Alloy e l'Alloy Analyzer

Tesina per il corso di Metodi Formali
nell'Ingegneria del Software
Anno 2006-07
Docente: prof. T.Mancini

Autore: Alessandro Pagliaro

Overview

- Nozioni di Base su Alloy
- Sintassi e Semantica dei costrutti Alloy
- Il tool Alloy Analyzer
- Formalizzazione in Alloy di UML Diagrams
- Verification Case Studies, Confronto Otter/Alloy

Alloy Basics

- Linguaggio Dichiarativo basato su FOL
- Da codice Alloy a istanza SAT a SAT-solver
- Vari SAT-solvers disponibili
- Universo di Alloy = Atomi e Relazioni
- Sintassi flessibile e vicina a FOL, Linguaggi OO nonché Teoria degli Insiemi
- Supporto per espressioni intere
- Costrutti che facilitano la modellazione (ereditarietà, molteplicità note, type-checking automatico...)
- Tool che offre efficaci mezzi per esplorare le istanze

Sintassi e Semantica di Alloy

Cosa vedremo

- Espressioni
- Formule
- Quantificatori e Multiplicity Keywords
- Moduli
- Signatures
- Facts
- Predicati e Funzioni
- Assertions
- Comandi: Check e Run

Espressioni

In Alloy esistono due tipi di espressioni: Espressioni Relazionali, che denotano un insieme di atomi, ed Espressioni Intere, di valore numerico.
Sono espressioni: variabili, Signatures (insiemi), costanti relazionali (none, univ), composizioni di espressioni tramite operatori.
Operatori Rel.: Unione (+), Differenza (-), Inters. (&) di insiemi, join (.), prodotto(->), Chiusura Trans.(^,*)
Operatori Interi: Somma (+), Differenza(-), Cardinalità (#).

Espressioni: Esempi

a

Persona

Persona - Anziano

Persona.nome

Persona.figli

Directory ^contents

Formule

Per Formula si intende l'applicazione di quantificatori ad espressioni relazionali o la comparazione di espressioni relazionali o intere ---> valore true/false

Operatori: Uguaglianza (numerica o insiemistico - estensionale, =), Subset (in), Not(not, !), And (and, &&), Or (or, ||), Se e solo se (Iff, <=>), Implicazione (implies, =>), confronti numerici (<, >, =<, >=).

Formule: Esempi

a in Persona

Maschio + Femmina = Persona

7 + 4 < 11

#Persona >= 10

Quantificatori

I quantificatori sono keywords speciali che servono ad esprimere i concetti di "per ogni" ed "esiste", nonché altre molteplicità comuni nell'informatica.

Utilizzo: <quantifier> varName: sigName | Formula
Quantificatori: all, some (1..*), one (1..1), lone (0..1), set(0..*), no.

I Quantificatori possono essere usati insieme ad una grande varietà di costrutti.

Quantificatori: Esempi

all p:Persona | p in EssereVivente

all p:Persona | some s:Stringa | p.nome = s

no Persona

Persona.nickname = lone Stringa

Moduli

Un modulo corrisponde ad un file Alloy (.als).
Un modulo può importare ed usare altri moduli, ma è l'unico analizzabile. Si tratta di una sorta di contenitore per le istruzioni dichiarative.

Struttura:

- Header (file path, module name);
- Imports (come gli import di Java o gli include di C);
- Paragraphs (Signatures, Facts, Predicates, Functions, Assertions, Commands).

Signatures

Costrutto fondamentale in Alloy: denota un insieme di atomi, i quali condividono le stesse relazioni e gli stessi vincoli. Analogo ai concetti di insieme, classe Java o classe UML. Abbreviato come "sig".

Sintassi:

[mult][abstract] sig sigName [extends | in SigRef] {
    relazione1: [mult] SigRef1
    relazione2: [mult] SigRef2...
} { <appended facts> }

Signatures: Esempi

```
sig Persona  {}
```

```
one sig Persona  {}
```

```
sig Stringa {}
   sig Persona  {  nome: one Stringa, tel: set Stringa }
```

```
sig Persona  {  tel: set Stringa }
```

Facts

Insieme di formule intese come vincoli globali del modello che devono essere sempre rispettati in ogni istanza legale. Presentano alto rischio di overconstraining. Gli "appended facts" valgono solo per la sig in cui sono dichiarati.

Sintassi:

```
fact  factName   {
        <formule di constraint>
}
```

Facts: Esempio

```
sig Persona  {}
sig Operoso in Persona  {}
sig Nullafacente in Persona  {}

//impongo disjointness e completeness della gerarchia
fact disjointAndComplete  {
        Operoso + Nullafacente = Persona
                &&
        Operoso & Nullafacente = none
}
```

Predicati

Per predicato si intende un insieme di constraints che deve valere solo quando invocato (da una formula o da un comando), ha valore true o false, e dipende da alcuni parametri in input.
Può essere utile invocare con i comandi un predicato vuoto, per ottenere un'istanza legale del modello.

Sintassi:

```
pred predName [param1,...,paramN]   {
        <formule di constraint>
}
```

Funzioni

Molto simili ai predicati, ma sono una "template expression" (e non constraint), cioè denotano un valore (un intero, o un insieme di atomi) e non un booleano.

Sintassi:

```
fun  funName [param1,...,paramN]: returnDomain   {
        <espressione dipendente dai params>
}
```

Uso di Predicati e Funzioni

Sono costrutti molto utili per la modellazione di problemi con aspetti dinamici (ad esempio operazioni di move o delete in un Filesystem model, o risoluzione del problema del River Crossing, nel senso di trovare una sequenza di stati legali che portano il sistema uno stato desiderato: predicati definiscono le funzioni di transizione).
Il loro uso esula dai nostri scopi di modellazione di proprietà statiche e verifica.

Assertions

Per Assertion si intende 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.

Sintassi:

```
assert  assertionName   {
        <list of  "challenge constraints">
}
```

Comandi (Run e Check)

Alloy mette a disposizione 2 comandi per richiedere al tool di effettuare un'analisi.
Run: trova un'istanza del modello che soddisfi un certo predicato o una certa funzione.
Check: trova un controesempio legale alla mia asserzione.

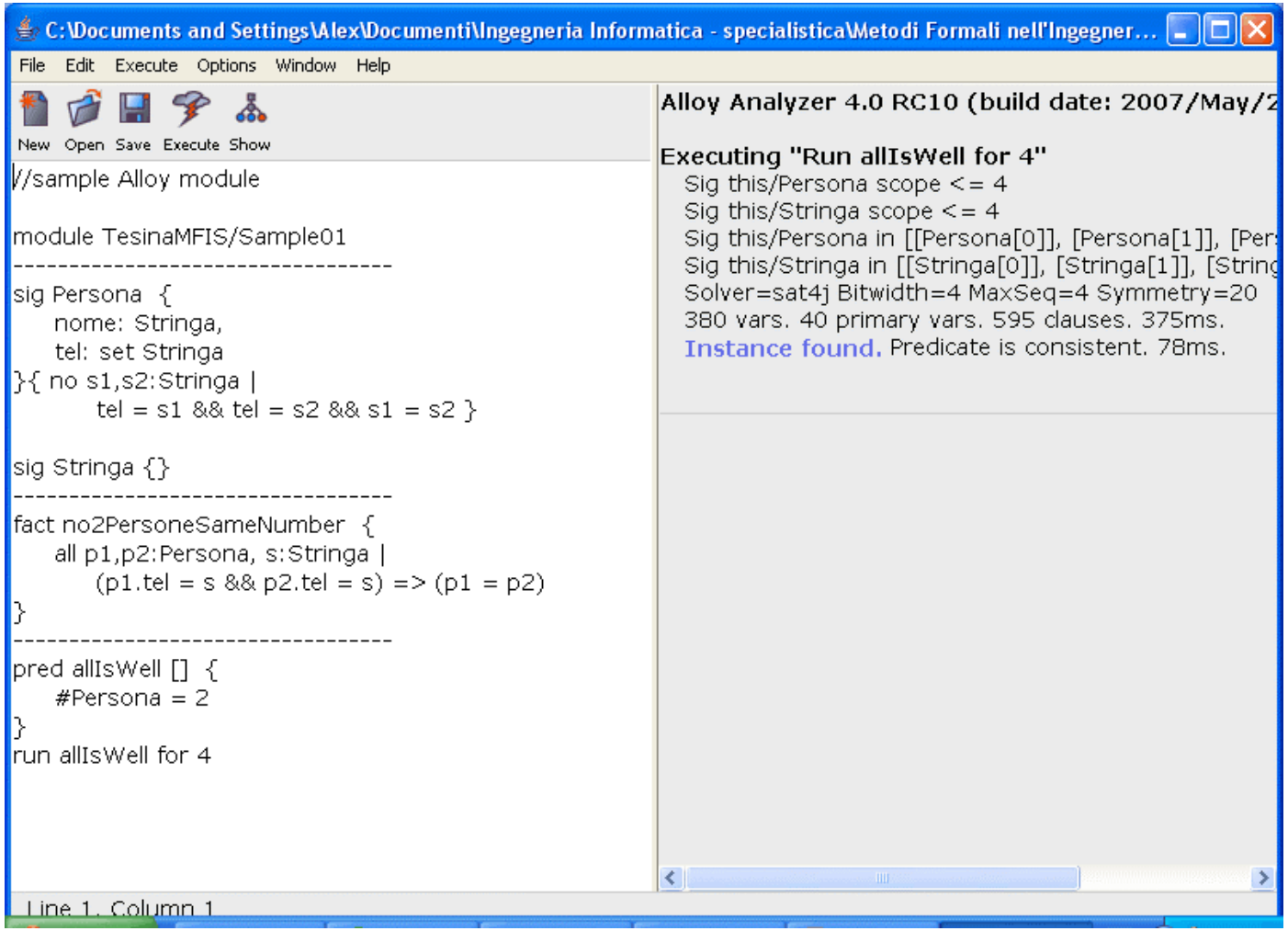
Sintassi (semplificata, analoga per entrambi):
run | check pred/fun/assertName for <int scope>

Esempio d'Uso: Assert + Check

```
sig Persona  {
        nome: lone Stringa  //0..1
}
sig Stringa  {}

assert allHaveNames  {
        all p:Persona | some s:Stringa |
                p.nome = s
}
check allHaveNames for 5  //l'asserzione è falsa
```

Il tool Alloy Analyzer

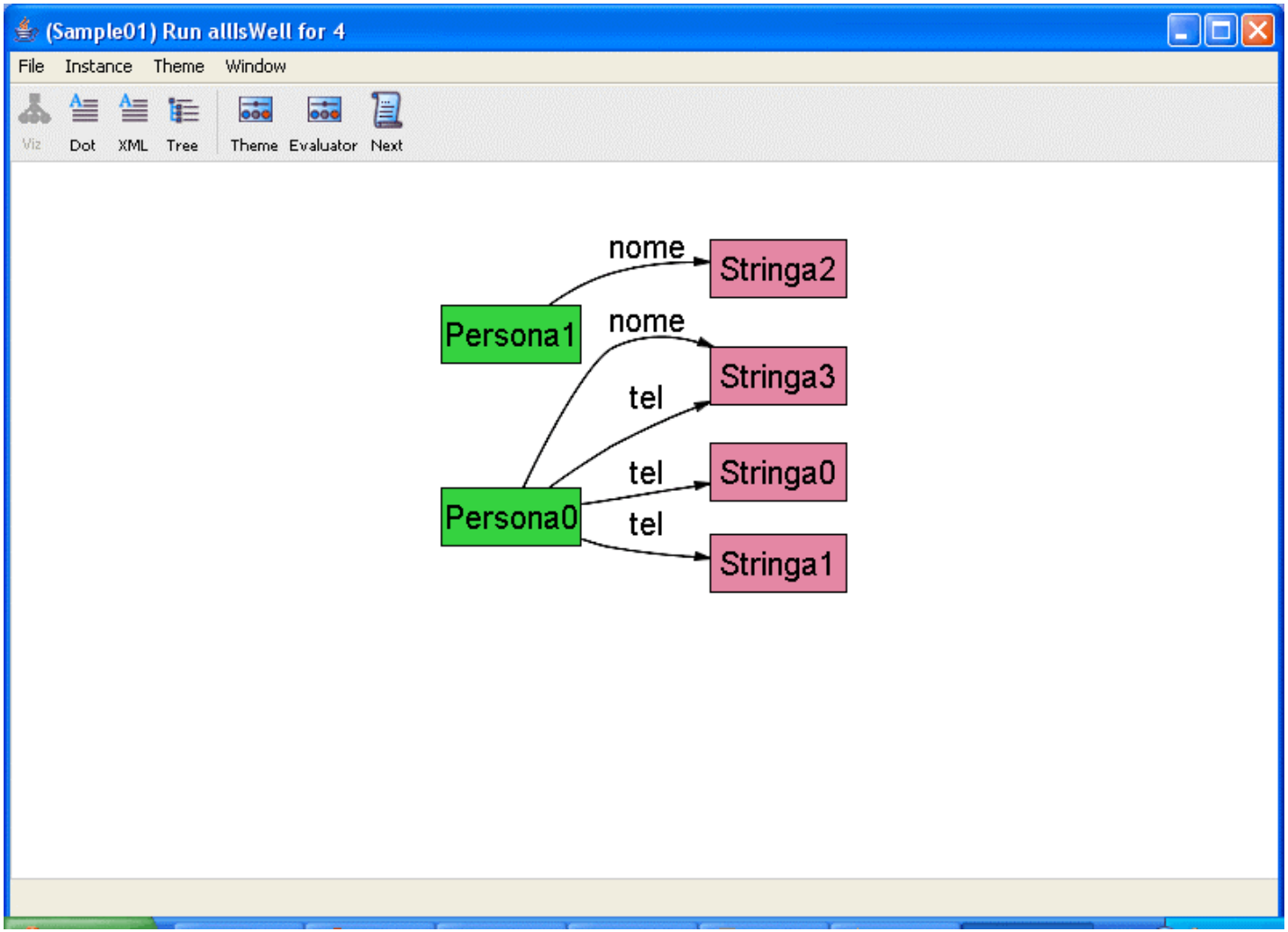
Schermata Principale



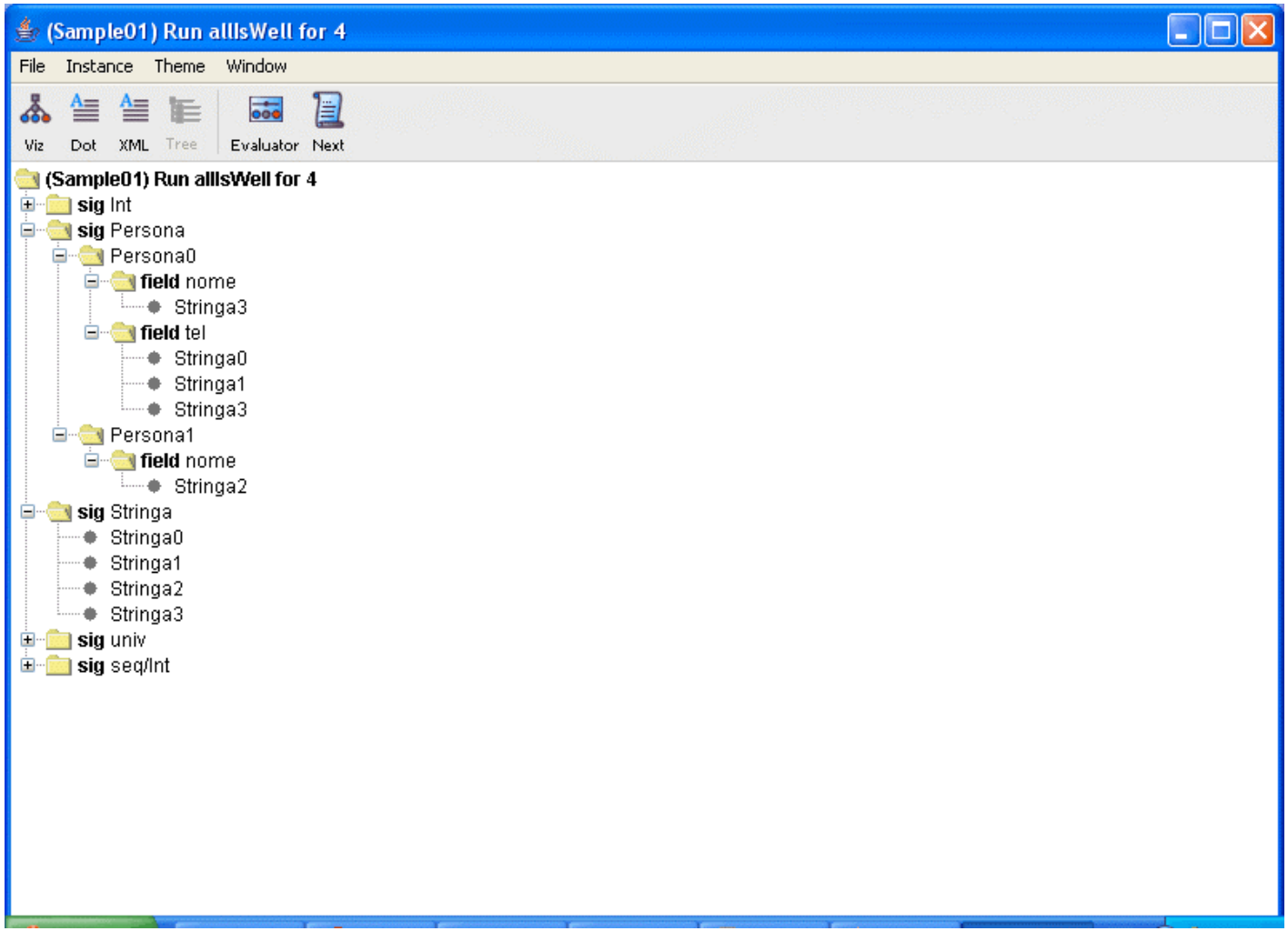
Alloy Analyzer features

- Split Screen: Text Editor + Message Box
- Informazioni su tempo impiegato, variabili usate...
- Links che conducono all'ispezione delle istanze
- 3 Modalità di visualizzazione: VIZ, TREE, XML
- Possibilità di importare "utility modules"
- Possibilità di visualizzare un Metamodel, ovvero una sorta di UML diagram del modello
- Possibilità di scegliere tra 6 SAT solvers, tra cui ZChaff
- Customizzazione delle istanze visuali

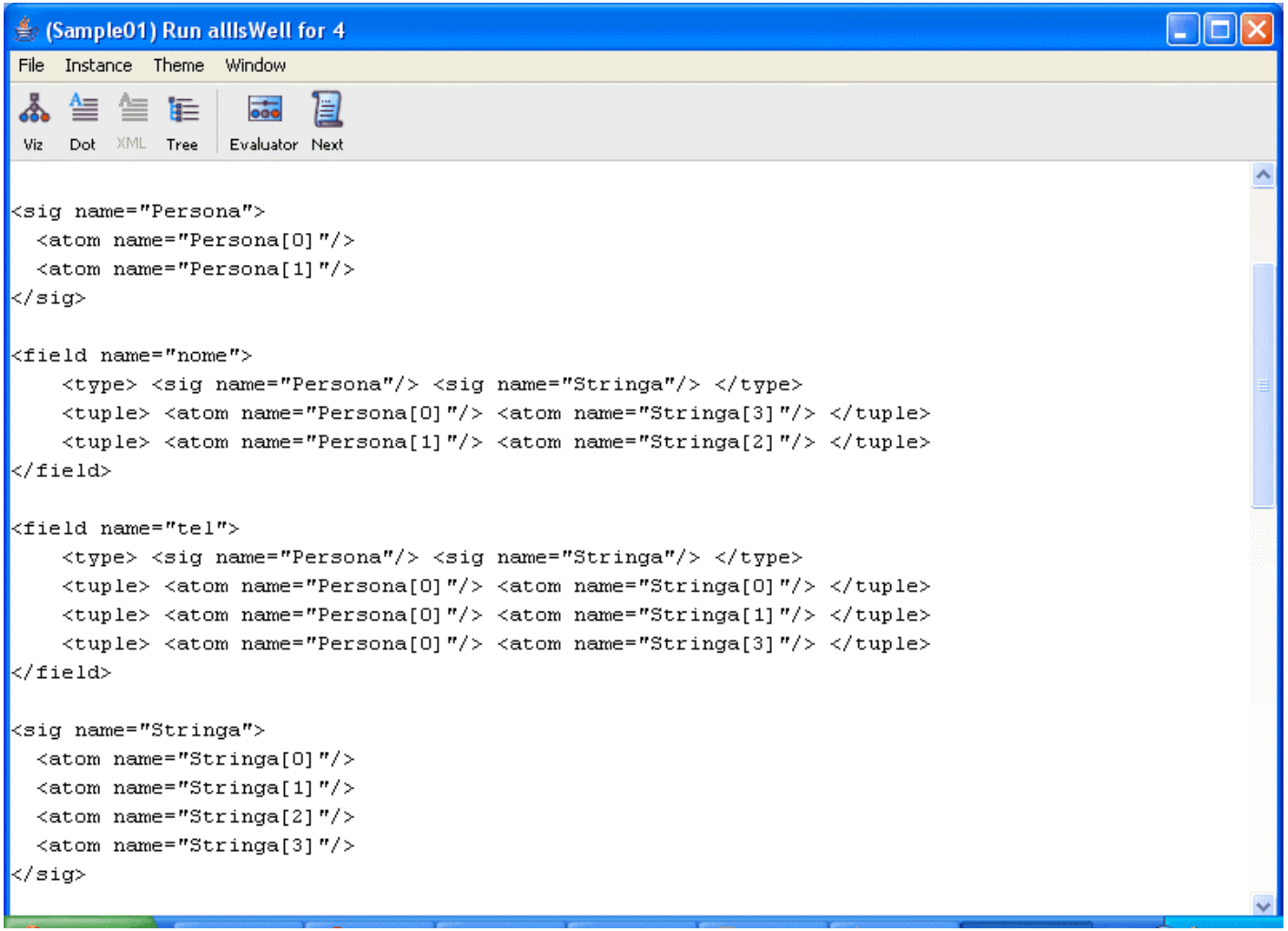
The VIZ View


(Sample01) Run allIsWell for 4
File
Instance
Theme
Window
Viz
Dot
XML
Tree
Theme
Evaluator
Next
nome
Stringa2
Persona1
nome
Stringa3
tel
tel
Stringa0
Persona0
tel
Stringa1

The Tree View



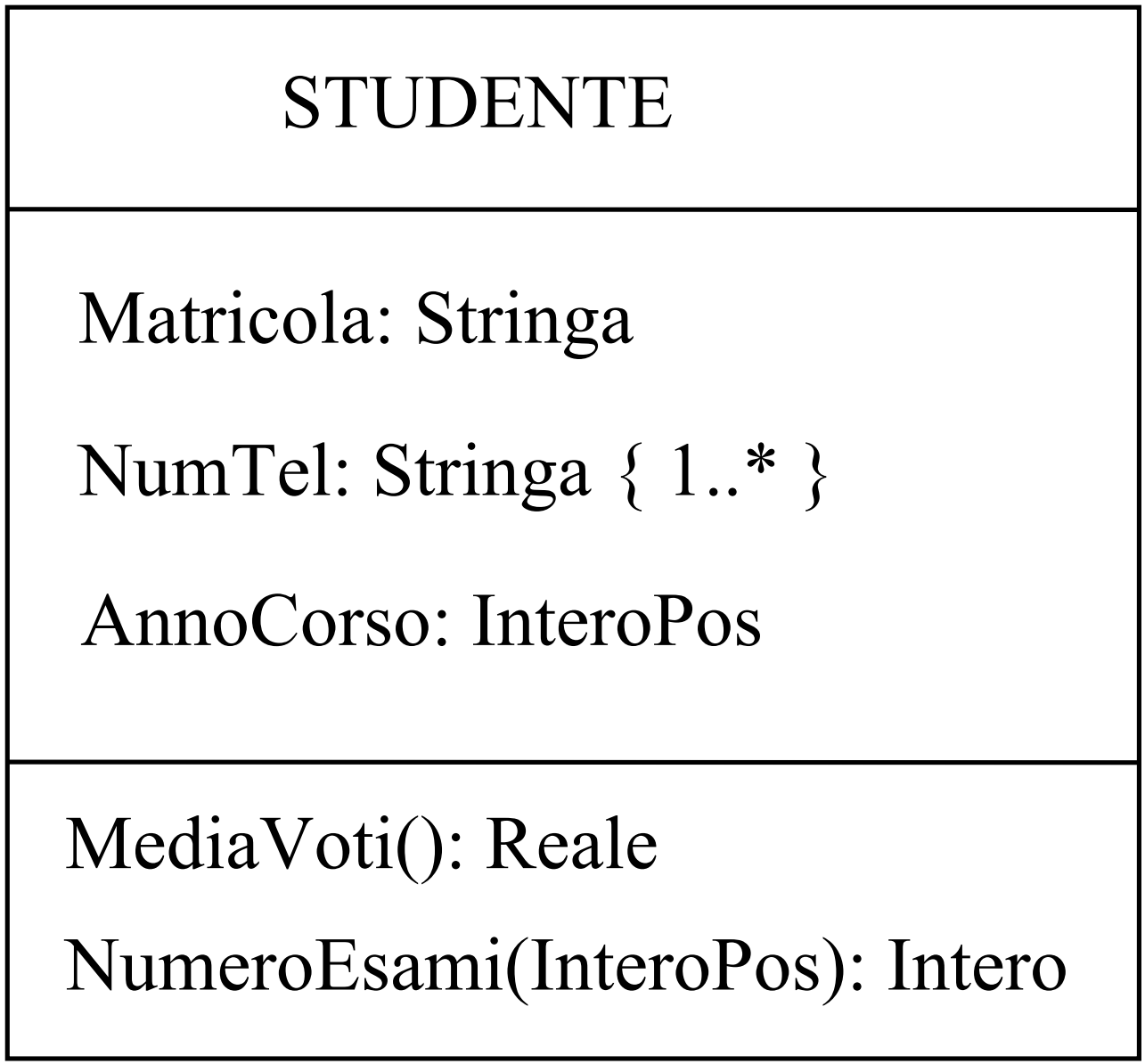
The XML View


(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>

Modellazione in Alloy di elementi degli UML Class Diagrams

Cosa vedremo

- Classi UML e Tipi
- Attributi di Classe (con molteplicità)
- Operazioni di Classe
- Associazioni binarie e loro molteplicità
- Generalizzazioni (IS-A)
- Gerarchie (disjoint e/o complete)
- Specializzazioni di Attributi
- Specializzazioni di Associazioni


STUDENTE
Matricola: Stringa
NumTel: Stringa { 1..* }
AnnoCorso: InteroPos
MediaVoti(): Reale
NumeroEsami(InteroPos): Intero

Modellazione di Classi UML

- *Classe UML* = Predicato Unario in FOL
- Mapping "naturale" sulle sigs
- Nel nostro caso: sig Persona {}
- Semplifica tutti i vincoli con type-checking

- *Tipo UML* = ancora un predicato unario in FOL
- Anche i tipi possono essere espressi con delle sigs, solitamente dal body vuoto
- esempio: sig Stringa {}

Attributi di Classe

Per modellare *attributi* in FOL dobbiamo avere:

- Un predicato binario che lega classe e attributo
- Type-checking sull'attributo
- Vincoli di molteplicità min e max (Theta(n^2) confronti, n è il valore della molteplicità)

Il concetto di attributo può essere modellato in Alloy come campo interno alla sig corrispondente alla classe dell'attributo, con le opportune molteplicità.
Vediamo alcuni esempi di confronto.

Attributi di Classe: Esempi

In FOL:

Studente/1 Stringa/1 hasNumTel/2

- for all X,Y

  Studente(X) && hasNumTel(X,Y) -> Stringa(Y)

- for all X Studente(X) -> exists Y hasNumTel(X,Y)

In Alloy:

```
sig Studente  {
        NumTel: some Stringa
}
sig Stringa  {}
```

Attributi di Classe: Esempi 2

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

sig CasoGenerale  {  //moltepl. X..Y
        attr: set X
} {  #attr >= X && #attr =< Y  } //molto conciso
```

Operazioni di Classe

Per modellare *operazioni* in FOL ci servono:

- Un predicato per la segnatura dell'op., di arità n+1 o n+2, contenente <this, n params, ret value>
- Type-checking su tutti i parametri dell'op
- Vincolo per esprimere l'unicità del risultato

Essendo un'op. una funzione n-aria, si tratta di un caso particolare di relazione, e può quindi essere modellata in modo simile agli attributi.

Operazioni di Classe: Esempi

In FOL:

Studente/1 Intero/1 InteroPos/1 numEsami/3

- for all X, P, R numEsami(X,P,R) ->
  Studente(X) & InteroPos(P) & Intero(R)

- for all X, P, R1, R2
  numEsami(X,P,R1) & numEsami(X,P,R2) ->
  R1 = R2

Operazioni di Classe: Esempi 2

In Alloy:

```
sig Intero  {}     sig InteroPos extends Intero  {}

sig Studente  {
        opNumEsami: InteroPos -> one Intero
}

sig CasoGenerale  {  //opX(this, P1...Pn, R)
        opX: P1 -> ( P2 -> ( ... Pn-1 -> ( Pn -> one R)
}
```

La Classe Studente in Alloy

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

        opMediaVoti: one Reale,
        opNumEsami: InteroPos -> one Intero
}
sig Stringa  {}   sig Intero  {}   sig Reale  {}
sig InteroPos extends Intero  {}
```

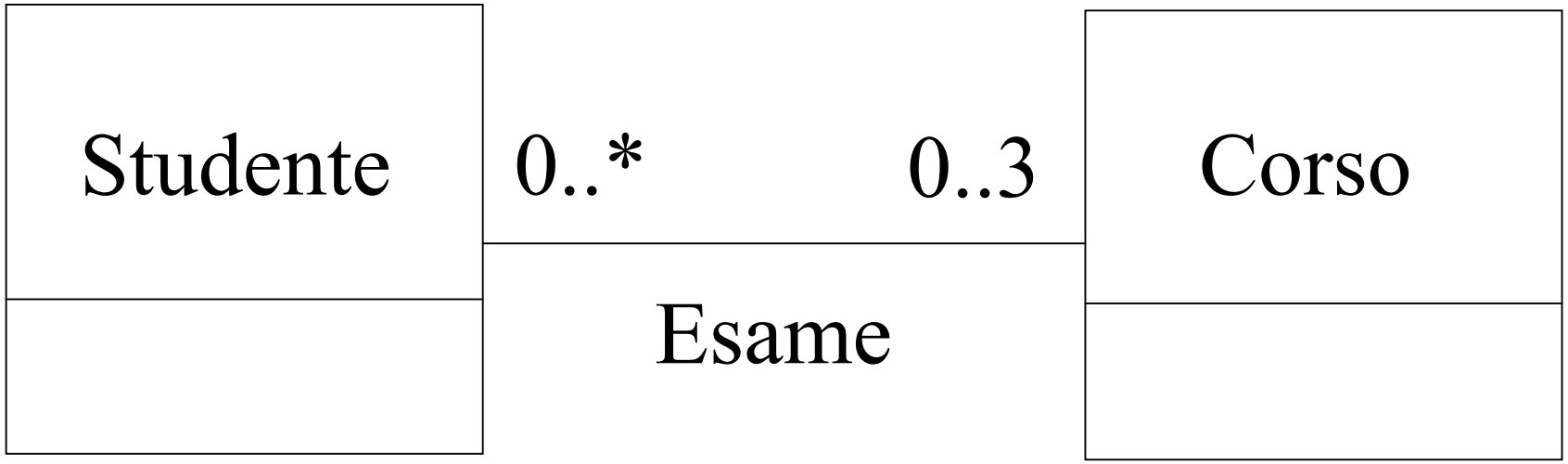
Modellazione di Associazioni

Per modellare *associazioni* in FOL c'è bisogno di:

- Un predicato n-ario per l'associazione n-aria
- Type-checking su tutte le classi coinvolte
- Vincoli sulle molteplicità come per gli attributi

Ancora una volta siamo di fronte ad un caso particolare di relazione: possiamo modellarla come campo interno, ma dobbiamo replicarla su tutte le classi coinvolte, con vincoli globali per includere tutte le classi coinvolte.

Esempio di Associazione



Esempio di Associazione 2

In FOL:

Studente/1 Corso/1 Esame/2

- for all X,Y Esame(X,Y) ->
  Studente(X) & Corso(Y)

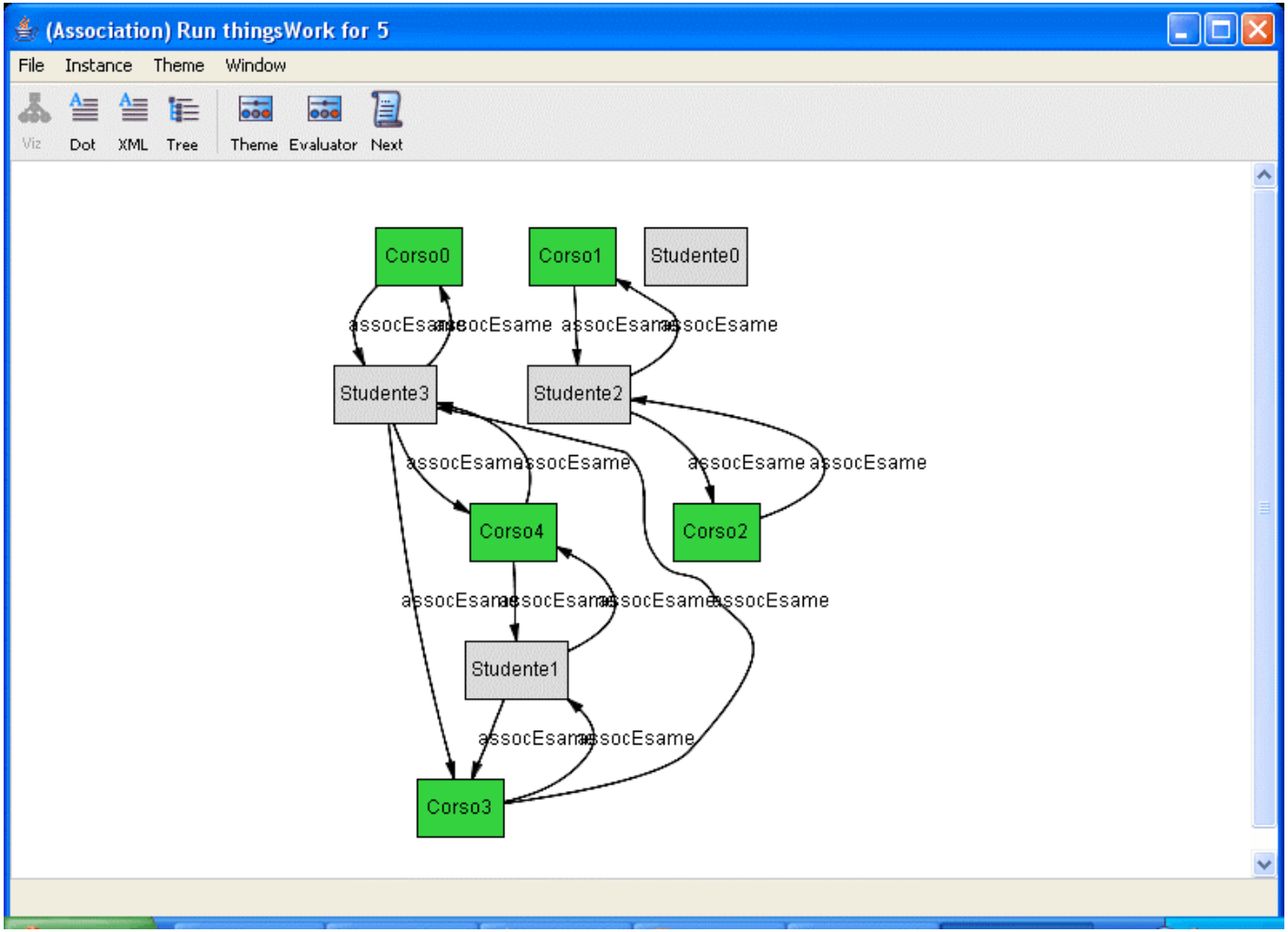
- for all X, Y1, Y2, Y3, Y4 (Studente(X) &&
  Esame(X,Y1) && ... && Esame(X,Y4)) ->
  ((Y1 = Y2) | (Y1 = Y3) | (Y1 = Y4) | (Y2 = Y3) |
  (Y2 = Y4) | (Y3 = Y4))

Esempio di Associazione 3

In Alloy:

```
sig Studente  {
        assocEsame: set Corso
}  {  #assocEsame =< 3  }
sig Corso  {
        assocEsame: set Studente
}
fact associationEsame  {
   all s:Studente, c:Corso |
      (s in c.assocEsame) <=> (c in s.assocEsame)
}
```

Esempio di Istanza del Modello



Associazioni Ternarie

Per quanto concerne le associazioni di higher arity, accenniamo a come si possano modellare le ternarie. Essenzialmente si tratta di un'estensione al metodo per le binarie: è necessario usare l'operatore prodotto come per le operazioni a N input (sono tutte relazioni particolari di arità N), e come per le associazioni binarie si deve imporre un fact che afferma la partecipazione alla associazione di tutte le classi coinvolte.

Generalizzazioni e Gerarchie

Per modellare *generalizzazioni* in FOL è sufficiente dichiarare: - for all X subClass(X) -> superClass(X)

Nel caso di *gerarchie* bisogna dichiarare ulteriori formule per disjointness e completeness:

- *complete*:

  for all X superClass(X) -> sub1(X) | ... | subN(X)

- *disjoint*:

  for all X subClassi(X) -> not subClassj(X)

  (for all i < j, AND di n*(n-1)/2 implicazioni)

IS-A e Gerarchie in Alloy

Alloy mette a disposizione un conveniente meccanismo di ereditarietà su Signatures: "extends" e "in"; la subclass eredita tutti i campi interni e appended facts della superclass.

In caso di gerarchie, "extends" impone la disjointness, mentre "in" no.
La completeness, se presente, va dichiarata mediante un semplice fact.

IS-A e Gerarchie: Esempi

```
sig Libro  {
        Titolo:Stringa
}

sig LibroStorico extends Libro  {
        Epoca: Stringa
}

/*LibroStorico appartiene all'insieme degli atomi
   Libro ed eredita tutte le relazioni Titolo */
```

IS-A e Gerarchie: Esempi 2

```
sig Persona  {
        Eta: Intero
}

sig Maschio, Femmina extends Persona  {
} //supponiamo sia disjoint e complete

//la disjointness è già implicata da extends
fact partitionPersona  {
        Maschio + Femmina = Persona
}
```

Specializzazioni

Possiamo avere *specializzazioni di attributi* (restrizioni di dominio o di molteplicità) o *di associazioni*.

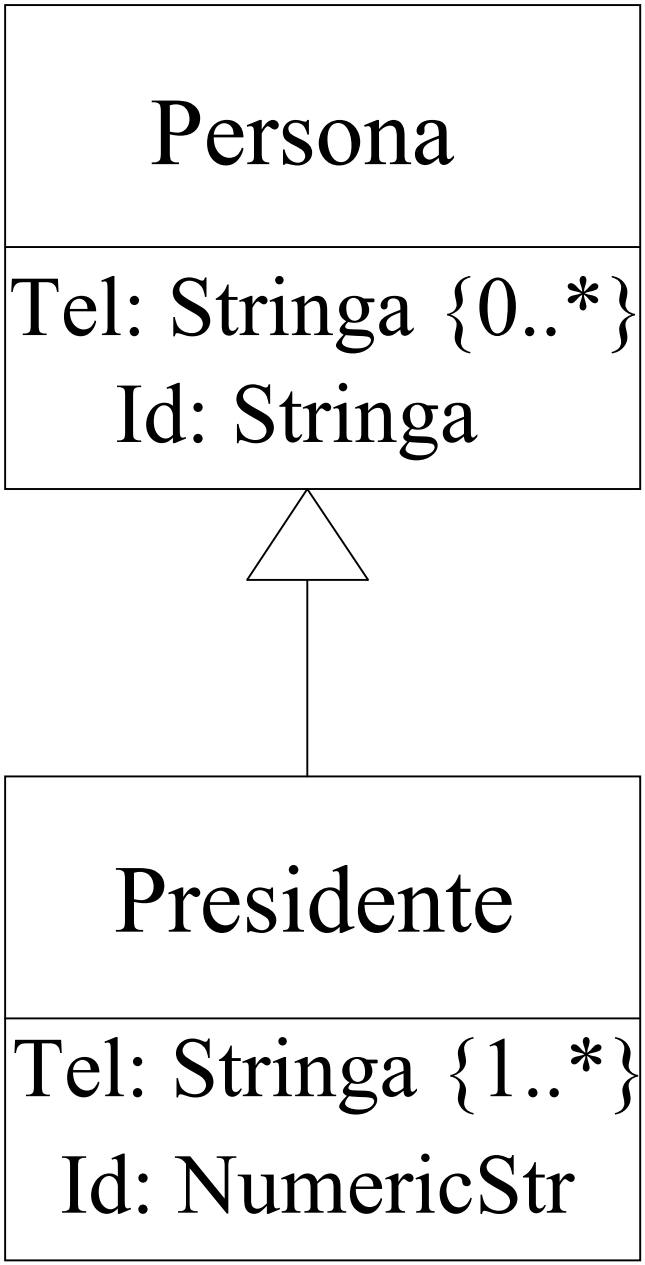
In FOL la specializzazione di Attributi non richiede formule particolari, l'attributo è interamente rimodellato nella sottoclasse.
Nel caso delle associazioni la formula è simile alle IS-A: - for all X1...Xn

subAssoc(X1...Xn) -> superAssoc(X1...Xn)

Spec. di Attributi in Alloy

Regola generale: imporre i vincoli di restrizione *negli appended facts* della sottoclasse in questione.

Nel caso di restrizioni di molteplicità si può utilizzare *l'operatore cardinalità* (#) per dare dei bounds agli attributi.
Nel caso di restrizioni di tipo bisogna *definire sotto-tipi e vincolare opportunamente i tipi degli attributi* negli appended facts.


Persona
Tel: Stringa {0..*}
Id: Stringa
Presidente
Tel: Stringa {1..*}
Id: NumericStr

Spec. di Attributi: Esempio

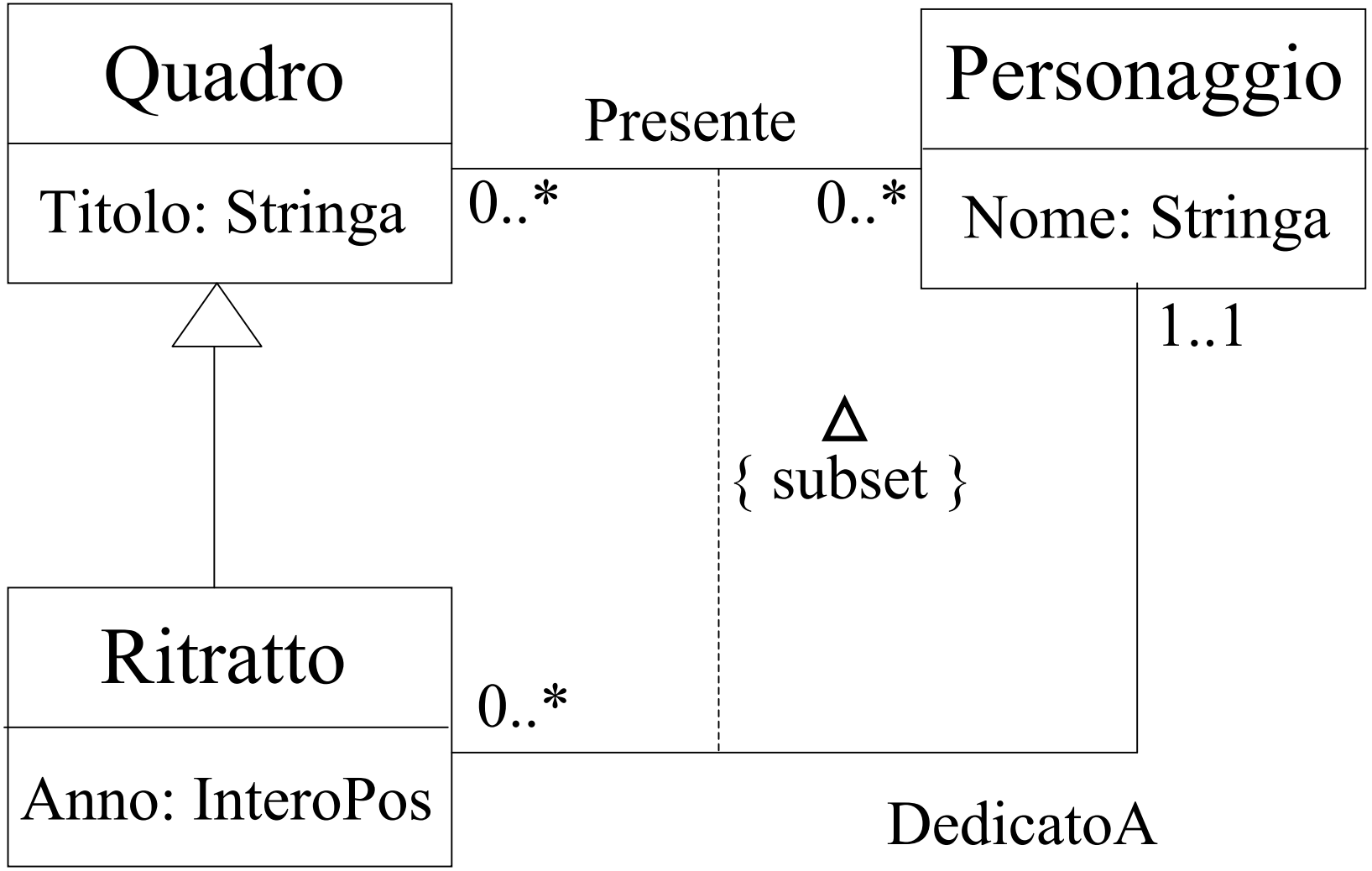
```
sig Persona  {
        Tel: set Stringa,
        Id: Stringa
}

sig NumericStr extends Stringa  {}

sig Presidente  {
} { #Tel >= 1 &&
        all s:Stringa | (s in Id) => (s in NumericStr) }
```

Spec. di Associazioni in Alloy

Nel caso di Specializzazioni di Associazioni è sufficiente replicare in un Fact il medesimo vincolo che abbiamo espresso in FOL.

Relativamente al diagramma nella prossima slide, il vincolo è:

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


Quadro
Titolo: Stringa
Presente
0..*
0..*
Personaggio
Nome: Stringa
1..1
{ subset }
Ritratto
Anno: InteroPos
0..*
DedicatoA

Analisi di Confronto tra Alloy ed Otter/Mace nel Contesto della Verifica di Proprietà

Tipi di Verifica - Overview

- *Conseguenze Implicite* = verificare se il modello implica una determinata proprietà (formula)
- *Single-Class Consistency* = il modello ammette un numero > 0 di istanze della classe
- *Sussunzione tra Classi* = esiste una coppia di classi tale che sono in relazione IS-A implicita?
- *Equivalenza tra Classi* = esiste una coppia di classi tale che l'una sussume l'altra?
- *Ristrutturazione* = il diagramma ristrutturato è equivalente a quello non ristrutturato?

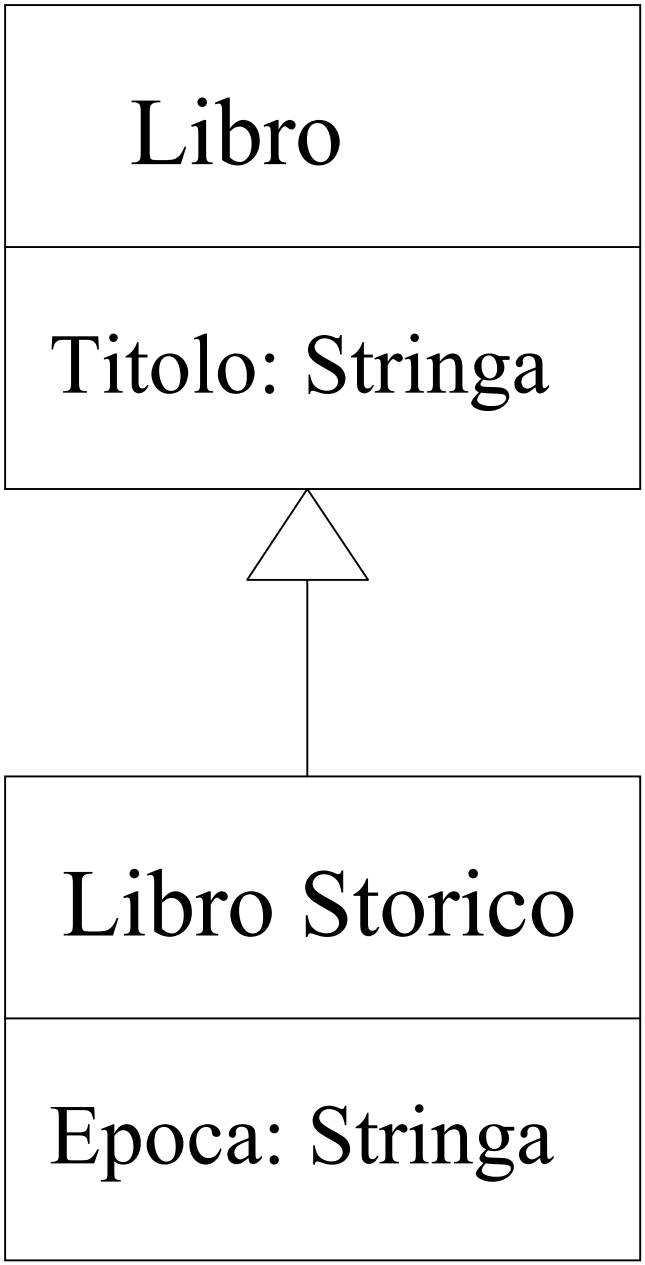
Case Study I: Cons. Implicite

Consideriamo uno scenario in cui abbiamo una semplice relazione IS-A (mostrata nella prossima slide). Vogliamo utilizzare Otter/Mace e Alloy per controllare le seguenti Conseguenze Implicite:


LibroStorico eredita l'attributo Titolo? (true)


Titolo è monovalore in LibroStorico? (true)


Libro eredita l'attributo Epoca? (false)


Libro
Titolo: Stringa
Libro Storico
Epoca: Stringa

Verifica in Otter

```
%%% 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))))
).
```

Verifica in Otter 2

```
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 3

Accorpiamo la verifica delle conseguenze 1 e 2:
"LibroStorico eredita titolo in modo monovalore".

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

Otter ha dichiarato valide le conseguenze 1 e 2, e mace ha trovato un controesempio per la 3 (scope 5).

Verifica in Alloy

```
//LibroStorico eredita Titolo, in modo monovalore
assert LSEreditaT  {
        all ls:LibroStorico | one s:Stringa |
                ls.Titolo = s
}
check LSEreditaT for 5
```

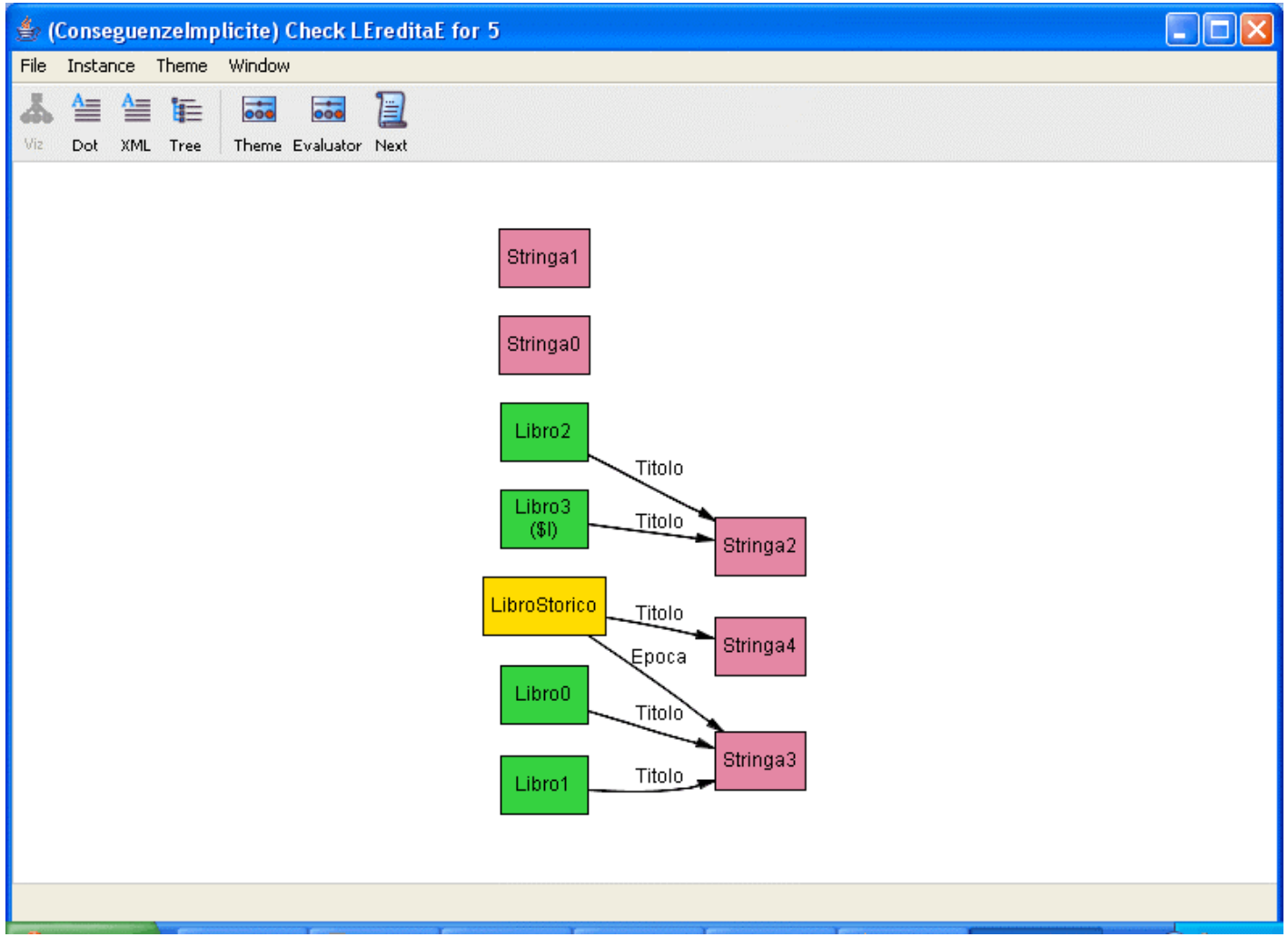
L'Analyzer non riesce a trovare un controesempio, ed impiega 0.468 secs con SAT4J (default solver), e 0.093 con ZChaff.

Verifica in Alloy 2

```
//Libro eredita Epoca (falso)
assert LEreditaE  {
        all l:Libro | some s:Stringa |
                l.Epoca = s
}
check LEreditaE for 5
```

L'Analyzer trova un controesempio, impiegando 0.094 secs con SAT4J (default solver), e 0.047 con ZChaff.

Verifica in Alloy 3



Performance Comparison

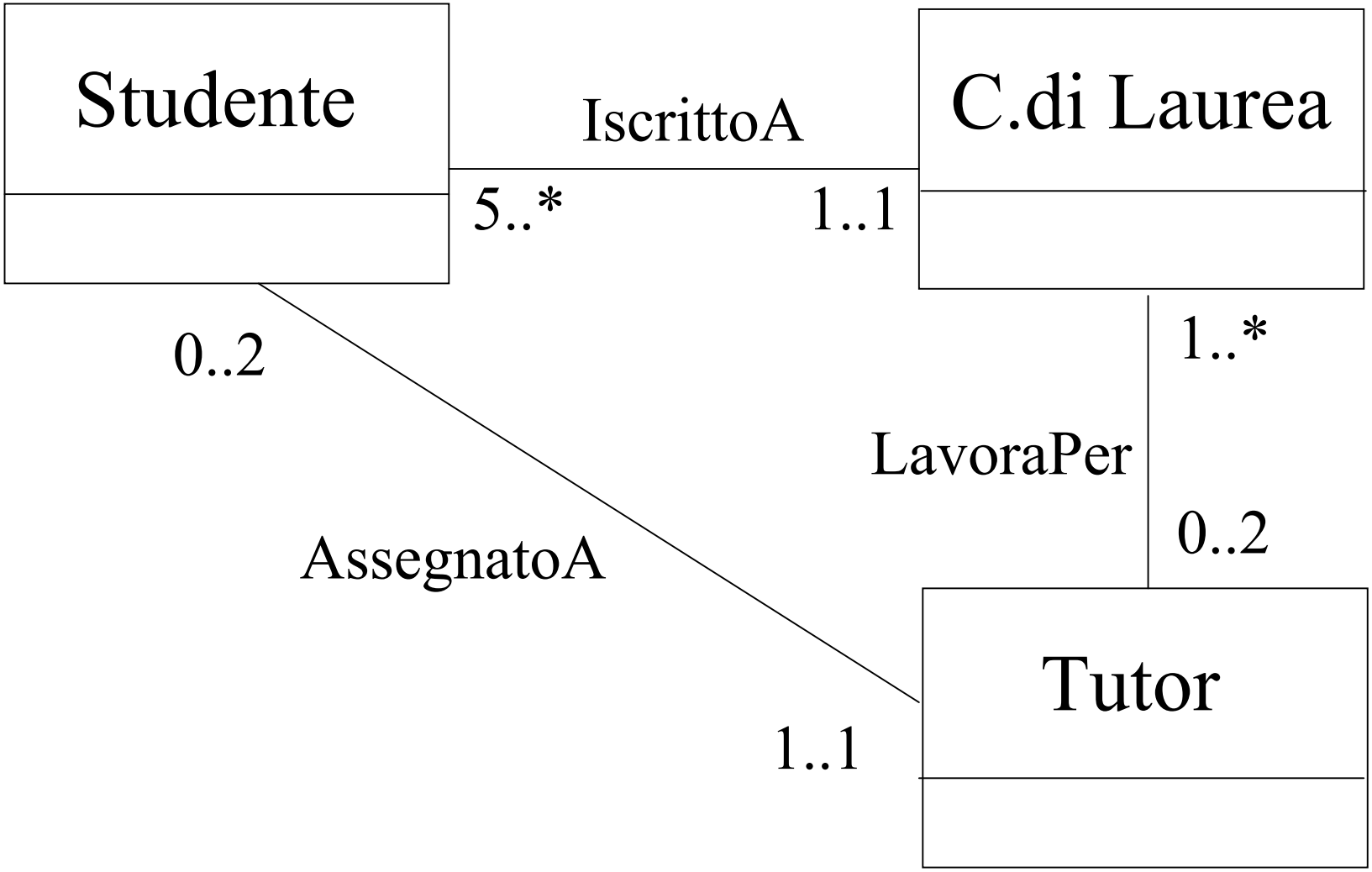
Confronto	Alloy SAT4J-AlloyZChaff	
assert 1	Zchaff	503% of SAT4J speed
assert 2	Zchaff	200% of SAT4J speed

Confronto	Alloy ZChaff-Otter/Mace	
assert 1	Alloy	656% of Otter speed
assert 2	Alloy	425% of Mace speed

Case Study II: Class Consistency

Questa volta ci occupiamo di un diagramma insoddisfacibile, mostrato nella slide successiva.
Il problema risiede nelle molteplicità: ogni Corso ha al massimo 2 Tutors, i quali possono occuparsi di al massimo 2 Studenti, ma un Corso richiede almeno 5 Studenti.

Vogliamo chiedere ai due tools se è possibile avere un'istanza legale in cui esiste almeno uno Studente, con scope 10.


Studente
IscrittoA
C.di Laurea
5..*
1..1
0..2
1..*
LavoraPer
0..2
AssegnatoA
Tutor
1..1

Notare: molteplicità min in FOL

% Studente IscrittoA 5..* CorsoDiLaurea

```
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)))).
```

Stesso caso in Alloy

```
// sig CorsoDiLaurea

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

Verifica in Otter

```
% Esiste almeno uno studente (false assertion)
someStudents <-> (
    (exists X Studente(X))
).
% Tesi
-(
    -someStudents     % mace2 non trova modelli
                      % time su scope 10 = 8.81 secs
    % si noti che è stato imposto che ogni atomo
    % debba far parte di una classe
).
```

Verifica in Alloy

```
//Ricerca di istanza legale con almeno uno Studente
assert noStudents  {
        no Studente
}
check noStudents for 10 int
```

L'Analyzer non riesce a trovare un controesempio, ed impiega 0.938 secs con SAT4J (default solver), e 0.844 con ZChaff.

Performance Comparison

Confronto Alloy SAT4J-AlloyZChaff
assert Zchaff 111% of SAT4J speed

Confronto Alloy ZChaff-Otter/Mace
assert Alloy 1044% of Otter speed

[Reflexive] Transitive Closure

La Chiusura Transitiva (nelle versioni Riflessiva e non) è una funzionalità offerta da Alloy che rende il linguaggio semanticamente più potente di Otter.
Si tratta di un operatore unario, applicabile ad una espressione relazionale, che denota l'insieme di tutti gli atomi raggiungibili applicando la relazione X o più volte, dove X è 0 se Riflessiva, 1 se Non Rifl.

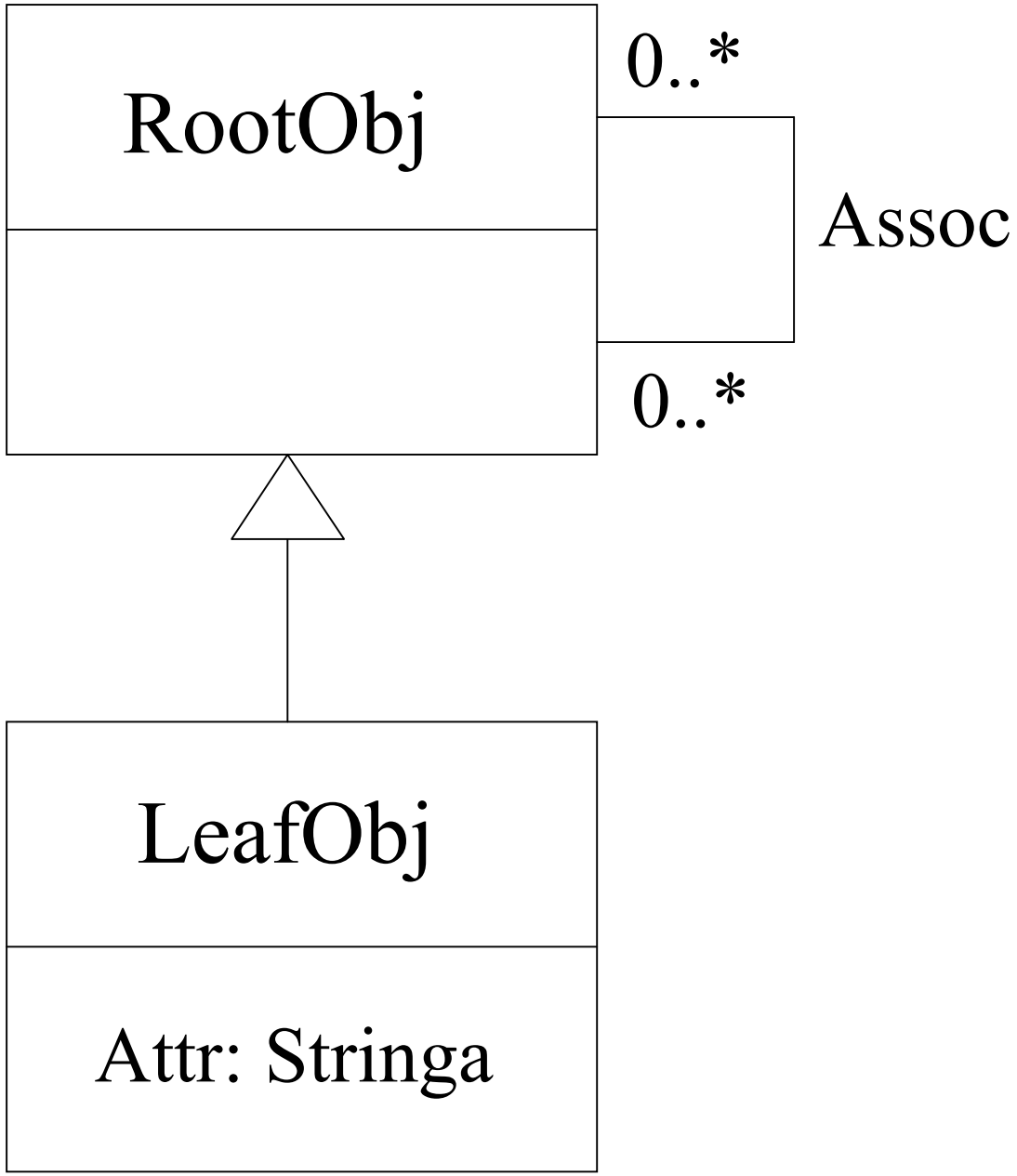
Sintassi:

NRTC: ^esprRelazionale

RTC: *esprRelazionale

[Reflexive] Transitive Closure 2

La TC può risultare utile in determinati scenari in cui abbiamo relazioni tra atomi di insiemi non disgiunti. Ad esempio uno dei tutorials presenti sul sito ufficiale di Alloy fornisce un esempio di TC modellando un Filesystem: se ad esempio una Dir *d* appartiene al FS, e condivide una relazione *contents* con altri oggetti del FS, l'espressione *d*.^*contents* denota l'insieme di tutti gli oggetti del FS raggiungibili da d. Nelle prossime slides verrà fornito un ulteriore esempio, più vicino ad UML.

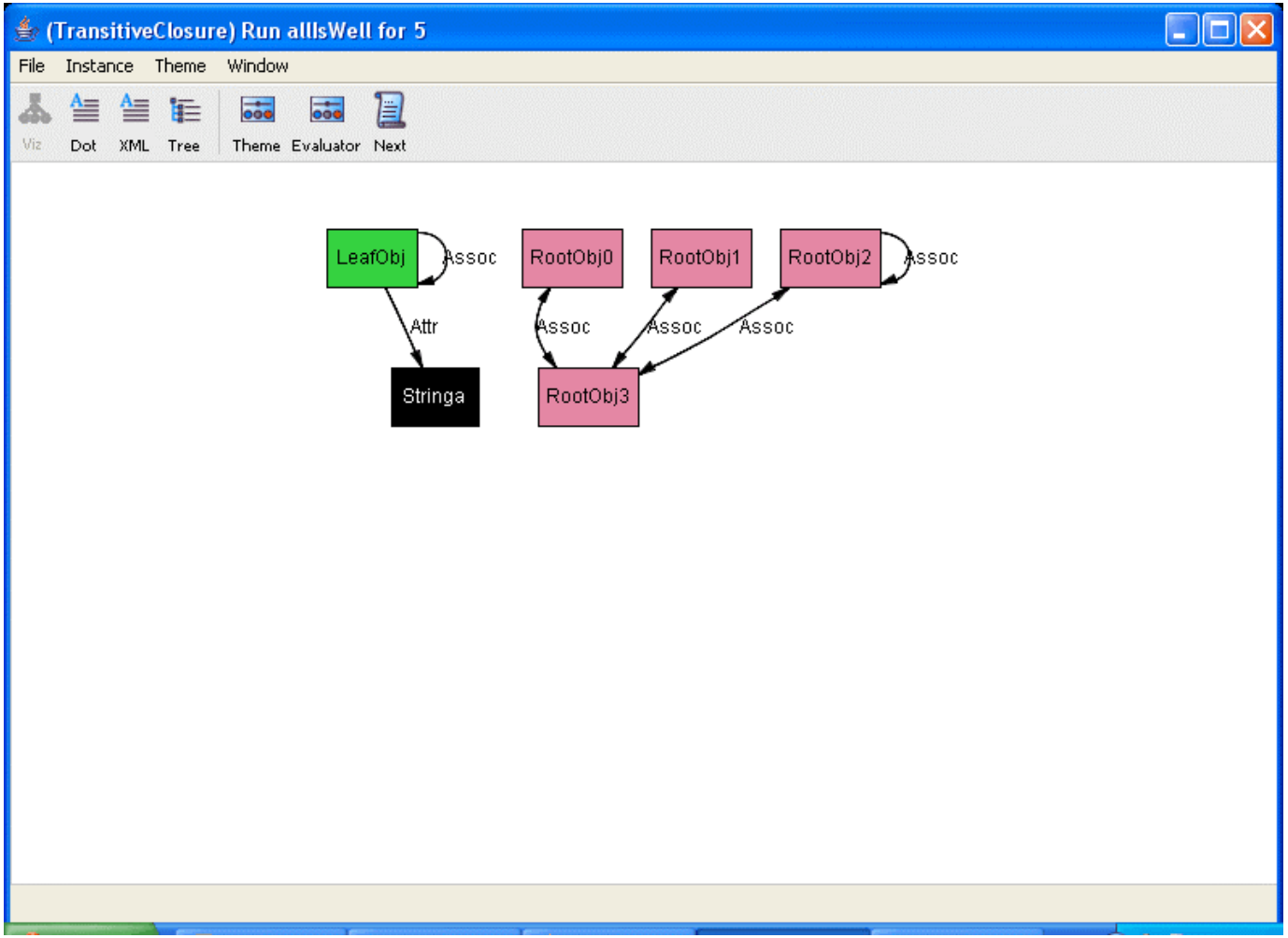

RootObj
0..*
Assoc
0..*
LeafObj
Attr: Stringa

Esempio di TC

Relativamente al diagramma precedente, supponiamo che esista questo vincolo: "Ogni associazione Assoc è condivisa da due oggetti istanza della stessa classe", cioè Root con Root e Leaf con Leaf.
Possiamo sfruttare la TC per aggiungere il vincolo:

```
fact constraintViaRTC  {
    all r: RootObj, d: r.*Assoc |
        (r not in LeafObj) => (d not in LeafObj)
        else (d in LeafObj)
}
```

Istanza corretta



Grazie per l'attenzione