta di complessità da UML ad Alloy sia notevole rispetto a UML-FOL: ricordiamo che secondo la metodologia vista a lezione sono necessarie, per un limite inferiore o superiore pari a X, formule con X variabili quantificate e Theta(X^2) diseguaglianze all'interno di tali formule.

Operazioni di Classe

Nel caso più generale possibile un'operazione è una funzione che associa un oggetto della Classe in questione ad un oggetto (un tipo o altra classe) risultato, anche in dipendenza da un numero arbitrario, ma finito, di parametri in input (istanze di tipi o classi). Per i nostri scopi di modellazione statica non è necessario occuparci di pre-condizioni e post-condizioni; in un certo senso siamo interessati più alla corretta sintassi (la cosiddetta *segnatura* in termini informatici, da non confondere con le sigs) dell'operazione di classe che alla sua semantica.
Nel corso di Metodi Formali l'espressione in FOL di un'operazione coinvolge il controllo sui tipi dei valori in gioco (this, input, return value) e il controllo sull'unicità del risultato. Le seguenti formule in pseudo-FOL possono aiutare il lettore ad avere un'idea più chiara:

- for all X,A1...An, B   classOp(X,A1...An,B) ->
  (isClass(X) && isRetType(B) && (isType1(A1) && ... && isTypeN(An)))

- for all X,A1...An, B1,B2   classOp(X,A1...An,B1) && classOp(X,A1...An,B2) -> B1 = B2

Come rappresentare tali vincoli in Alloy? Poiché l'operazione è un caso particolare di relazione, possiamo includerla, con le dovute cautele, nei "campi" di sig insieme agli Attributi di classe.
Ma procediamo in modo graduale: consideriamo prima un'operazione a zero argomenti, come MediaVoti nel nostro esempio, e notiamo come sia praticamente identica ad un attributo:

```
sig Studente  {
        //Attributi
        Matricola: Stringa,
        NumTel: some Stringa,
        AnnoCorso: InteroPos,

        //Operazioni
        opMediaVoti: one Reale
}

sig Reale  {}
```

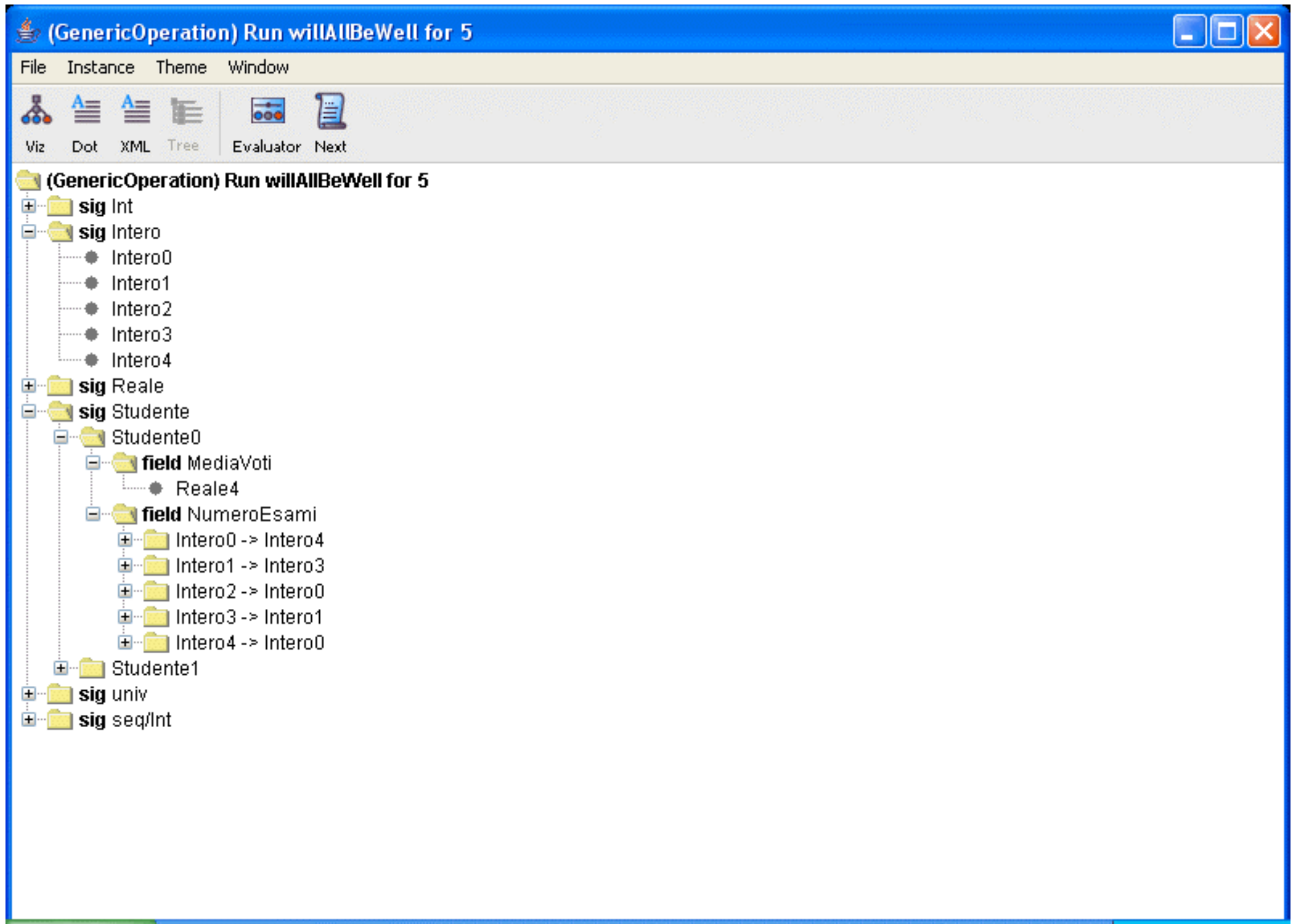
Abbiamo affermato, traendo inoltre vantaggio dal type-checking "automatico" offerto da Alloy, che per ogni atomo Studente esisterà uno ed un solo valore Reale corrispondente, sebbene non ci importi quale, in questo contesto.

Per operazioni ad un argomento, come NumeroEsami(InteroPos): Intero, dobbiamo dire ad Alloy che per ogni atomo Studente **e** per ogni intero positivo in input (ovviamente sempre all'interno di un bound limitato sul numero di interi da considerare) dobbiamo avere un unico Intero di ritorno. Possiamo realizzare tale istruzione molto agevolmente utilizzando una versione dell'operatore prodotto corredata di molteplicità a destra; ci assicuriamo cioè che esista un risultato per ogni possibile valore in input:

```
sig Studente  {
        //Attributi
        Matricola: Stringa,
        NumTel: some Stringa,
        AnnoCorso: InteroPos,

        //Operazioni
        opMediaVoti: one Reale,
```

```
        opNumeroEsami: Intero -> one Intero
}
```

Come si comporta il nostro Alloy model dopo questa affermazione? La seguente immagine mostra il risultato di tipo Tree View (molto comodo in questo caso) in uno scenario dove devono esistere almeno 2 Studenti e lo scope è ridotto a 5:



Notiamo come il "mondo degli interi" sia ridotto a 5 elementi, esistono esattamente 2 studenti, e riguardo a Studente0, egli ha Reale4 come media dei voti ed è previsto un possibile output di NumeroEsami per ogni possibile coppia <this, Intero> in input.

Nel caso più generale possiamo iterare la procedura; ad esempio per una operazione AdvancedOp con due input Reali e un return value Intero avremmo:

```
sig Studente  {
        //Attributi
        Matricola: Stringa,
        NumTel: some Stringa,
        AnnoCorso: InteroPos,

        //Operazioni
        opMediaVoti: one Reale,
        opNumeroEsami: Intero -> one Intero,
        opAdvancedOp: (Reale -> (Reale -> one Intero))
```

}

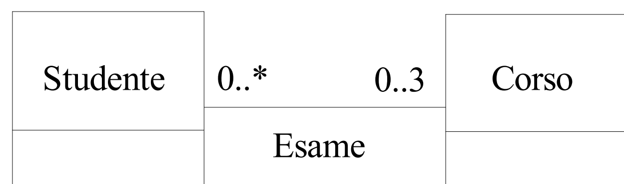
L'Analyzer considererà tutte le possibili combinazioni in input ed assegnerà un unico return value al valore di ritorno, il tutto con type-checking automatico. Per N input avremo bisogno di N operatori prodotto (->) annidati.
Cogliamo l'occasione per accennare che la documentazione di Alloy a volte non è chiarissima sull'uso dei vari operatori e sulle loro molteplicità, e spesso è onere dell'utilizzatore sperimentare per comprendere l'esatta semantica dei costrutti.

Sezione 4.2 - Associazioni e loro Gerarchie

In questa parte tratteremo gli argomenti legati ad Associazioni binarie con arbitraria molteplicità, Associazioni ternarie e Gerarchie sia singole (IS-A), sia multiple (Disjointness e Completeness).
Data la varietà di argomenti, dovremo esibire esempi differenti caso per caso; si cercherà comunque di mantenere una progressione incrementale nella modellazione dei vari concetti, dalle basi agli aspetti più complessi.

Associazioni Binarie e loro Molteplicità
Cominciamo la modellazione dei vari "flavours" di Associazione con una semplice Associazione binaria.
Ricordiamo che la formalizzazione in FOL per il caso generale prevede type-checking, come al solito, per le N Classi che partecipano all'Associazione, quest'ultima incarnata da un simbolo di predicato n-ario, più i vincoli per le molteplicità che ricalcano lo stesso schema delle molteplicità di Attributi (molteplicità Inf o Sup pari ad i => formule con i variabili quantificate e Theta(i^2) disuguaglianze).
In pseudo-FOL:
- someAssociation(X1, ..., Xn) //predicato n-ario
- for all X1...Xn someAssoc(X1...Xn) -> isClass1(X1) && ... && isClassN(Xn)
- controllo sulla molteplicità (cfr. molteplicità di Attributi di Classe).

Consideriamo il seguente esempio di Associazione binaria:



L'Associazione Esame altro non è che una relazione, ancora una volta, che lega istanze della Classe Studente con istanze della Classe Corso, in modo vincolato da molteplicità ed in modo

che la coppia <someStudente, someCorso> sia unica, e la relazione sia presente in entrambi i sensi.

Ancora una volta possiamo far ricorso ai "campi interni" delle sigs di Alloy per modellare la relazione in questione con molta eleganza e semplicità:

```
sig Studente  {
        assocEsame: Corso
}

sig Corso  {
        assocEsame: Studente
}
```

Aggiungiamo quindi le molteplicità, in modo del tutto analogo a come abbiamo trattato gli Attributi di classe:

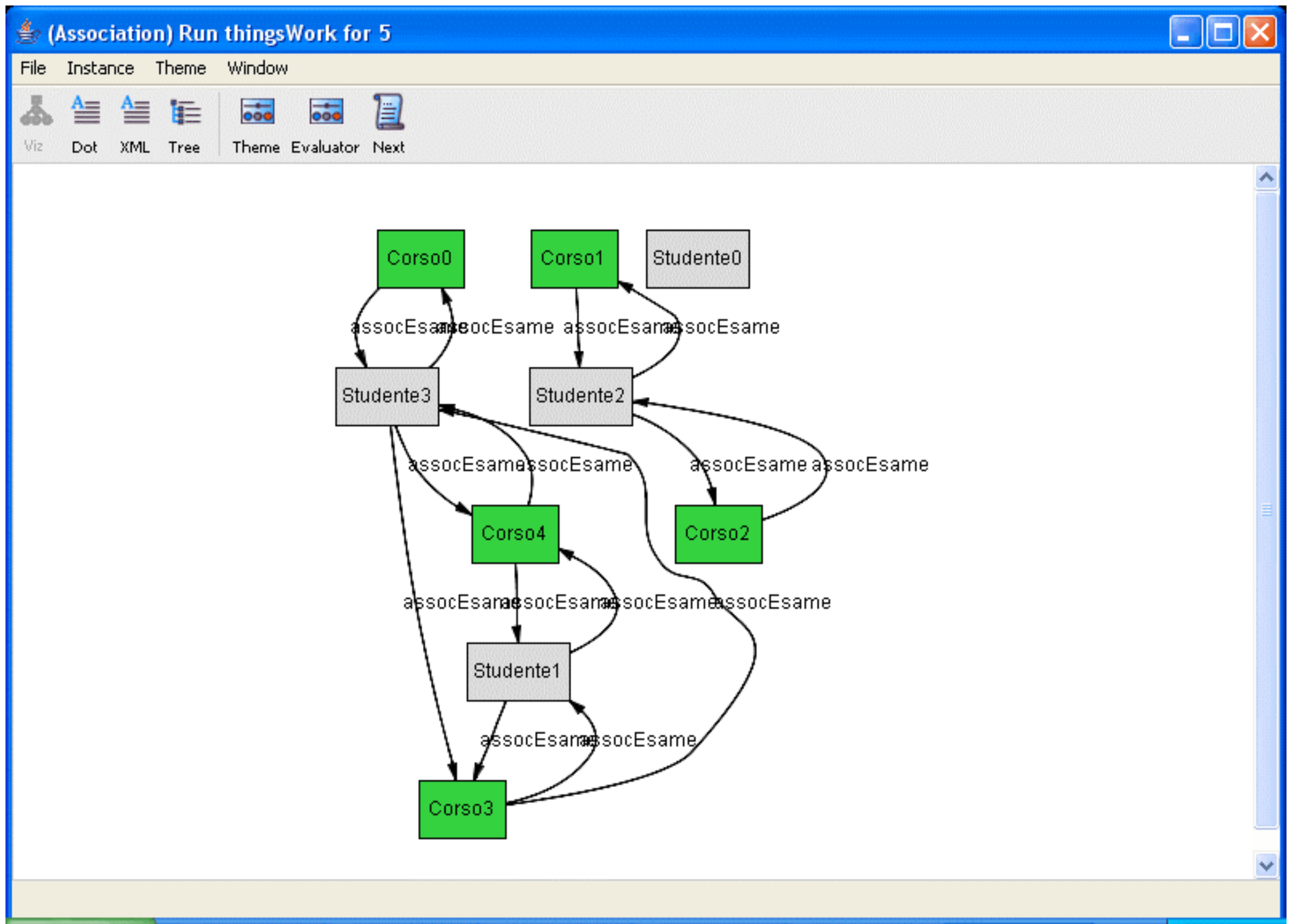
```
sig Studente  {
        assocEsame: set Corso  //questo impone il limite inferiore a 0...
} {  #assocEsame =< 3  } //...e questo il superiore a 3 (appended fact)

sig Corso  {
        assocEsame: set Studente //ovvero 0..*
}
```

Sebbene possa sembrare una modellazione sufficiente, al lettore attento non sfuggirà che secondo queste dichiarararazioni è legale avere uno Studente S associato ad un Corso C senza che il corso C sia esso stesso in relazione con lo Studente S.
Manca cioè un vincolo, comune alle due sigs Studente e Corso, che impone la presenza di coppie uniche e relazioni presenti da ambo le parti. Poiché si tratta di un vincolo comune a più sigs conviene esprimerlo tramite un Fact:

```
fact associationEsame  {
        all s:Studente, c:Corso |
                ((c in s.assocEsame) <=> (s in c.assocEsame))
}
```

Si presti particolare attenzione alla struttura del vincolo; esso afferma che per ogni atomo Studente e per ogni Corso vale la seguente proprietà: un corso C appartiene all'insieme dei corsi che dividono la relazione Esame con uno Studente S **se e solo se** quello stesso Studente S è nell'insieme degli studenti che condividono la relazione Esame con il corso C.

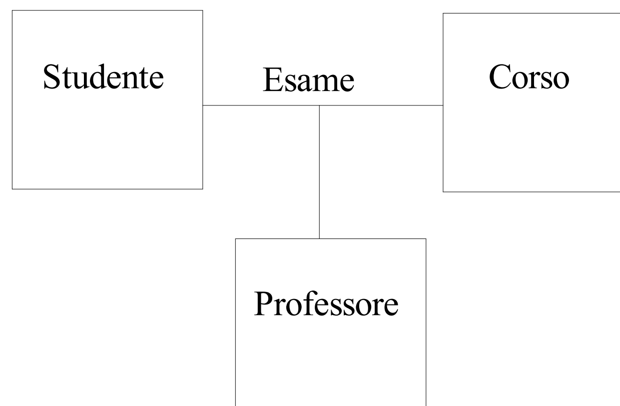
La seguente immagine è la VIZ View di un'istanza positiva restituita dall'Alloy Analyzer, con uno scope di 5 e il vincolo che ci fossero almeno 3 studenti ed almeno 1 corso. Notiamo come Studente0 non abbia passato nessun esame, Studente1 ha affrontato Corso3 e Corso4, Studente2 Corso1 e Corso2, mentre Studente3, particolarmente laborioso, ha sostenuto Corso0, Corso4 e Corso3.



Accenniamo inoltre al fatto che qualora l'Associazione da esprimere in Alloy possedesse degli Attributi, l'unico modo per modellare la loro presenza sarebbe quello di inserirli come campi interni nelle sigs che patecipano all'associazione (come un normale Attributo di classe), con un vincolo globale di tipo Fact che obblighi tali attributi delle varie classi ad avere valori coincidenti per preservare la consistenza.

Associazioni Ternarie

Consideriamo il seguente esempio di Associazione Ternaria:



Possiamo adottare un approccio simile a quello utilizzato per le binarie, ma con le dovute precauzioni. I vincoli e la formalizzazione dell'Associazione si complicano notevolmente, come era prevedibile.
Similmente a come siamo proceduti per le Operazioni di Classe, nei campi delle sigs in questione non abbiamo più un insieme di atomi, ma il prodotto di due insiemi, che dobbiamo opportunamente vincolare:

```
sig Studente  {
        assocEsame: Corso -> Professore
}

sig Corso  {
        assocEsame: Studente -> Professore
}

sig Professore  {
        assocEsame: Studente -> Corso
}
```

Come esprimere i vincoli necessari? Come di consueto il type-checking è automatico, e in questa sede non affronteremo molteplicità per le Ternarie diverse da 0..*. Dobbiamo però imporre il vincolo di presenza di triple uniche nel modello, nonché la molteplicità a 0..*.
Un possibile modo di organizzare i vincoli è il seguente:

```
sig Studente  {
        assocEsame: Corso set->set Professore
}

sig Corso  {
        assocEsame: Studente set->set Professore
}

sig Professore  {
        assocEsame: Studente set ->set Corso
}

fact ternaryAssocEsame  {
        all s:Studente, c:Corso, p:Professore |
                ( ((c->p in s.assocEsame) => (s->p in c.assocEsame)&&(s->c in p.assocEsame))
                        &&
                 ((s->p in c.assocEsame) => (c->p in s.assocEsame)&&(s->c in p.assocEsame))
                        &&
                 ((s->c in p.assocEsame) => (c->p in s.assocEsame)&&( s->p in c.assocEsame))
                )
}
```

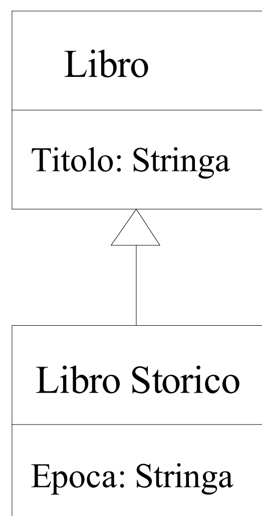
In linguaggio naturale, ogni Studente o Corso o Professore può essere legato ad un numero arbitrario di relazioni ternarie di tipo Esame, ma deve valere il seguente vincolo: riguardo a ogni Studente S, se un Corso C e un Professore P partecipano con S ad una relazione ternaria Esame,

allora segue che C sa di partecipare a quella relazione con S e P, e anche P sa di partecipare alla relazione con S e C; similmente per l'insieme dei Corsi e dei Professori.

Le molteplicità di tipo *set* a destra e a sinistra dell'operatore prodotto sono ridondanti ('->' è equivalente a 'set->set') ma sono state comunque inserite per evidenziare il fatto che non stiamo imponendo alcun constraint di molteplicità sulle varie classi coinvolte nell'Associazione: tutti partecipano con molteplicità 0..*, ovvero non affermiamo che per ogni professore debbano esistere un numero minimo o massimo di studenti o corsi che partecipano con esso alla relazione Esame.
Per ulteriori dettagli di sintassi e semantica dell'operatore prodotto con molteplicità si consultino le sezioni 2.1 (sulle Espressioni in Alloy) e 2.3 (Quantificatori in Alloy).

Generalizzazioni (Relazioni IS-A)
Essendo il concetto di Classe UML esprimibile in FOL come un predicato unario C/1, il concetto di sottoclasse che è anche istanza di una superclasse ha un'espressione molto concisa, del tipo:
- for all X   isSubClass(X) -> isSuperClass(X)

Anche in Alloy è estremamente semplice tradurre la proprietà di ereditarietà grazie alla possibilità di estendere le segnature (per i dettagli cfr. sezione 2.5 sulle Signatures). Se ad esempio avessimo la situazione seguente:



Potremmo tradurre in Alloy come:

```
sig Libro  {
        Titolo: Stringa
}

sig LibroStorico extends Libro  {
        Epoca: Stringa
}

sig Stringa {}
```

In tal modo tutte le istanze di LibroStorico (gli atomi appartenenti all'insieme LibroStorico) sarebbero un *subset* degli atomi di tipo Libro; ogni atomo LibroStorico eredita tutte le relazioni e gli appended facts relativi alla classe Libro.

Gerarchie (Disjointness & Completeness)
Supponiamo di avere una superclasse S e N sottoclassi B(i) in relazione IS-A con S. In termini di FOL, oltre al vincolo di Generalizzazione già citato in precedenza potremmo aver bisogno di ulteriori vincoli per eventuali casi di Disjointness e Completeness.
Il caso di Completeness è molto semplice e sintetico:
- for all X   isSuperClass(X) -> isSubClass1(X) || ... || isSubClassN(X)
Tuttavia per esprimere la Disjointness abbiamo bisogno di n*(n-1)/2 congiunzioni di formule del tipo mostrato di seguito che escludono l'appartenenza a due distinte classi:
- for all i [1,n] (for all j < i (for all X   isClassI(X) -> not isClassJ(X)))

Ancora una volta la sintassi Alloy ci favorisce, in quanto esprimere tutte le possibili combinazioni di gerarchia è estrememente semplice grazie alle keywords **extends**, **in** e gli operatori insiemistici.

Supponiamo di avere una superClasse Super e due sottoclassi, SubA e SubB. Abbiamo ovviamente quattro casi possibili:

CASO I : La gerarchia non è né disjoint, né complete. In tal caso usiamo l'operatore *in*, che a differenza di *extends* non implica la disjointness delle sottoclassi.

sig Super {}

sig SubA in Super {}

sig SubB in Super {}

CASO II : La gerarchia è disjoint, ma non complete. Come il precedente, ma stavolta con l'operatore *extends*:

sig Super {}

sig SubA extends Super {}

sig SubB extends Super {}

Caso III : La gerarchia è complete, ma non disjoint. Usiamo *in* per la non-disjointness, ed aggiungiamo un vincolo esterno con operatori insiemistici per la completeness:

sig Super {}

sig SubA in Super {}

sig SubB in Super {}

fact completeHierarchy { SubA + SubB = Super //+ è l'unione }

Caso IV : La gerarchia è sia disjoint che complete. Ci riduciamo ad una prevedibile composizione dei casi precedenti:

sig Super {}

sig SubA extends Super {}

sig SubB extends Super {}

fact completeAndDisjointHierarchy { SubA + SubB = Super //+ è l'unione }

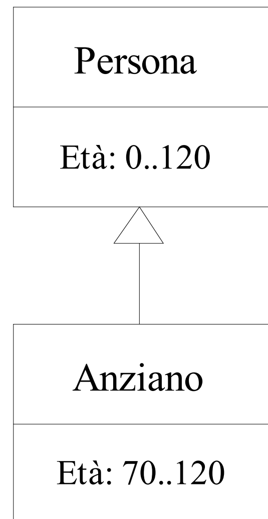
Ciò conclude la trattazione sulle Associazioni.

Sezione 4.3 - Specializzazione di Attributi e Associazioni

Ora verrà affrontata la questione relativa alla modellazione di Specializzazioni, sia di Attributi sia di Associazioni (le cosiddette associazioni Subset).

Specializzazione di Attributi
Nel caso di un Attributo di una sottoclasse che sia anche presente nella relativa superclasse, ma tale da essere nella subclass solo un sottoinsieme dell'insieme di attributi originario, parliamo di specializzazione di attributo.
In seguito considereremo specializzazioni di tre tipi: restrizioni di dominio sugli interi, restrizioni di molteplicità e restrizioni di dominio arbitrario (sulle stringhe, ad esempio).
Nell'approcio di formalizzazione in FOL visto nel corso, poiché la sottoclasse ridefinisce ogni suo attributo con le procedure viste nella sezione 4.1, compresi gli attributi specializzati, non vi è alcuna differenza "di trattamento" tra questi ultimi e gli attributi esclusivi alla sottoclasse.

In Alloy questo non è possibile, poiché le regole di namespace non permettono overloading nel nome degli attributi tra signatures in rapporti di Generalizzazione (*extends* e *in*).
Possiamo però risolvere agevolmente il problema imponendo dei vincoli (appended facts) alla sig subclass, in modo da restringere come desideriamo le relazione a cui la sig partecipa.
Si consideri il seguente esempio:



Per rendere significativo l'esempio stavolta dovremo modellare il campo interno (Attributo in questo caso) Età come una relazione con l'insieme degli Interi, in questo modo:

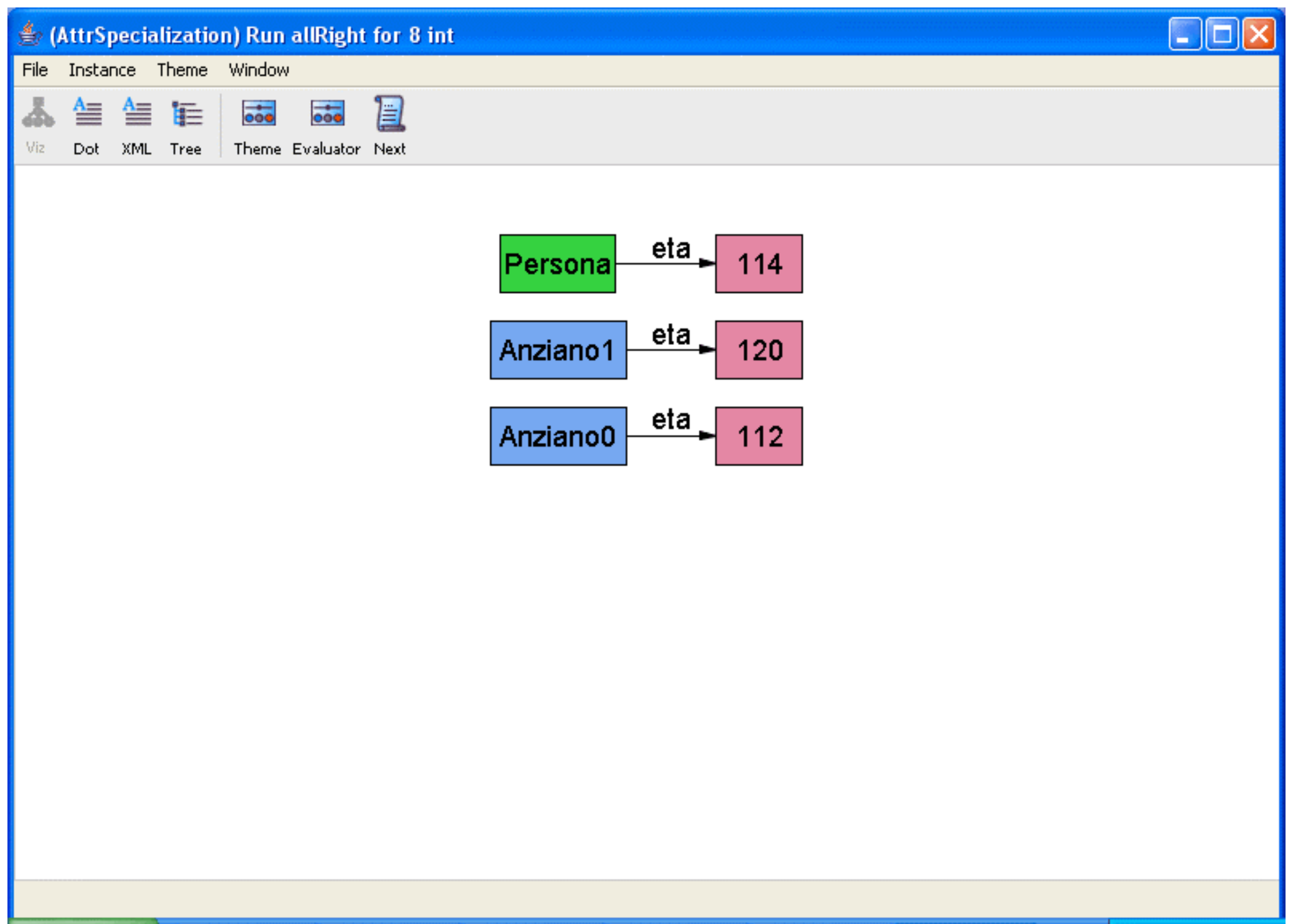
```
sig Persona  {
        Eta: Int
} { Eta >= 0 && Eta =< 120  }

sig Anziano extends Persona  {
        //nessun ulteriore attributo
} { Eta >= 70 && Eta =< 120  } //limite sup ridondante, ma è riportato per chiarezza
```

La seguente figura mostra il risultato fornito dall'Alloy Analyzer in seguito ad una richiesta di modello con esattamente tre persone e due anziani.

Come piccola technical note, accenniamo al fatto che lo scope minimo richiesto per la valutazione di tale istanza dipende dal valore del massimo numero intero che abbiamo usato, in questo modo:

- scope minimo = (numero di bit necessari per il max int value) + 1

Nel nostro caso è stato necessario uno scope di almeno 8 (roofOfPwr2(120) = 127 -> 7 bit + 1 = 8).



Nel caso di restrizioni di cardinalità, la procedura da seguire dipende da caso a caso. In generale dovrebbe essere sufficiente aggiungere uno o più constraints sotto forma di appended facts nella sottoclasse, utilizzando opportunamente l'operatore cardinalità (#) per restringere le molteplicità.

Se ad esempio avessimo una superclasse Super che ha un attributo someAttrib costituito da una o più Stringhe, mentre la cottoclasse Sub deve avere una e una sola Stringa per someAttrib, scriveremmo:

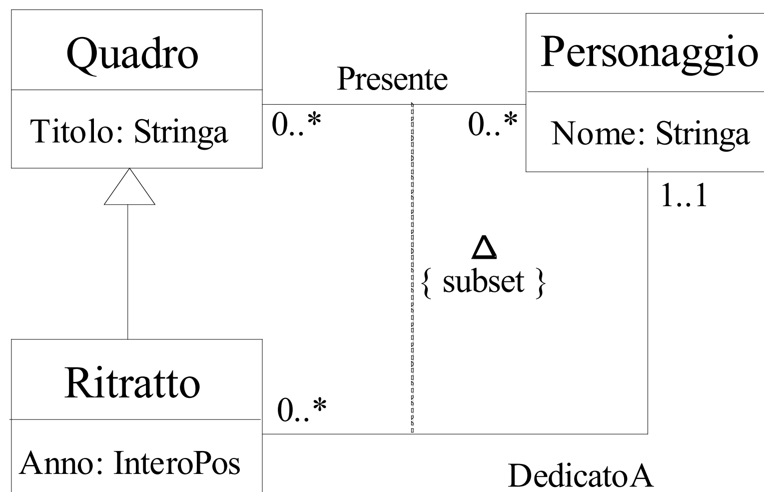
```
sig Super  {
        someAttrib: some Stringa
}

sig Sub extends Super  {

} { #someAttrib = 1 }

sig Stringa  {}
```

Nel terzo ed ultimo caso, in cui abbiamo una specializzazione di dominio arbitraria, possiamo affrontare il problema tramite constraints di ereditarietà negli appended facts: se ad esempio la classe Super appena vista avesse un ulteriore attributo someOtherAttrib, di tipo Stringa, e volessimo specializzarlo nella sottoclasse Sub come Stringa alfanumerica, potremmo scrivere:

```
sig Stringa  {}

sig StringaAlfanumerica extends Stringa  {}

sig Super  {
        someAttrib: some Stringa
        someOtherAttrib: Stringa
}

sig Sub extends Super  {

} { #someAttrib = 1 &&
    all s:Stringa  |   (s in someOtherAttrib) => (s in StringaAlfanumerica)
  }
```

In tal modo tutte le istanze di Sub avrebbero un atomo StringaAlfanumerica come attributo someOtherAttrib.

Specializzazione di Associazioni (Subset Associations)
Il caso della Specializzazione di Associazione richiede un esempio per forza di cose più ricco di attori (e ciò si tradurrà, com'è naturale, in un blocco di codice Alloy più consistente del solito), e tuttavia sia il concetto, sia la traduzione in Alloy del vincolo di Subset non presentano particolari difficoltà di comprensione.
Consideriamo il seguente esempio:



L'Associazione DedicatoA è subset dell'Associazione Presente; l'espressione mediante FOL di un vincolo del genere sarebbe molto semplice ed intuitiva:
- for all X1, X2   assocDedicatoA(X1, X2) -> assocPresente(X1, X2)

Vediamo come sia possibile dire altrettanto in Alloy; useremo un vincolo di tipo Fact per modellare il rapporto di Specializzazione:

```
//signatures

sig Quadro  {
        Titolo: Stringa,
        assocPresente: set Personaggio
}

sig Ritratto extends Quadro  {
        Anno: InteroPos,
        assocDedicatoA: one Personaggio
}

sig Personaggio  {
        Nome: Stringa,
        assocPresente: set Quadro,
        assocDedicatoA: set Ritratto
}

sig Stringa  {}

//vincoli per le associazioni bidirezionali

fact associationPresente  {
        all q:Quadro, p:Personaggio  |
                (p in q.assocPresente) <=> (q in p.assocPresente)
}

fact associationDedicatoA  {
        all r:Ritratto, p:Personaggio  |
                (p in r.assocDedicatoA) <=> (r in p.assocDedicatoA)
}

//vincolo di specializzazione di associazione

fact associationSpecialization  {
        all r:Ritratto, p:Personaggio  |
                (p in r.assocDedicatoA) => (p in r.assocPresente)
}
```

L'esempio non ha bisogno di molte spiegazioni: ci siamo infatti limitati ad imporre lo stesso semplice vincolo espresso in precedenza in FOL.

Sezione 4.4 - Cenni sulla Traduzione di Vincoli Esterni

In questa ultima parte della panoramica fornita riguardo una possibile formalizzazione in Alloy degli UML Diagrams verranno forniti alcuni brevi cenni su come mappare alcune tipologie di Vincoli esterni nel linguaggio in esame.

Contenimento stretto di tipi
Nel caso avessimo due tipi di dato (che ricordiamo essere espressi come due distinte Signatures in Alloy) in relazione di contenimento stretto, come ad esempio Intero e InteroPos, e volessimo modellare questo rapporto, è sufficiente ricordare che dichiarando la sig InteroPos come estensione (extends) o sottoinsieme (in) di Intero avremmo esattamente il risultato voluto, poiché si tenga conto del fatto che tali keywords non impongono mai la completeness.
In FOL avremmo invece dovuto dichiarare che:
- for all X   InteroPos(X) -> Intero(X) && existsX   not InteroPos(X) && Intero(X)

Un esempio di uso del contenimento stretto di tipi è stato fornito nella sezione 4.3, dedicata alle varie tipologie di specializzazione di attributi (Stringa e StringaAlfanumerica).

Dipendenze Funzionali
Con riferimento al precedente esempio di Associazione Ternaria (assocEsame(Studente, Corso, Professore)) , se avessimo un vincolo di tipo "Ogni studente può sostenere un esame per un certo corso al più una volta, indipendentemente dal professore coinvolto", esso dovrebbe essere tradotto in FOL da una formula di questo tipo:
- for all XYZW   Esame(X,Y,Z) && Esame(X,Y,W) -> Z = W

In Alloy tale vincolo potrebbe essere espresso con un Fact che ricalca la formula FOL appena vista (si confronti con il codice Alloy dell'esempio in questione):

```
fact dipendenzaFunz  {
        all s:Studente, c: Corso, p1, p2: Professore  |
                ((c->p1 in s.assocEsame) && (c->p2 in s.assocEsame)) => (p1 = p2)
}
```

In questo modo l'Alloy Analyzer ci fornisce, correttamente, solo istanze in cui per ogni Studente abbiamo solo esami con corsi distinti.

Identificatori di Classe
Ipotizziamo di avere una classe Studente con un attributo Matricola di tipo Stringa, e che la matricola identifichi univocamente uno Studente. In FOL scriveremmo:
- for all XYZ isStudente(X) && isStudente(Y) && hasMatricola(X,Z) &&
    hasMatricola(Y,Z) -> X = Y

Come per l'esempio precedente, il vincolo è banalmente traducibile in Alloy con un fact:

```
fact classId  {
        all s1,s2:Studente, m:Stringa  |
                ((s1.matricola = m) && (s2.matricola = m)) => (s1 = s2)
}
```

Con queste osservazioni chiudiamo il capitolo dedicato alla modellazione in Alloy degli elementi degli UML Diagrams visti nel corso di Metodi Formali.

Parte V: Analisi di Confronto tra Alloy ed Otter/Mace nel Contesto della Verifica di Proprietà degli UML Diagrams

In questa sezione passeremo all'ambito della verifica di proprietà: tratteremo dei semplici esempi didattici per dimostrare come i metodi di formalizzazione visti finora siano validi, ovvero implichino determinate caratteristiche nel nostro modello; per fare ciò utilizzeremo, in chiave comparativa, sia il linguaggio Alloy sia la sintassi di Otter/Mace, ovviamente con i rispettivi tools, mettendo in evidenza aspetti di performance, espressività ed "ease of use".
Il capitolo sarà organizzato in 3 parti: nella parte 5.1 daremo una breve overview su alcune possibili tipologie di casistiche di verifica a cui possiamo essere interessati, essenzialmente quelle affrontate nel corso di Metodi Formali; nella parte 5.2 procederemo al confronto tra Alloy ed Otter/Mace esaminando alcuni esempi pratici e il comportamento dei vari tools; infine nella parte 5.3 vedremo i possibili utilizzi degli operatori di Chiusura Transitiva di Alloy, con loro sintassi, semantica ed applicazione nel nostro contesto.

Sezione 5.1 - Overview delle Tipologie di Verifica

Conseguenze Implicite
Per verifica di Conseguenze Implicite si intende il dimostrare che una generica formula FOL, la quale dovrebbe essere una desiderata conseguenza, anche se non esplicitamente espressa, della nostra modellazione, sia effettivamente implicata dalla formalizzazione.
In altri termini il nostro insieme di formule modella la nuova formula, che è per noi la proprietà di interesse.
Tale casisitica di verifica può essere eseguita in Otter/Mace aggiungendo la formula al modello e verificando la validità con il Theorem Prover o il Model Finder, mentre in Alloy possiamo dichiarare un'assertion e sfidare l'Analyzer a trovare un constroesempio (sempre entro uno scope finito) in cui i vincoli specificati non sono rispettati.

Single-Class Consistency
Un controllo di Consistenza di una Singola Classe, o Single-Class Consistency, mira a provare che per una specifica classe i vincoli definiti rendano possibile lo scenario in cui esistano un numero maggiore di zero di istanze di quella classe; in altre parole vorremmo verificare che la nostra modellazione non vieti inavvertitamente, per ogni istanziazione legale, l'esistenza di un membro della particolare classe in esame. Risulta piuttosto chiaro che un diagramma in cui è presente una classe che non può avere istanze presenta degli errori che devono essere corretti.
Mediante Otter/Mace possiamo cercare di risolvere il problema individuando un modello in cui compaia un'istanza di tale classe, mediante il comando 'mace2'; in Alloy è sufficiente richiedere un'istanza positiva tramite predicato vuoto con imposizione di cardinalità ("forzare" il numero di istanze tramite l'operatore cardinalità '#').

Sussunzione tra Classi
Nel caso di verifica di Sussunzione tra due classi siamo interessati a verificare se tra di loro esiste una relazione di generalizzazione (cioè una IS-A) non esplicitamente modellata nella nostra formalizzazione.

Il controllo può essere effetuato mediante formule analoghe a quelle per le IS-A in Otter/Mace, cercando una dimostrazione con il Theorem Prover ('otter' command), mentre in Alloy possono essere usate delle assertions che cercano un contenimento tra signatures.

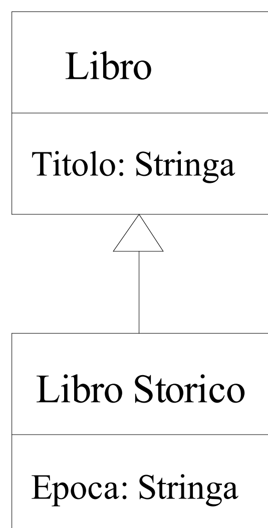
Equivalenza tra Classi
Questa tipologia è un caso particolare della precedente, e consiste nell'avere una coppia di classi in relazione reciproca di Sussunzione: dette le classi A e B, A sussume B e B sussume A.
Il risultato è spesso indesiderato in quanto le classi hanno le stesse istanze, e quindi una è ridondante e complica inutilmente il diagramma.
Tale tipologia di verifica si affronta come per il caso precedente, ripetendo il controllo su entrambe le classi.

Ristrutturazione di Parti del Diagramma
Sotto il nome di Ristrutturazione del Diagramma UML si annoverano tutte le modifiche che portano ad un diagramma di migliore qualità (come ad esempio alta coesione, basso accoppiamento, alta modularizzazione, eliminazione di aspetti ridondanti o inutili), facendo bene attenzione che il nuovo diagramma sia equivalente all'originale; ciò, nel contesto in esame, si traduce in una equivalenza tra le formalizzazioni dei due diagrammi.
Possiamo verificare tali equivalenze con Otter/Mace raggruppando le varie formule che costituiscono le due diverse versioni del diagramma sotto due distinti simboli di variabile e verificare l'equivalenza tramite il "se e solo se", utilizzando il Theorem Prover ('otter' command).

Sezione 5.2 - Verification Case Studies: Otter/Mace vs Alloy

Verranno ora affrontati degli studi di caso relativi alle tipologie di verifica di Conseguenze Implicite e Single Class Consistency.

Case Study 1: Conseguenze Implicite relative a Generalizzazioni
Consideriamo il seguente semplice scenario di IS-A, già esaminato nella sezione 4.2 dedicata alle Generalizzazioni:



Procediamo quindi a formalizzare il frammento di diagramma, prima in Otter/Mace FOL, poi in Alloy.

```
% Formalizzazione in sintassi Otter

% modellazione della classe Libro
all X Y ((Libro(X) & Titolo(X,Y)) -> (Stringa(Y))).
all X ((Libro(X)) -> (exists Y (Titolo(X,Y)))).
all X Y Z ((Libro(X) & Titolo(X,Y) & Titolo(X,Z)) -> (Y = Z)).

% modellazione della classe LibroStorico
all X Y ((LibroStorico(X) & Epoca(X,Y)) -> (Stringa(Y))).
all X ((LibroStorico(X)) -> (exists Y (Epoca(X,Y)))).
all X Y Z ((LibroStorico(X) & Epoca(X,Y) & Epoca(X,Z)) -> (Y = Z)).

% modellazione della relazione IS-A
all X ((LibroStorico(X)) -> (Libro(X))).

% disjointness tra classi e tipi
all X ((Libro(X)) -> (-Stringa(X))).

% appartenenza di ogni oggetto ad una classe
all X ((Libro(X)) | (Stringa(X)) | (LibroStorico(X))).
```

```
//Formalizzazione in sintassi Alloy

sig Libro  {
        Titolo: Stringa
}

sig LibroStorico extends Libro  {
        Epoca: Stringa
}

sig Stringa  {}
```

La codifica nei due linguaggi è piuttosto eloquente: grazie ai potenti costrutti per l'ereditarietà ('extends') e al concetto di Signature e di campo interno alle sigs, la formalizzazione in Alloy è più breve, più chiara per chi gode di un background informatico, più intuitiva, meno error-prone, contiene type-checking automatico ed è meno stressante da scrivere, poiché ad esempio l'uso delle parentesi, rigidissimo in Otter, è ridotto al minimo in Alloy.

Ipotizziamo ora di essere interessati alle seguenti Conseguenze Implicite, delle quali le prime due sono vere, mentre la terza è falsa:
1- LibroStorico eredita l'attributo Titolo; //True
2- Titolo è monovalore in LibroStorico; //True
3- Libro eredita l'attributo Epoca. //False

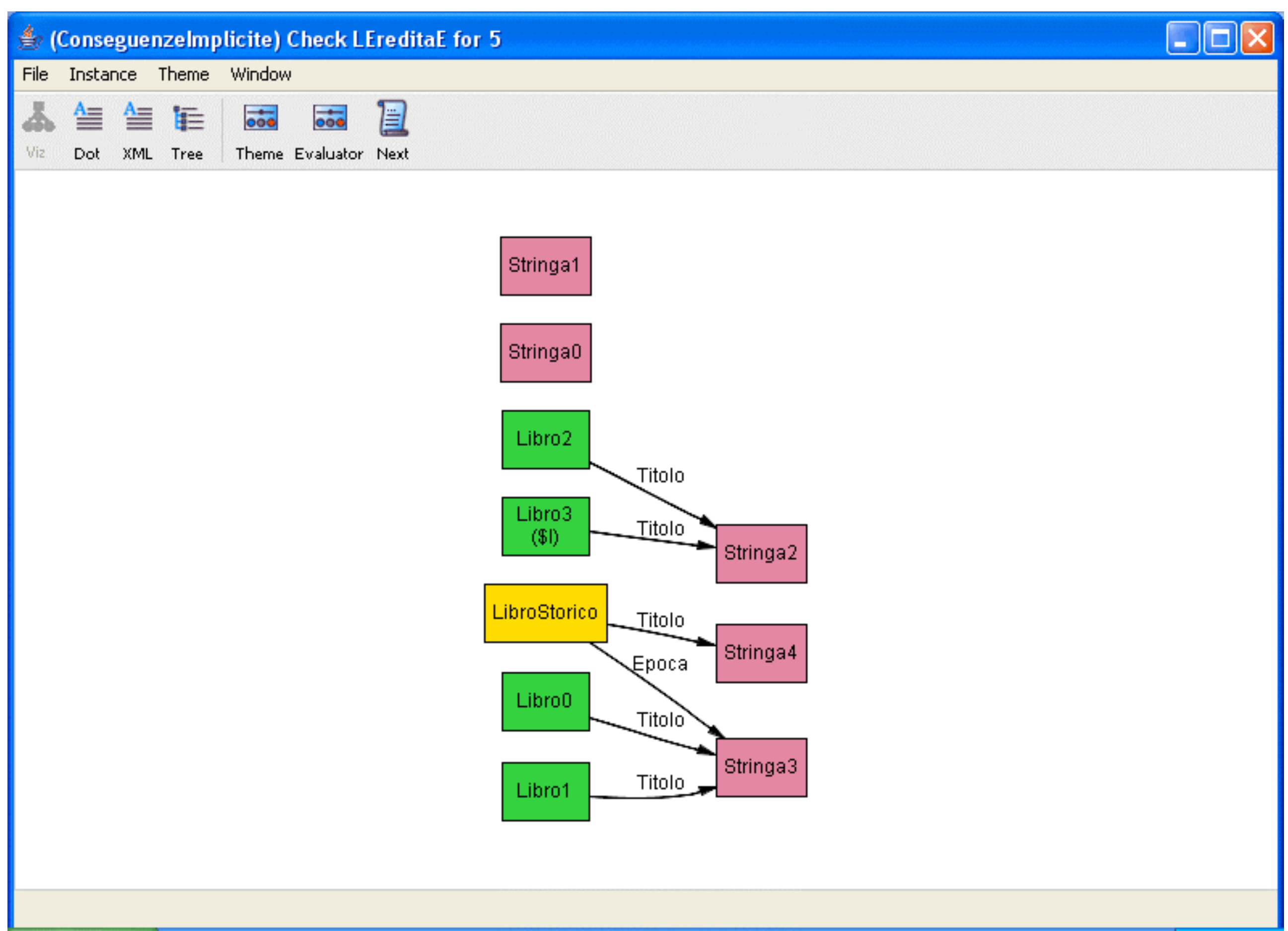
Possiamo effetuare le verifiche necessarie nei due linguaggi inserendo nuovo codice nel seguente modo:

```
% Conseguenze Implicite in sintassi Otter: Dichiarazioni

%%% C1 = LibroStorico eredita Titolo (true)
C1 <-> (
  all X Y ((LibroStorico(X) & Titolo(X,Y)) -> (Stringa(Y)))
).

%%% C2 & C3 = Titolo è monovalore in LibroStorico (true)
C2 <-> (
  all X ((LibroStorico(X)) -> (exists Y (Titolo(X,Y))))
).

C3 <-> (
  all X Y Z ((LibroStorico(X) & Titolo(X,Y) & Titolo(X,Z)) -> (Y = Z))
).

%%% C4 = Libro eredita l'attributo Epoca (false)
C4 <-> (
  all X Y ((Libro(X) & Epoca(X,Y)) -> (Stringa(Y)))
).

% Verifica in Otter, una riga alla volta

- (
        C1                  % finished in 0.55 secs (otter)
        % C2 & C3           % finished in 0.61 secs (otter)
        % C4                % finished in 0.19 secs  (mace2)
        % C1 & C2 & C3      % finished in 0.61 secs (otter)
).

-----------------------------------------------------------------

//Verifica di Conseguenze in sintassi Alloy

//LibroStorico eredita Titolo, che è monovalore (true assertion)
assert LSEreditaT  {
        all ls:LibroStorico  |   one s:Stringa  |
                ls.Titolo = s
}
check LSEreditaT for 5          //finished in 0.468 secs with default SAT solver

//Libro eredita Epoca (false assertion)
assert LEreditaE  {
        all l:Libro |  some s:Stringa  |
                l.Epoca = s
}
check LEreditaE for 5           //finished in 0.094 secs with default SAT solver
```

L'analisi ci fornisce una gran quantità di informazioni interessanti:

Relativamente ad Otter/Mace, abbiamo dovuto aggiungere delle formule ulteriori che poi abbiamo verificato; per le prime due il Theorem Prover ha dato esito positivo, con un tempo di 0.55 secondi e 0.61 secondi per le istruzioni separate, ed impiegando 0.61 secondi per C1, C2 e C3 congiunte (abbiamo aggiunto tale dato per il paragone con Alloy); riguardo alla terza formula è stato cercato un modello tramite il Model Finder con scope 5, che ha mostrato l'esistenza di un Libro che non eredita l'attributo Epoca, impiegando 0.20 secondi.

In Alloy ci siamo limitati a definire due assertions, cioè bodies contenenti vincoli che dovrebbero essere ridondanti, unificando le prime due conseguenze implicite da dimostrare.
L'Analyzer non è riuscito a trovare controesempi nello spazio di ricerca 5 che abbiamo indicato, come ci aspettavamo, ed ha impiegato 0.468 secondi con SAT4J, il SAT solver di default; riguardo alla seconda assertion, quella non valida, è riuscito a trovare un controesempio legale in appena 0.094 secondi, sempre con SAT4J.
Accenniamo appena alle performance di Alloy con ZChaff, un SAT solver che abbiamo visto nel corso di Metodi Formali: un ottimo 0.093 secondi sulla prima assertion e 0.047 secondi sulla seconda.
Di seguito è mostrata l'istanza di controesempio restuituita dall'Alloy Analyzer; si noti come venga indicato Libro3 come possibile istanza di Libro che non eredita Epoca, mediante la variabile 'l' utilizzata nella assertion LEreditaE:



Possiamo concludere l'analisi di questo case study con alcuni valori di performance comparison in percentuale:

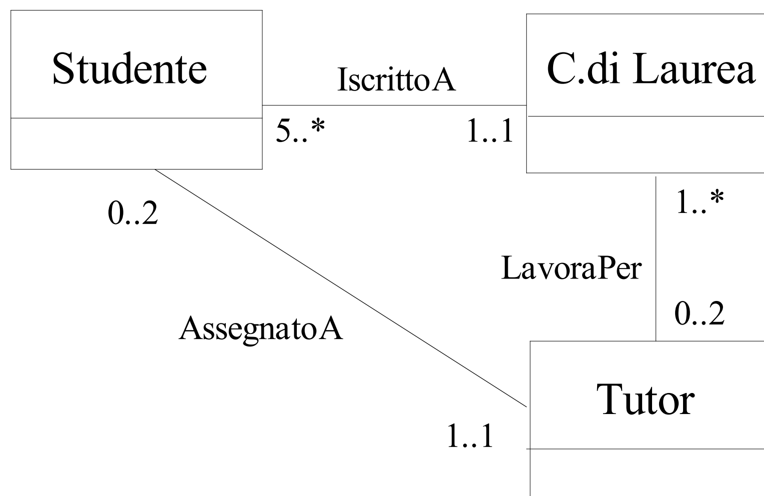
- Performance Comparison tra Alloy SAT4J e Alloy ZChaff:
    assert 1: ZChaff esibisce una performance del 503 % circa rispetto SAT4J;
    assert 2: ZChaff è veloce il 200 % circa rispetto SAT4J;

- Performance Comparison tra Alloy ZChaff e Otter/Mace:
    conseguenza 1 e 2: Alloy ha fornito una performance del 656 % circa rispetto ad Otter;
    conseguenza 3: Alloy è stato il 425 % circa più veloce rispetto a Mace.

Sia la lunghezza e chiarezza delle codifiche, sia le performance comparisons dovrebbero essere abbastanza eloquenti sul confronto tra i due linguaggi e tools.

****

Case Study 2: Class Consistency
Passiamo ora al problema della Consistenza di Classi, mostrando come un diagramma UML può, a causa dei vincoli imposti, vietare l'esistenza di istanze di una o più classi.
Considereremo stavolta uno scenario insoddisfacibile a causa di particolari molteplicità imposte nelle relazioni, evidenziando aspetti interessanti mentre ripetiamo la procedura di confronto dello studio di caso precedente.
Il diagramma in questione è il seguente:



Notiamo che l'origine del problema è la combinazione delle molteplicità imposte nelle varie associazioni: un Corso non può avere più di 2 Tutors, i quali non possono occuparsi di più di 2 Studenti ciascuno; inoltre ogni studente deve avere esattamente 1 Tutor, essere iscritto ad esattamente 1 Corso, ed ogni Tutor deve lavorare per almeno un Corso; da tutto ciò consegue che il massimo numero di studenti "coperti" in termini di Tutors da un Corso è 2*2 = 4, ma un Corso deve avere almeno 5 Studenti, e quindi non può esistere alcuna istanza finita del modello.

Procediamo alla formalizzazione in Otter/Mace, e quindi in Alloy:

% Formalizzazione in sintassi Otter

```
% Schema riepilogativo
% STUDENTE 5..* ------- IscrittoA ------- 1..1 CORSO_DI_LAUREA
% STUDENTE 0..2 ------- AssegnatoA ------- 1..1 TUTOR
% CORSO_DI_LAUREA 1..* ------- LavoraPer ------- 0..2 TUTOR

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Associations' type-checkings
all X Y ((IscrittoA(X,Y)) -> (Studente(X) & CorsoDiLaurea(Y))).
all X Y ((AssegnatoA(X,Y)) -> (Studente(X) & Tutor(Y))).
all X Y ((LavoraPer(X,Y)) -> (CorsoDiLaurea(X) & Tutor(Y))).

%%% Molteplicità di Studente
% IscrittoA 1..1 CorsoDiLaurea
all X ((Studente(X)) -> (exists Y (IscrittoA(X,Y)))).
all X Y Z ((Studente(X) & IscrittoA(X,Y) & IscrittoA(X,Z)) -> Y = Z).
% AssegnatoA 1..1 Tutor
all X ((Studente(X)) -> (exists Y (AssegnatoA(X,Y)))).
all X Y Z ((Studente(X) & AssegnatoA(X,Y) & AssegnatoA(X,Z)) -> Y = Z).

%%% Molteplicità di CorsoDiLaurea
% IscrittoA 5..* Studente
all X ((CorsoDiLaurea(X)) -> (exists Y Z W K J
                    (IscrittoA(Y,X) & IscrittoA(Z,X) & IscrittoA(W,X) &
                     IscrittoA(K,X) & IscrittoA(J,X) &
                    (Y != Z) & (Y != W) & (Y != K) & (Y != J) &
                    (Z != W) & (Z != K) & (Z != J) &
                    (W != K) & (W != J) &
                    (K != J)))).
% LavoraPer 0..2 Tutor
all X Y Z W ((CorsoDiLaurea(X) & LavoraPer(X,Y) & LavoraPer(X,Z)
                               & LavoraPer(X,W)) -> ((Y = Z) | (Y = W) | (Z = W))).

%%% Molteplicità di Tutor
% AssegnatoA 0..2 Studente
all X Y Z W ((Tutor(X) & AssegnatoA(Y,X) & AssegnatoA(Z,X)
                                      & AssegnatoA(W,X)) ->
                               ((Y = Z) | (Y = W) |
                                (Z = W))).
% LavoraPer 1..* CorsoDiLaurea
all X ((Tutor(X)) -> (exists Y (LavoraPer(Y,X)))).

% Disgiunzione tra classi non legate da gerarchie
all X -(Studente(X) & CorsoDiLaurea(X)).
all X -(CorsoDiLaurea(X) & Tutor(X)).
all X -(Studente(X) & Tutor(X)).

% Ogni oggetto appartiene ad almeno una classe
all X (Studente(X) | CorsoDiLaurea(X) | Tutor(X)).
```

```
----------------------------------------------------------------

//Formalizzazione in sintassi Alloy

sig Studente  {
        IscrittoA: one CorsoDiLaurea,
        AssegnatoA: one Tutor
}

sig CorsoDiLaurea  {
        IscrittoA: set Studente,
        LavoraPer: set Tutor
}  {  #IscrittoA >= 5 &&
     #LavoraPer =< 2
   }

sig Tutor  {
        LavoraPer: some CorsoDiLaurea,
        AssegnatoA: set Studente
} {  #AssegnatoA =< 2  }

----------------------------------------------------------------

fact associationIscrittoA  {
        all s:Studene, c:CorsoDiLaurea |
                (c in s.IscrittoA) <=> (s in c.IscrittoA)
}

fact associationAssegnatoA  {
        all s:Studene, t:Tutor |
                (t in s.AssegnatoA) <=> (s in t.AssegnatoA)
}

fact associationLavoraPer  {
        all t:Tutor, c:CorsoDiLaurea |
                (c in t.LavoraPer) <=> (t in c.LavoraPer)
}
```

Ancora una volta è il caso di notare la sensibilmente maggior brevità ed eleganza del codice Alloy rispetto alla formalizzazione in FOL di Otter; salta particolarmente all'occhio il grande numero di confronti nelle formule che descrivono molteplicità minima e/o massima: per un valore di molteplicità pari ad N il numero di confronti in FOL cresce come N^2 (più precisamente come il noto N*(N-1)/2), mentre in Alloy è sufficiente aggiungere un appended fact che imponga un vincolo di cardinalità.
Notiamo anche la comodità delle molteplicità speciali di Alloy, come "some" e "one", che ci ha permesso di modellare alcune associazioni con il minimo sforzo.
Come di consueto Alloy richiede dei constraints aggiuntivi per "adattare" il concetto di Associazione a quello di campo interno di una sig, ma lo sforzo di aggiungere un fact per ogni associazione è ampiamente ripagato dai vantaggi ottenuti: il tempo impiegato in questa sede per scrivere, correggere e testare il file Otter ha richiesto circa 5 volte il tempo dedicato ad Alloy.

Procediamo ora alla (vana, come si è mostrato) ricerca di un'istanza legale del nostro modello utilizzando i due tools.

La ricerca è piuttosto semplice: useremo Mace per ricercare un modello in cui esista una qualche istanza di Studente; in Alloy useremo un'assertion per cercare di ottenere un'istanza legale e non vuota.

```
% Ricerca di un modello finito tramite Mace

% Esiste almeno uno studente (false assertion)
someStudents <-> (
    (exists X Studente(X))
).

% Tesi
-(
    -someStudents     % mace2 non trova alcun modello
                      % time taken su scope 10 = 8.81 secs
).


-----------------------------------------------------------------

//Ricerca di un'istanza legale con almeno uno Studente in Alloy

assert noStudents  {
        no Studente
}
check noStudents for 10 int   //no counterexample found; time = 0.938 secs with SAT4J
                              //                         time = 0.844 secs with ZChaff
```

Come ci aspettavamo i due tools non sono stati in grado di trovare un modello. Si tenga presente che nel caso non sapessimo a priori dell'inconsistenza del diagramma, e a patto che le formalizzazioni fossero corrette, l'informazione fornita non sarebbe certo stata irrilevante, ma sintomo di un serio problema di Class Consistency. Il controllo può essere ripetuto in maniera del tutto analoga per le altre classi presenti.

La keyword "int" è stata aggiunta nel comando check per ampliare la bitwidth degli interi che l'analizzatore può trattare, in modo da permettere un'analisi corretta del modello.

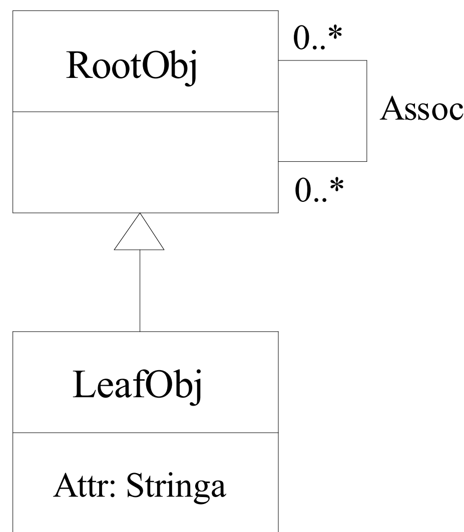
Per quanto riguarda le performances, stavolta il solver ZChaff ha esibito una performace del solo 111% circa rispetto a SAT4J; quest'ultimo è risultato essere superiore a Mace in misura del 939%, mentre ZChaff del 1044% circa.

Sezione 5.3 - Beyond Otter: La [Reflexive] Transitive Closure

Una potenzialità offerta dal linguaggio Alloy e non presente in Otter/Mace è data dai due operatori unari per la Non-Reflexive Transitive Closure o Chiusura Transitiva Non Riflessiva (^expr) e la Reflexive Transitive Closure o Chiusura Transitiva Riflessiva (*expr); entrambi si applicano ad un'espressione relazionale.
Qual è la semantica di tali operatori? Essi restituiscono l'insieme di tutti gli atomi ottenibili applicando un certo numero arbitrario di volte (precisamente *zero o più volte* per la riflessiva, *una o più volte* per la non riflessiva) la relazione expr.

Tale funzionalità può risultare utile in determinati scenari in cui siamo in presenza di relazioni tra insiemi non disgiunti e vogliamo imporre determinati vincoli alle relazioni legali. Particolarmente conveniente è in tali casi l'uso dell'operatore di subset *in*.

Ipotizziamo il seguente scenario:



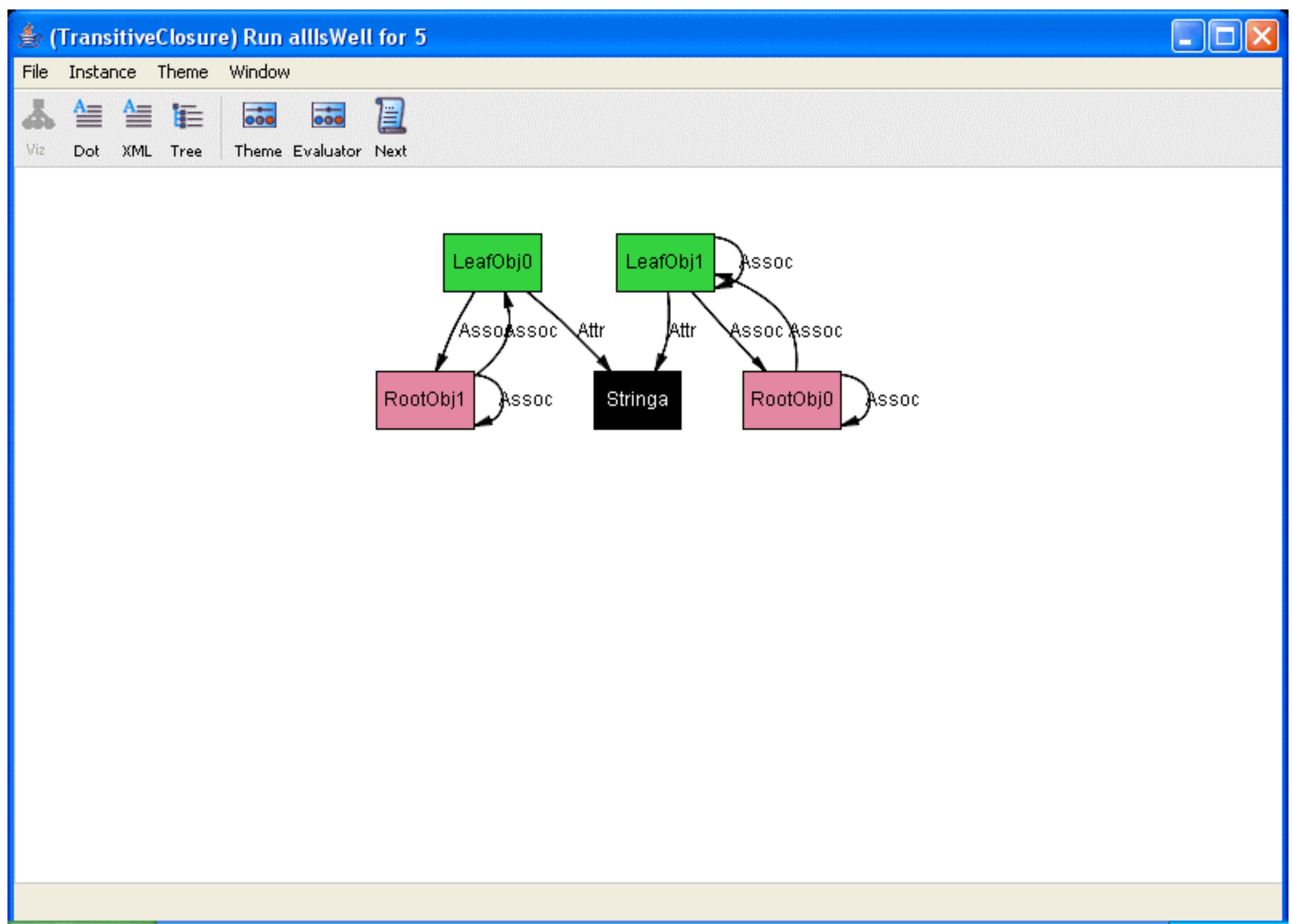
Assumiamo di avere il vincolo aggiuntivo: "Ogni istanza di RootObj può condividere l'associazione Assoc solo con RootObjs che non sono anche LeafObjs, mentre ogni istanza di LeafObj può condividere Assoc solo con altre istanze di LeafObj". In altre parole l'associazione Assoc deve insistere su classi del medesimo tipo.

Supponiamo di codificare questo scenario in Alloy nel seguente modo:

```
//signatures

sig RootObj  {
        Assoc: set RootObj
}

sig LeafObj extends RootObj  {
        Attr: Stringa
}

sig Stringa  {}

//vincolo per l'associazione
```

```
fact association  {
        all r1,r2: RootObj  |
                (r2 in r1.Assoc) <=> (r1 in r2.Assoc)
}

//imponiamo anche nessuna stringa isolata, per comodità
fact noStrayStrings  {
        all s:Stringa  |   some l:LeafObj  |
                s in l.Attr
}

//empty predicate per trovare un'istanza valida
pred allIsWell []  {
        #RootObj >= 3 && #LeafObj >=1
}
run allIsWell for 5
```

Ovviamente, non avendo inserito nulla per modellare il nostro vincolo aggiuntivo, otterremmo istanze indesiderate come la seguente, in cui atomi LeafObj partecipano a relazioni Assoc:



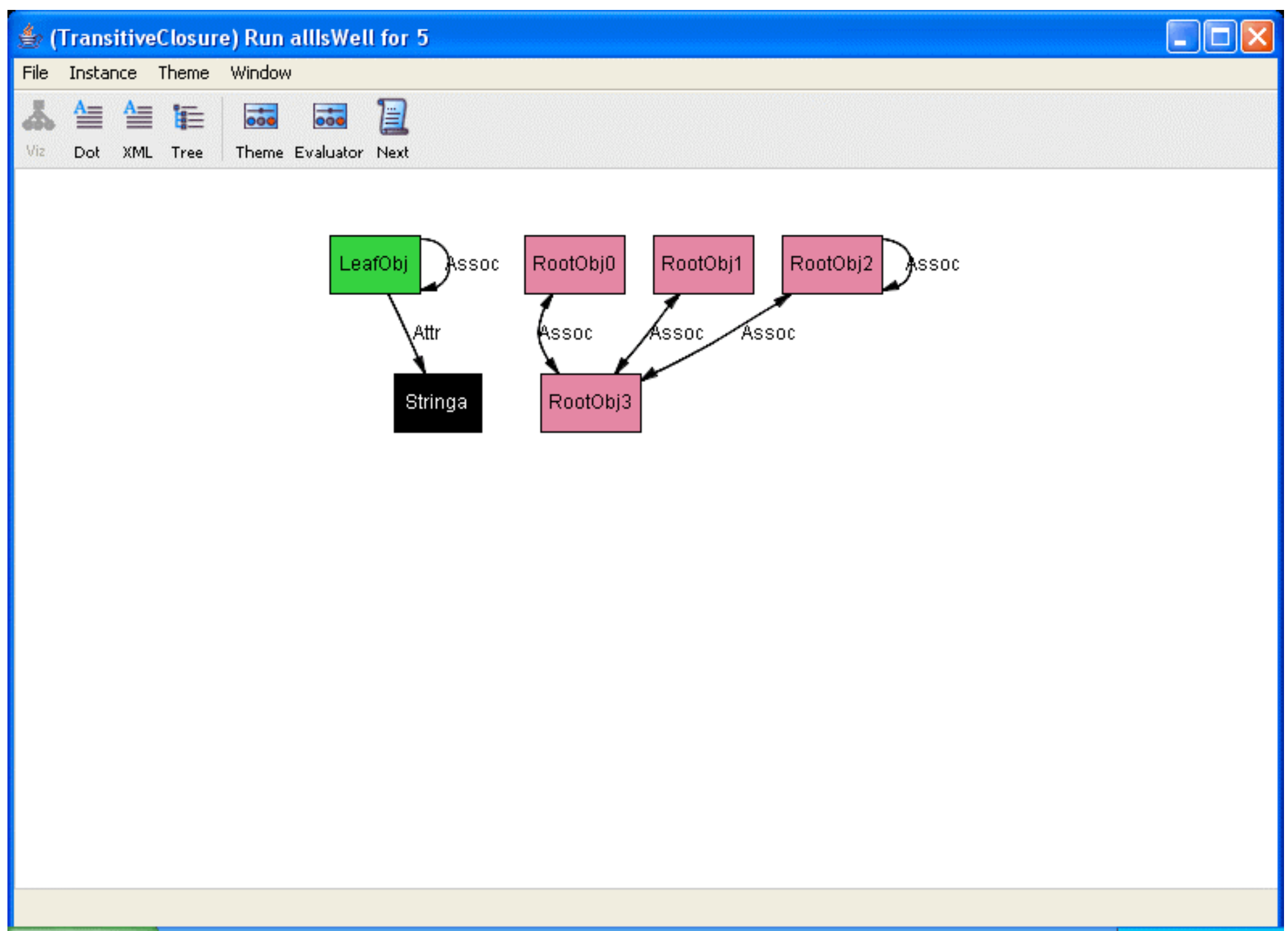
Potremmo usare, in questa circostanza, l'operatore per la Chiusura Transitiva aggiungendo il seguente vincolo di tipo Fact:

```
fact constraintViaRTC  {
```

```
all r:RootObj, d: r.*Assoc  |
        ((r not in LeafObj) => (d not in LeafObj)) &&
        ((r in LeafObj) => (d in LeafObj))
}
```

Abbiamo affermato che per ogni atomo r di tipo RootObj (il quale quindi potrebbe anche essere un LeafObj), e *per ogni atomo d appartente all'insieme ottenuto applicando zero o più volte la relazione Assoc*, tale atomo d deve appartenere alla stessa classe di r: la relazione può esistere solo tra RootObjs oppure tra LeafObjs.
Si noti come in questo caso era indifferente utilizzare la Chiusura Riflessiva o Non Riflessiva: infatti il vincolo impone che seguendo le associazioni Assoc si "rimanga" sempre su atomi della stessa classe; è ovvio che applicando la relazione zero volte insistiamo sempre sulla stessa classe, pertanto in questo caso l'applicazione "zero o più volte" equivale a "una o più volte".

Aggiungendo la nostro modulo il vincolo in questione otteniamo istanze di questo tipo:



Com'è facile constatare, ora il modello soddisfa il vincolo esterno originale, e l'Associazione coinvolge soltanto classi dello stesso tipo.

Conclusioni

In questa relazione abbiamo effettuato un'analisi critica del linguaggio Alloy, basato su FOL, e del tool associato, l'Alloy Analyzer.
Abbiamo visto come l'universo di Alloy sia interamente composto da atomi e relazioni tra atomi, e come sintassi e semantica di Alloy investano, convenientemente, ambiti plurivalenti come la teoria degli insiemi e i linguaggi Object-Oriented.
Successivamente è stata data una overview sulla quasi totalità dei costrutti di Alloy e le features rese disponibili dal tool di Analisi.
L'attenzione si è quindi focalizzata su come sia possibile utilizzare Alloy per formalizzare gli aspetti dei diagrammi UML visti nel corso di Metodi Formali nell'Ingegneria del Software; le caratteristiche del linguaggio e il più alto livello di astrazione hanno permesso una modellazione in generale più sintetica ed intuitiva rispetto al tradizionale FOL, nonché facilmente analizzabile con ottime performances e consultabile in modo user-friendly tramite l'Analyzer.
L'ultima parte del lavoro ha affrontato l'argomento delle possibili casistiche di verifica di proprietà delle formalizzazioni ottenute, mettendo in risalto le prestazioni e l'ease-of-use relativamente all'altro tool per FOL visto nel corso, Otter/Mace. Si è anche illustrato un operatore, la chiusura transitiva, che rende Alloy più espressivo del linguaggio di Otter/Mace.
Abbiamo anche evidenziato, dove incontrati, i problemi legati all'uso di Alloy, come ad esempio la documentazione non sempre chiara e completa, la difficoltà nello scovare overconstraining e l'occasionale difficoltà di uso di alcuni costrutti in determinate situazioni, come ad esempio la modellazione di relazioni di arità maggiore o uguale a tre.

Bibliografia

- Dispense del corso di Metodi Formali nell'Ingegneria del Software, prof. T.Mancini, anno 2006-07;

- Tutorials dal sito ufficiale di Alloy: "http://alloy.mit.edu/";

- The Alloy3 Reference Manual (il manuale per Alloy 4.0 è in questo momento ancora TBA), disponibile sul sito ufficiale.