Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

IL LINGUAGGIO ALLOY COME AUSILIO NELLA SPECIFICA FORMALE DI MODELLI UML

Relatore:
Chiar.mo Prof.
Davide Rossi

Presentata da:
Tiziano Verdone

Sessione I
Anno Accademico 2013-2014

ABSTRACT

UML è ampiamente considerato lo standard de facto nella fase iniziale di modellazione di sistemi software basati sul paradigma Object-Oriented; il suo diagramma delle classi è utilizzato per la rappresentazione statica strutturale di entità e relazioni che concorrono alla definizione delle specifiche del sistema; in questa fase viene utilizzato il linguaggio OCL per esprimere vincoli semantici sugli elementi del diagramma. Il linguaggio OCL però soffre della mancanza di una verifica formale sui vincoli che sono stati definiti. Il linguaggio di modellazione Alloy, inserendosi in questa fase, concettualmente può sopperire a questa mancanza perchè può descrivere con le sue entità e relazioni un diagramma delle classi UML e, tramite propri costrutti molto vicini all'espressività di OCL, può specificare vincoli semantici sul modello che verranno analizzati dal suo ambiente l'Alloy Analyzer per verificarne la consistenza.

In questo lavoro di tesi dopo aver dato una panoramica generale sui costrutti principali del linguaggio Alloy, si mostrerà come è possibile creare una corrispondenza tra un diagramma delle classi UML e un modello Alloy equivalente. Si mostreranno in seguito le analogie che vi sono tra i costrutti Alloy e OCL per la definizione di vincoli formali, e le differenze, offrendo nel complesso soluzioni e tecniche che il modellatore può utilizzare per sfruttare al meglio questo nuovo approccio di verifica formale. Verranno mostrati anche i casi di incompatibilità. Infine, come complemento al lavoro svolto verrà mostrata, una tecnica per donare una dinamicità ai modelli statici Alloy.

Keywords: Alloy, Diagramma delle classi UML, OCL

Indice

Introduzione **4**

1 Il linguaggio Alloy **6**
1.1 Caratteristiche del linguaggio Alloy 6
1.2 Osservazioni preliminari 7
1.3 Primo esempio in Alloy 8
1.4 Primo esempio in OCL 10
1.5 I costrutti del linguaggio 11
1.5.1 Signature 11
1.5.2 Relazioni 12
1.5.3 Join relazionale 13
1.5.4 Espressioni relazionali 14
1.5.5 Formule 14
1.5.6 Espressioni derivate da formule 15
1.5.7 Fact 15
1.5.8 Predicati 15
1.5.9 Funzioni 16
1.5.10 Asserzioni 16

2 Traduzione di un diagramma delle classi UML **17**
2.1 Molteplicità particolari 17
2.2 Bidirezionalità associazioni 18
2.3 Classe singleton 20
2.4 Gli attributi 20
2.4.1 Attributi numerici e atomi Int 20
2.4.2 Attributi di tipo stringa 21
2.4.3 Attributi booleani 22
2.4.4 Enumerazioni 22
2.5 Gerarchie e sottotipi 23

3 Alloy vs OCL **25**
3.1 Precedenza degli operatori 26
3.2 context e self in OCL 27
3.3 Sintassi OCL 28
3.4 Operazioni insiemistiche 29
3.5 Operazioni in OCL 30
3.6 if-then-else statement 31
3.7 Insiemi Alloy e Collezioni OCL 31
3.8 Bag in Alloy 33
3.9 Cosa non si può fare in Alloy 35

4 Modellazione dinamica **37**
4.1 Elaborazione di un modello dinamico 37
4.2 Operazioni come predicati 38

Conclusioni **41**

Introduzione

Nelle prime fasi di un processo di sviluppo di un sistema software si avrà una specifica dei requisiti tra sviluppatore e committente; essa è usata per definire le caratteristiche del sistema. La fase di specifica dei requisiti è molto critica, se i requisiti non prendono in esame alcuni aspetti il progetto rischierebbe degli inutili ritardi e inevitabili manutenzioni; per questo motivo è necessario disporre di una specifica chiara, senza ambiguità e consistente. Il linguaggio naturale non consente di ottenere specifiche dotate di tutte le proprietà elencate, ad esempio un testo scritto in linguaggio naturale ha al suo interno una serie di ambiguità che il lettore disambigua sulla base delle proprie conoscenze; tuttavia ogni individuo può avere dei background diversi e di conseguenza è possibile che disambigui le frasi in linguaggio naturale differente. Si utilizzano allora delle notazioni formali, di stampo matematico. Il linguaggio UML è ampiamente considerato lo standard de facto nella fase iniziale di modellazione di sistemi software basati sul paradigma Object-Oriented; il suo diagramma delle classi è utilizzato per la rappresentazione statica strutturale di entità e relazioni che concorrono alla definizione delle specifiche del sistema; in questa fase viene utilizzato il linguaggio OCL per esprimere vincoli semantici sugli elementi del diagramma. Questi due linguaggi partecipano alla definizione delle notazioni formali; il linguaggio OCL però soffre della mancanza di una verifica sui vincoli che sono stati definiti. Il linguaggio di modellazione Alloy, inserendosi in questa fase, concettualmente può sopperire a questa mancanza perchè può descrivere con le sue entità e relazioni un diagramma delle classi UML e, tramite propri costrutti molto vicini all'espressività di OCL, può specificare vincoli semantici sul modello che verranno analizzati dal suo ambiente l'Alloy Analyzer per verificarne la consistenza e analizzare comportamenti.

Il metodo principale usato dall'analizzatore Alloy per effettuare le verifiche di consistenza, si basa sulla trasformazione automatica del modello in una, a volte immensa, proposizione logica su cui lavorerà uno dei SAT-solver messi a disposizione dall'ambiente.

In questo lavoro di tesi, si esaminerà a fondo il linguaggio Alloy, studiando vari aspetti che possono portare, senza una pretesa di esaustività, ad un corretto accostamento con lo standard UML/OCL. Nello specifico si affronteranno i seguenti argomenti:

Capitolo1: si offrirà una panoramica sul linguaggio Alloy, studiandone i costrutti linguistici e dando un esempio esplicativo adatto ad illustrare le potenzialità espressive

e di verifica.

Capitolo2: si mostrerà come è possibile creare una corrispondenza tra un diagramma delle classi UML e un modello Alloy equivalente offrendo delle tecniche che permettano di gestire aspetti non controllabili nativamente dal linguaggio Alloy.

Capitolo3: si mostreranno le analogie che vi sono tra i costrutti linguistici Alloy e OCL per l'espressività nella definizione di vincoli formali; si mostreranno anche le differenze offrendo nel complesso soluzioni e tecniche che il modellatore può utilizzare per sfruttare al meglio questo nuovo approccio di verifica formale.

Capitolo4: come complemento si mostrerà una tecnica che permette di donare una dinamicità ai modelli Alloy che per la loro natura sono statici. Queste tecnica può permettere di specificare operazioni sulle entità in gioco analizzando le pre e post condizioni.

In ogni capitolo, per poter usufruire di una fedele e onesta visione, si analizzeranno anche gli aspetti incompatibili mostrando degli esempi chiarificatori.

Capitolo 1

Il linguaggio Alloy

1.1 Caratteristiche del linguaggio Alloy

Alloy è un linguaggio sviluppato al MIT e giunto oggi alle versione 4.2, consente di definire dei modelli software.

I concetti utilizzati in Alloy sono molto simili a quelli tipici dei linguaggi di programmazione ad oggetti, tuttavia in questo caso il concetto di classe non contiene al suo interno una descrizione degli algoritmi utilizzati per compiere determinate operazioni ma vi saranno delle descrizioni delle proprietà definite mediante l'uso della logica del primo ordine. Al linguaggio Alloy è sempre associato un potente e veloce tool automatico, per simulare le specifiche e verificarne la validità, questo tool è in grado di stabilire se le specifiche date sono soddisfacibili, cioè se potrà esistere un'implementazione che le soddisfa, e se da esse possono essere tratte delle particolari conseguenze.

La trattazione non ha lo scopo di analizzare ogni minimo aspetto del linguaggio, sia per quanto riguarda la sintassi che per quanto riguarda le metodologie di progettazione ma ci si limiterà a dare una presentazione e una panoramica che descriva le potenzialità che verranno utilizzate in questo lavoro di tesi; per chi è interessato ad una trattazione più specifica si consiglia la lettura del manuale ufficiale "Software Abstraction: Logic, Language and Analysis" di Daniel Jackson, l'ideatore del linguaggio.

Alloy è un linguaggio dichiarativo, in poche parole non si esprimono degli algoritmi ma affermazioni che potranno essere vere o false, e quindi idoneo a descrivere lo stato di un sistema presentando le sue proprietà e vincoli che verrano espressi secondo la logica del primo ordine. Per l'analisi dei modelli, l'analizzatore, il tool accostato al linguaggio, preso in input un codice sorgente scritto in Alloy, trasforma il codice in un'istanza SAT e usa uno tra i vari SAT-solver a disposizione per verificarne la consistenza.

In Alloy tutto si fonda sul concetto di insiemi di atomi e relazioni; ogni atomo è un'entità primaria indivisibile e immutabile in quanto non può essere divisa ulteriormente e le sue proprietà, una volta definite non variano nel tempo. Una relazione consiste in

un insieme di tuple di arietà maggiore o uguale a due che stabilisce un legame tra gli atomi del modello; gli insiemi di tuple che rappresentano le relazioni non possono essere in relazione tra di loro altrimenti avremmo una logica del secondo ordine. Il linguaggio non differenzia in maniera netta un atomo da un insieme di atomi: nelle espressioni quando si fa riferimento ad un singolo atomo, esso sarà sempre visto come un insieme di cardinalità uno; è quindi lecito utilizzare operatori insiemistici, come l'unione o l'intersezione anche con singoli atomi. Come già accennato, anche se il linguaggio si fondi sul concetto di atomo e relazione, molto ingegnosamente si hanno anche delle somiglianze con i linguaggio Object-Oriented, ad esempio :

1. la definizione di un tipo di atomi può essere vista come la dichiarazione di una classe e quindi tutti gli atomi appartenenti al proprio insieme di definizione, saranno visti come istanze della classe.

2. si può estendere un tipo con un altro, tramite la keyword "extends"

Essendo Alloy un linguaggio dichiarativo, non esistono costrutti come il while per il controllo del flusso, dato che quest'ultimo non esiste; non esistono variabili come quelle definite nei linguaggi imperativi ma etichette.

1.2 Osservazioni preliminari

E' bene sottolineare da subito che bisogna sempre impostare uno scope per tutte le entità in gioco; anche se è possibile enumerare le possibili istanze-configurazioni valide del modello che vengono generate una alla volta, non è possibile determinare e quindi decidere se un'affermazione è sempre vera o sempre falsa con un numero imprecisato e quindi potenzialmente infinito di entità in gioco : l'analizzatore continuerebbe all'infinito o fino alla fine della memoria disponibile senza dare un responso. Allo stesso modo, in questo contesto con scope finito però bisogna fare ugualmente attenzione, in quanto se l'analizzatore non individua nessuna situazione indesiderata non si può avere la certezza che con scope di grado maggiore si abbia lo stesso comportamento; tuttavia se l'ambiente è sufficientemente grande, non trovare controesempi significa con buona probabilità che il modello è corretto.

Un'altra determinante considerazione che è sempre bene tenere a mente è che anche se un modello scritto in Alloy risulta essere consistente con specifiche chiare e non ambigue, non esiste alcun modo automatico per stabilire se il software che in seguito verrà realizzato sarà aderente o meno al modello specificato.

In conclusione diciamo che Alloy può essere utilizzato anche in altri ambiti, ovunque si abbia la necessità di lavorare con varie configurazioni di sistemi da "dare in pasto" al SAT-solver; cioè oltre ad essere usato nell'ingegnerizzazione dei requisiti per descrivere il dominio e le sue proprietà ad esempio può essere usato per trovare le soluzioni di uno schema a sfondo logico-matematico come il Sudoku, il "problema delle otto regine" etc..

1.3 Primo esempio in Alloy

Vediamo un primo intuitivo esempio senza ricorrere a tutte le spiegazioni necessarie alla comprensione dei costrutti Alloy utilizzati; per una loro corretta comprensione si rimanda ai paragrafi successivi dove verranno elencati e studiati approfonditamente. Al solo scopo di illustrare brevemente le potenzialità espressive del linguaggio, non si partirà da notazioni formali in UML/OCL ma si descriverà il contesto da modellare in linguaggio naturale; in seguito verrà mostrata la versione OCL equivalente per dare al lettore un primo confronto, senza soffermarsi sulle differenze che verranno analizzate e risolte nel capitolo 3.

Vogliamo modellare un mondo in cui ci sono delle isole e delle città collegate da strade; ogni città è situata su una e una sola isola. La definizione del modello Alloy si basa sulla seguente composizione di signature e relazioni:

```
sig Isola {}

sig Citta { situata: one Isola }

sig Strada {
  partenza: one Citta,
  arrivo: one Citta
}
```

In un'analisi sono stati individuati i seguenti vincoli che devono essere sempre rispettati :

1. ogni strada, che per come è stata definita collega due città, non può arrivare nella città di partenza.
2. non dobbiamo avere due strade uguali, ovvero che partono e arrivano nelle stesse città.
3. deve esistere almeno un'isola con esattamente tre città.
4. non vogliamo avere ponti tra le isole, ovvero strade che arrivano in un'isola diversa da quella di partenza.
5. se due città sono situate sulla stessa isola, devono per forza essere collegate.

```
fact noStradeCircolari {
  all s:Strada | s.partenza != s.arrivo
}

fact noStradeUguali {
  all s1,s2:Strada | (s1.partenza = s2.partenza and s1.arrivo = s2.arrivo) implies s1 = s2

  //o in modo equivalente
  //no disj s1,s2:Strada | s1.partenza = s2.partenza and s1.arrivo = s2.arrivo
}

fact isolaConTreCitta {
```

```
  some i:Isola | #situata.i = 3
}

fact noPonti {
  all s:Strada | s.partenza.situata = s.arrivo.situata
}

fact cittaCollegateStessaIsola {
  all disj c1,c2:Citta | {
    (c1.situata = c2.situata) =>
    (some s:Strada | s.partenza = c1 and s.arrivo = c2)
  }
}
```

Possiamo passare a questo punto ad utilizzare l'analizzatore per avere delle risposte su comportamenti del nostro modello così definito; ad esempio possiamo porre la domanda <<è vero che ogni isola ha almeno una città?>> tramite la assert:

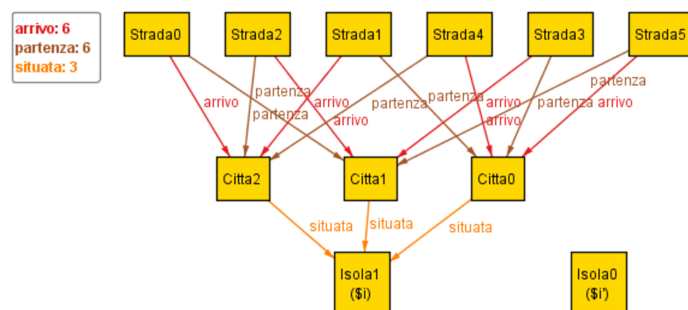
```
assert noIsolaDeserta {
  all i:Isola | some c:Citta | c.situata = i
}

check noIsolaDeserta for 7
```

il "for" serve ad impostare lo scope per il nostro modello; nel caso in questione si genereranno massimo sette istanze diverse per ogni signature concreta ma avremmo potuto anche esplicitare il numero esatto di istanze per determinate signature ad esempio scrivendo:

```
check noIsolaDeserta for exactly 7 but exactly 5 Citta, exactly 3 Isola
```

L'analizzatore ci mostra subito un controesempio; vediamo che tutti i vincoli che avevamo stabilito sono stati attuati e allo stesso tempo esiste un'isola, la numero zero, con cui non è associata nessuna città.



Possiamo quindi stabilire se questa situazione possa non essere tollerabile e quindi trasformarla successivamente in un vincolo, cambiando la keyword "assert" con la keyword "fact" continuando le analisi e verifiche sul modello.

Un'altra modalità di analisi permette di far creare delle istanze del modello rispetto ad un predicato, anche vuoto, che deve essere rispettato.

Consideriamo quindi la situazione in cui nel nostro modello deve esistere una strada con un solo senso di marcia, ovvero una strada per cui non esista la sua versione opposta; possiamo quindi verificare un predicato ed utilizzare il comando “run” :

```
pred StradaOpposta {
  some s:Strada | no opposta:Strada | {
    s.partenza = opposta.arrivo
    s.arrivo = opposta.partenza
  }
}

run StradaOpposta for 7
```

Notiamo che l’analizzatore non troverà nessuna istanza valida del modello (anche aumentando lo scope).

```
Executing "Run StradaOpposta for 7"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   5246 vars. 175 primary vars. 11317 clauses. 90ms.
   No instance found. Predicate may be inconsistent. 1123ms.
```

Il lettore attento avrà capito che il vincolo “cittaCollegateStessaIsola” per come è stato definito data una strada, “crea” automaticamente sempre la versione opposta. Grazie a questa verifica formale quindi possiamo accorgerci di situazioni potenzialmente indesiderate che possono celarsi “indisturbate” in un modello che ad una prima vista poteva risultare corretto.

1.4 Primo esempio in OCL

La versione OCL equivalente all’esempio appena visto in Alloy è il seguente:

```
-- classes

class Isola
end

class Citta
end

class Strada
end

-- associations

associations Situata between
  Citta[*] role contiene
  Isola[1] role situata
end

associations Partenza between
  Strada[*] role stradePartenza
  Citta[1] role partenza
end
```

```
associations Arrivo between
  Strada[*] role stradeArrivo
  Citta[1] role arrivo
end

-- OCL constraints

constraints

context Isola
  inv isolaConTreCitta:
    Isola.allInstances->exists(i | i.contiene->size() = 3)

context Strada
  inv noStradeCircolari:
    self.partenza <> self.arrivo

context Strada
  inv noPonti:
    self.partenza.situata = self.arrivo.situata

context Strada
  inv noStradeUguali:
    Strada.allInstances->forAll(s |
      (self.partenza = s.partenza and self.arrivo = s.arrivo) implies self = s
    )

context Citta
  inv cittaCollegateStessaIsola:
    Citta.allInstances->forAll(c |
      (c <> self and c.situata = self.situata) implies
      (Strada.allInstances->exists(s | s.partenza = c and s.arrivo = self))
    )
```

Oltre a forti similitudini, notiamo da subito che ogni vincolo è definito in relazione ad un contesto e che nelle associazioni c'è sempre un ruolo, con relativa molteplicità, per ogni entità coinvolta; quest'ultimo approccio consente di poter esprimere il vincolo "isolaConTreCitta" in maniera molto più leggibile rispetto la versione Alloy. Come preannunciato in precedenza, questo esempio serve solo a dare un primo confronto estetico tra il linguaggio Alloy ed il linguaggio OCL; tutte le differenze verranno analizzate e risolte nei capitoli successivi.

1.5 I costrutti del linguaggio

1.5.1 Signature

```
[abstract] sig Nome { [Corpo] } [extends altrasig]
```

Le signature sono il costrutto fondamentale in Alloy, paragonabili alle classi nel diagramma delle classi UML. Quando si definisce un signature si definisce un insieme di atomi che condividono le stesse relazioni e gli stessi vincoli. Nel seguito quando si parlerà di "atomo di tipo A", si intenderà un atomo appartenente all'insieme di atomi determinato dalla definizione di una signature A.

Si potrà definire una signature astratta anteponendo la keyword “abstract” nella definizione, ciò imporrà all'analizzatore di creare un tipo senza crearne istanze valide, questo meccanismo è utilizzato per esprimere quindi interfacce e classi astratte.

Alloy tratta i medesimi concetti di eredità (singola) e gerarchia dei principali linguaggi di programmazione Object-Oriented, cioè è possibile far ereditare ad una nuova signature le caratteristiche di un'altra signature astratta o concreta già definita utilizzando la keyword “extends” specificando non obbligatoriamente nuove relazioni; ogni nuovo sottotipo di atomo potrà essere utilizzato dovunque si possa utilizzare il suo supertipo.

1.5.2 Relazioni

Una relazione viene definita all'interno di una signature e serve a legare i tipi di atomi tra di loro, può essere dunque vista come un'associazione nei diagrammi delle classi. Nel mondo Alloy una relazione, una volta definita, denoterà un insieme di tuple di atomi. Vediamo subito un esempio di definizione di relazione singola, ovvero la relazione che lega due tipi di atomi avente un'unica direzione di navigabilità :

```
sig Studente { iscritto: one CorsoDiLaurea }
```

con questo codice, in tutte le possibile configurazioni del modello, ogni atomo di tipo Studente dovrà essere collegato ad atomi di tipo CorsoDiLaurea e “iscritto” sarà l' insieme composto dalle coppie $<$Studente,CorsoDiLaurea$>$; nel caso preso in esame si è utilizzato il quantificatore “one” per specificare la molteplicità della relazione, esso simboleggia il “uno e uno solo” evitando che uno studente possa essere iscritto a due o più corsi di laurea differenti (o a nessuno).

Alloy mette a disposizione, oltre a “one” altri tre quantificatori per esprimere la molteplicità delle relazioni tra atomi “di partenza” A e atomi “di arrivo” tipo B :

lone ogni atomo di tipo A è associato a zero o un atomo di tipo B

set ogni atomo di tipo A è associato a zero o molti atomi di tipo B

some ogni atomo di tipo A è associato ad almeno un atomo di tipo B

è possibile anche specificare relazioni riflessive quando il tipo B coincide con il tipo A. Se non si specifica nessun quantificatore, verrà usato di default il quantificatore “one”.

Per specificare una relazione doppia, ovvero una relazione che lega un tipo di atomo di partenza con una coppia di atomo di arrivo, si usa la notazione :

```
sig A { rel: B [quant] -> [quant] C }
```

Se non si specifica nessun quantificatore, verrà utilizzato il quantificatore “set-set” dove ogni atomo di tipo A potrà essere associato a qualsiasi coppia $<$B,C$>$. Il quantificatore “set” nelle relazioni doppie è irrilevante: le seguenti definizioni di relazione sono equivalenti :

```
sig A { rel: B -> C }
sig A { rel: B set -> C }
sig A { rel: B -> set C }
sig A { rel: B set -> set C }
```

A differenze delle relazioni singole, dove la comprensione sull'uso dei quantificatori è immediata, nelle relazioni doppie si possono incontrare casi di molteplicità che sfuggono ad una prima intuizione creando non poche incomprensioni; verranno quindi spiegati in dettaglio i casi più frequenti diversi da quello di default:

one-one ogni atomo di tipo A è associato a tutte le #C (o #B) coppie <B,C> dove tutti gli atomi di tipo B e C compaiono esattamente una volta, nel modello il numero di atomi di tipo B deve essere uguale al numero di atomi di tipo C altrimenti si avrà sicuramente un'inconsistenza.

one-set ogni atomo di tipo A è associato ad esattamente #C coppie <B,C>, dove ogni atomo di tipo C è sempre diverso e gli atomi di tipo B possono essere ripetuti o mancare.

some-set ogni atomo di tipo A è associato ad almeno #C coppie <B,C>, dove ogni atomo di tipo C compare almeno una volta e gli atomi di tipo B possono essere ripetuti o mancare.

lone-set ogni atomo di tipo A è associato al massimo a #C coppie <B,C> dove ogni atomo di tipo C compare al massimo una volta e gli atomi di B possono essere ripetuti o mancare.

il "#X", come l'operatore di cardinalità, simboleggia il numero di atomi di tipo X presenti nel'istanza del modello che viene creata. Scambiando nei casi descritti i due moltiplicatori tra loro si otterrà un comportamento speculare.

1.5.3 Join relazionale

L'operatore sicuramente più utilizzato e importante è il join relazionale o "dot join" perchè simboleggiato dal punto "."; questo operatore opera tra due insiemi di atomi, vediamo di illustrare il suo funzionamento con la seguente situazione:

```
insieme X : {(A1,B1), (A2,B4), (A2,B5)}
insieme Y : {(B1,C2,D2), (B1,C3,D1), (B3,C3,D1), (B4,C4,D2)}
```

X.Y sarà l'insieme composto dalle coppie (A1,C2,D2) (A1,C3,D1) (A2,C4,D2)

Cioè si costruiranno le nuove tuple concatenando le tuple del primo insieme con quelle del secondo e lasciando quelle che fanno "match", ovvero quelle dove l'ultimo atomo della tupla del primo insieme è uguale al primo atomo della tupla del secondo insieme ed eliminando infine l'atomo di congiunzione. Questo operatore di solito verrà utilizzato tra un insieme singoletto contenente un solo atomo e una relazione, che ricordiamo è

un insieme di tuple, ma come nel caso precedente può essere usato anche con insiemi di tuple di arietà maggiore; ad esempio se "esami" rappresenta la relazione che parte dagli atomo di tipo Studente e arriva agli atomi di tipo Esame, l'espressione "s.esami" dove s è un arbitrario atomo di tipo Studente (visto automaticamente come un insieme) denoterà l'insieme degli esami sostenuti dallo studente s. L'operatore dot join quindi funge implicitamente anche come navigatore tra gli elementi del modello.

1.5.4 Espressioni relazionali

Le espressioni in Alloy esprimono e rappresentano un insieme di atomi; possono essere definite semplicemente indicando una signature e quindi rappresentando gli atomi definiti da quest'ultima, oppure semplicemente indicando un atomo arbitrario di un certo tipo; tramite le operazioni insiemistiche come l'unione(+), l'intersezione(&), la differenza insiemistica(-), il join relazionale (.) o il prodotto cartesiano(→) si possono quindi definire espressioni più complesse. Queste operazioni insiemistiche possono essere applicate solo tra insiemi costituiti da tuple con la stessa arietà ed è possibile costruire espressioni rappresentanti insiemi costituiti da elementi disomogenei tra loro, ad esempio prendendo sempre l'insieme degli studenti e la relazione "esami", in maniera del tutto lecita:

```
Studenti.esami + Int
```

denoterà l'insieme composto da tutti gli esami sostenuti dagli studenti e da tutti gli interi nel modello (che avranno il proprio scope deciso dal modellatore).

1.5.5 Formule

Le formule denotano sempre un valore booleano; in maniera primitiva esse possono essere create in tre modi:

1. confronto tra espressioni relazionali tramite gli operatori di uguaglianza(=), differenza(!=), e inclusione(in, not in)

2. applicazione di quantificatori ad espressioni relazionali, ad esempio la dicitura "some X" avrà valore vero se esiste almeno un atomo di tipo X nel modello

3. confronto tra due espressioni rappresentanti atomi di tipo Int tramite gli operatori : <, =< (minore-uguale), >, >= (maggiore-uguale), =, !=. Una tipica situazione si ha quando si usa l'operatore di cardinalità "#" su un'espressione ottenendo un atomo di tipo Int; la dicitura "#X > 3" avrà valore vero se l'insieme rappresentanto dall'espressione X avrà più di tre elementi.

Le formule potranno essere combinante tra loro tramite gli operatori di congiunzione(and, &), disgiunzione(or, ||) e implicazione logica(implies, =>, <=, <=>) esprimendo una

nuova formula più complessa. Ogni formula potrà essere negata, anteponendo il "not" o il "!".

Con questi ingredienti quindi si potranno formare enunciati con variabili, in stile logica del primo ordine, ad esempio con

```
all x:Exp1 | some y:Exp2 | Formula[x,y]
```

si indica che esistono due insiemi di riferimento (Exp1 e Exp2) sui cui elementi vengono applicati dei quantificatori logici, nel caso specifico il "per ogni" e "esiste" rispettivamente, che verificano una formula Alloy in cui compaiono gli elementi in questione; questo enunciato che sarà a tutti gli effetti una nuova formula potrà assumere quindi valore vero o falso a seconda del modello istanziato. Nella definizione di enunciati non si potrà lavorare con i sottoinsiemi di un insieme, cioè non si potrà dire "per tutti i sottoinsiemi SA di A vale Formula(SA)" altrimenti avremmo una logica del secondo ordine.

1.5.6 Espressioni derivate da formule

In Alloy è possibile creare delle espressioni racchiudendo una formula nelle parentesi graffe e specificando la formazione delle tuple componenti il nuovo insieme/espressione :

```
{ x1:Exp1, x2:Exp2, ... , xn:Expn | F[x1,x2,...,xn] }
```

Tale espressione corrisponde a tutte le tuple di n elementi, il cui i-esimo elemento appartiene all'insieme Exp(i), che verificano la Formula F.

1.5.7 Fact

I fatti "fact" sono formule che svolgono il ruolo di vincoli globali del sistema che creano una selezione tra tutte le possibili istanze del modello, ovvero verranno create solo le istanze legali dove queste formule sono vere. Bisogna fare attenzione all'uso dei fatti in quanto sono la prima causa dell'overconstraints dove l'eccessiva selezione causa un modello non popolato da atomi o incosistente. Un fatto potrà essere costituito da un numero di formule che verranno interpretate tutti in congiunzione tra di loro; si può anche assegnare un nome :

```
fact [nome] {
  Formula1[]
  Formula2[]
  ...
  FormulaN[]
}
```

1.5.8 Predicati

```
pred nome[parametri] { [corpo] }
```

I predicati sono costrutti costituiti da un nome, da parametri, ed un corpo formato da formule che mirano a fornire un valore booleano a seconda dei parametri in input. I predicati potranno essere invocati dal comando “run” specificando lo scope oppure possono fare le veci di formule e quindi riutilizzabili ad esempio nei fact facendo attenzione ai parametri in gioco.

Quando l’analizzatore trova un’istanza valida rispetto ad un predicato o un controesempio rispetto ad una asserzione, è possibile esplorare graficamente l’istanza del modello che è stata generata, continuando ad effettuare “domande” su quella particolare configurazione e quindi a poter navigare su atomi precisi del modello; ad esempio riprendendo l’esempio di introduzione al linguaggio, con “Citta$2.situata” si ottiene l’insieme {Isola$1}.

A volte è utile usare il comando run con un predicato vuoto senza parametri, quindi sempre verificato, per creare una o più istanze valide del modello, in modo da avere una panoramica concreta del modello sul quale si vuole lavorare.

1.5.9 Funzioni

Le funzioni come i predicati possono avere dei parametri con i quali lavorare, ma al contrario dei predicati che “ritornano” un valore booleano, le funzioni servono a far tornare un insieme di atomi, e quindi a svolgere il ruolo di espressione. Ad esempio possiamo scrivere :

```
fun tra[da: Int , a:Int] : set Int {
  { x:Int | da =< x && a >= x }
}
```

tra[2,6] denoterà l’insieme {2,3,4,5,6}

1.5.10 Asserzioni

```
assert nome { corpo }
```

Le asserzioni sono strutturalmente uguali ai fatti ma hanno un ruolo e un utilizzo diverso; mentre i fatti sono dei vincoli globali le asserzioni sono delle “query” al modello; in pratica si “sfida” il tool analizzatore a trovare un modello legale in cui la nostra asserzione non è verificata, mostrandola in caso di successo (per il tool, un insuccesso per il modellatore). Come già spiegato nell’introduzione, l’analizzatore effettua le proprie ricerche a seconda di uno scope ben preciso, quindi se non viene trovato nessun esempio che smentisce l’asserzione non vuole dire che quell’asserzione è sempre verificata, sta al modellatore una volta ottenuti i dati validare il contesto o proseguire con ulteriori esami.

Capitolo 2

Traduzione di un diagramma delle classi UML

Come già spiegato nel capitolo procedente, in Alloy si descrive un modello definendo le strutture delle entità in gioco con le relazioni che intercorrono tra di esse, quindi senza considerare delle istanze concrete, esattamente come avviene nel diagramma delle classi con classi e associazioni; le istanze che poi verranno generate dall'analizzatore, partendo dal modello definito, costituiranno un possibile diagramma degli oggetti UML. Grazie a questa visione comune, è molto semplice trovare le corrispondenze tra i costrutti dei due linguaggi, almeno nei fondamenti : una classe corrisponderà ad una signature, un'associazione ad una relazione etc. tuttavia esistono delle differenze sulla gestione degli elementi del modello; questa parte della tesi vuole quindi cercare di risolvere queste discordanze dando delle tecniche che consentano la più corretta traduzione di un diagramma delle classi in un modello Alloy equivalente.

2.1 Molteplicità particolari

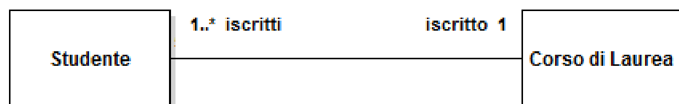
Come già sappiamo Alloy permette di utilizzare dei quantificatori per esprimere la molteplicità delle relazioni, ma con questi mezzi non riusciamo a coprire tutti i casi che possiamo avere in UML, ad esempio non possiamo esprimere la molteplicità 5 o la molteplicità (2..3). Fortunatamente possiamo trovare rimedio usando la seguente tecnica : si usa il quantificatore "set" per definire la molteplicità della relazione e tramite l'operatore di cardinalità si aggiunge a fine signature una "sig fact", ovvero un fatto "privato" sui campi della signature, come nell'esempio sottostante; questo tipo di vincolo poteva comunque essere definito con un fatto normale che andava ad operare con il "per ogni" :

```
sig A { rel: set B } { #rel >= 2 and #rel =< 3 }

o in modo equivalente senza il sig fact
fact { all a:A | #a.rel >= 2 and #a.rel =< 3 }
```

2.2 Bidirezionalità associazioni

In UML un'associazione è una relazione strutturale tra due o più entità che descrive connessioni tra le potenziali istanze. In base al numero di elementi coinvolti nella relazione si hanno diverse specializzazioni : associazioni binaria, associazione ternaria, ... associazione n-aria. Questa associazione è così importante da essere considerata la "colla" che unisce il sistema, senza di esse le istanze delle entità sarebbero destinate ad una vita isolata. Il caso decisamente più frequente è costituito dall'associazione binaria che come lecito attendersi coinvolge due entità. In un'associazione binaria entrambe le classi hanno una responsabilità sull'associazione, questo vuol dire che bisogna esprimere le molteplicità in entrambi i versi e non in uno solo come nelle relazioni in Alloy; pertanto, data un'associazione binaria è possibile navigare dagli oggetti di un tipo a quelli di un altro. Qualora nella stesura dei vincoli Alloy avessimo la necessità di una doppia bidirezionalità nelle relazioni, come avviene in UML, procedendo ad una giusta traduzione, dobbiamo creare delle relazioni inverse dalla signature di arrivo verso la signature di partenza. Illustriamo ciò con l'esempio di uno studente iscritto ad un corso di laurea



dove oltre ad avere la relazione "iscritto", tra Studente e CorsoDiLaurea dobbiamo creare anche l'inversa, chiamata in questo caso "iscritti" :

```
sig Studente { iscritto: one CorsoDiLaurea }
sig CorsoDiLaurea { iscritti: some Studente }
```

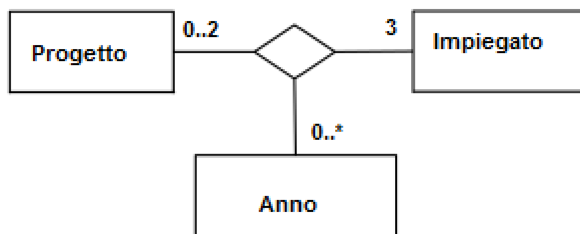
Anche se ad una prima vista, il nostro modello sembra essere ben tradotto, dato che abbiamo due associazioni singole diverse, al lettore attento non sarà sfuggito il fatto che anche se uno Studente s è iscritto ad un CorsoDiLaurea c, non è detto che c a sua volta sia legato allo studente s, ovvero manca un vincolo semantico che unisca queste due entità creando la vera bidirezionalità. Tutto ciò può essere raggiunto impostando il seguente fact :

```
fact bidirezionalità {
  all s:Studente, c:CorsoDiLaurea | s in c.iscritti <=> c in s.iscritto
}
```

dove il simbolo "<=>" è esattamente il "se e solo se" e può essere espresso anche con la keywork "iff". Questo fact può essere generalizzato a qualunque associazione binaria, ricordando, come nell'esempio, che nella logica del primo ordine che utilizza Alloy, un atomo, preso singolarmente viene visto come un insieme singoletto e quindi è lecito utilizzare l'operatore "in" che simboleggia il "è sottoinsieme di ".

Discorso analogo per le associazioni ternarie, ma con un fact per la bidirezionalità leggermente più complesso. Per illustrare anche il ruolo delle molteplicità prendiamo in esame la seguente situazione : in un'azienda abbiamo le entità Impiegato, Progetto e

Anno in relazione fra loro; ogni impiegato può partecipare al massimo a due progetti l'anno e ogni anno se un progetto è attivo deve avere esattamente tre impiegati (un progetto può essere rieseguito in un anno diverso).



per la tecnica che è stata individuata, in Alloy per poter gestire al meglio le molteplicità conviene non utilizzare quantificatori nelle definizioni delle relazioni, lasciando quindi il default "set-set" come nel codice seguente

```
sig Impiegato { assocAP: Anno->Progetto }
sig Progetto { assocAI: Anno->Impiegato }
sig Anno { assocPI: Progetto->Impiegato }

fact bidirezionalità {
  all a:Anno, i:Impiegato, p:Progetto | {
    (a->i in p.assocAI => (a->p in i.assocAP and p->i in a.assocPI)) and
    (a->p in i.assocAP => (a->i in p.assocAI and p->i in a.assocPI)) and
    (p->i in a.assocPI => (a->i in p.assocAI and a->p in i.assocAP))
  }
}
```

a questo punto possiamo "giocare" con l'operatore di cardinalità e il join relazionale per ottenere le molteplicità volute. Senza ricorrere ad ulteriori spiegazioni sulla formazione dei vincoli, diciamo che questa tecnica è del tutto generale e applicabile ad ogni associazione ternaria a patto di rispettare l'ordine dei join rispetto la relazione (quella verso le altre due entità delle ternaria) definita nella signature per la quale si vuole definire la molteplicità, come nel seguente codice :

```
fact molteplicità {

  //molteplicità per Progetto dove assocAI: Anno->Impiegato
  all a:Anno, i:Impiegato | #(assocAI.i.a) =< 2

  //molteplicità per Impiegato dove assocAP: Anno->Portiere
  all a:Anno, p:Progetto  | #(assocAP.p.a) = 3

  //non c'è bisogno di manipolare quella per Anno, lasciata a set-set (0..*)
}
```

I casi di bidirezionalità di associazioni con arietà maggiore di 3 non sono trattati in questo testo, anche perchè abbastanza rari nella realtà; comunque se si è prestata attenzione alla formazione della bidirezionalità nel caso ternario non si dovrebbero incontrare difficoltà dato che si può seguire la stessa logica di costruzione.

2.3 Classe singleton

In Alloy è estremamente semplice definire una classe "singleton", ovvero una classe che dovrà avere esattamente una sola istanza nel modello. L'unico quantificatore che si può utilizzare nella definizione di una signature è appunto "one"

```
one sig A { ... }
```

con questo codice, in qualsiasi istanza legale del modello esisterà sempre uno e un solo atomo di tipo A.

2.4 Gli attributi

Il linguaggio Alloy mette a disposizione in maniera primitiva solo il tipo di dato intero, rappresentato dagli atomi di tipo Int; sarà quindi possibile esprimere in una signature rappresentante una classe UML una relazione verso questo tipo di atomo simboleggiando un attributo intero. Nel seguito vedremo come poter gestire gli attributi booleani; per tutti gli altri tipi di attributo si dovranno creare relative relazioni verso nuovi tipi di atomi definiti con una signature vuota, come nell'esempio "Data" :

```
sig Data {}
sig Spettacolo { data: Data }
```

possiamo a questo punto proseguire impostando il vincolo che non possono esserci due spettacoli con la stessa data:

```
fact { no disj s1,s2:Spettacolo | s1.data = s2.data }
```

Si consiglia di attuare la stessa tecnica anche per gli attributi di tipo stringa nonostante il linguaggio metta a disposizione questo tipo di attributo ; vedremo più avanti le motivazioni per questa scelta.

2.4.1 Attributi numerici e atomi Int

Come già spiegato in precedenza, un attributo di tipo intero sarà a tutti gli effetti una relazione verso un atomo Int. Questo tipo di atomo è l'unico che può godere implicitamente di un ordinamento e quindi di poter usufruire di operatodi di confronto (maggiore, minore etc..); inoltre il linguaggio mette a disposizione gli usuali operatori aritmetici per l'addizione(+) e la sottrazione(-). Tuttavia data l'ambiguità sintattica che si crea con gli operatori insiemistici di unione e differenza insiemistica che usano i medesimi simboli, si consiglia di usare sempre le funzioni definite nel file di libreria per la gestione degli interi che può essere importata ad inizio sorgente tramite l'istruzione:

```
open util/integer
```

Essa consente di usufruire delle operazioni di addizione "add", sottrazione "sub", moltiplicazione "mul", divisione "div", resto modulo "rem", valore assoluto "signum" ed altri.

Ovviamente come per tutte le entità del modello, anche gli atomi di tipo Int avranno un proprio scope; di default se non si specifica un Bitwidth, ci saranno 16 atomi di tipo Int, da quello rappresentante il -8 a quello rappresentante il +7. Bisogna fare massima attenzione a questo intervallo in quanto per rendere coerente il linguaggio con la propria filosofia di atomi e relazioni, qualsiasi riferimento numerico, che sia il risultato di una somma o la cardinalità di un insieme ottenuta con l'operatore "#", sarà associato sempre ad un atomo di tipo Int e ciò può provocare degli "overflow". Ad esempio se si sta utilizzando lo scope di default e si sta lavorando con un insieme di 10 elementi, l'operatore di cardinalità associato a quell'insieme non restituirà 10, perchè l'intero più grande è 7, ma restituirà -6 ! In parole povere, nel "conteggio" degli elementi, dopo il 7 si è ripartiti dal -8 arrivando infine ad una cardinalità sbagliata e addirittura negativa, e come si può facilmente immaginare questi overflow possono portare a svariate situazioni indesiderate che si celaranno inosservate in una disattenta analisi del modello. Per poter usufruire di uno scope di interi più ampio, quando si verifica un predicato con il comando run o quando si analizza un'asserzione con il comando check, dopo aver stabilito lo scope per le altre entità bisognerà specificare a parte quello per gli interi come di seguito:

```
run nomePredicato for 8, but 7 Int
```

in questo caso, a differenza dello scope per le altre entità, non avremo 7 atomi di tipo Int diversi ma avremo interi con Bitwidth di valore 7, ovvero ci saranno 2^7 interi distinti da -64 a +63. E' buona norma comunque continuare a preoccuparsi degli overflow anche nei casi di scope più grandi.

La più grande limitazione che si è riscontrata sul linguaggio Alloy è sulla gestione dei tipi numerici con virgola : i float e i double. Non c'è nessun modo per poter usufruire di questo tipo di dato, sia a livello nativo che con tecniche derivate; il modellatore quindi dovrà essere sempre cosciente di questa mancanza e procedere ad un utilizzo di Alloy solo quando ha la sicurezza di non aver bisogno di questo tipo di dato.

2.4.2 Attributi di tipo stringa

In realtà il linguaggio Alloy mette a disposizione anche il tipo di dato "String" che però inaspettatamente non offre nessuna utilità, nonostante sia un elemento fondamentale di qualsiasi sistema software, sia in fase di progettazione che in fase di sviluppo, e pertanto personalmente ne sconsiglio l'utilizzo. Vediamo di illustrarne il motivo con il seguente codice:

```
sig Persona {
  nome: String,
  cognome: String
}

fact unoSolo { one p:Persona | p.nome = "Mario" and p.cognome = "Rossi" }

run {} for 6
```

In qualsiasi istanza legale del modello, per quanto riguarda gli atomo di tipo String, ci saranno sempre e solo quelli definiti esplicitamente in qualche parte del codice e nessun altro; se non vi saranno definizioni esplicite si incapperà in un'inconsistenza del sistema (!). Nel caso preso in esame avremo i due atomi "Mario" e "Rossi" definiti nel fact che saranno gli unici a poter essere raggiunti tramite le relazioni "nome" e "cognome" da tutti gli atomi di tipo Persona esistenti nell'istanza generata; inoltre facendo qualche semplice prova si potrà constatare come si possano avere delle istanze legali del modello in cui due persone diverse o più avranno entrambe nome e cognome "Mario Rossi" nonostante il chiaro vincolo "unoSolo" e quindi rendendo del tutto inutile il "model checking" voluto.

2.4.3 Attributi booleani

Alloy non mette a disposizione in maniera primitiva il tipo di attributo booleano ma unendo le proprietà dei singleton, delle signature astratte e dell'estensione si può usare la seguente tecnica:

```
abstract sig Booleano {}
one sig Vero extends Booleano {}
one sig Falso extends Booleano {}

sig Studente {
  laureando: Booleano,
  cfu: Int
}

fact { all s:Studente | s.cfu > 170 <=> s.laureando = Vero }
```

Nel modello ci saranno sempre esattamente un atomo di tipo Vero e un atomo di tipo Falso; qualsiasi riferimento al tipo di dato Booleano come "laureando" sarà una relazione verso uno dei due atomi simulando il comportamento di un valore booleano. Anche se la tecnica è del tutto funzionale, esiste una differenza da tenere a mente: un Booleano anche rappresentando un valore di verità conserverà la propria natura di atomo e quindi non sarà idoneo ad essere usato come una formula, ad esempio non si potrà usare l'operatore di negazione "!" su di esso ma si dovrà specificare "a mano" uno dei due valore possibili come nel fatto del codice precedente.

2.4.4 Enumerazioni

La tecnica precedente può essere generalizzata con la creazione di più di due valori distinti, creando quindi tipi enumerativi

```
abstract sig Colore {}
one sig Blu, Giallo, Rosso extends Colore {}

sig Bandiera { colore: one Colore }
```

2.5 Gerarchie e sottotipi

Come già presentato all'inizio dello scorso capitolo, in Alloy è possibile definire delle gerarchie tra le signature usando la keyword "extends".

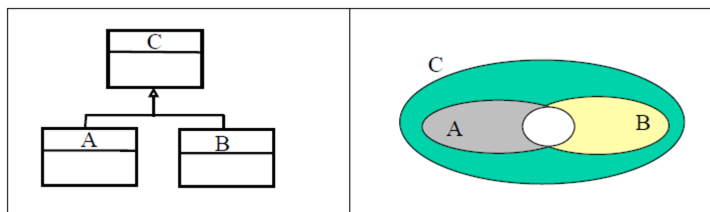
```
sig Impiegato { anni: Int }
sig CapoReparto extends Impiegato { dirige: Reparto }
sig Fattorino extends Impiegato { guida: Mezzo }

fact nonMinorenni{ all i:Impiegato | i.anni >= 18 }
```

con questo codice abbiamo creato una gerarchia di signature dove CapoReparto e Fattorino sono signature specializzate rispetto la signature Impiegato con cui sono legate. I due nuovi tipi di signature ereditano automaticamente l'attributo "anni" e specificano una nuova relazione ciascuno che la signature Impiegato non avrà . Tutti i nuovi atomi di tipo CapoReparto e di tipo Fattorino faranno parte dell'insieme di atomi di tipo Impiegato e di conseguenza verranno considerati dovunque si faccia riferimento a quest'ultimi come nel vincolo "nonMinorenni"; non potranno esistere atomi di tipo Fattorino o di tipo CapoReparto in relazione ad interi minori di 18.

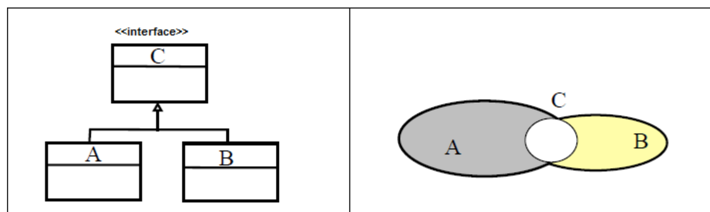
La extends opera in modalità "disjointness", ovvero nessun atomo Fattorino farà parte dell'insieme degli atomi CapoReparto e viceversa; sostituendo la keyword "in" al posto della "extends" si potrà definire una gerarchia che non opera in modalità "disjointness". Inoltre è possibile estendere una classe astratta creando quattro tipi di gerarchia diversi :

- tipo1: supertipo concreto e possibile intersezione dei sottotipi



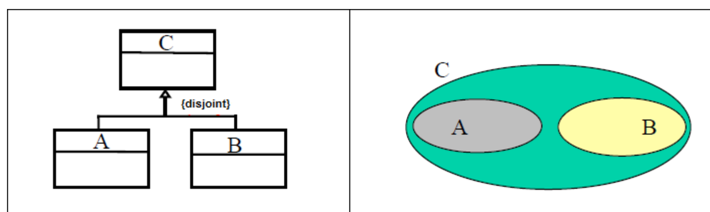

```
sig C {...}
sig A in C {...}
sig B in C {...}
```

- tipo2: supertipo astratto e possibile intersezione dei sottotipi concreti



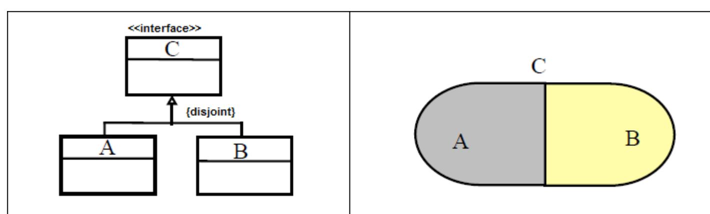
```
abstract sig C {...}
sig A in C {...}
sig B in C {...}
```

- tipo3: supertipo concreto e disjointness dei sottotipi




```
sig C {...}
sig A extends C {...}
sig B extends C {...}
```

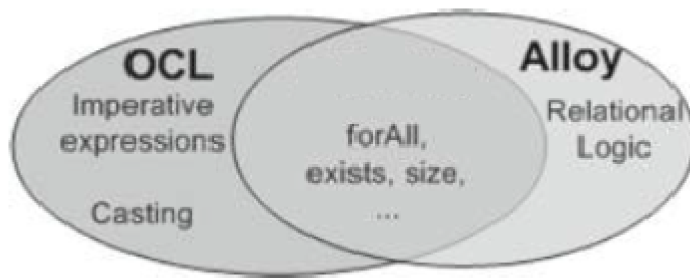
- tipo4: supertipo astratto e disjointness dei sottotipi concreti




```
abstract sig C {...}
sig A extends C {...}
sig B extends C {...}
```

Capitolo 3

Alloy vs OCL



Nel capitolo precedente ci siamo occupati della traduzione di un diagramma delle classi in un modello Alloy equivalente con il quale poter lavorare controllando il comportamento e la consistenza del sistema. Dopo aver quindi effettuato le varie analisi e verifiche avremo vincoli e asserzioni sul modello da utilizzare per esprimere invarianti in linguaggio OCL che è attualmente lo standard per la specifica formale. Tuttavia si può incontrare anche la situazione inversa in cui si vuole analizzare un diagramma delle classi già corredato con sintassi OCL per validarne la consistenza. In questo capitolo pertanto ci occuperemo di confrontare questi due linguaggi analizzandone:

1. gli aspetti linguistici e sintattici. Constateremo, senza pretese di esaustività, di disporre di una mappatura che ci permetterà di poter trasformare facilmente formule ed espressioni Alloy in costrutti OCL equivalenti e viceversa.

2. la gestione sugli elementi del modello. Vedremo come il linguaggio OCL presenti più tipi di insieme con diverse modalità sulla gestione degli elementi; offriremo quindi una soluzione che permetta di ottenere la stessa espressività anche in Alloy.

3.1 Precedenza degli operatori

Sicuramente il primo aspetto linguistico da considerare per procedere ad un giusto esame dei due linguaggi è la precedenza degli operatori; vediamo quindi le due liste di operatori in ordine di livello di precedenza, dal più alto al più basso, iniziando ad impararne le differenze sintattiche:

- Alloy

1. disgiunzione: or, ||
2. implicazione: implies, =>, <=, <=> (doppia)
3. congiunzione: and, &&
4. negazione: not, !
5. inclusione e uguaglianza tra insiemi: in, not in, =, !=
6. confronto tra interi: =, !=, <, =<, >=
7. somma e differenza tra interi: +, -
8. cardinalità insieme: #
9. intersezione: &
10. prodotto cartesiano: →
11. dot join: dot (".")
12. altri: ~, *, ^

- OCL

1. implicazione: implies
2. congiunzione, disgiunzione: and, or
3. uguaglianza: =, <> (diverso)
4. confronto numerico: <, >, <=, >=
5. decisione: if then else endif
6. somma e differenza tra numeri: +, -

7. prodotto e divisione tra numeri: *, /

8. negazione: not

9. navigazione: dot ("."), →, @pre

vediamo le differenza di interpretazione per :

```
Exp1 or Exp2 implies Exp3
```

usando le parantesi, possiamo illustrare le differenze tra due linguaggi; in OCL si effettuerà prima la disgiunzione logica tra Exp1 e Exp2 che determinerà l'implicazione

```
(Exp1 or Exp2) implies Exp3
```

mentre in Alloy dato che l'operatore di implicazione gode di priorità maggiore, otteremo il seguente comportamento, semanticamente diverso

```
Exp1 or (Exp2 implies Exp3)
```

3.2 context e self in OCL

A differenza di Alloy dove i fact sono globali, gli invarianti OCL hanno sempre un contesto sul quale operano, ovvero si deve specificare la parte del modello, generalmente una classe UML; nella definizione si farà uso della keyword "self" per riferirsi all'istanza generica della classe (relativa al contesto), in piccola analogia con la keywork "this" in Java. Di conseguenza se in Alloy abbiamo il fact

```
fact { all p:Persona | p.anni > 0 }
```

il medesimo concetto in OCL si esprimerà con :

```
context Persona
  inv etàPositiva: self.anni > 0
```

Si può vedere quindi l'uso del contesto in OCL come un fact Alloy dove si imposta il "per ogni" (all...) sulle istanze di una determinata signature. Anche se negli invarianti OCL con l'uso di "self" si fa riferimento solo ad un'istanza generica, non c'è carenza di espressività in quanto si può sempre far riferimento alle altre istanze in gioco della stessa entità, ad esempio :

```
context Studente
  inv matricolaUnivoca:
    Studente.allInstances()->forAll(s:Studente |
      s <> self implies s.matricola <> self.matricola)
```

esprime il già visto concetto di unicità di attributo. Nell'equivalente versione Alloy quindi ci sarà un "per ogni" che lavora su coppie di atomi studente; un atomo per il "self" e un altro atomo per l'elemento scandito dalla "forAll" :

```
fact {
  all s1,s2:Studente | s1 != s2 => s1.matricola != s2.matricola
}
```

Non è obbligatorio far riferimento all'istanza generica self in un invariante OCL, come già abbiamo avuto modo di vedere nel paragrafo 1.4 con :

```
context Isola
  inv isolaConTreCitta:
    Isola.allInstances->exists(i | i.contiene->size() = 3)
```

3.3 Sintassi OCL

In OCL si lavora solamente con collezioni di elementi e con operazioni applicabili su di esse. Questo tipo di sintassi comunque riesce ad avere la stessa espressività di quella vista in Alloy potendo usufruire di composizioni e concatenazioni di operazioni.

OCL ha quattro diverse tipologie di collezione; gli insiemi con cui si lavora in Alloy possono essere comparati alle collezioni OCL di tipo "Set", di conseguenza quando nei prossimi paragrafi si accosteranno questi due concetti si sottointenderà di lavorare con dei Set OCL. A fine capitolo verrà mostrato come gestire in Alloy il tipo di collezione OCL "Bag".

Lavorando con collezioni e operazioni, si avrà una sintassi leggermente diversa a livello estetico da quella di Alloy, usufruendo però della medesima espressività; ad esempio invece di avere l'operatore di unione tra due insiemi, avremo che su una prima collezione viene applicata l'operazione di unione che come parametro avrà una seconda collezione. Le operazioni potranno esprimere risultati diversi :

booleano : come ad esempio la "->isEmpty()"

collezione : come ad esempio la "->intersection(Coll c)"

queste operazioni contribuiranno a formare formule ed espressioni; esse potranno essere concatenate e combinate per formarne di più complesse seguendo la stessa interpretazione in Alloy vista in precedenza. Ad esempio con l'espressione:

```
Set{1,2,3}->intersection(Set{2,3}->union(Set{3,4,5}))
```

si denoterà la collezione di interi {2,3} derivata dalla composizione della union "dentro" la intersection; e concatenando la isEmpty

```
Set{1,2,3}->intersection(Set{2,3}->union(Set{3,4,5}))->isEmpty()
```

si denoterà una formula che avrà nel caso specifico valore falso.

Le corrispondenti formule in Alloy potranno essere :

```
no (1+2+3) & ((2+3) + (3+4+5))
#((1+2+3) & ((2+3) + (3+4+5))) = 0
```

Ogni combinazione denotante un booleano sarà una formula ed ogni combinazione denotante una collezione sarà un'espressione; nel seguito useremo "Formula" e "Coll" rispettivamente in relazione all'ambiente OCL per mostrare i legami con i costrutti Alloy.

3.4 Operazioni insiemistiche

Le operazioni insiemistiche in OCL a differenza di quelle in Alloy sono permesse solo tra insiemi omogenei, ovvero non è possibile fare l'unione tra l'insieme degli studenti e l'insieme Set{1,2,3} mentre in Alloy questo è del tutto lecito. Questa differenza non porta con sè particolari problemi in quanto l'uso di insiemi con elementi disomogenei è un caso particolare, se non molto raro, sia nella definizione e costruzione di specifiche sia nell'implementazione dove la maggior parte dei linguaggi Object-Oriented sono a tipizzazione. Verranno elencate le operazioni OCL per la gestione insiemistica, accostandole ai rispettivi costrutti Alloy :

- Unione

```
(OCL)   Coll1->union(Coll2)
(Alloy) aExp1 + aExp2
```

se in Alloy sappiamo con sicurezza che una delle due espressioni denota un insieme di un solo elemento, è più coerente accostarla all'operazione OCL "including" per una maggiore chiarezza estetica :

```
(OCL)   Coll->including(elem)
```

- Intersezione

```
(OCL)   Coll1->intersection(Coll2)
(Alloy) aExp1 & aExp2
```

- Differenza

```
(OCL)   Coll1 - Coll2
(Alloy) aExp1 - aExp2
```

Da notare come questa operazione OCL abbia stranamente una diversa sintassi rispetto a tutte le altre, non vi è infatti l'usuale freccia ma vi è il simbolo "-" usato anche in Alloy.

In maniera analoga all'unione, se in Alloy sappiamo con sicurezza che una delle due espressioni denota un insieme di un solo elemento, è più coerente accostarla all'operazione OCL "excluding" per una maggiore chiarezza estetica :

```
(OCL)   Coll->excluding(elem)
```

In OCL è trattata anche l'operazione di differenza simmetrica

```
(OCL)   Coll1->symmetricDifference(Coll2)
```

In Alloy non è previsto nessun operatore che consenta questa operazione, tuttavia è possibile incontrare la situazione semanticamente affine:

```
(Alloy) (aExp1 + aExp2) - (aExp1 & aExp2)
```

- Prodotto Cartesiano

```
(OCL)   Coll1->product(Coll2)
(Alloy) aExp1->aExp2
```

3.5 Operazioni in OCL

Verrano mappate le principali operazioni OCL con le corrispettive versioni Alloy in modo da poter dare, a livello visivo, un ausilio per un eventuale traduzione dei vincoli del modello tra i due linguaggi.

- forAll

```
(OCL)   Exp->forAll(x | Formula[x] )
(Alloy) all x:Exp | Formula[x]
```

- exists

```
(OCL)   Exp->exists(x | Formula[x] )
(Alloy) some x:Exp | Formula[x]
```

- size

```
(OCL)   Exp->size()
(Alloy) #Exp
```

- isEmpty/notEmpty

```
(OCL)   Exp->isEmpty()
(Alloy) no Exp

(OCL)   Exp->notEmpty()
(Alloy) some Exp
```

- includes/ includesAll

```
(OCL)   Exp->includes(elem)
(Alloy) elem in Exp

(OCL)   Exp1->includesAll(Exp2)
(Alloy) Exp2 in Exp1
```

- excludes / excludesAll

```
(OCL)   Exp->excludes(elem)
(Alloy) elem not in Exp

(OCL)   Exp1->excludesAll(Exp2)
(Alloy) Exp2 not in Exp1
```

- select() : Exp

```
(OCL)   Exp->select(x | Formula[x])
(Alloy) { x: Exp | Formula[x] }
```

- reject() : Exp

```
(OCL)   Exp->reject(x | Formula[x])
(Alloy) { x: Exp | not Formula[x] }
```

- sum : Integer

```
(OCL)   Exp->sum()
(Alloy) sum(Exp)
```

- one : boolean

```
(OCL)   Exp->one(x | Formula[x])
(Alloy) one x:Exp | Formula[x]
```

3.6 if-then-else statement

In Alloy per poter esprimere lo statement "if-then-else" si fa ricorso all'operatore di implicazione logica

```
(OCL)   if oExp1 then oExp2 else oExp3
(Alloy) aExp1 implies aExp2 else aExp3
```

3.7 Insiemi Alloy e Collezioni OCL

Una tra le principali differenze tra il linguaggio Alloy ed il linguaggio OCL è sulla gestione degli insiemi: in Alloy un insieme è sempre senza ripetizioni e senza un ordinamento sugli elementi; in OCL invece si hanno quattro tipi di insiemi, chiamate collezioni :

Set: come in Alloy, senza ripetizioni e senza ordinamento sugli elementi

Bag: con possibili ripetizioni e senza ordinamento sugli elementi

OrderedSet: senza ripetizioni e con ordinamento sugli elementi

Sequence: con possibili ripetizioni e con ordinamento sugli elementi

In questo lavoro di tesi si darà, nel paragrafo successivo, una soluzione per poter gestire le collezioni di tipo “Bag” in Alloy. Per quanto riguarda le collezioni di tipo “Ordered-Set” e “Sequence” purtroppo, data l’impossibilità di poter impostare un ordinamento sugli elementi di un insieme arbitrario in Alloy, non è stata trovata nessuna soluzione che permetta una loro totale traduzione; tuttavia esistono delle disposizioni sintattiche messe a disposizione in maniera nativa che consentono una parziale somiglianza di comportamento.

- il modulo ordering

E’ possibile impostare un ordinamento sugli atomi degli insiemi semplici, ovveri gli insiemi di atomi definiti da una signature. Importando il modulo “ordering” con l’istruzione open da inserire ad inizio sorgente

```
open util/ordering[Giornata]

sig Giornata { ... }
```

si informa l’analizzatore che vogliamo disporre di un ordinamento sugli atomi di tipo Giornata. Per procedere ai confronti si utilizzano le funzioni messe a disposizione dal modulo:

siano g1 e g2 due atomo di tipo Giornata, la formula

```
g1 in prevs[g2]
```

sarà vera se g1 $<$ g2, mentre

```
g1 in nexts[g2]
```

sarà vera se g1 $>$ g2.

- la keyword seq

In Alloy si può usare il quantificatore speciale “seq” nella definizione di una relazione, vediamo un esempio:

```
sig Studente { esami: seq Esame }
sig Esame {}
```

ogni atomo di tipo Studente, tramite la relazione “esami”, sarà associato a delle coppie <Int,Esame> dove gli atomi di tipo Int saranno non negativi e consecutivi, instaurando in questo modo una sequenza di esami che potranno comparire più volte; ovvero uno studente s potrà essere associato ad esempio all’insieme

```
{ 0->Esame$2, 1->Esame$3, 2->Esame$1, 3->Esame$2 }
```

inoltre, se nel codice compare la keyword “seq” si potrà usufruire di alcune funzioni, da applicare in questo caso insieme ad un atomo di tipo Studente e alla relazione “esami”, verranno mostrate le principali:

s.esami.elems: denota gli esami sostenuti dallo studente s

s.esami.first: denota il primo esame sostenuto dallo studente s

s.esami.last: denota l'ultimo esame sostenuto dallo studente s

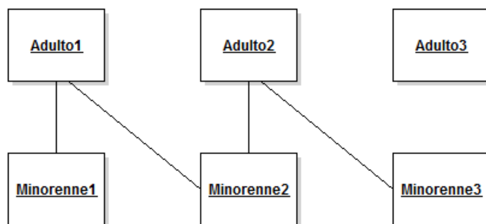
s.esami.hasDups: denota un valore booleano a seconda della presenza di esami ripetuti dallo studente s

s1.esami.append[s2.esami]: denota la concatenazione degli esami sostenuti dallo studente s1 e dallo studente s2 rispettando la continuità degli indici.

s.esami.inds: ritorna gli interi utilizzati per definire la sequenza di esami.

3.8 Bag in Alloy

Prendiamo adesso il seguente caso ordinario in UML/OCL: ci sono due classi, Adulto e Minorenne, ogni minorenne può essere associato a uno o due Maggiorennni (banalmente i genitori), quindi possiamo avere la seguente situazione:



In OCL anche essendo gli insiemi Adulto e Minorenne dei Set, l'espressione :

```
Adulto.allInstances.figli
```

dove "figli" è il ruolo della associazione da Adulto a Minorenne, darà in output automaticamente un Bag (!) : {Minorenne1, Minorenne2, Minorenne2, Minorenne3}, invece in Alloy l'espressione equivalente

```
Adulto.figli
```

darà in output un Set senza ripetizioni : {Minorenne1, Minorenne2, Minorenne3}

Sfruttando le funzioni in Alloy, che godono anche dell'overloading, e la possibilità di riferirsi allo stesso tempo sia un atomo generico e sia all'insieme di tutte le istanze in gioco tramite la keyword "univ", si è pensato di definire un modulo, del tutto generale e non ristretto al caso specifico Adulto-Minorenne, che consenta di creare e gestire insiemi di coppie <elemento-volte> simboleggiante un bag OCL da poter inserire ad inizio sorgente Alloy. Questo modulo scritto appositamente per questo lavoro di tesi, avrà le seguenti funzioni applicabili su insiemi sia Set che Bag

- funzione che trasforma un Set in un Bag semplicemente creando un insieme di coppie <elemento,1> dove 1 simboleggia il numero di volte, che l'elemento compare nel Bag

```
fun toBag[s: set univ] : univ->Int {
  s->1
}
```

- funzione che trasforma un Bag in un Set

```
fun toSet[s: univ->Int] : univ set {
  s.Int
}
```

- unione tra due insiemi di tipo Bag

```
fun union[s1: univ->Int, s2: univ->Int] : univ->Int {
  {
    x:(s1+s2).Int, num: Int | {
      x in (s1+s2).Int
      num = add[x.s1, x.s2]
    }
  }
}
```

- unione tra un Bag ed un Set; secondo lo standard OCL darà in output un Bag

```
fun union[s1: univ->Int, s2: set univ] : univ->Int {
  union[s1, toBag[s2]]
}
```

- unione tra un Set ed un Bag; secondo lo standard OCL darà in output un Bag

```
fun union[s1: set univ, s2: univ->Int] : univ->Int {
  union[toBag[s1], s2]
}
```

- intersezione tra due Bag

```
fun intersection[s1: univ->Int, s2: univ->Int] : univ->Int {
  {
    x: s1.toSet & s2.toSet, num:Int | {
      x in (s1.toSet & s2.toSet)
      num = integer/min[x.s1 + x.s2]
    }
  }
}
```

- intersezione tra un Bag ed un Set; secondo lo standard OCL darà in output un Set

```
fun intersection[s1: univ->Int, s2: set univ] : set univ {
  s1.toSet & s2
}
```

- joinBag : un join tra due insiemi Set che però ritorna un Bag implementando il comportamento cercato visto all'inizio del paragrafo tra le entità Adulto e Minorenne

```
fun joinBag[s: set univ, rel: univ->univ] : univ->Int {
  {
    x:s.rel, volte:Int | {
      x in s.rel
      volte = #{ y:s | x in y.rel }
    }
  }
}
```

ogni funzione può essere applicata direttamente sul primo parametro e chiamata senza quest'ultimo; quindi avremo implicitamente a disposizione una somiglianza sintattica tra la versione nativa del join in Alloy e il nuovo join definito. Ad esempio, sia a un atomo di tipo Adulto con:

```
a.figli
```

si denoterà il normale insieme di tipo Set senza ripetizioni, e con

```
a.joinBag[figli]
```

si denoterà un insieme di tipo Bag con possibili ripetizioni.

3.9 Cosa non si può fare in Alloy

Nonostante la forte espressività di Alloy, non è possibile definire ogni tipo di vincolo rispetto al linguaggio OCL, vediamo un caso rilevante.

Vogliamo definire un mondo composto da studenti ed esami; ogni studente sostiene degli esami e ad ogni esame è associato un valore numerico simboleggiante i cfu. In OCL è molto semplice esprimere il vincolo:

"ogni studente non può eccedere i 180 cfu"

```
class Studente
end

Class Esame
attributes
  cfu: Integer
end

association Esami between
  Studente[*] role sostenuto
```

```
  Esame[*] role sostiene
end

constraints

context Esame
inv intervallo: self.cfu > 0 and self.cfu <= 10

context Studente
inv limite: self.sostiene.cfu->sum() <= 180
```

in Alloy, applicando anche la bidirezionalità, tutto ciò può essere espresso come :

```
sig Studente {
  sostiene: set Esame
}

sig Esame {
  cfu: Int,
  sostenuto: set Studente
} { cfu > 0 and cfu =< 10 }

fact bidirezionalita {
  all s:Studente, e:Esame | e in s.sostiene <=> s in e.sostenuto
}

fact limite {
  all s:Studente | sum(s.sostiene.cfu) =< 180
}
```

Come possiamo vedere la sintassi è quasi identica e gode in entrambi i casi un'ottima leggibilità; c'è però il problema che in Alloy l'espressione "s.sostiene.cfu" denota un Set in cui non ci sono ripetizioni, quindi se in una istanza legale del modello abbiamo due o più esami con lo stesso numero di cfu x, nell'insieme ottenuto dall'espressione "s.sostiene.cfu" vi sarà solo un riferimento ad x, e quindi probabilmente rendendo non veritiero il vincolo che vogliamo imporre; anche applicando la tecnica precedentemente descritta, cioè importando il modulo e utilizzando la sintassi : "s.sostiene.joinBag[cfu]" otterremmo si, un bag denotante il giusto insieme di crediti con le relative molteplicità, ma incapperemmo in un errore di compilazione perchè l'operatore "sum" vuole solo insiemi unari di interi e non insiemi di coppie.

Capitolo 4

Modellazione dinamica

Il linguaggio Alloy è stato concepito per una modellazione statica che difatti è stata utilizzata per questo lavoro di tesi; come complemento finale al lavoro svolto vedremo come sia possibile donare della dinamicità ai modelli Alloy. Tramite la tecnica che verrà descritta in questo capitolo, semplicemente aumentando l'arietà delle relazioni che possono subire dei cambiamenti nel tempo, si creerà in maniera del tutto implicita una visione dinamica restando a tutti gli effetti conformi ad un modello Alloy statico. In coerenza con il tipo di studio affrontato nei capitoli precedenti, non si mostreranno delle metodologie progettuali volte a risolvere determinate situazioni pratiche ma verrà mostrato e illustrato in dettaglio un caso base da cui partire per affrontare casi più complessi. Questo nuovo approccio risulta pertinente al lavoro di tesi, in quanto con questa nuova visione possiamo dare una specifica sulle operazioni delle classi UML previamente tradotte in Alloy analizzando le pre e post condizioni. Bisogna comunque ammettere che la tecnica impesantisce notevolmente il modello, sia a livello di numero di variabili in gioco con conseguente risposta "lenta" da parte dell'analizzatore, sia a livello di complessità sul numero di aspetti che il modellatore è costretto a gestire; quindi se ne consiglia un utilizzo solo per delle verifiche singole e non estremamente complesse.

4.1 Elaborazione di un modello dinamico

Partiamo considerando un semplice caso statico : un modello in cui ci sono dei libri che possono essere riposti in degli scaffali; ogni libro può essere riposto al massimo su uno scaffale e ogni scaffale ha una sua capacità massima.

```
sig Libro { riposto: lone Scaffale }
sig Scaffale {
  contiene: set Libro,
  capienza: Int
} { capienza > 0 }

fact bidirezionalita { all l:Libro, s:Scaffale | l in s.contiene <=> s in l.riposto }
fact limite { all s:Scaffale | #s.contiene =< s.capienza }
```

Ogni istanza valida del modello simboleggerà una probabile configurazione che può essere vista come uno "snapshot" riferito ad un determinato istante di tempo. Per poter donare dinamicità al modello bisogna quindi creare innanzitutto una signature vuota che chiameremo Tempo ed informare l'analizzatore che vogliamo avere un ordinamento su questi tipi di atomi con l'istruzione ad inizio sorgente :

```
open util/ordering[Tempo]
```

un riferimento a questa signature dovrà essere aggiunto nella definizione delle relazioni che possono avere riferimenti diversi nel tempo, lasciando inalterate le altre, come di seguito :

```
sig Libro { riposto: Scaffale lone -> Tempo }
sig Scaffale {
  contiene: Libro -> Tempo,
  capienza: Int
}

fact bidirezionalita { all l:Libro, s:Scaffale, t:Tempo | l in s.contiene.t <=> s in l.riposto.t }
fact limite { all s:Scaffale, t:Tempo | #s.contiene.t =< s.capienza }
```

Si può notare come i nuovi vincoli nella versione dinamica abbiano subito semplicissime modifiche : c'è solo l'aggiunta di un join con un atomo di tipo tempo per selezionare una determinata situazione temporale; "capienza" invece viene lasciata inalterata sia nella definizione della signature Scaffale sia nei vincoli, perchè è un tipo di informazione che si vuole mantenere fissa e quindi non risente di un criterio temporale.

"ordering" è un modulo Alloy scritto dai progettisti del linguaggio; esso ci permetterà di usufruire anche delle seguenti funzioni :

first[] : ritorna il primo elemento

last[] : ritorna l'ultimo elemento

next[t:Time] : ritorna l'atomo successivo a t nell'ordinamento

prev[t:Time] : ritorna l'atomo precedente a t nell'ordinamento

nexts[t:Time] : ritorna l'insieme degli atomi successivi a t

prevs[t:Time] : ritorna l'insieme degli atomi precedenti a t

4.2 Operazioni come predicati

A questo punto possiamo procedere con la stesura di tre possibili operazioni. Ognuna di queste operazioni avrà sempre un parametro di tipo Tempo; esso servirà per specificare la situazione "pre", mentre la situzione "post" verrà indicata con "next[t]" ovvero l'istante di tempo successivo al parametro t. Dopo aver fatto le assunzioni per la situazione iniziale bisogna utilizzare i due consecutivi atomi di tempo per manipolare nel modo aspettato

TUTTE relazioni rispetto la situazione temporale menzionando anche quelle ininfluenti ai fini di una corretta applicazione dell'operazione. Questo obbligo di gestione serve per creare una continuità di legami tra gli atomi nel modello che potrebbe altrimenti essere interrotta con le "chiamate" dell'operazione. Infine va precisato che per poter usufruire di questa nuova visione dinamica, gli atomi di tempo non devono essere visti come istanti di tempo che scandiscono in maniera precisa il lasso di tempo totale sul quale si sta lavorando, ma devono essere visti come gli istanti di tempo in cui "succede qualcosa", ovvero istanti in cui una delle operazioni definite viene verificata; in seguito vedremo come impostare la non concorrenza delle operazioni per uno stesso istante di tempo.

- aggiungi

Precondizione: non esiste nessuno scaffale contenente il libro l

Postcondizione: il libro l è stato aggiunto allo scaffale s

```
pred aggiungi[t:Time, s:Scaffale, l:Libro] {
  //pre
  no s:Scaffale | l in s.contiene.t

  //post
  contiene.(next[t]) = contiene.t + s->l
}
```

Per ribadire il concetto menzionato precedentemente, diciamo che se nella definizione del modello avessimo avuto anche un'altra relazione dinamica "rel" estranea all'operazione di aggiunta, avremmo comunque dovuta gestirla nella post-condizione con:

```
rel.(next[t]) = rel.t
```

altrimenti nelle istanze valide del modello, l'analizzatore per l'istante di tempo t avrebbe potuto gestire questa relazione "a piacimento" e quindi interrompendo molto probabilmente il legame tra gli atomi che invece si vuole continuare ad avere.

- sposta

Precondizione: i due parametri "da" e "a" di tipo Scaffale devono essere diversi

Precondizione: lo scaffale "da" contiene il libro l

Precondizione: lo scaffale "a" non contiene il libro l

Postcondizione: il libro l è stato spostato in modo corretto

```
pred sposta[t:Time, da:Scaffale, a:Scaffale, l:Libro] {
  //pre
  da != a
  l in da.contiene.t
  l not in a.contiene.t

  //post
  contiene.(next[t]) = contiene.t + a->l - da->l
}
```

- rimuovi

Precondizione: lo scaffale s contiene il libro l

Postcondizione: il libro l è stato rimosso dallo scaffale s

```
pred rimuovi[t:Time, s:Scaffale, l:Libro] {
  //pre
  l in s.contiene.t

  //post
  contiene.(next[t]) = contiene.t - s->l
}
```

Le operazioni che sono state definite, oltre ad essere usate nella fase di specifica, possono essere utilizzate per analizzare i possibili comportamenti del sistema. Per usufruire di questo servizio è d'obbligo specificare un ulteriore vincolo che imponga al sistema di effettuare un'operazione per volta

```
fact {
  all t:Time-last | one s:Scaffale, l:Libro | {
    //solo aggiungi
    (aggiungi[t,s,l] and not rimuovi[t,s,l] and not (one da:Scaffale | sposta[t,da,s,l])) or
    //solo rimuovi
    (not aggiungi[t,s,l] and rimuovi[t,s,l] and not (one da:Scaffale | sposta[t,da,s,l])) or
    //solo sposta
    (not aggiungi[t,s,l] and not rimuovi[t,s,l] and (one da:Scaffale | sposta[t,da,s,l]))
  }
}
```

Inoltre si tende a specificare anche la situazione all'istante iniziale, per poter analizzare meglio la serie di "snapshot" che verranno generate; ad esempio

"all'inizio tutti gli scaffali sono vuoti"

```
fact inizio {
  no contiene.first
}
```

Con queste dovute precauzioni, quindi possiamo scrivere altri vincoli di stampo dinamico; come ad esempio :

"non è vero che non si possono avere tre aggiunte di seguito per uno stesso scaffale"

```
check {
  no t:Tempo | some s:Scaffale | {
    let t2 = next[t], t3 = next[t2] | {
      some l:Libro | aggiungi[t,s,l]
      some l:Libro | aggiungi[t2,s,l]
      some l:Libro | aggiungi[t3,s,l]
    }
  }
} for 7
```

l'analizzatore ci mostrerà un controesempio.

Conclusioni

Nella trattazione abbiamo illustrato come il linguaggio di modellazione Alloy può essere inserito nel già consolidato ambiente UML/OCL per l'analisi dei requisiti e l'analisi del dominio offrendo delle tecniche e soluzioni per minimizzare le differenze linguistiche e di gestione. Il risultato finale è un nuovo approccio che a differenza dello standard de facto esistente consente di esplorare il funzionamento di un modello, testare proprietà, scovare errori e mettere in risalto le incosistenze. Nel capitolo 3 è stata mostrata una mappatura, senza pretese di esaustività, tra le principali operazioni in OCL e i costrutti Alloy che rafforza l'idea di una ipotetica automatizzazione di traduzione tra i due linguaggi.

L'approccio Alloy può essere utilizzato nelle prime fasi del ciclo di vita di qualsiasi sistema software, e incontra maggiori successi nei sistemi "safety-critical" dove piccoli errori di specifica possono portare anche a situazioni disastrose. In questi sistemi, la fase finale di testing non può assicurare la totale correttezza del comportamento del sistema perchè una simulazione può rilevare sì la presenza di bachi ma non la totale assenza.

L'uso di un nuovo approccio può però scoraggiare un progettista, dato che richiederà sempre preparazioni e competenze ulteriori rispetto lo standard a cui si è abituati ma l'alto costo di un fallimento nei sistemi dove l'affidabilità è il principio basilare rende le compagnie propense ad accettare questo onere per assicurarsi che il prodotto software sia il più fidato possibile. Tutti gli esempi base che sono stati usati in questo lavoro di tesi, cercano quindi di consentire la più veloce comprensione del linguaggio Alloy e delle tecniche che sono state introdotte.

Bibliografia

[1] D. Jackson : "Software Abstraction - Language and Analysis". MIT Press 2006

[2] Alloy documentation [online] : http://alloy.mit.edu/alloy/documentation.html

[3] K. Anastasakis : "A model driven approach for the automated analysis of UML class diagram". Tesi Università di Birmingham, 2009

[4] K. Anastasakis, B. Bordbar, G. Georg, I. Ray : "UML2Alloy: A Challenging Model Transformation" 2007

[5] S.M.A. Shah, K. Anastasakis, B. Bordbar : "From UML to Alloy and Back Again". School of Computer Science, The University of Birmingham 2010

[6] A. Hall : "Seven myths of formal methods". IEEE Software 1990

[7] A. Garis, A.Cunha, D. Riesco : "Translating Alloy specifications to UML Class Diagram Annotated with OCL" 2010

[8] J. Cabot, M. Gogolla : "Object Constraint Language : A definitive guide" 2010

[9] M. Gogolla, J. Bohling, M. Richters : "Validating UML and OCL models in USE by automatic snapshot generation. Software and Systems Modeling" 2005

[10] N. Asoudeh, R. Khosravi : "Alloy as a language for Domain Modeling", School of Electrical and Computer Engineering Teheran 2007

[11] Y. He : "Comparison of the Languages Alloy and UML", Software Engineering Research and Practice (SERP) 2006

[12] G. Genova, J. Llorens, P. Martinez : "Semantics of the Minimum Multiplicity in Ternary Associations in UML". Computer Science Department, Carlos III University of Madrid 2001

[13] T. Massoni, R. Gheyi, P. Borba : "Formal Refactoring for UML Class Diagrams", Informatics Center – Federal University of Pernambuco (UFPE) 2005

[14] T. Massoni, R. Gheyi, P. Borba : "A Model-driven Approach to Program Refactoring", Informatics Center – Federal University of Pernambuco (UFPE) 2005

[15] M. Vaziri, D. Jackson : "Some Shortcomings of OCL, the Object Constraint Language of UML". MIT Press 2000

[16] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, R. Drechsler : "Verifying UML/OCL Models Using Boolean Satisfiability". Group for Computer Architecture, University of Bremen 2010

[17] B. Meyer : "Object-Oriented Software Construction". Prentice Hall 1997

[18] A. Evans, R. France, K. Lano, B. Rumpe : "The UML as a Formal Modeling Notation". 2000

[19] B. Beckert, U. Keller, P. H. Schmitt : "Translating the Object Constraint Language into First-order Predicate Logic". Institut fur Logik, Komplexitat und Deduktionssysteme, Universitat Karlsruhe 2002

[20] B. Meyer : On Formalism in Specifications. University of California, 1985