Università degli Studi
di Roma "Sapienza"

Facoltà di Ingegneria

Tesina del corso "Metodi Formali
nell'Ingegneria del Software"

Java Modeling Language (JML)

Autore:
Alberto Cerullo

SOMMARIO

Introduzione

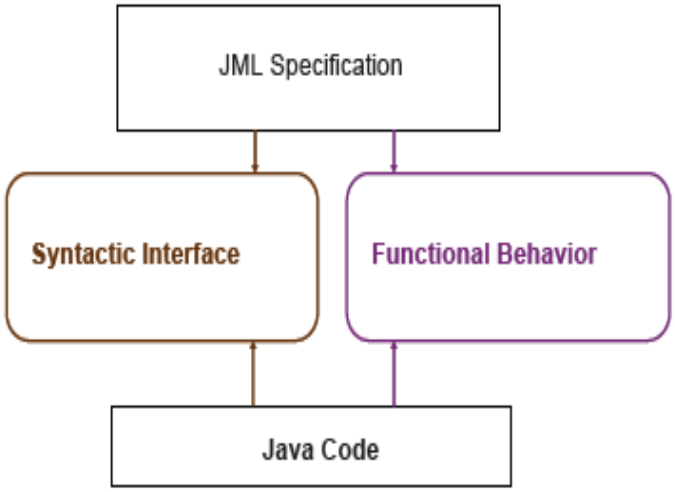
In questo lavoro presenteremo JML, un linguaggio per la definizione di specifiche formali in Java.
L'ambito di applicazione di tale linguaggio è quello della correttezza dei programmi, una delle tematiche affrontate nel corso di "Metodi Formali nell'Ingegneria del Software", nonché uno degli argomenti cardine delle tecniche formali.
L'obiettivo del lavoro è quello di dimostrare come è possibile definire specifiche per classi Java attraverso una sintassi molto simile a Java. In questa maniera, infatti, si facilita il lavoro di progettazione dell'architettura delle classi, non essendo necessario lo studio e la definizione di specifiche attraverso un formalismo completamente diverso da Java, che richiederebbe uno sforzo maggiore.

L'argomento verrà affrontato e diviso in sei parti:

- La prima avrà come obiettivo principale quello di fare una panoramica sul linguaggio JML, sulle sue caratteristiche e sui tool che lo utilizzano.
- La seconda parte servirà per delineare i costrutti e gli aspetti principali di JML.
- Nella terza verranno discusse le caratteristiche di tool utili per l'assertion checking a runtime e per la static verification.
- Nella quarta verranno fatti degli esempi pratici d'utilizzo dei costrutti di JML applicati ad alcuni tool che lavorano con questo linguaggio.
- Nella quinta verranno presentati brevemente altri programmi che utilizzano JML.
- Nell'ultima parte verranno tratte le conclusioni finali relative al lavoro svolto ed al linguaggio oggetto di studio.

Capitolo 1. Panoramica su JML

JML è l'acronimo di Java Modeling Language ed è un linguaggio di specifica progettato per moduli Java: classi, interfacce e metodi. Le sue funzionalità sono sintetizzabili in due profili principali: specifica del comportamento delle classi Java e definizione ed implementazione delle decisioni di progetto.
La definizione formale è esprimibile attraverso la seguente frase: "JML is a behavioral interface specification language tailored to Java"; dunque, tramite JML, si può definire sia l'interfaccia sintattica del codice scritto in Java sia il suo comportamento.
Con la locuzione "interfaccia sintattica" si intende la parte relativa ai nomi, alla visibilità e alle informazioni sui tipi relativi ad una classe o ad un metodo. Ad esempio, l'interfaccia sintattica di un metodo può essere scorta nella sua intestazione dove sono presenti il tipo di ritorno, i parametri formali con i relativi tipi ed il tipo di eccezioni che può lanciare.
Il comportamento di un codice Java, invece, descrive ciò che bisognerebbe aspettarsi a runtime quando il codice viene eseguito. Tipicamente questo può essere garantito definendo pre-condizioni e post-condizioni nelle specifiche.



JML è stato ideato anche per rispondere alle esigenze della metodologia di "Progettazione a Contratto" (Design by Contract), che consiste in una serie di tecniche utili per la progettazione di software di qualità. In particolare, l'idea che sta alla base di questa metodologia afferma che ogni classe ed ogni suo cliente (ossia le altre classi che la utilizzano) definiscano tra loro un "contratto".
Per far sì che il contratto sia valido, il cliente deve garantire alcune condizioni prima di chiamare un metodo della classe a cui è vincolato, mentre quest'ultima garantisce che dopo la chiamata del metodo verranno mantenute e/o soddisfatte alcune proprietà. La novità di questa metodologia di progettazione è che i contratti non vengono definiti esternamente al codice, ma diventano un tutt'uno con esso. Infatti le specifiche sono definite all'interno del codice del programma e sono tradotte in codice eseguibile dal compilatore. Perciò, ogni violazione del contratto, mentre il programma è in esecuzione, viene subito riconosciuta e opportunamente segnalata.

Attraverso l'uso di precondizioni e postcondizioni, dichiarate nella definizione del contratto, il cliente di un metodo deve assicurare la precondizione avendo garanzia della soddisfacibilità della postcondizione, mentre il metodo, a sua volta, deve assicurare la postcondizione, assumendo come vera la precondizione.

Diversi linguaggi di programmazione dispongono di estensioni per la gestione della "Progettazione a Contratto" come ad esempio ADA, il C++ o hanno un linguaggio a parte che si preoccupa di rispondere alle esigenze di questa tipologia di design. E' il caso, ad esempio, di Promela, il linguaggio di SPIN.

JML fa uso di una sintassi Eiffel-like, il primo linguaggio creato appositamente per il problema della "Progettazione a Contratto", anche se rispetto a quest'ultimo, risulta essere maggiormente espressivo. Tale sintassi, infatti, è combinata con l'espressività ed il formalismo tipico dei linguaggi di specifica orientati ai modelli. Come avviene in Eiffel, JML fa uso di asserzioni le quali, attraverso espressioni Java, sono utili per definire pre-condizioni, post-condizioni ed invarianti.
L'ovvio vantaggio di usare una notazione molto simile a Java per la definizione delle asserzioni consiste, dal punto di vista dei programmatori, nella facilità di imparare tale formalismo e nella rapidità di utilizzo, rispetto ad un linguaggio che faccia uso di una notazione matematica creata allo scopo.
Tuttavia, le espressioni in Java mancano di una certa forma di espressività che rendono altri linguaggi specializzati in asserzioni maggiormente convenienti. Il Java Modeling Language risolve questo problema estendendo le espressioni Java con diversi costrutti specifici, come ad esempio i quantificatori.

JML, nonostante la similarità degli acronimi, è diverso da UML. Mentre quest'ultimo si pone l'obiettivo di aiutare ad ampio spettro il progettista nelle diverse fasi dei processi d'analisi e di progettazione, JML tenta di descrivere il comportamento di classi ed interfacce Java registrando scelte dettagliate per la progettazione e decisioni di implementazione.

Più simile a JML, invece, è OCL (Object Constraint Language) anch'esso utilizzato per la definizione di precondizioni e postcondizioni.
OCL non è progettato esplicitamente per un unico linguaggio di programmazione, come JML con Java, ma è più generico.
Lo svantaggio di mancanza di generalità che paga JML, però, è ripagato dalla facilità d'utilizzo della sua sintassi.
La definizione, ad esempio, di un invariante in JML:

invariant var!=null && var.length>0; (JML)

è senz'altro maggiormente comprensibile rispetto a:

inv: var <> null and var->size() >0 (OCL)

Oltre a queste differenze di natura sintattica, ci sono anche differenze, non meno importanti, dal punto di vista semantico. JML si basa sulla semantica di Java; quindi ci si può aspettare lo stesso comportamento e le stesse regole per operazioni e costrutti. Per OCL non è esattamente così; difatti solo da poco è stata definita una semantica per tale formalismo.

L’importanza che sta avendo JML, oltre per i motivi già elencati, è dovuta al fatto che ha una comunità molto ampia di sviluppatori che si interessa a questo progetto (JML, infatti, è un progetto Open source).
Il progetto legato a JML è relativamente maturo. Sourceforge, infatti, ne cataloga la qualità come “Beta quality”. Ci sono circa 23 gruppi nel mondo che lavorano su questo linguaggio e sui vari tool ad esso associati ed oltre 135 pubblicazioni sull’argomento, anche se manca una documentazione formale del linguaggio.
Esistono, inoltre, diversi tool che lavorano con JML, ognuno con scopi diversi. Esistono, infatti, tool per fare assertion checking a runtime, static analysis, verifica formale tramite theorem prover, runtime debugging, unit testing, documentation, etc.
Una tale ampia offerta, permette di scegliere il livello di completezza e accuratezza nelle verifiche formali di un programma in base ai differenti costi, utilizzando sempre lo stesso linguaggio di specifica.
Si potrebbe, ad esempio, iniziare da una tecnica che richiede meno tempo e fatica, come ad esempio, il checking di asserzioni a runtime per poi passare a tecniche più articolate o ad un merge di tali tecniche.

Capitolo 2. Costrutti ed aspetti di JML

Il Java Modeling Language è un progetto di ricerca realizzato presso l'Iowa State University.
In questa sezione introdurremo gli aspetti più importanti di JML descrivendo la sintassi e la semantica dei costrutti principali, nonché di quelli maggiormente utili ai nostri scopi.

JML permette di definire asserzioni all'interno del codice Java di una classe.
L'inserimento delle asserzioni, però non è invasivo. Infatti, il codice JML è inserito all'interno di commenti opportunamente marcati.

Due sono le tipologie di commenti valide per l'inserimento di codice JML in file .java:

- La prima modalità consiste nel commento ad una linea, seguito dal simbolo '@':

  ```
  //@   statement JML
  ```

- La seconda, invece, corrisponde al commento su più linee. Come in Java, si usano /* e */ per aprire e chiudere il commento. Si aggiunge,però, anche qui il simbolo speciale '@' sia all'apertura che alla chiusura del commento:

  ```
  /*@
          statement JML
  @*/
  ```

Commenti del tipo // @ oppure /* @ @ */, cioè che non presentano il simbolo speciale '@' come primo elemento subito dopo il backslash vengono ignorati.

Le asserzioni possono essere inserite negli header dei metodi, nel corpo delle classi(ad esempio per la definizione del comportamento degli invarianti o dei metodi) e nel corpo dei metodi.

Le asserzioni JML sono diverse dalla annotazioni Java (introdotte a partire da Java 5).
La possibile confusione potrebbe derivare dall'uso, che fanno entrambe, del carattere speciale '@'. Tuttavia oltre alla sintassi differente, le annotazioni vengono usate liberamente all'interno del codice ed hanno uno scopo d'utilizzo diverso. Senza entrare particolarmente nel dettaglio, attraverso le annotazioni è possibile associare metadati agli elementi di un programma (package, classi, metodi,etc.) e salvarli tramite il compilatore nei file .class. Durante l'esecuzione del programma è possibile, poi, ottenere tali metadata per determinare come interagire con gli elementi del programma stesso oppure per cambiarne il comportamento.

Alcune importanti keyword appartenenti al linguaggio JML sono:

- Keyword di alto livello in classi ed interfacce:
  - ***invariant*** : serve per definire gli invarianti
  - ***spec_public***: definisce la visibilità di un elemento del codice
  - ***non_null*** : non è ammesso che l'elemento, cui è associato, sia null

- Keyword per metodi e costruttori :
  - ***requires***: permette di definire le precondizioni
  - ***ensures***: serve per dichiarare una postcondizione
  - ***assignable***: identifica campi della classe che possono essere modificati dal metodo
  - ***pure***: serve per gestire il side-effect
  - ***old(var)***: riferimento al valore originario della variabile var
  - ***forall***: quantificatore universale
  - ***exists***: quantificatore esistenziale
  - ***result***: valore di ritorno del metodo
  - ***assert***: indica una proprietà che deve essere garantita ad un certo punto del codice
  - ***nothing***: viene usata come elemento di altre clausole (ad esempio assignable) per indicare nessun elemento o un insieme vuoto.

Nei paragrafi seguenti introdurremo queste ed altre parole chiave del linguaggio JML nei loro ambiti d'utilizzo.

2.1 Predicati ed espressioni

I predicati JML sono espressioni di specifica (definiti "spec-expressions") che hanno un valore di tipo booleano.
Tali espressioni, infatti, sono espressioni Java (che assumono lo stesso valore definito nelle specifiche del linguaggio Java), estese con espressioni primarie JML ed alcuni operatori ad hoc.
Questo produce delle conseguenze sull'interpretazione delle asserzioni. Infatti, un'asserzione è considerata rispettata e quindi valida se e solo se la sua interpretazione:

- restituisce il valore true,
- non causa eccezione.

Questa interpretazione delle asserzioni è chiamata "strong validity".

La precedenza tra gli operatori usati in JML è simile a quella in Java e può essere riassunta con il seguente specchietto (sono stati considerati gli operatori principali, trascurando quelli poco comuni e che non verranno usati in questo lavoro).

() \forall \exists \max \min \num_of \product \sum [] . e chiamate di metodi
< <= > >= <: instanceof
== !=
&
^
|
&&
||
= => <==
<==> <=!=>

Nelle asserzioni non si possono usare operatori quali ++ o -- e gli operatori di assegnamento che causerebbero side effect. All'interno delle asserzioni, infatti, è possibile usare solo metodi puri, cioè metodi che non fanno side effect.
Gli operatori che JML aggiunge rispetto a quelli già previsti in Java sono i seguenti = = > , < = = , < = = > , < = ! = > e <: che prevedono la seguente sintassi e semantica:

a = => b	a implica b
a <= = b	a segue da b (o similmente b implica a)
a <= => b	vale a se e solo se vale b
a <= ! => b	not(vale a se e solo se vale b)
a <: b	il tipo a è un sottotipo di b (l'operatore è anche riflessivo)

Tutte le keyword JML che possono essere usate nelle espressioni e che iniziano con un carattere alfabetico, presentano un backslash (\) davanti al nome, usato per non creare confusione con i nomi delle variabili usate all'interno di un programma.
La keyword **\result** può essere usata solo nelle post-condizioni di un metodo non void (clausola "ensures") [In realtà può essere usata anche nella clausole duration e workingspace, ma il loro significato ed utilizzo esula dagli scopi di questo lavoro].
\result rappresenta il valore di ritorno di un metodo.
\old e \pre hanno la sintassi ***\old (espr)*** e ***\pre(espr)*** ed indicano entrambe il valore che aveva l'espressione espr nel pre-stato di un metodo [per la definizione di pre-stato vedere la sezione 2.2]. La clausola \old va usata nelle postcondizioni, mentre \pre nelle asserzioni che compaiono nel corpo dei metodi (statement "assert" e "assume") e mai nelle post condizioni.

Nell'utilizzo di \old() e \pre() non vanno incluse variabili libere. Ad esempio è illegale la seguente espressione:

(\forall int i; 0 <= i && i < 7; \old(i < y)); *//illegale*

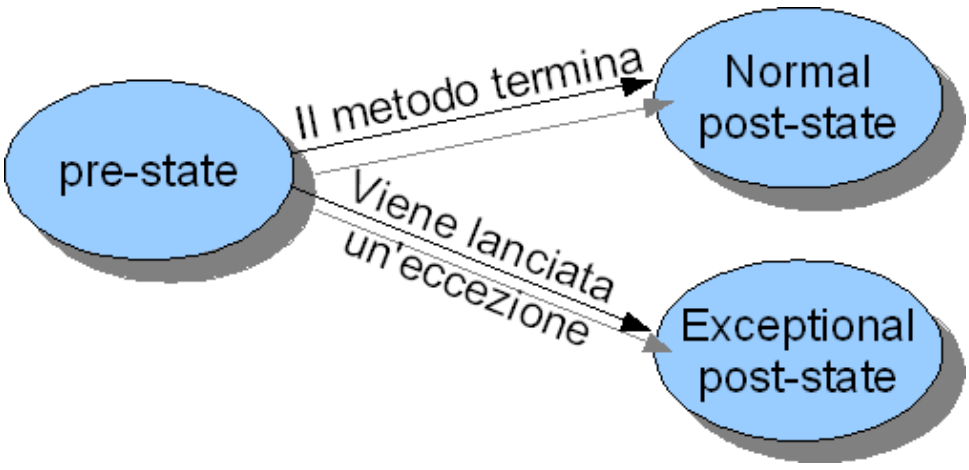
dato che \old() include un'occorrenza libera della variabile quantificata 'i',
mentre sono legali:

```
(\forall int i; 0 <= i && i < 7; i < \old(y));        // ok
\old((\forall int i; 0 <= i && i < 7; i < y));        // ok
```

Nella prima espressione, infatti, \old() non include la variabile quantificata "i" e nella seconda \old() include 'i', ma non è un'occorrenza libera di una variabile quantificata, dato che 'i' è limitata dalla dichiarazione all'interno di \old().

2.2 Ciclo di vita di un metodo

Ogni metodo annotato con specifiche JML deve essere invocato in uno stato in cui le precondizioni siano verificate. Tale stato viene chiamato "pre-state".
Se la precondizione o le precondizioni non sono verificate non è assicurato nulla sul risultato della chiamata del metodo, che potrebbe comportare un non ritorno del metodo stesso oppure un ciclo o un cambio arbitrario dello stato.
Se le precondizioni sono verificate, ci sono due possibili stati in cui ci si può trovare: il metodo termina senza lanciare eccezioni ("normal post-state") oppure può terminare con il lancio di un'eccezione("exceptional post-state"), se questa non è ereditata dalla classe java.lang.Error. Nel primo caso, cioè quando si è nel "Normal post-state" le post-condizioni sono soddisfatte; invece in un "Exceptional post-state" l'eccezione lanciata deve essere permessa nelle specifiche attraverso la clausola "*signals_only*" e lo stato d'eccezione deve garantire le corrispondenti condizioni definite nella clausola dell'eccezione. Di default la clausola signals_only non viene specificata e permette che siano lanciate le eccezioni presenti nella clausola throw nell'intestazione di un metodo.



In realtà esistono anche altri due stati in cui può terminare l'esecuzione di un metodo e sono lo stato "diverge" tipico dei cicli infiniti e dei casi in cui il chiamante non riceve il ritorno della propria chiamata (può essere segnalato tramite la key-word "*diverges*") ed inoltre il caso d'errore della Java Virtual Machine, che è fuori dal controllo del programmatore.

2.3 Quantificatori

In JML è possibile definire diversi tipi di quantificatori all'interno delle asserzioni.
La sintassi generale per l'uso dei quantificatori è la seguente:

(\quantificatore variabili ;[range ;] body);

Come possiamo vedere oltre al quantificatore ci sono tre parti: la prima è la dichiarazione di una o più variabili quantificate; la seconda, chiamata range, che può anche non essere presente, consiste in una o più espressioni che vincolano l'uso delle variabili e funge da filtro per i valori che possono essere assunti dalle variabili; infine una terza parte chiamata body utile per rappresentare la condizione da verificare relativa alle variabili del quantificatore. Il body è un'espressione di tipo booleano e deve essere vera per tutti i valori delle variabili che cadono nel range.

Esistono quattro tipologie di quantificatori: il quantificatore universale *"\forall"* e quello esistenziale *"\exists"*, i quantificatori generalizzati *\product, \min, \max* ed il quantificatore numerico: *\num_of.*

Un esempio pratico di quantificatore universale è il seguente:

(\forall int i,j; 0 <= i && i < j && j < a.length; a[i] < a[j]);

La riga precedente serve per descrivere la condizione che il vettore 'a' è ordinato.

La stessa espressione poteva essere scritta in maniera equivalente anche così:

(\forall int i,j; ;0 <= i && i < j && j < a.length ==> a[i] < a[j]);

Tuttavia è da preferire la prima forma perché senza il range il \forall non può essere eseguito. Può, però, essere usata la forma senza range ad esempio per scopi documentativi.

In generale, il \forall è eseguibile (cioè valutabile a runtime) se la condizione che viene specificata dal range individua un dominio finito. Quindi ad esempio la condizione *0<=i && i< a.length*; è eseguibile, mentre *i>0* non lo è.

I quantificatori \max,\min,\sum sono quantificatori generalizzati che restituiscono il massimo, il minimo o la somma dei valori definiti nel body , sempre rispettando le condizioni indicate nel range.
Per la computazione delle somme e dei prodotti JML sfrutta l'aritmetica di Java. Inoltre quando il range dei predicati non è soddisfacibile, la somma e la sottrazione restituiscono 0, mentre il prodotto 1.

Il quantificatore numerico \num_of, infine, restituisce il numero di valori per le variabili quantificate per le quali il range e l'espressione nel body sono vere. Sia il predicato del range che il body devono essere di tipo booleano e l'intera espressione quantificata è di tipo long.

2.4 Precondizioni e Postcondizioni

La semantica di un metodo Java può essere definita per mezzo di precondizioni e postcondizioni.
Una precondizione è una (o più) condizione che deve essere vera immediatamente prima dell'esecuzione di un'operazione (invocazione di un metodo o di un costruttore), mentre una postcondizione è una condizione che deve risultare vera subito dopo l'esecuzione di una operazione. Quest'ultima condizione per le postcondizioni vale se l'operazione ritorna senza lanciare un'eccezione.
Per definire in JML una precondizione si usa la clausola "requires", mentre per la postcondizione si usa "ensures".
La sintassi delle due clausole è la seguente:

requires pred;

ensures pred;

dove pred esprime un qualunque predicato.

In una singola specifica può essere incluso un numero qualsiasi di clausole requires. Se sono previste più clausole, queste hanno lo stesso significato di una singola clausola requires formata dalla congiunzione dei diversi predicati.
Dunque:

requires P; equivale a ***requires P && Q;***
requires Q;

Quando la clausola "requires" è omessa in una specifica vale il valore di default che è \not_specified nel caso lightweight, mentre è true per il caso heavyweight. [per i concetti "lightweight" ed "heavyweight" consultare la sezione 2.14].
Quanto appena detto per "requires", vale anche per la clausola "ensures".

Una post-condizione può contenere la clausola "\old(espr)". L'espressione contenuta nella parentesi è valutata nel pre-stato dell'operazione, non nel post-stato, e permette di rappresentare una relazione tra il pre-stato ed il post-stato.

Infine è possibile definire diverse pre-condizioni, non correlate tra di loro, con conseguenti post-condizioni per uno stesso metodo. Per fare ciò si può usare la keyword "also" nella seguente maniera:

requires P1 ;
ensures Q1;
also
requires P2 ;
ensures Q2;

oppure si può usare la seguente sintassi.

```
requires  P1 || P2
ensures \old(P1) ==>Q1;
ensures \old(P2) ==>Q2;
```

Un esempio del caso precedente è il seguente:

```
requires 0 <= a && a <= 150;
assignable age;
ensures age == a;
also
requires a < 0;
assignable age;
ensures age == \old(age)
&& \only_assigned(\nothing);
```

che può essere scritta anche nel seguente modo:

```
requires (0 <= a && a <= 150) || a < 0;
assignable age;
ensures \old(0 <= a && a <= 150)
==> (age == a);
ensures \old(a < 0)
==> (age == \old(age)
&& \only_assigned(\nothing));
```

In questo esempio se la variabile ‘a’ è compresa tra 0 e 150, la variabile ‘age’ assume il valore di ‘a’, invece se ‘a’ è minore di 0 ‘age’ mantiene il suo valore.

2.5 Invarianti

Un invariante è una proprietà che deve essere sempre mantenuta in ogni stato di una classe e deve essere vera quando il controllo non è interno ai metodi dell’oggetto. Perciò, un invariante è una proprietà che deve essere garantita quando termina l’esecuzione del costruttore di una classe Java ed all’inizio ed alla fine di un metodo.
Tipicamente, nella “Progettazione a contratto”, un invariante si definisce per un valore pubblico; tuttavia in JML si possono dichiarare invarianti anche elementi con visibilità più ristretta. Questi vengono definiti “representation invariant”, a differenza dei precedenti che vengono chiamati “type invariant”.

La sintassi per un invariante è (ovviamente all'interno dei commenti JML):

invariant <espressione_booleana>;

dove <espressione_booleana> sta ad indicare una qualunque espressione che si vuole verificare.

Ad esempio:

```
public int priorityLevel;
public int timeStamp;
/*@ public invariant
@ priorityLevel >= 0 && timeStamp >= 0;
@*/
```

afferma che le variabili priorityLevel e timeStamp devono essere sempre maggiori od uguali a 0.
Gli invarianti sono implicitamente inclusi all'interno delle pre- e post-condizioni e deve essere garantita la loro veridicità anche quando viene lanciata un'eccezione.
Tuttavia, possono verificarsi situazioni in cui sia necessario violare la condizione di invarianza. Per tal motivo è possibile utilizzare la keyword ***helper***. Definendo helper un costruttore o un metodo, questi sono esentati dal soddisfare l'invariante. Questi metodi devono essere dichiarati privati e possono essere usati, ad esempio, per casi di debugging.

Concludiamo questo paragrafo, sottolineando un altro importante uso degli invarianti. Infatti, si può usare un invariante per documentare e stabilire decisioni di progetto definendole direttamente nel codice.

Un esempio di quanto appena affermato è il seguente:

```
public class Directory {
private File[] files;
/*@ invariant
files != null
&&
(\forall int i; 0 <= i && i < files.length; files[i] != null && files[i].getParent()!=null);
@*/
```

Nel precedente esempio abbiamo reso esplicito come vogliamo che sia caratterizzata ed usata la variabile files[]. In particolare, abbiamo definito un array di elementi della classe File, in cui ogni cella è non vuota e contiene un elemento che restituisce un valore diverso da null se gli si applica il metodo getParent().
Attraverso l'uso esplicito di invarianti, evidentemente, migliora la comprensione del codice.

2.6 Assert

In JML è possibile definire asserzioni oltre che prima dell'header di metodi e costruttori o nel corpo delle classi, anche all'interno dei metodi.
A tale scopo si usa la keyword ***"assert"*** che serve per definire una proprietà che deve essere vera nel punto del codice dove si trova l'asserzione.
Anche in Java come in JML è presente lo statement assert. Per tal motivo JML distingue tra gli assert presenti nelle annotazioni e quelli che sono presenti esternamente alle annotazioni. Quelli esterni, ovviamente, appartengono al linguaggio Java e come tali vanno trattati. Infatti, a differenza degli assert JML l'espressione che segue l'"assert" Java può, eventualmente, fare side effect (in teoria questa pratica è fortemente sconsigliata).
In JML, invece, è presente un predicato che non può fare side effect, ma che usa una qualsiasi espressione booleana da verificare.
La sintassi di assert di JML è, dunque, la seguente:

assert pred [: expression];

La parte opzionale ": expression" deve essere di tipo String e corrisponde a ciò che deve essere visualizzato in caso di fallimento dell'asserzione.

Un esempio di questo costrutto può essere la seguente porzione di codice di un metodo, che vedremo in maggior dettaglio nell'esempio 4.2:

```
if (i <= 0 || j < 0) {
...
} else if (j < 5){
//@ assert i > 0 && 0 < j && j < 5;
...
} else {
//@ assert i > 0 && j > 5;
...
}
```

Nelle posizioni di codice dove è presente l'assert viene assicurato che il valore delle variabili i e j assuma un valore in un range ben preciso.

2.7 Clausola assignable

La clausola "assignable" viene usata nelle specifiche di un metodo per indicare tutti gli oggetti che non sono locali al metodo (ossia parametri e variabili della classe) e che possono essere modificati dal metodo.

La sintassi di assignable è la seguente:

assignable obj [, obj]

dove obj è l' oggetto da modificare. All' interno di una singola clausola assignable (come possiamo vedere dalla sintassi) è possibile indicare una lista di oggetti, separandoli semplicemente tramite una virgola.

Nell'uso di "assignable" va considerato che si devono sempre indicare gli oggetti e non i riferimenti.
Ad esempio se si vuole asserire che un metodo possa modificare un array 'arr' è errato il seguente codice:

```
//assignable arr;
public static void metodo(int [] arr){
```

L'errore consiste nel fatto che 'arr' non può essere modificato , dato che solo i suoi elementi possono esserlo.
Il codice corretto per l'esempio precedente è, dunque, il seguente:

```
//assignable arr[*];
public static void metodo(int [] arr){
```

in cui si asserisce che 'metodo' può modificare gli elementi dell'array 'arr'.

In generale la clausola assignable:

- deve indicare per i metodi statici gli oggetti che si vogliono modificare tra quelli riferiti dai parametri formali (ad esempio elementi di un array, campi di un oggetto, etc.)
- deve indicare nel caso degli array gli elementi (es. *assignable arr[1], arr[5];*) oppure il range degli elementi che vanno modificati (es. *assignable arr[1..8]*) oppure va usata la forma *assignable arr[*]*, come abbiamo visto nell' esempio precedente, per indicare che tutti gli elementi dell'array 'arr' sono modificabili.
- è usata nella forma ".*" per gli oggetti (ad esempio se c è l'oggetto di una classe C, va scritto *assignable c.*;*) per indicare che può essere modificato un campo dell'oggetto.

Questa clausola è opzionale, dunque può essere omessa, ma può essere utile per una maggiore comprensione del metodo e di ciò che deve fare.
E' da notare che l'assenza della clausola assignable deve essere intesa come l'assenza di "promesse" sulla modifica di elementi. Dunque, un metodo che non presenta nella propria specifica questa clausola può modificare un elemento oppure no.

Infine, qualora si voglia asserire in maniera esplicita, che il metodo non modifica nulla va usata l'espressione "*assignable \nothing*". L'espressione precedente indica che non ci sono variabili a cui può essere assegnato un valore; dunque i valori non sono modificati al termine della chiamata del metodo.

2.8 non-null e nullable

Una variabile può essere definita "non_null" all'interno delle specifiche JML. Tale particolare variabile indica che in ogni stato pubblico, la variabile non-null deve essere sempre diversa da null. Dunque, dopo l'esecuzione del costruttore e quando non è in esecuzione un metodo della classe cui appartiene la variabile, quest'ultima risulta sempre diversa da null.
Lo stesso risultato può essere ottenuto usando un invariante del tipo *"nome_variabile !=null"*.
Per asserire, invece, che una variabile può anche essere null, si può usare la parola chiave *"nullable"*. La scelta di usare "nullable" solitamente è fatta per rendere più chiara la specifica. Infatti di default JML considera tutti gli elementi come "nullable".

Esempi di applicazione di questa clausola sono i seguenti:

```
public class Directory {
private /*@ non_null @*/ File[] files;
void createSubdir(/*@ non_null @*/ String name){
...
Directory /*@ non_null @*/ getParent(){
...
}
}
```

Come possiamo vedere la keyword non_null può essere applicata a variabili, metodi e parametri formali di metodi.

2.9 Clausola signals ed "exceptional postcondition"

Una post-condizione d'eccezione (exceptional postcondition) serve per affermare ciò che è vero quando un metodo lancia un'eccezione. Per poterla utilizzare si usa la keyword "signals" che specifica l'exceptional postcondition, la proprietà che è garantita alla fine dell'invocazione del metodo (o del costruttore), quando l'invocazione del metodo (o del costruttore) termina lanciando un'eccezione.
La sintassi della clausola signals è la seguente:

signals (Exc [e]) Pred;

dove Exc è una sottoclasse di java.lang.Exception. Se Exc è una checked exception (ad esempio se non deriva da java.lang.RuntimeException), deve essere una delle eccezioni presenti nel clausola throws del metodo (o costruttore) o una superclasse o sottoclasse delle eccezioni dichiarate.
Informalmente la clausola oggetto d'esame afferma che se l'invocazione termina con il lancio dell'eccezione di tipo Exc, allora il predicato Pred è garantito nello stato finale per questa eccezione Exc.

In maniera equivalente potevamo scrivere la clausola come:

signals (java.lang.Exception e) (e instanceof Exc) ==> Pred;

Come visto nel caso delle precondizioni e postcondizioni anche per la signals, qualora siano presenti clausole multiple di questo tipo in una specifica, queste hanno lo stesso valore di un'unica clausola costruita come la congiunzione dei predicati di ciascuna singola clausola.
Ad esempio hanno lo stesso valore le seguenti formule:

signals (E1 e) R1;
signals (E2 e) R2;

e

signals (Exception e) ((e instanceof E1) ==> R1)
&& ((e instanceof E2) ==> R2);

Notiamo nel precedente esempio che se viene lanciata un'eccezione che è sia del tipo E1 che del tipo E2, allora sono garantite, quando termina l'invocazione del metodo, sia R1 che R2.

E' doveroso segnalare che una clausola signals non sta ad indicare che una certa eccezione deve essere lanciata, ma serve per specificare cosa accade quando si verifica una certa eccezione.

Alla stessa famiglia di signals, appartiene anche la clausola **signals_only**, utile per specificare quali eccezioni possono essere lanciate da un metodo ed, implicitamente, quali eccezioni non vengono lanciate.
La sintassi della clausola signals_only è la seguente:

signals_only E1, E2, ..., En;

che è equivalente a

signals (java.lang.Exception e)
e instanceof E1
|| *e instanceof E2*
|| ...
|| *e instanceof En;*

La differenza tra queste due clausole è, dunque, la seguente: signals indica che un'eccezione può essere lanciata durante l'esecuzione di un metodo e contemporaneamente segnala quale stato deve essere garantito dopo che l'eccezione viene lanciata; mentre signals_only indica semplicemente quali eccezioni possono verificarsi.

Se all'interno della specifica di un metodo non è presente nessuna clausola signals (ricordiamo,infatti, che signals è una clausola opzionale), viene usato il valore di default. Nelle specifiche heavyweight (la tipologia più formale di specifica) la clausola è (Exception) true, mentre per il lightweight il default è \not_specified e per il

normal_behavior è (*Exception) false*. La semantica del valore di default indica che possono essere lanciate solo le eccezioni presenti nella clausola throws.
Infine, se si desidera che un metodo non lanci eccezioni basta specificare la seguente clausola signals: *signals (Exception) false;*

2.10 Side effect in JML

Una della più importanti restrizioni in JML consiste nella mancanza di side effect delle espressioni usate nelle asserzioni.
Infatti, le espressioni Java di assegnazione quali *"=" , "+="*, etc. e gli operatori di incremento *"++"* e decremento *"--"* non sono utilizzabili. Inoltre, all'interno delle asserzioni, proprio per garantire che non ci sia side effect nelle specifiche, possono essere usati solo metodi "puri".
Un metodo o un costruttore è definito puro se non produce nessun side effect sullo stato del programma. All'interno delle specifiche JML per definire un metodo come puro si può usare la parola chiave *"pure"* nella dichiarazione di un metodo.

La sintassi è dunque:

public /*@ pure @*/ tipo_ritorno metodo_get()

Un metodo puro è solitamente chiamato anche metodo "query", perché come le query restituisce un risultato senza effettuare delle modifiche al risultato o su altri dati.
La scelta di vietare il side effect, che ad una prima vista potrebbe sembrare troppo restrittiva, rappresenta invece un importante vincolo, perché evita che eventuali modifiche, fatte nelle asserzioni, ricadano in maniera dannosa sul codice, danneggiando in maniera più o meno grave l'architettura generale dell'applicazione che si sta realizzando.

2.11 Specifiche informali

In JML è possibile definire anche specifiche informali. Questa tecnica è usata principalmente nella fase iniziale della progettazione, quando vengono organizzati i metodi e si definiscono i comportamenti rispetto a pre- e post-condizioni. Una descrizione informale è del tipo:

(* testo che descrive la specifica *)

La specifica informale, che deve essere scritta tra parentesi tonde, può essere combinata con specifiche formali. La convenienza di usare questa forma si riscontra maggiormente nei casi in cui non è semplice definire in maniera formale la specifica di un metodo.
Tuttavia, pur essendo molto utili nel caso in cui si voglia definire una specifica informale, tali specifiche sono spesso ambigue e/o incomplete. Inoltre, i tool non possono manipolare questo tipo di specifiche. Ad esempio, a runtime un assertion checker non ha modo di verificare se la proprietà definita in maniera testuale sia rispettata o meno.

Dunque, solitamente si consiglia di usare questa forma di notazione nelle prime fasi di definizione delle specifiche, progettando per tempo la sostituzione della forma testuale con un'espressione formale maggiormente adeguata agli scopi, qualora fosse possibile.

Per concludere mostriamo un semplice esempio

```
public class Account {
/*@ requires (* x is positive *);
@ ensures \result >= 0 &&
@ (* \result è l'update del conto dopo il deposito*)
@*/
public static double deposita_sul_conto(double x) { ... }
```

la classe Account presenta il metodo deposita_sul_conto che prevede come precondizione che x sia un valore positivo, mentre come postcondizione assicura che il risultato sia non negativo e che corrisponde al valore del conto dopo il deposito.

2.12 Information Hiding in JML

Attraverso JML si può gestire la nozione di information hiding facendo leva sui livelli di privacy definibili in Java. Di base vengono mantenute le regole Java sulla visibilità, ma vengono aggiunte alcune regole extra.
Il livello di visibilità di un contesto, sia esso la definizione di un invariante, la specifica di un metodo o la dichiarazione di un campo, può essere: public, protected, private o default (che corrisponde alla visibilità del package).
In breve, la regola JML afferma che un contesto di un'annotazione non può riferirsi ad elementi che hanno un livello di visibilità inferiore.

I metodi ed i costruttori definiti public hanno conseguentemente anche le specifiche JML con visibilità pubblica. Lo stesso dicasi per quegli elementi definiti private o protected. Per tutti gli elementi per cui non è definita la visibilità, le specifiche sono visibili solo per i client all'interno dello stesso package, a meno dell'utilizzo della keyword JML "*public*", che rende un'asserzione JML pubblica.
Per garantire l'information hiding non è permesso l'utilizzo di variabili private all'interno di specifiche pubbliche. Per tal motivo se si ha la necessità di utilizzare variabili private all'interno delle asserzioni, bisogna definire tali variabili con la parola chiave: *"spec_public"* o "*spec_protected*" nella dichiarazione di tali variabili. In questa maniera pur essendo considerate private da Java, tali variabili hanno visibilità pubblica nelle specifiche JML.

La sintassi è:

private /*@ spec_public @*/ tipo variabile;

Spec_public non è semplicemente uno strumento per cambiare la visibilità ad un elemento delle specifiche, ma è utile soprattutto per la dichiarazione delle variabili modello, come vedremo nel prossimo paragrafo.

2.13 Variabili Modello

JML permette di gestire eventuali modifiche all'implementazione del codice, senza stravolgere le specifiche già definite per una classe o per un metodo. Tali modifiche, infatti, potrebbero influire sul codice dei clienti delle variabili o del metodo che ha subito le modifiche.
Per fare questo si può usare una "variabile modello", ossia una variabile definita ed usata solo all'interno delle specifiche e non disponibile nel codice Java, all'esterno delle annotazioni. Per descrivere meglio questa possibilità facciamo un semplice esempio.
Immaginiamo di avere la variabile "nome" che viene usata nelle specifiche di una classe, ad esempio nella postcondizione del costruttore e che al suo posto si decida di usare la nuova variabile "nomeCompleto".

Vogliamo quindi passare da

private /@ spec_public non_null @*/ String nome;*

a

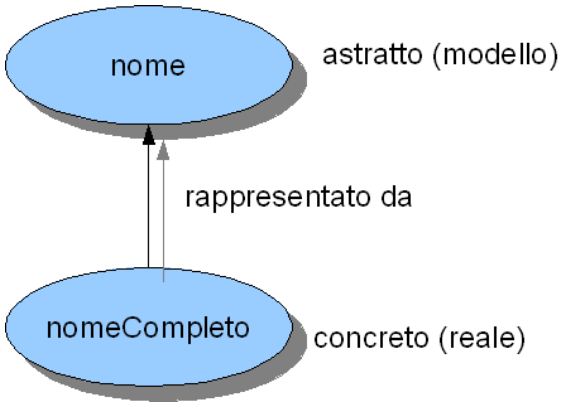
private /@ non_null @*/ String nomeCompleto;*

Il nostro obiettivo è quello di non modificare le specifiche già definite che fanno uso di "nome", modificando questa variabile con "nomeCompleto".
Possiamo risolvere il problema in questa maniera:

```
//@ public model non_null String nome;
private /*@ non_null @*/ String nomeCompleto;
//@ private represents nome <- nomeCompleto;
```

Il nuovo campo "nomeCompleto" è usato all'interno del codice, ma la sua esistenza è nascosta al cliente che può continuare ad usare la variabile "nome" all'interno delle sue specifiche.
La clausola "*represents*" definisce una funzione d'astrazione che mappa una rappresentazione concreta di un valore in una astratta. Dunque il valore della variabile "nome" è lo stesso di "nomeCompleto".

Quanto abbiamo visto per le variabili può essere definito non solo per variabili, ma anche per metodi, classi ed interfacce.



2.14 Specifiche Lightweight e Heavyweight

In JML non è necessario definire una specifica completa per ogni metodo o per ogni classe. E' possibile, infatti, usare lo stile di specifica chiamato "lightweight" con il quale l'utente specifica solamente ciò che maggiormente gli interessa o ritiene più importante da verificare.
All'opposto della tipologia "lightweight" , è possibile usare la "heavyweight" in cui ogni elemento deve essere presentare un'opportuna asserzione JML. In questo caso è permesso all'utente di omettere solo quelle parti per le quali i valori di default sono appropriati.
La sintassi per definire questa metodologia di specifica consiste nell'utilizzo delle keyword: "normal_behavior", "exceptional_behavior" e "behavior" (tutte usate nella metodologia heavyweight –la lightweight non presenta keyword).

Le differenze principali tra queste tre keyword sono:

- *normal_behavior*: comportamento di default, non vengono lanciate eccezioni. Se si parte da uno stato in cui è vera la precondizione, si deve arrivare in uno stato in cui ottiene il metodo ritorna.
- *behavior*: prevede la possibilità di lanciare eccezioni.
- *exceptional_behavior*: vengono sempre lanciate eccezioni, del tipo definite nelle specifiche. Non si giunge mai ad un normal post-state.

Capitolo 3: JMLC ed ESC/Java2

In questo capitolo vedremo due tool che lavorano con JML: **JMLC** ed **ESC/Java2**.
Sono stati scelti questi due tool, perché rappresentano bene due delle principali categorie dei tool attualmente disponibili che permettono di verificare se moduli Java, annotati con JML, siano conformi alle loro specifiche.

Tali due categorie sono:

- il runtime assertion checking (di cui fa parte JMLC)
- la static verification (di cui ne è un esempio ESC/Java2)

Queste due categorie rappresentano due forme complementari di controllo delle asserzioni, la cui origine può collocarsi addirittura nei primi lavori degli anni 1950 di Von Neuman e Turing.

Il runtime assertion checking riguarda il test delle specifiche durante l'esecuzione del programma; qualsiasi violazione viene catturata e riportata attraverso speciali errori che vengono visualizzati all'utente.
La static verification, invece, usa tecniche logiche per provare, prima del runtime, che nessuna violazione delle specifiche avrà luogo una volta in cui il codice sarà eseguito.
L'aggettivo statico enfatizza che la verifica avviene proprio attraverso un'analisi statica e non dinamica del codice.

3.1 JMLC

Il compiler JML (jmlc) permette di compilare codice Java annotato con asserzioni JML, così come il comando javac permette di compilare classi Java. La differenza tra i due meccanismi è che all'interno del bytecode creato con la compilazione, vengono aggiunte le annotazioni JML, utili anche dopo la compilazione.
Lo scopo principale è la verifica a tempo di esecuzione delle asserzioni. Tale verifica è trasparente al cliente dell'applicazione, a meno che non venga violata un'asserzione (sia essa una pre-condizione o una post-condizione di un metodo od ancora un invariante). Inoltre, a parte leggere differenze nelle performance spaziali e temporali, il comportamento del programma è mantenuto conforme a quello originario compilato semplicemente con javac. Ciò è garantito, tra l'altro, dal fatto che nelle asserzioni JML non è permesso il side-effect.
Questo tool, infatti, non tenta di modificare il codice, ma di assisterlo evitando che si presentino situazioni non desiderate o contrarie alle specifiche iniziali.

Per poter compilare il codice java di una classe basta usare il seguente comando:

jmlc *NomeClasse.java*

così come si farebbe per compilare in Java, usando javac al posto di jmlc.

Il risultato di questo comando è il file *NomeClasse.class*, che viene posto all'interno della stessa directory del file .java originario.

Altri comandi utili con jmlc sono i seguenti:

***jmlc –Q** *.java*

che permette di compilare contemporaneamente più file .java e

***jmlc –d** ../bin NomeClasse.java*

che permette di stabilire in che directory inserire i file .class.

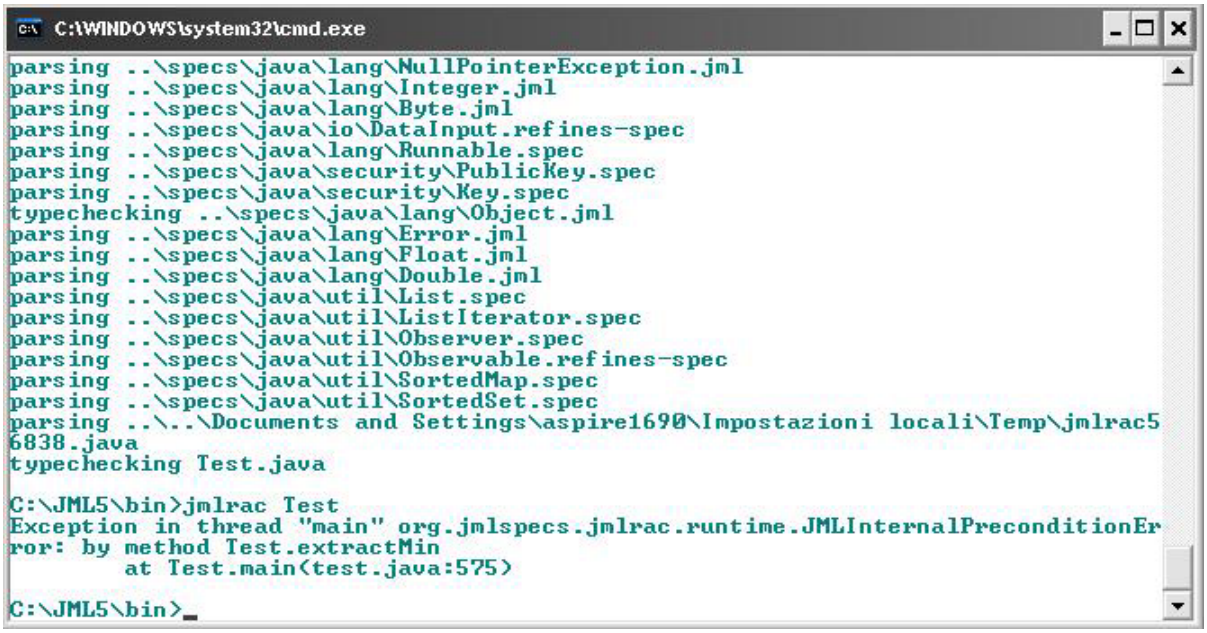
Per poter eseguire il codice compilato con jmlc, possiamo usare il classico comando di Java, includendo la libreria *jmlruntime.jar* che contiene le classi necessarie per essere interpretate dalla Java Virtual Machine.
Per poter effettuare questa operazione in maniera automatica può essere usato il comando jmlrac che ha la seguente sintassi:

***jmlrac** NomeClasse*

Ovviamente tale comando permette il runtime assertion checking solo delle classi che sono state compilate tramite jmlc e che presentano un main.

Il java runtime assertion checking (jmlrac) è uno strumento, realizzato da Gary Leavens, Yoonsik Cheon ed altri presso l'Iowa State University, che permette di verificare asserzioni JML a runtime: durante l'esecuzione, tutte le asserzioni sono testate e qualsiasi violazione produce un Errore. Una violazione in un'asserzione può consistere in un errore nel codice o un errore nelle specifiche.
Molte proprietà sono testate in diversi punti del codice, infatti, allo stato attuale, è uno dei migliori meccanismi di testing di proprietà che si hanno a disposizione.

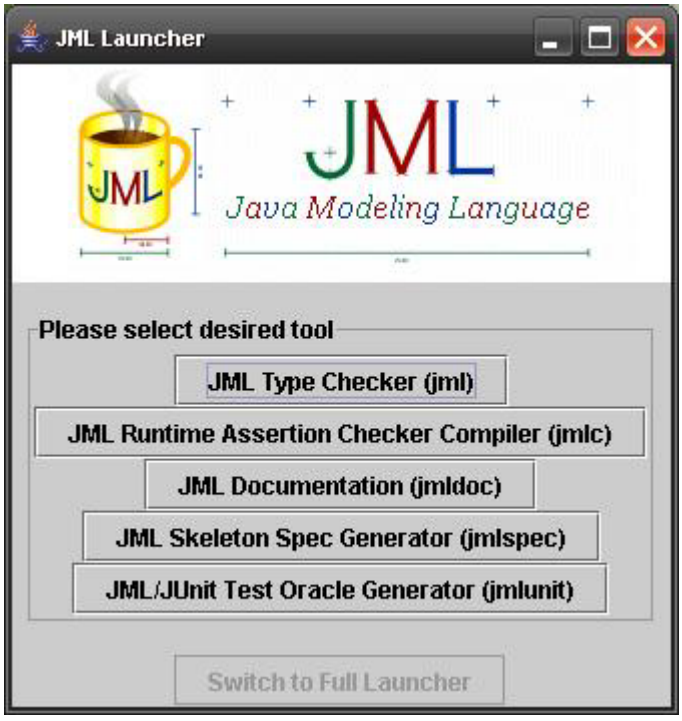
Ecco cosa è visibile all'utente del tool JMLC quando effettua prima la compilazione di una classe (nel nostro esempio Test.java) e poi prova a farne il controllo delle asserzioni a runtime.



Nell'esempio mostrato nella precedente immagine, la compilazione è terminata senza problemi; infatti il controllo su Test.java non ha dato esiti negativi. Eventuali problemi sarebbero stati riportati subito dopo l'espressione *"typechecking Test.java",* con la linea di codice dell'errore e la causa.
Il controllo dell'asserzione a runtime, invece, è fallito. Come possiamo vedere nella figura, viene riportata un'eccezione: una precondizione relativa ad un metodo è stata violata.

Alternativamente all'esecuzione del tool attraverso linea di comando, si può usare un'interfaccia grafica accessibile tramite il file *jml-release.jar*

Appena fatta partire l'interfaccia grafica, l'utente si trova di fronte questa schermata:



che permette di scegliere tra i seguenti elementi:

- Il type checker (jml), un tool utile per il check delle specifiche. Può essere usato come un veloce sostituto di jmlc se non c'è la necessità di compilare codice.

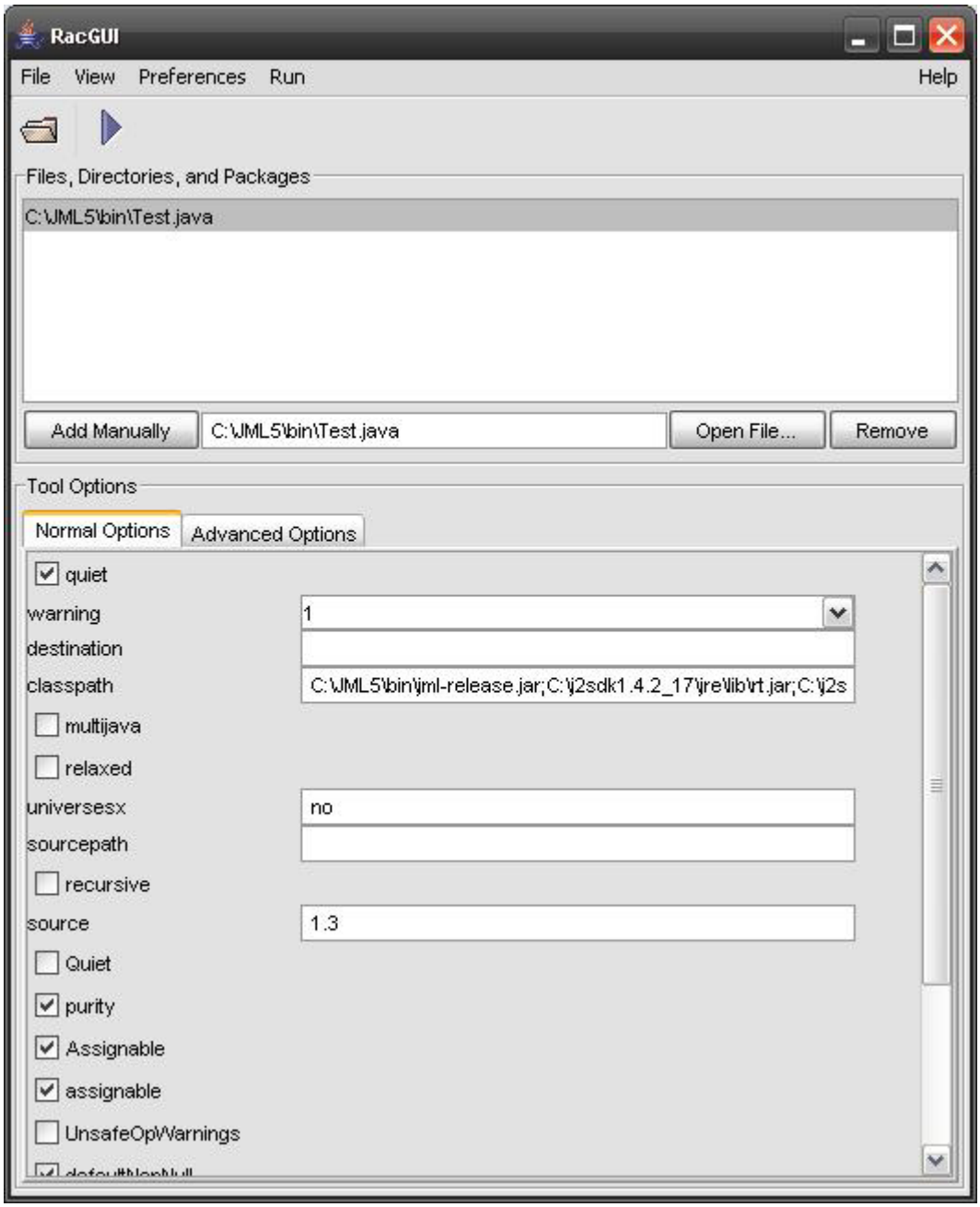
- Il compilatore JML (jmlc) è un'estensione del compilatore Java e permette di compilare file Java con annotazioni JML in bytecode Java. Tra le istruzioni previste c'è quella del runtime assertion checking che controlla specifiche JML come precondizioni, postcondizioni, invarianti.

- Il generatore di documentazione (jmldoc) produce pagine HTML contenente sia commenti Javadoc che specifiche JML

- Un generatore dello "scheletro" della classe arredato con specifiche (jmlspec)

- Il tool per il tesing (jmlunit)che combina JML con JUnit, un noto tool per il test in Java.Utilizzando jmlunit il programmatore è libero dallo scrivere codice che decida e verifichi il successo o il fallimento attraverso test. Il tool, infatti, permette, sfruttando le specifiche JML, processate da jmlc di decidere se il codice testato lavora in maniera appropriata.

Provando ad accedere ad uno qualsiasi dei precedenti tool si può visualizzare la seguente finestra:



La finestra permette di poter definire diverse opzioni, utili nell'esecuzione del tool.

Tra le più importanti segnaliamo:

- quiet: in questa maniera può essere usato il tool su più file contemporaneamente e di diverso tipo (.java , .jml, .refines-java)
- warning: permette di scegliere la profondità dei warning
- destination: directory dove salvare gli output
- classpath: permette di definire il CLASSPATH necessario per poter eseguire il tool. E' necessario, infatti, passare la directory in cui sono salvate le specifiche JML e le librerie Java.
- purity: controlla gli elementi definiti puri nelle specifiche JML
- assignable: controlla se metodi che presentano una clausola assignable non chiamino metodi senza questa clausola.

- defaultNonNull: definisce di default come non nulli i parametri formali, le variabili ed il ritorno dei metodi.
- verbose: permette di visualizzare informazioni aggiuntive durante l'esecuzione del tool.

3.1.1 Versione usata

Il JML compiler è stato sviluppato presso l'Università dell'IOWA ed è scaricabile dal sito http://www.eecs.ucf.edu/%7Eleavens/JML/index.shtml Le release di questo tool, che sono allo stato attuale nella fase di beta-test, funzionano su piattaforme che presentano come versioni Java la J2SDK 1.4.1 o 1.4.2.
Durante il lavoro per questa tesina ho potuto constatare che ci sono dei problemi di compatibilità tra le ultime release del tool (che è arrivato alla 5.6) e le varie versioni del JDK.
Una versione stabile, che è quella usata per questo lavoro, è la release 5.5 che funziona correttamente con il JDK 1.4.2_17 (l'ultimo rilascio della Sun della 1.4.2). Per tutte le versioni precedenti provate del JDK, sono stati trovati problemi durante l'esecuzione del tool.

Tra gli sviluppi futuri di JMLC c'è quello di garantire una completa compatibilità con le versioni superiori di Java a partire dalla 1.5.

3.2 ESC/Java2

Esc/Java2 è un tool di programmazione che cerca di trovare errori run-time in programmi Java attraverso un'analisi statica del programma sul quale è eseguito. Possiamo, dunque, definirlo come un checker statico che prova la correttezza di specifiche a tempo di compilazione, in maniera completamente automatica. Lo scopo principale del tool è quello di individuare possibili errori che vanno oltre i semplici errori riportati dalla verifica statica eseguita dai compilatori.
ESC/Java2 è un'estensione di ESC/Java, prodotto nato nei laboratori della Compaq Research e rispetto alla versione precedente è perfettamente compatibile con la versione 1.4 di Java ed è consistente con le versioni più recenti di JML.

Il funzionamento del tool può essere sintetizzato attraverso i seguenti passi:

- Partendo dal codice sorgente Java e dalle specifiche JML, vengono generati alcuni predicati da verificare.
- I predicati,insieme ad un modello logico del funzionamento di Java, vengono elaborati da un theorem prover, che verifica che le condizioni siano soddisfatte.
- Se le condizioni non sono soddisfatte viene segnalato un potenziale bug.

Nel primo passo, vengono fatti alcuni passaggi di semplificazione utili per arrivare alla generazione finale dei predicati che verranno passati al theorem prover. Il codice Java annotato con JML viene trasformato in una forma semplificata di un linguaggio definito da Dijkstra chiamato *Guarded Command* e da questa forma vengono poi generate l'insieme di condizioni da passare al prover. La trasformazione in Guarded Command può essere ottenuta (se lo si desidera) tramite linea di comando digitando *escj –pgc NomeClasse.java*
Il theorem prover che viene usato è *Simplify*, il quale accetta in input una sequenza di formule del primo ordine, di cui cerca di dimostrare la validità. Questo prover non è in grado in ogni situazione di dimostrare che una formula valida sia effettivamente valida; però, non afferma mai che una formula non valida sia valida. Per verificare la validità di una formula F, Simplify procede cercando di dimostrare che ¬F sia contraddittoria. Se ¬F è valida, allora il prover assume che F non lo sia e produce un controesempio, cioè un insieme di formule atomiche che rendono invalida la formula F. Purtroppo, a causa della sua verbosità, il controesempio che viene prodotto non sempre è di facile comprensione e quindi risulta di scarsa utilità.(per maggiori dettagli su Simplify vedere la sezione 3.2.1)
ESC/Java2 non è completo, ma trova diversi potenziali bug in maniera rapida. E' ottimo per provare l'assenza o la possibile presenza di eccezioni a runtime (ad esempio eccezioni di tipo Null-, ArrayOutOfBounds,ClassCast-) e per la verifica di proprietà relativamente semplici.
A differenza di JMLC, il tool esposto nel precedente paragrafo, che semplicemente **testa** la correttezza di una specifica, ESC/Java ne **prova** la correttezza.
Tuttavia, mentre con il runtime assertion checking, possiamo liberamente scegliere cosa specificare, con ESC/Java per poter accettare in maniera formale una specifica dobbiamo specificare tutte le proprietà su cui si basa la specifica.
Dunque ESC/Java è indipendente da qualsiasi suite di test sui programmi e provvede a dare un alto grado di sicurezza nella progettazione. Inoltre, per il controllo automatico delle asserzioni in maniera statica non c'è bisogno di alcun input dell'utente.

Le asserzioni in ESC/Java2 supportano il controllo modulare. Infatti, quando viene controllato il corpo di una routine r, sia esso un metodo o un costruttore, ESC/Java2 non esamina il corpo delle routine che usano r. Piuttosto, fa affidamento sulle specifiche di queste routine ed in particolare su come sono definite le asserzioni requires (ossia le pre-condizioni) ed altri costrutti sia JML che Java quali header dei metodi, tipi di ritorno e clausole throws. In sintesi, per controllare il corpo della routine r, ESC/Java2 non esamina i chiamanti di r, ma assume che r sia invocata solo in accordo alle specifiche.

In maniera dettagliata i passi che esegue ESC/Java2 sono i seguenti:

- fase di parsing (in cui viene controllata la sintassi)
- fase di typechecking (controllo sui tipi)
- fase di controllo statico (vengono cercati potenziali bugs). In questa fase viene usato, in maniera trasparente all'utente, il theorem prover chiamato Simplify.

Le prime due fasi producono "cautions" ed "errori".
La terza ed ultima fase produce "warnings".

ESC/Java2 restituisce warnings su potenziali errori a run-time. La versione corrente di questo tool (la b4) verifica solo il corpo di metodi e costruttori e non produce warnings per potenziali errori quali inizializzazioni statiche o inizializzazioni per campi statici. Inoltre le eccezioni lanciate dal sistema Java a runtime vengono trattate come errori a runtime. Infatti, alcuni delle potenziali condizioni d'errore riconosciute da ESC/Java2, verrebbero individuate dal sistema a tempo d'esecuzione e porterebbero di conseguenza il lancio di un'eccezione (ad esempio per NullPointerException, IndexOutOfBoundsException, ClassCastException, ArrayStoreException, ArithmeticException e NegativeArraySizeException).

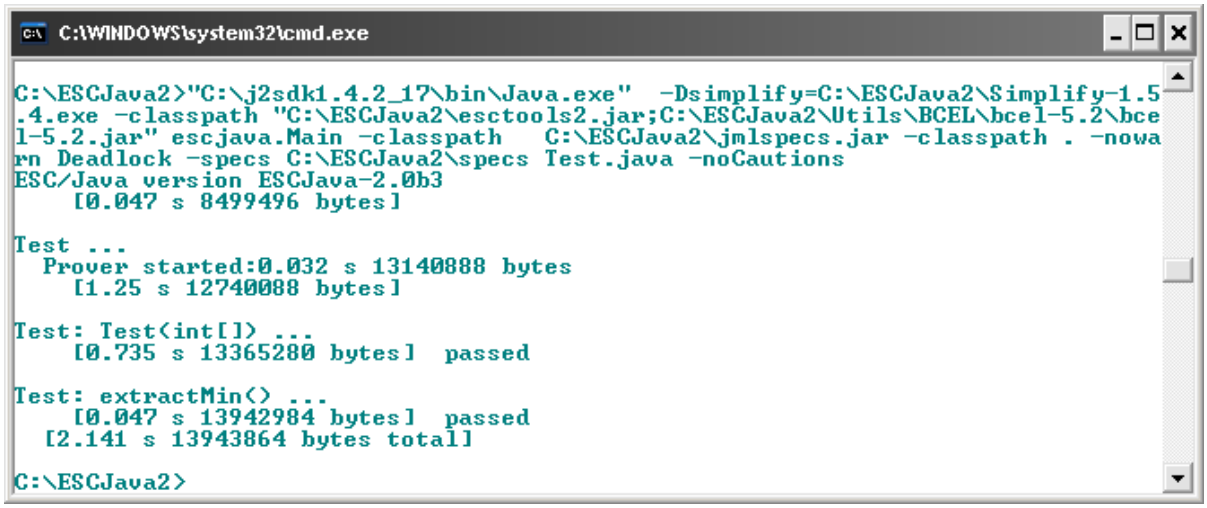
I messaggi d'errore, invece, vengono restituiti a causa di programmi mal formati. Infatti, prima dell'analisi relativa al rispetto delle specifiche da parte del codice del programma, devono essere effettuate le operazioni di parsing, risoluzione di nomi, controllo sui tipi, sia del codice Java, sia delle asserzioni JML. Quando un costrutto illegale viene scoperto in questa fase preliminare, viene lanciato un messaggio d'errore, facilmente riconoscibile perché presenta la dicitura *Error* invece di *Warning*. Ovviamente per poter passare alla successiva fase di controllo d'errore a runtime è necessario che tutti gli errori di questo tipo vengano risolti.
Volendo fare un parallelo con il compilatore Java possiamo affermare che i messaggi d'errore di ESC/Java2 sono simili ai messaggi d'errore di compilazione.

Per poter usare il tool ESC/Java2 basta digitare il seguente comando da prompt dei comandi :

escj *NomeClasse.java*

Come si può vedere nella figura che segue, in cui è stato provato il tool con una classe d'esempio Test.java, ESC/Java2 fa partire il prover e verifica metodo per metodo tutta la classe. Per ogni metodo restituisce il tempo necessario per la verifica e se questa ha avuto esito positivo.

L'esito positivo è indicato dalla dicitura "*passed*". La presenza di errori o warning, viene segnalata con la stampa a video del warning (o errore), la causa e la linea di codice dove è presente il problema ed inoltre la dicitura "*failed*" vicino al metodo che presenta il warning (o l'errore).



3.2.1 Il theorem prover Simplify

3.2.1.1 Introduzione a Simplify

Simplify è il theorem prover usato in ESC/Java.
L'obiettivo di ESC, come abbiamo visto, è quello di provare, a tempo di compilazione, l'assenza di errori a run-time. La metodologia di ESC consiste dapprima nel processare il codice sorgente con un "generatore di verifica delle condizioni", che produce formule del primo ordine e poi sottomettere queste condizioni da verificare al theorem prover.
L'input di Simplify è, dunque, una formula del primo ordine.

Quando si deve verificare la validità di una formula F, Simplify come molti theorem prover procede testando la soddisfacibilità della formula negata ¬F.
Per verificare se una formula è soddisfacibile, Simplify effettua una ricerca di tipo backtracking, guidata dalla struttura proposizionale della formula, cercando di trovare *un'assegnazione di soddisfacibilità*, ossia un'assegnazione di valori di verità che rendono la formula vera e che è consistente con la semantica della teoria sottostante. Il prover, infatti, si basa su degli algoritmi specifici per verificare la consistenza dell'assegnazione di soddisfacibilità.

Vediamo un esempio per chiarire quanto detto.

Vogliamo verificare la soddisfacibilità della formula F

$$x < y \;\Rightarrow\; (x - 1 < y \;\wedge\; x < y + 2)$$

passiamo alla sua negata ¬F, che possiamo scrivere così:

$$x < y \;\wedge\; (x - 1 \geq y \;\vee\; x \geq y + 2)$$

La formula ¬F è divisibile nelle seguenti formule atomiche:

$$x < y$$
$$x - 1 \geq y$$
$$x \geq y + 2$$

Ogni assegnazione di verità che deve soddisfare ¬F deve assegnare a x < y vero, quindi la ricerca in backtracking inizia postulando x < y. Il procedimento continua considerando le due possibilità $x - 1 \geq y$ e $x \geq y + 2$. Una delle due deve essere pari a true, affinché ¬F sia soddisfatta. La ricerca, dunque, procede nella seguente maniera:

si assume $x < y$.
consideriamo la clausola $x - 1 \geq y \vee x \geq y + 2$
primo caso,assumiamo $x - 1 \geq y$
si verifica l'inconsistenza di $x < y \;\wedge\; x - 1 \geq y$
facciamo backtracking dal primo caso (si elimina l'assunzione $x - 1 \geq y$*)*
secondo caso, assumiamo $x \geq y + 2$
si verifica l'inconsistenza di $x < y \;\wedge\; x \geq y + 2$
facciamo backtracking dal secondo caso
(poiché sono esauriti tutti i casi e non è stato trovata un'assegnazione di soddisfacibilità terminiamo)
viene restituito ¬F insoddisfacibile, dunque F è soddisfacibile.

Riassumendo, l'idea base della tecnica del backtracking si basa su un insieme di percorsi da esplorare, guidati dalla struttura proposizionale della formula, mentre il test per la consistenza di ogni percorso è fatto attraverso algoritmi specifici che riflettono la semantica dei predicati.

Prima di introdurre gli algoritmi usati da Simplify, descriviamo brevemente la teoria su cui si fonda il prover.
L'input di Simplify è una formula del primo ordine con simboli di funzione e d'uguaglianza.

Il linguaggio include i connettivi logici $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ ed i quantificatori universale $\forall$ ed esistenziale $\exists$.

I simboli di funzione hanno una semantica predefinita. Possiamo dividere tale semantica in alcune "teorie":

1. teoria dell'uguaglianza: definisce la semantica della relazione d'uguaglianza "=".
2. teoria dell'aritmetica: definisce i simboli di funzione "+", "-", "*" e i simboli di relazione: ">", "<","$\geq$" e "$\leq$ ". Questi simboli di funzione assumono il significato classico.
3. La teoria delle mappe: che fa uso delle funzioni "select" e "store" che sono usate, ad esempio, per rappresentare array o insiemi.

3.2.1.2 La strategia di ricerca di Simplify

Come già accennato nel paragrafo precedente, la strategia di ricerca di Simplify è essenzialmente sviluppata sulla struttura proposizionale delle formule passate in input al prover.
Il prover fa uso di una struttura dati globale che prende il nome di "***context***" e che è costituita dalla congiunzione delle formule e dalle assunzioni che vengono definite nel caso di ricerca corrente.
Alcune delle componenti fondamentali di context sono:

- Il valore booleano ***refuted***, che viene settato ogni qual volta si verifica che il contesto è inconsistente, che causa il backtracking dell'algoritmo di soddisfacibilità.
- L'insieme ***lits***, che è un insieme di formule atomiche chiamate ***letterali***
- L'insieme ***cls***, che è un insieme di clausole, ognuna delle quali è costituita dalla disgiunzione di un insieme di letterali

L'algoritmo di ricerca di Simplify, denominato ***Sat***, fa uso di tre interfacce:

- ***AssertLit(P)***: che aggiunge il letterale P all'insieme lits e che eventualmente setta refuted se P rende lits inconsistente. Inoltre il context permette che clausole in cls possano essere cancellate e letterali siano eliminati da clausole.
- ***Push()***: salva lo stato di context
- ***Pop()***: recupera il precedente stato di context

Per verificare la validità di una formula F, Simplify inizializza il context con ¬F ed usa la procedura ricorsiva Sat per verificare se il contesto è soddisfacibile, trovando un'assegnazione di soddisfacibilità. Se una tale assegnazione viene trovata Simplify la riporta all'utente come controprova di F. Altrimenti se Sat completa la ricerca senza trovare un'assegnazione per ¬F, Simplify riporta la validità della formula F.

L'output di Sat è un insieme di assegnazioni di verità che rendono soddisfatto il contesto, dove ogni assegnazione è costituita da un *monomio*, ossia la congiunzione di tutte le variabili rese true dall'assegnazione con la negazione di tutte le variabili rese false.

L'algoritmo Sat è implementato con la ricerca in backtracking che prova a creare un'estensione consistente dell'insieme lits includendo un letterale da ogni clausola in cls. Per ridurre l'esplosione combinatoria, Sat chiama la procedura Refine prima di ogni caso di split. La procedura, come possiamo vedere nell'algoritmo sottostante, si basa su un valore booleano globale che registra se il raffinamento è abilitato.

```
proc Refine() ≡
  while refinement enabled do
    disable refinement;
    for each clause C in cls do
      RefineClause(C);
      if refuted then
        return
      end
    end
  end
end

proc RefineClause(C : Clause) ≡
  if C contains a literal l such that [lits ⇒ l] then
    delete C from cls;
    return
  end;
  while C contains a literal l such that [lits ⇒ ¬l] do
    delete l from C
  end;
  if C is empty then

    refuted := true
  elseif C is a unit clause {l} then
    AssertLit(l);
    enable refinement
  end
end
```

La notazione $[P \Rightarrow Q]$ afferma che Q è una conseguenza logica di P.
$[lits \Rightarrow l]$ indica, perciò, che l è una conseguenza di lits e va intesa, quindi, come equivalente a $l \in lits$.
Refine, dunque, permette di ridurre clausole rimuovendo letterali inconsistenti con lits e inconsistenti con il contesto ed inoltre di rimuovere clausole che sono già soddisfatte da lits.
Definito Refine, presentiamo lo pseudocodice per l'algoritmo Sat che effettua il backtracking:

```
proc Sat() ≡
  enable refinement;
  Refine();
  if refuted then return
  elsif cls is empty then
    output the satisfying assignment lits;
    return
  else
    let c be some clause in cls, and l be some literal of c;
    Push();
    AssertLit(l);


     delete c from cls;
     Sat()
     Pop();
     delete l from c;
     Sat()
   end
 end
```

L'algoritmo Sat dapprima chiama la procedura Refine, se refuted diventa pari a true il contesto è insoddisfacibile e Sat correttamente ritorna senza restituire nessuna assegnazione di soddisfacibilità come output. Se l'insieme cls è vuoto e quindi sono state eliminate tutte le clausole, il contesto è equivalente all'insieme lits, che viene restituito in output. Se cls, invece, non è vuoto viene scelta una clausola da cls e un letterale in tale clausola e dopo aver salvato il contesto si effettua una chiamata ricorsiva a Sat(). Se si scopre l'inconsistenza di l si recupera il contesto e si effettua il backtracking eliminando l e richiamando Sat().

Capitolo 4 Esempi d'applicazione

In questo capitolo proponiamo alcuni esempi esplicativi delle potenzialità di JML e dei tool che ne fanno uso, nonché dei loro eventuali limiti.
Negli esempi saranno presentati i concetti più importanti che sono stati analizzati nel Capitolo_2.

4.1 Esempio: Precondizioni, Postcondizioni ed invarianti

In questo paragrafo studieremo un esempio di classe Java annotata con asserzioni JML in cui sono presenti i principali costrutti illustrati nei capitoli precedenti.
In particolar modo ci interesseremo di precondizioni, postcondizioni ed invarianti.
Per fare ciò introduciamo la classe Java Massimo, una classe utile per la ricerca del massimo in un vettore di interi positivi.
Il metodo preposto a tale ricerca è massimo(int[] vett) per il quale possiamo definire le seguenti precondizioni e postcondizioni.

Precondizione: ogni elemento dell'array è maggiore di 0
Postcondizione: il risultato è un valore presente nell'array ed è il più grande di quelli presenti.

La precondizione espressa in logica del primo ordine è la seguente:

$$\forall X \quad X \geq 0 \ \wedge \ X < vett.length \rightarrow vett[X] > 0.$$

la postcondizione:

$$\begin{array}{ll} (\exists X & X \geq 0 \ \wedge \ X < vett.length \ \wedge \ vett[X] = result) \ \wedge \\ (\forall X & X \geq 0 \ \wedge \ X < vett.length \rightarrow result \geq vett[X]) \end{array}$$

La traduzione in JML, come possiamo notare, è molto simile alla versione in FOL.

Precondizione:

```
requires vett!=null && (\forall int i; 0 <= i && i < vett.length; vett[i] > 0);
```

Postcondizione:

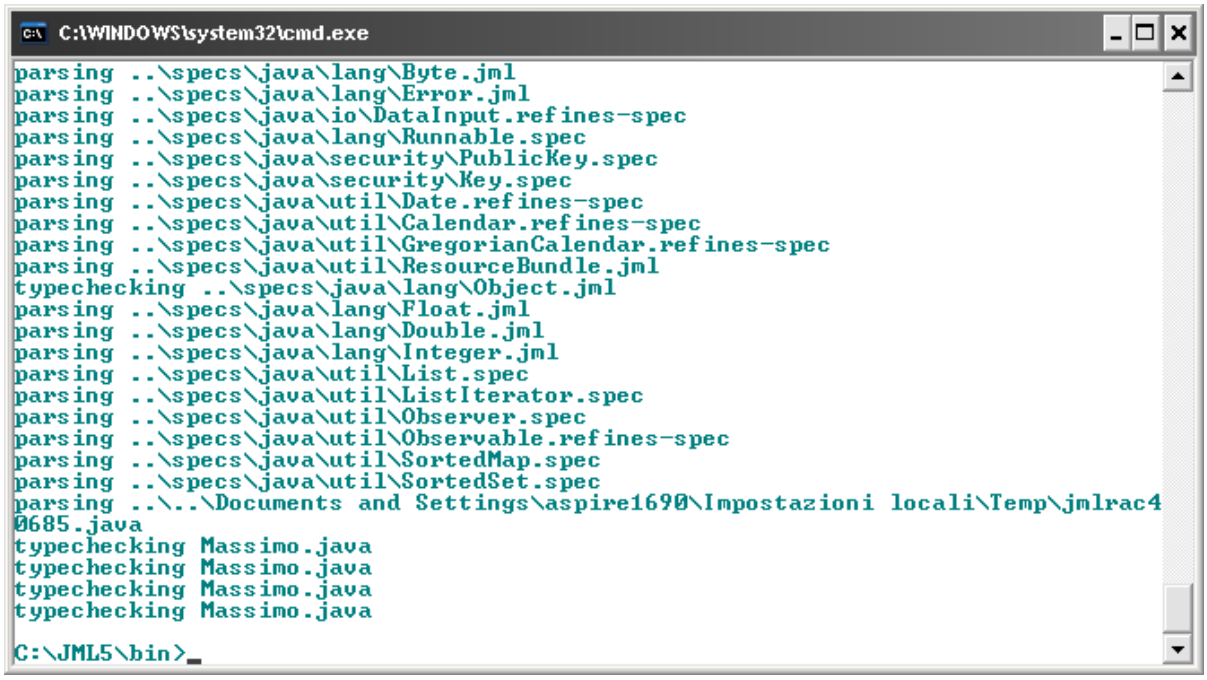
```
ensures  (\exists int i; 0 <= qi && i < vett.length; vett[i] == \result)  &&
                (\forall int j; 0 <= j && j < vett.length; \result >= vett[j]) ;
```

La variabile MIN dichiarata come final nella classe Massimo, costituirà il nostro invariante.

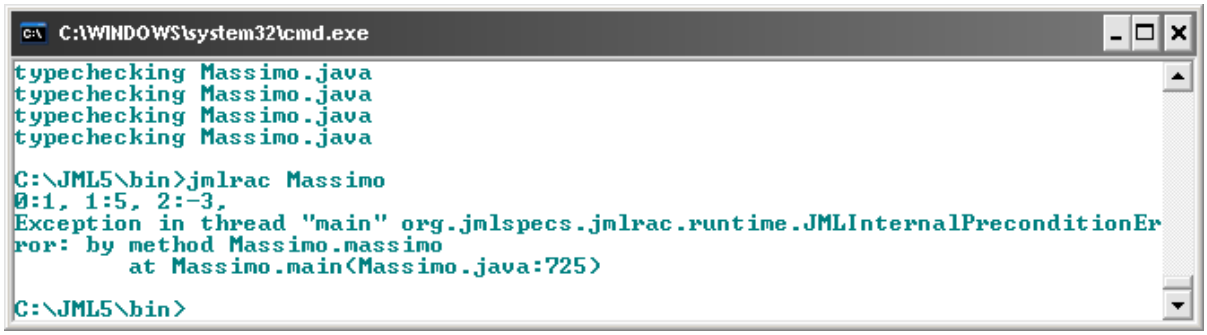
La classe Massimo corredata con le annotazioni JML è, dunque, la seguente:

```
public class Massimo {
        //@ invariant MIN == 0;
        public final static int MIN = 0;
        /*@ public behavior
         requires vett!=null;

         @*/
        public static void stampaVettore (int[] vett) {
                for (int i = 0; i < vett.length; i++)
                System.out.print(i + ":" + vett[i] + ", ");
                System.out.println();
        }
        /*@ public behavior
         requires vett!=null && (\forall int i; 0 <= i && i < vett.length; vett[i] > 0);
         ensures  (\exists int i; 0 <= i && i < vett.length; vett[i] == \result)  &&
                        (\forall int j; 0 <= j && j < vett.length; \result >= vett[j]) ;

        @*/
        public static int massimo(int[] vett) {

                int result = MIN;
                int i = 0;
                        while(i < vett.length) {
                                if (vett[i] > result)
                                result = vett[i];
                                i++;
                        }
                return result;
        }

        public static void main(String[] args) {
                int[] a = { 1, 5, 3 };
                stampaVettore(a);
                System.out.println("massimo: " + massimo(a) +
                "\n****");
                }
        }
```

Utilizziamo ora JMLC per verificare se ci sono violazioni delle specifiche nella classe. Come possiamo vedere dalle figure che seguono, la compilazione (eseguita con il comando *jmlc*) non ha riportato errori, così come il runtime assertion checking. Il metodo main, infatti, provato sull'array d'esempio costituito dai valori 1, 5 e 3 ha restituito in maniera corretta il valore massimo pari a 5.

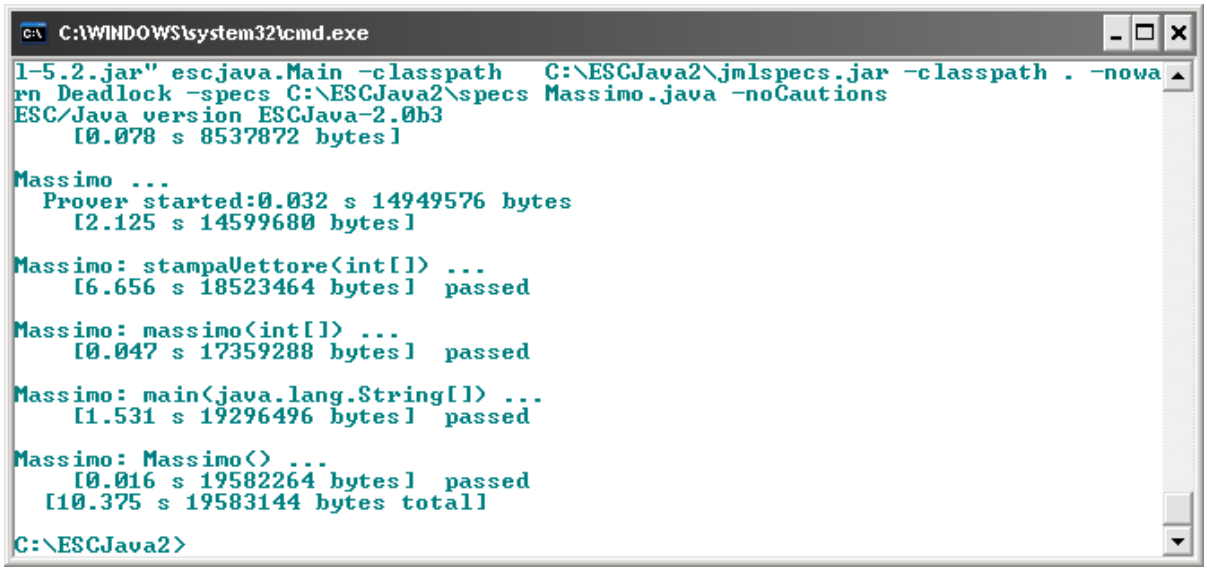


Modifichiamo ora l'array passato come parametro al metodo massimo() all'interno del main. Il nuovo array contiene un valore negativo pari a –3.
Dopo aver compilato, il runtime assertion checker lancia, in maniera corretta ed opportuna, un'eccezione specificando che questa è stata causata da una violazione della Precondizione (*JMLInternalPreconditionError*).



Proviamo ora ad effettuare una static verification tramite ESC/Java2, sulla classe Massimo con l'array con tutti i valori maggiori di 0. Ricordiamo che non bisogna compilare il codice, perché il controllo viene fatto in maniera statica direttamente sulla classe Java.

Come possiamo notare dalla figura che segue, tutti i metodi della classe Massimo rispettano le specifiche e non viene lanciato nessun Warning.
Il tool impiega circa 10 secondi per completare la verifica.



4.2 Esempio: Assert

In questo paragrafo vedremo un esempio d'utilizzo della clausola "assert".
Come detto nel paragrafo 2.6, tramite questa clausola possiamo definire una proprietà che deve essere vera nel punto del codice dove si trova l'asserzione.
Il suo utilizzo è necessario, soprattutto per prevenire possibili errori nella realizzazione del codice.
L'esempio che stiamo per vedere, infatti, dimostra come un codice che in prima apparenza possa sembrare corretto, nasconde, invece, degli errori che lo rendono non conforme alle specifiche iniziali.

Ipotizziamo di avere, dunque, un metodo che prenda in ingresso due valori interi ed effettui alcuni controlli sul range di tali valori.

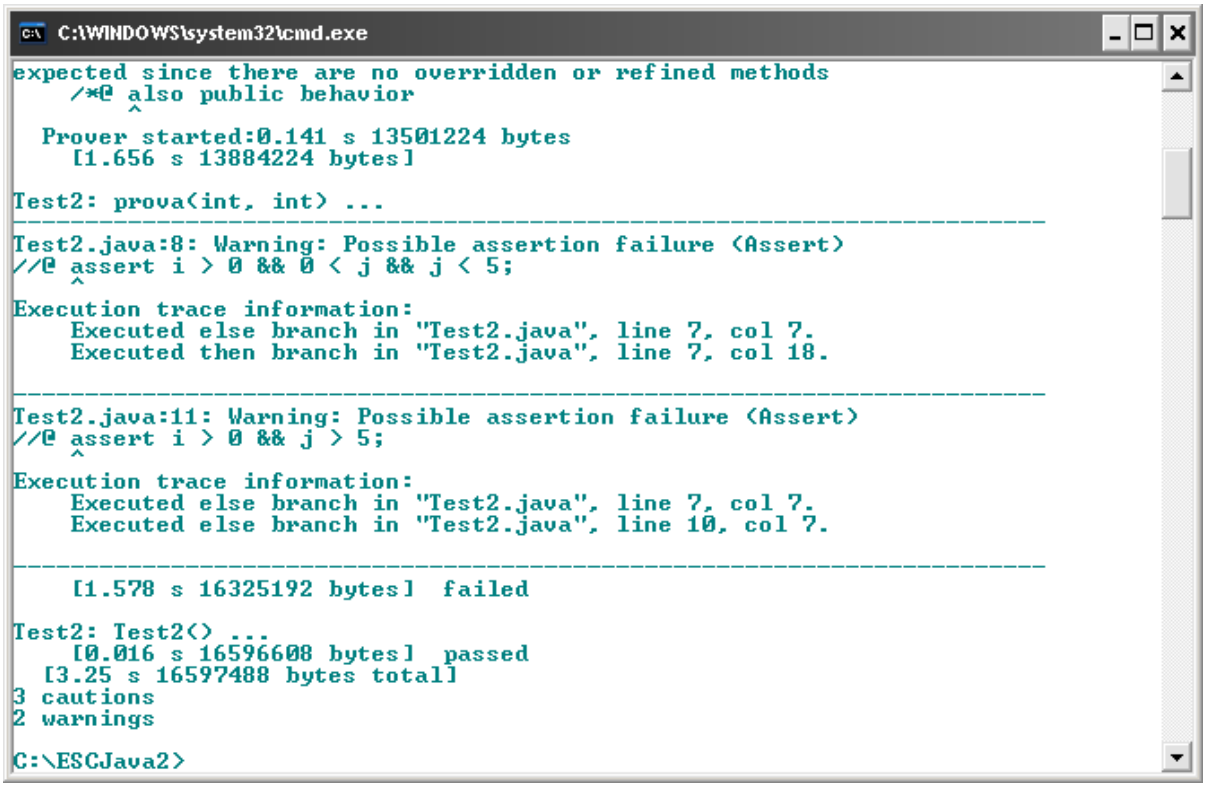
Tale metodo è il seguente:

```
1   int prova(int i, int j) {if (i <= 0 || j < 0) {
2     System.out.println("i <= 0 || j < 0");
3           } else if (j < 5) {
4                   //@ assert i > 0 && 0 < j && j < 5;
5                   System.out.println("j < 5");
6                   } else {
7                           //@ assert i > 0 && j > 5;
8                           System.out.println("j>5");
9                             }}
```

in grassetto sono riportati gli statement JML relativi agli assert.

Nella realizzazione del metodo prova() c'è un evidente errore. Le asserzioni non coincidono con quanto espresso dal codice. Alla riga 5, secondo l'asserzione, ci si arriva se il valore della variabile è strettamente maggiore di i ed il valore di j è compreso tra 0 e 5 estremi non inclusi. In realtà, si può arrivare alla riga 5 con un valore di j pari esattamente a 0, il che non è conforme alle specifiche. Per motivi simili anche l'asserzione alla riga 7 produce problemi. Infatti si può entrare nell'else finale anche se la variabile j è esattamente pari a 5. Il che non è ammissibile secondo l'assert.

Se proviamo ad eseguire ESC/Java2 sul codice appena proposto il risultato è il seguente:



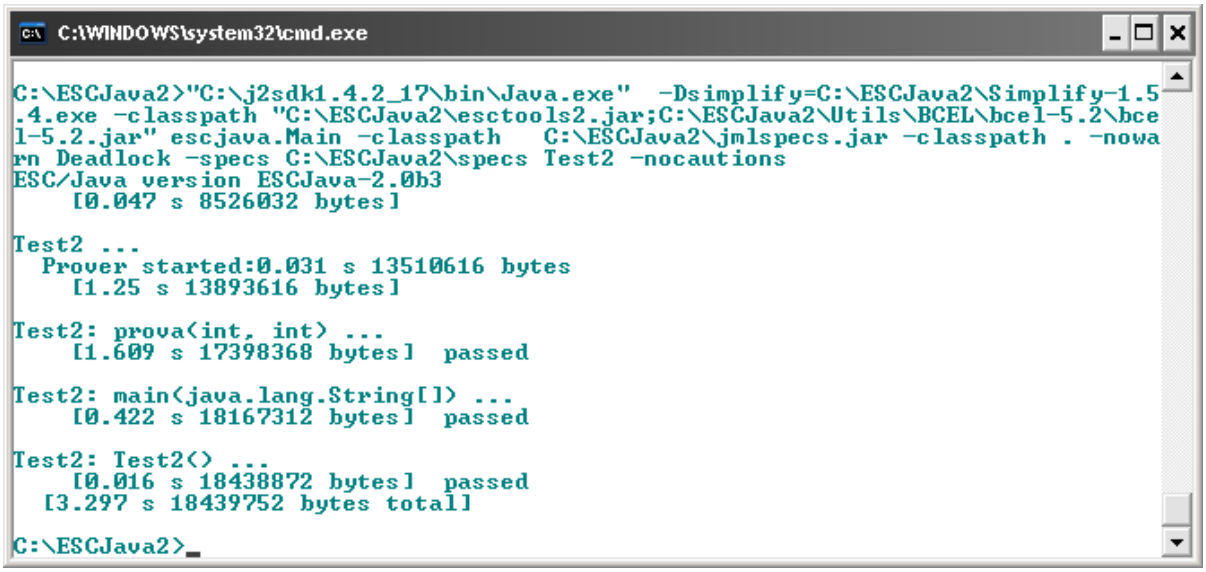
Come possiamo notare lo static checking è riuscito subito a scoprire il sottile problema, restituendo due warning relativi ad un possibile fallimento delle asserzioni, indicando le linee di codice che procurano il problema..
Ritenendo corretto quanto espresso attraverso le asserzioni, dato che rappresentano le specifiche del nostro problema, correggiamo il codice considerando i casi in cui la variabile j sia pari a 0 ed i casi in cui sia pari a 5.

Il codice corretto diventa dunque il seguente:

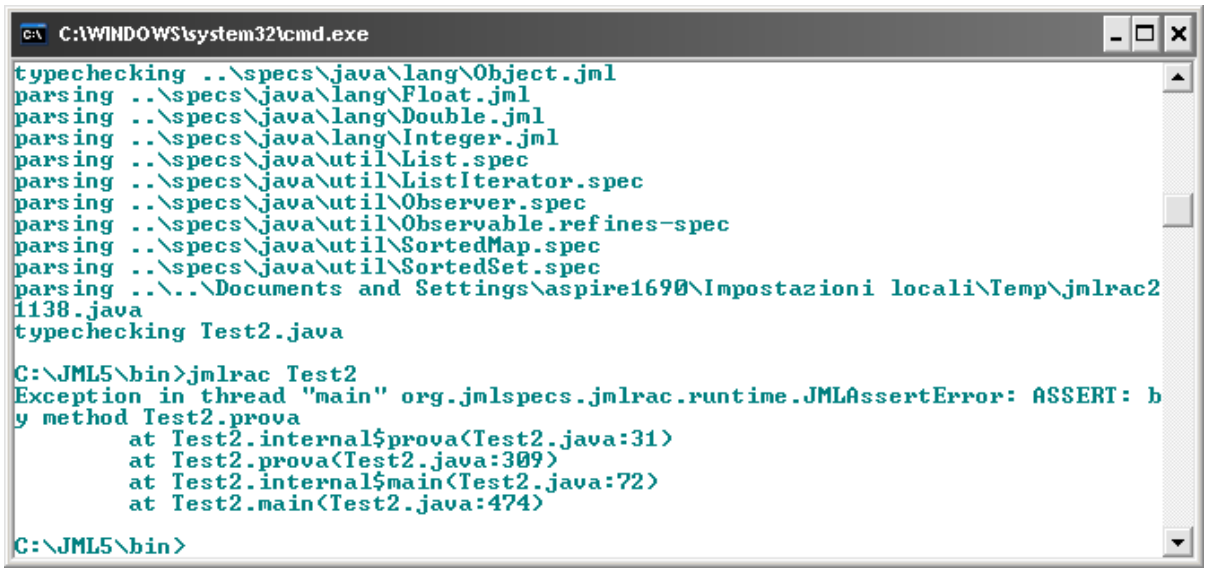
```
1     void prova(int i, int j) {if (i <= 0 || j <= 0) {
2       System.out.println("i <= 0 || j < 0");
3               } else if (j < 5) {
4                       //@ assert i > 0 && 0 < j && j < 5;
5                       System.out.println("j < 5");
6                               } else if(j>5){
7                                       //@ assert i > 0 && j > 5;
8                                       System.out.println("j>5");
9                                        }}
```

dove abbiamo provveduto a modificare la riga 5 inserendo il caso j<=0 e la riga 10 in cui abbiamo inserito il controllo relativo alla variabile j.

Il codice opportunamente modificato passa il test di ESC/Java2 come si può vedere nella figura che segue.



Come abbiamo dimostrato ESC/Java2 è uno strumento che in maniera molto rapida ci permette di verificare la correttezza delle specifiche.
Qualora avessimo usato un runtime checker come jmlc avremmo avuto bisogno di una approfondita suite di test per trovare il problema del nostro codice.
Infatti il codice non presenta errori di sintassi e tramite jmlc e jmlrac può essere facilmente compilato ed eseguito su istanze che non violino le asserzioni (cioè quando j=0 oppure j=5).
Nel caso in cui fosse presente un metodo main in cui viene lanciato il metodo prova con parametri di ingresso per i e j pari rispettivamente a 3 e 0 otterremmo il seguente risultato:



Come possiamo vedere la compilazione ha avuto esito positivo, mentre durante l'esecuzione del codice viene lanciata un'eccezione relativa alla violazione dell'asserzione.

4.3 Esempio: problemi con ESC/Java2 – bugs non trovati

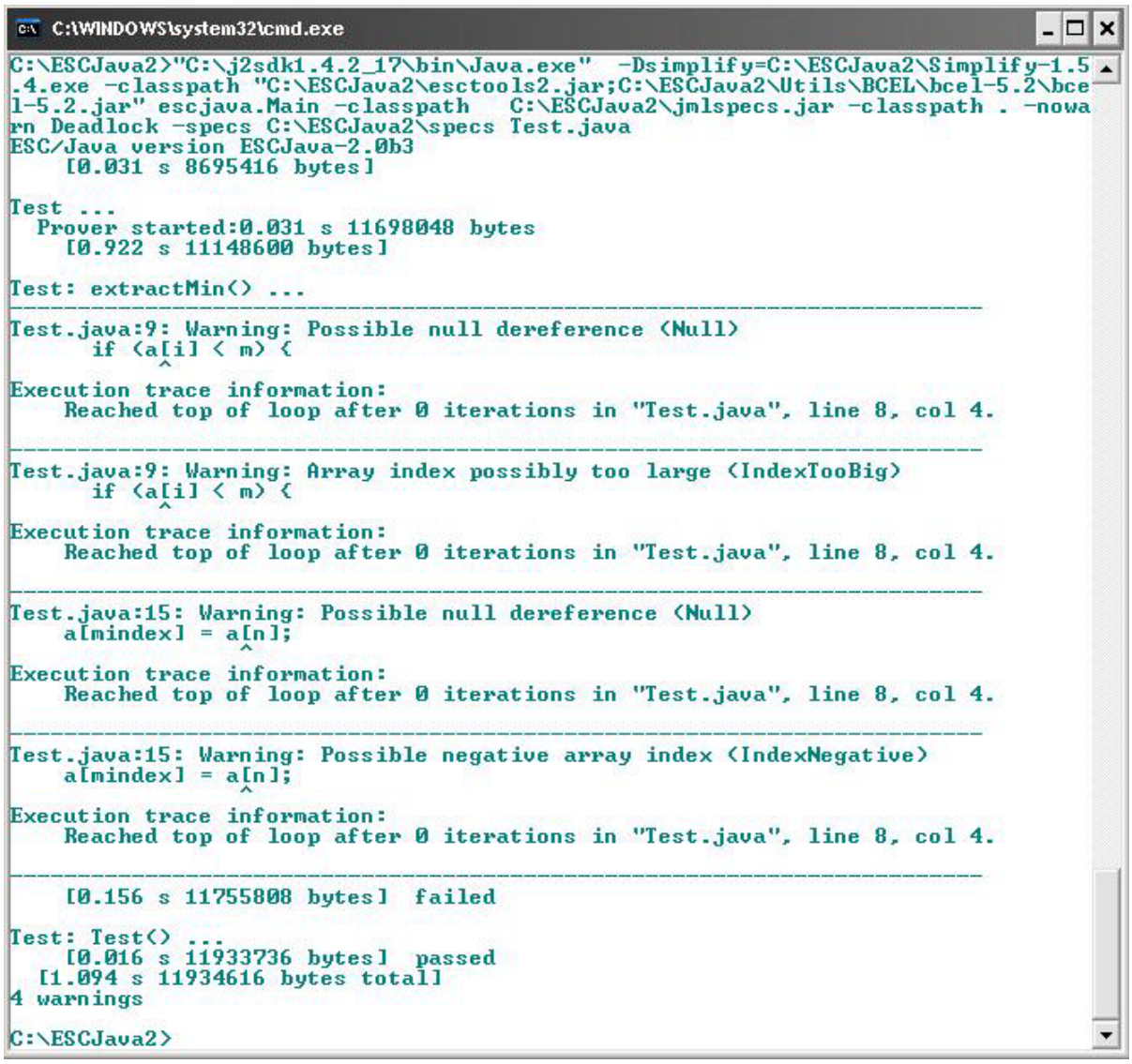
Proviamo ora un particolare caso di test con ESC/Java2.
Partiamo da una classe senza annotazioni JML, che costruiremo passo passo grazie all'aiuto del nostro tool.
L'obiettivo di questo esempio è di verificare se tutti i possibili bug vengono scovati da ESC/Java2.

Senza trascrivere completamente la classe(tralasciamo la definizione del costruttore, che può essere vista nei file allegati) la classe Test è la seguente:

```
1 public class Test {
2  int[] a;
3   int n;
4
5 public  int extractMin() {
6   int m = Integer.MAX_VALUE;
7   int mindex = 0;
8   for (int i = 0; i < n; i++) {
9    if (a[i] < m) {
10      mindex = i;
11    m = a[i];
12    }
13   }
14   n--;
15   a[mindex] = a[n];
16   return m;
17  }
18 }
```

Questa classe presenta una variabile intera ‘n’ che serve per rappresentare il numero di elementi dell’array ‘a’, un array di interi.
Il metodo extractMin() itera sui primi ‘n’ elementi dell’array, tenendo traccia del valore più piccolo presente nell’array. Dopo aver terminato il ciclo, copia l’elemento più piccolo nella posizione dell’array con indice più alto (indicato dalla variabile ‘n’), diminuisce di una unità il valore di ‘n’ e restituisce il valore minimo trovato.

Proviamo ad utilizzare ESC/Java2 con questa classe. (ricordiamo che per controllare le nostra definizione della classe dobbiamo digitare sulla linea di comando *escj Test.java*)
Il risultato è il seguente:



Come possiamo vedere vengono lanciati 4 warnings.
Studiamoli in dettaglio.

Il primo warning è relativo alla riga 9.
Warning: Possible null dereference (Null) e fa riferimento all’istruzione if(a[i] < m)

Possiamo risolvere questo warning, inserendo nella seconda riga la seguente porzione di codice:

```
/*@ non_null */ int[] a;
```

in questa maniera specifichiamo che l'array di interi 'a' non deve mai essere null.

Il secondo warning è relativo sempre alla riga 9, ma tratta un'altra potenziale fonte d'errore. "Array index possibly too large".
L'indice 'i' dell'array 'a' all'interno del ciclo for potrebbe assumere dei valori (fino al valore pari ad 'n') maggiori rispetto a quelli consentiti dalla grandezza dell'array stesso. Dobbiamo dunque definire dei valori per la variabile 'n'. Per tal motivo correggiamo la riga 3 con il seguente codice:

```
  int n;
//@ invariant 0 <= n && n <= a.length;
```

Abbiamo, dunque, definito un'invariante per la variabile 'n'. Questa potrà assumere solo i valori compresi tra 0 e la lunghezza dell'array 'a'.

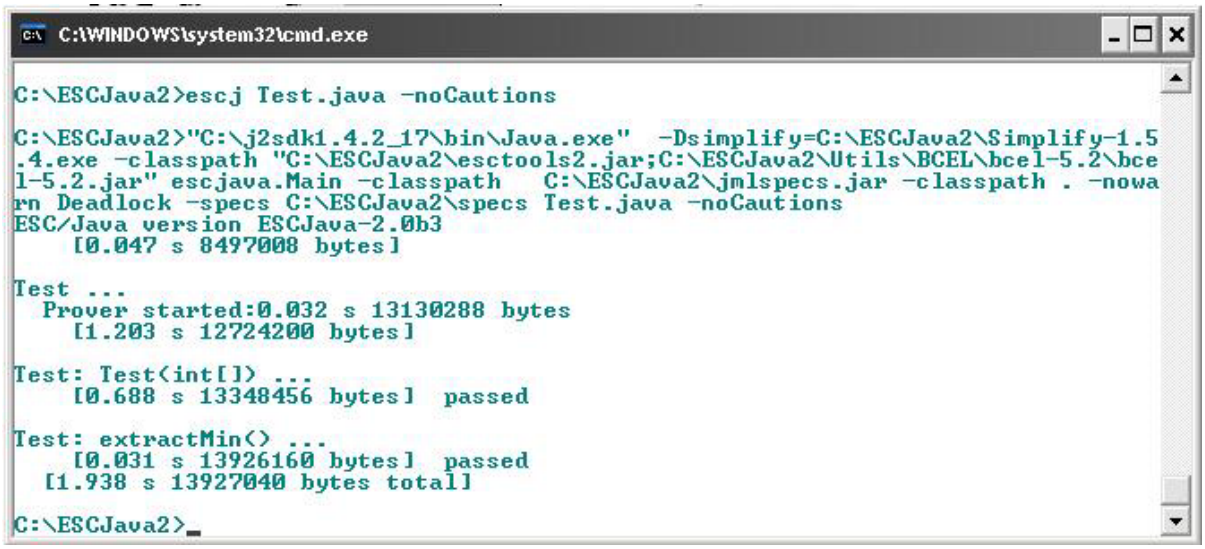
Il penultimo warning riguarda la linea di codice 15 e si riferisce ad un possibile riferimento ad una variabile nulla (Possible null dereference (Null)). Questo problema è stato già risolto trovando una soluzione al primo warning.

L'ultimo warning fa riferimento ad un possibile valore negativo per l'indice del'array 'a' nella riga 15 (Possibile negative array index (Index Negative).
La soluzione a questo warning consiste nel definire una precondizione che assicuri che il valore della variabile 'n' sia sia maggiore di 0.

Ecco, dunque, che introduciamo la seguente porzione di codice:

```
//@ requires n >= 1;
    int extractMin() {…}
```

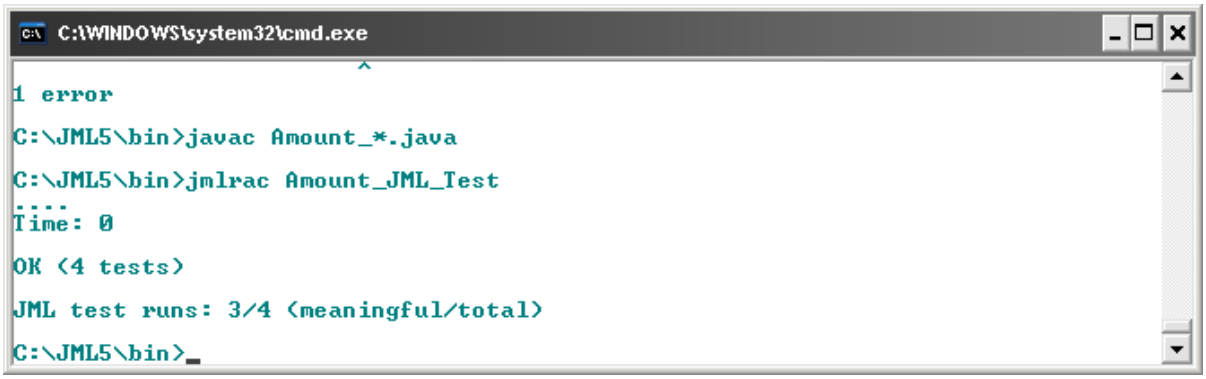
Provando ora con ESC/Java2 il codice opportunamente modificato otteniamo il seguente risultato, che conferma che tutte le possibili situazioni di warning sono state risolte.



Tuttavia, però, **non sono considerati possibili warnings sul numero delle chiamate di extractMin()** che potrebbero non assicurare la pre-condizione.
Se ad esempio un client della classe Test chiama il metodo extractMin() un numero di volte superiore alla dimensione dell'array, si presenta una violazione della pre-condizione relativa al valore della variabile "n", perché all'interno del metodo la variabile globale "n" viene decrementata ogni volta che si usa il metodo extractMin() con il rischio che questa diventi inferiore ad 1, contrariamente a quanto richiesto dalle specifiche.

ESC/Java si accorge del problema solo se all'interno di un proprio metodo main vengono effettate un numero di chiamate di extractMin() che portano alla violazione della precondizione.

Utilizzando il tool jmlunit per JMLC possiamo verificare che il problema esposto sopra non viene scoperto, ma vengono effettuati un numero di test non sufficienti per scovare il problema.



Tuttavia, come abbiamo visto per ESC/Java, se modifichiamo il codice con un main ad hoc JMLC scova il problema.
Predisponendo, infatti, un metodo main in cui inizializziamo un array con quattro elementi e provando ad eseguire il metodo extractMin() un numero di volte superiore a quattro, il tool verifica che effettivamente c'è stata una violazione della pre-condizione, come si può verificare nella seguente immagine.

4.4 Considerazioni su JMLC ed ESC/Java2 alla luce degli esempi proposti

Alla luce degli esempi proposti deduciamo che nonostante le ottime opportunità nella verifica di specifiche formali offerte da JMLC ed ESC/Java2, sono presenti dei limiti

In particolare, JMLC, nonostante sia un meccanismo molto rapido di verifica, richiede un pool di test ampio per poter verificare che il programma realizzato sia esente da violazioni della specifica. Per tal motivo è opportuno accompagnare il suo utilizzo insieme ad altri strumenti come, per esempio, ESC/Java2.

I Limiti di ESC/Java2, invece, risiedono nella sua non accuratezza e non completezza. I rischi che ne seguono possono, perciò, essere il segnalare una specifica corretta o il fallimento nella ricerca di warning in specifiche non corrette.
Come già detto in precedenza, infatti, ESC/Java segnala molti bug, ma potrebbe non segnalarli tutti.
Tipi d'errore che non vengono segnalati sono, ad esempio, gli overflow aritmetici ed i cicli infiniti, che non vengono verificati.
Ovviamente questo rappresenta una grossa limitazione nell'utilizzo di questo tool.

Allo stato attuale, infatti rispetto alla propria evoluzione come prodotto, ESC/Java2 è un'alfa release.

4.5 Un intero progetto documentato con JML: Fashion District

In questo paragrafo vedremo come JML può essere usato come valido strumento nella definizione di specifiche formali durante la realizzazione di un intero progetto.
A tale scopo è stato preso in esame il progetto "Fashion District", progetto proposto per il corso di Progettazione del Software del prof. Mancini (http://www.dis.uniroma1.it/~tmancini/index.php?currItem=teaching.prosw.materiale.progetti)

4.5.1 Fashion District: Requisiti

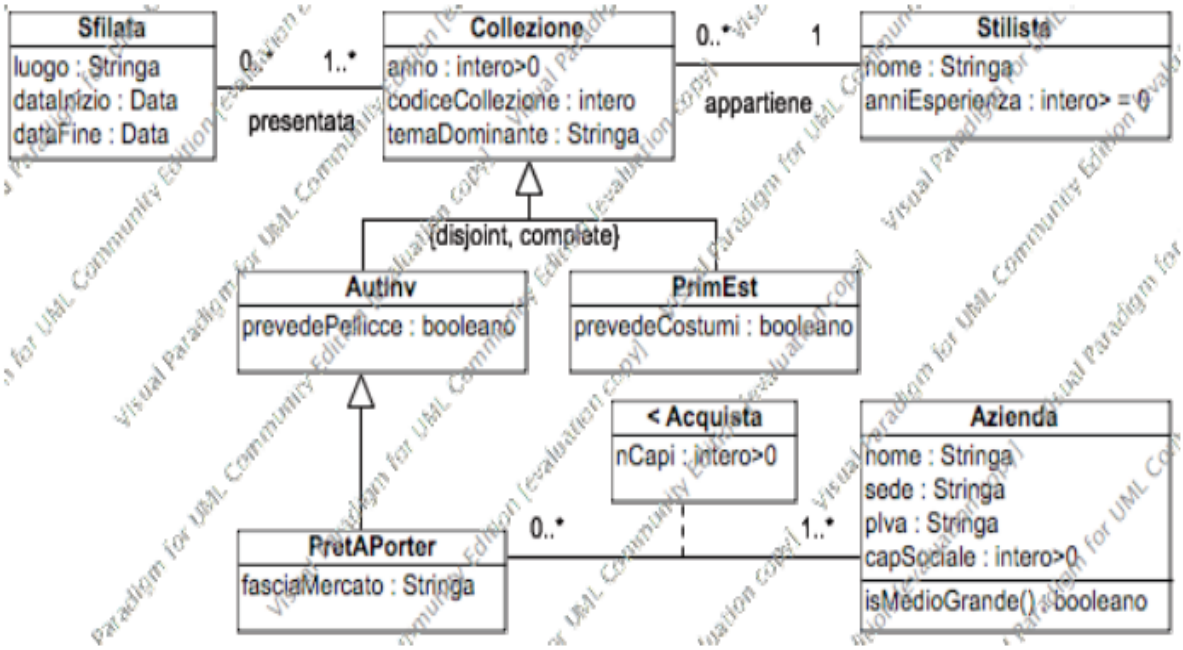
L'agenzia ha interesse a mantenere nel sistema informazioni circa le collezioni di capi d'abbigliamento prodotte, che sono caratterizzate da un identificativo (un intero), dagli stilisti che le realizzano, dall'anno e dal tema dominante (una stringa). Degli stilisti interessa il nome d'arte ed il numero di anni di esperienza (un attributo). Per semplicità, si può assumere che una collezione appartenga ad un solo stilista (cioè, se gli stilisti sono più di uno, sono comunque rappresentati da un unico nome d'arte, ad es. Dolce & Gabbana). Le collezioni possono essere presentate in sfilate, oppure essere pubblicizzate mediante altri canali (ad esempio riviste specializzate, video, portale Web dell'agenzia, etc.). Di una sfilata interessa il luogo in cui si tiene, la data di inizio e quella di fine.
Le collezioni possono essere di due tipi: primavera-estate e autunno-inverno. Delle collezioni primavera-estate si `e interessati a conoscere se prevedono o meno costumi da bagno, mentre delle collezioni autunno-inverno se prevedono o meno pellicce.
L'applicazione deve consentire di effettuare indagini di mercato. Dato che il segmento di mercato più rappresentativo `e costituito dalle collezioni pret-a-porter autunno-inverno, si vogliono rappresentare esplicitamente tali collezioni, caratterizzandole in base alla fascia di mercato (una stringa che rappresenta se la fascia di mercato `e costituita da teenager, da tempo libero, etc.). Queste collezioni vengono acquistate da una o più aziende (almeno una), di cui interessa nome, partita IVA e capitale sociale. `E inoltre di interesse conoscere il numero di capi di una collezione acquistati da ciascuna azienda.
L'applicazione deve fornire le seguenti funzionalità agli operatori dell'agenzia:

- Data una sfilata `s`, verificare se `e una sfilata di esordienti, ovvero se la media degli anni di esperienza degli stilisti che hanno una collezione che sfila in `s`, `e minore di 5. Si assuma per semplicità che ogni stilista ha al più una collezione che sfila in `s`.

- Data una collezione `c`, se `c` `e pret-a-porter autunno-inverno si vuole restituire il numero totale di capi acquistati da aziende medio-grandi, cioè con capitale sociale maggiore di 10 milioni di euro, altrimenti (`c`è una collezione invernale non pret-a-porter oppure una collezione primavera-estate) il valore restituito deve essere 0.

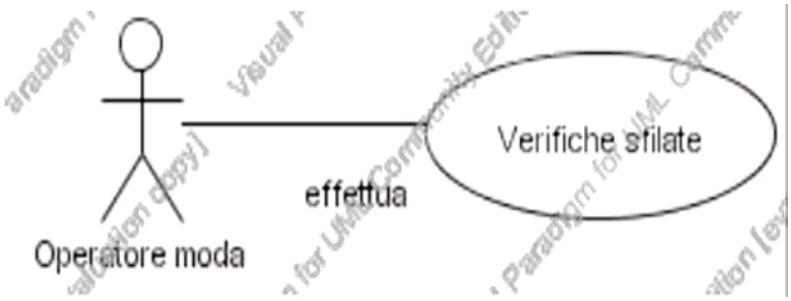
4.5.2 Fashion District: Analisi

L'analisi del progetto inizia con la definizione di due diagrammi fondamentali dello schema concettuale: il diagramma delle classi ed il diagramma degli use case.



Attraverso il diagramma delle classi possiamo rappresentare le classi del sistema software e le relative proprietà.
Ogni classe è rappresentata con le associazioni a cui partecipa e relative molteplicità.
In particolare notiamo la classe Collezione che viene definita come superclasse della generalizzazione disgiunta e completa delle sottoclassi AutInv e primEst che modellano rispettivamente i concetti di collezione autunno-inverno e collezione primavera-estate.
Nel diagramma viene rappresentata, inoltre, anche la classe PretAPorter che è in relazione is-a con la classe AutInv e che presenta l'associazione "Acquista" che modella la relazione con la classe Azienda relativa al numero di capi della collezione acquistati da una o più aziende. La classe Azienda presenta l'operazione isMedioGrande() che certifica se l'azienda ha un capitale sociale maggiore di 10 milioni. Infine nel diagramma sono presenti le classi Sfilata e Stilista che modellano gli omonimi concetti con le relative proprietà e le relazioni che presentano con la classe Collezione.

Definito il diagramma delle classi, passiamo a definire un altro importante elemento della fase di analisi: il diagramma degli use case. Attraverso tale diagramma, infatti, possiamo rappresentare l'interazione tra l'utente ed il sistema software.



In questo particolare caso vogliamo modellare la possibilità che hanno gli Operatori dell'agenzia di moda di effettuare verifiche sulle sfilate. Il diagramma nella figura della pagina precedente, descrive esattamente questo comportamento.

Definiamo ora le specifiche degli use case e delle classi. La prima specifica è relativa all'operazione che verifica se una sfilata è fatta da esordienti

SpecificaUseCase VerificheSfilate

verificaEsordienti(s : Sfilata) : booleano
pre: nessuna
post: Sia C l'insieme degli oggetti di classe Collezione coinvolti in link di tipo 'presentata' con s, ovvero:
C = { c in Collezione | <s, c> in presentata }.
Sia inoltre
T = { t in Stilista | <c,t> in appartiene e c in C }
l'insieme degli oggetti di classe Stilista coinvolti in link di tipo appartiene con oggetti in C.
result vale true se e solo se
$(\sum_{(t \in T)} t.anniEsperienza) / |T| < 5.$

La seconda è relativa al numero di capi di collezioni pret-a-porter acquistati da aziende medio-grandi.

calcolaVendite(c : Collezione): intero >= 0
pre: nessuna
post: Se c non e' di classe PretAPorter, allora result = 0.
Altrimenti (c e' di classe PretAPorter), detto
$L = \{ <c,a> \text{ in } c.acquista \mid a.isMedioGrande() = true \}$
result e' pari a $\sum_{(l \in L)} \ln Capi$

Infine l'unica specifica per il diagramma delle classi è quella relativa alla classe Azienda ed alla sua operazione isMedioGrande()

SpecificaClasse Azienda
isMedioGrande() : booleano
pre: nessuna
post: result = true se e solo se this.capSociale > 10 000 000.
FineSpecifica

che possiamo già descrivere usando la notazione JML che verrà poi inserita all'interno della classe Azienda.java, prima del metodo isMedioGrande().

La notazione JML di questa operazione è, dunque, la seguente:

```
/*@    requires  capSociale > 10000000;
       ensures \result == true;
       also
       requires  capSociale <= 10000000;
       ensures \result == false;
 @*/
```

Come possiamo vedere sono previste due possibili postcondizioni (separate dalla parola chiave "also"), dipendenti dalle due diverse precondizioni che si possono verificare nell'invocazione dell'operazione. Infatti la specifica JML asserisce che qualora il capitale sociale dell'Azienda (indicato con la variabile "capSociale") sia superiore a 10 milioni il risultato dell'operazione sarà true, altrimenti sarà false.

4.5.3 Fashion District: Fase di Progetto

Nell'evoluzione di un progetto, alla fase di analisi segue quella di progetto. In questo paragrafo vedremo, applicate all'esempio che stiamo studiando (Fashion District), come unire le tecniche già note per la progettazione con l'uso di JML.
Lo scopo di questa fase è di tener conto di come deve essere la realizzata la nostra applicazione e non solo di cosa dobbiamo realizzare. Ossia va tenuta in debita considerazione la tecnologia che si usa per la realizzazione dell'applicazione.
A tale scopo prendiamo subito alcune importanti decisioni sui tipi e le strutture dati che saranno usati.
Tali decisioni possono essere riassunte nella seguente tabella:

Tipo UML	Tipo Java	Note
Stringa	String	-
Data	Data	Versione senza side effect
intero>0	int	Verifica lato server
intero	int	-
booleano	boolean	-
Insieme	HashSet	Implementa l'interfaccia Set

Un'annotazione importante va fatta sulla tabella appena esposta.
Il tipo UML intero>0 tradotto con un semplice "int" in Java richiede la verifica lato server che il valore intero sia effettivamente >0.
Questa verifica può essere garantita in JML con l'introduzione di un invariante preposto a tale scopo. Nel progetto che stiamo realizzando due attributi richiedono tale verifica e sono "anniEsperienza", attributo della classe Stilista e "capSociale" della classe Azienda.
Per tal motivo introdurremo nel codice delle classi relative i seguenti invarianti:

```
//@ invariant capSociale>0;

//@ invariant anniEsp>=0;
```

che garantiscono il rispetto delle proprietà definite in fase di specifica.
In generale qualora si debba definire delle restrizioni sui tipi, basta definire un invariante nelle specifiche JML della classe.

La fase di progetto continua con la specifica realizzativi delle classi e degli use case, che consistono in una sorta di raffinamento delle specifiche fatte in fase d'analisi.
Nel progetto in esame la specifica delle operazioni presenti nelle classi coinvolge, come già visto, solo la classe Azienda e la sua operazione isMedioGrande(), che possiamo così rappresentare:

SpecificaClasse Azienda
+ isMedioGrande() : boolean
pre: nessuna
algoritmo: se this.capSociale > 10 000 000 allora ritorna true;
altrimenti ritorna false.
FineSpecifica

Tale specifica è molto simile a quella vista in fase d'analisi. Notiamo solamente il "+" che indica che l'operazione dovrà avere visibilità pubblica.
Pertanto, la corrispondente specifica JML risulta invariata rispetto a quella presentata nel paragrafo 4.5.2.

Vediamo ora la specifica realizzativi degli use case.

SpecificaUseCase VerificheSfilate

```
+verificaEsordienti(s : Sfilata) : boolean
pre: nessuna
post: algoritmo:
        somma = 0;
        quanti = 0;
        per ogni 'l' in s.presentata {
        somma = somma + l.Collezione.appartiene.Stilista.anniEsperienza;
        }
        ritorna ( (somma/|s.presentata|) < 5 );

+calcolaVendite(c : Collezione): int
pre: nessuna
post: Se c non e' di classe PretAPorter, allora ritorna 0.
        Altrimenti (c e' di classe PretAPorter) {
                result = 0;
                        per ogni 'l' in c.acquista {
                                se l.Azienda.isMedioGrande() = true {
                                result += l.nCapi;
                        }
                }
        }
        ritorna result;
```

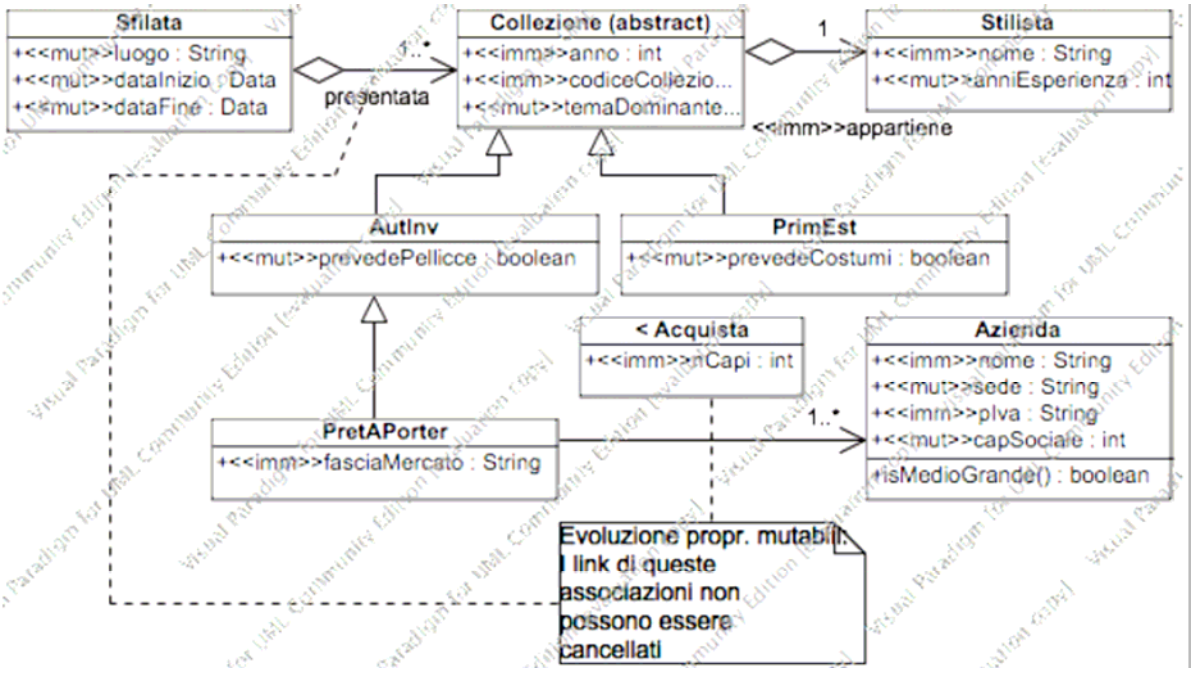
Le specifiche tradotte in JML sono rispettivamente:

```
/*@  requires s!= null;
     ensures (* \result ==  verifica se la media degli anni d'esperienza degli
     stilisti< 5   *);
@*/
```

```
/*@    requires c!= null && PretAPorter.class.isInstance(c);
       ensures  (* \result == numero di capi acquistati da aziende con capitale
       maggiore di 10 milioni *);
       also
       requires requires c!= null && !PretAPorter.class.isInstance(c);
       ensures \result ==0;
@*/
```

Come possiamo vedere dal codice appena esposto, alcune delle postcondizioni non possono essere espresse con i costrutti messi a disposizione da JML, che non ha la sufficiente complessità espressiva per rappresentare le condizioni da verificare. Per tal motivo si è deciso di definire tali specifiche in maniera informale con il costrutto (* … *) come visto nel paragrafo 2.11.

Concludiamo la fase di progetto, realizzando il diagramma delle classi realizzativo



Il diagramma delle classi è stato opportunamente modificato definendo le responsabilità per ciascuna relazione e dichiarando tutti gli attributi sia delle classi che delle associazioni se siano di tipo mutabile o immutabile. Infine la classe Collezione verrà realizzata come

classe abstract, poiché la sua generalizzazione costituita dalle classi AutInv e PrimEst, è completa.

Alla luce di quanto detto, nelle specifiche JML dovremmo tener conto dei seguenti aspetti:

- Gli attributi di tipo Stringa o che sono una classe Java, definiti come immutabili nel diagramma delle classi realizzativi, vanno definiti nelle specifiche JML come "*non_null*".
- Gli attributi che hanno un livello di visibilità basso, ma che sono necessari per la definizione di precondizioni o postcondizioni nelle specifiche di metodi, vanno definite con visibilità pubblica nelle specifiche attraverso il costrutto */*@ spec_public @*/*
- Tutti i metodi che non effettuano side effect e che sono usati nelle specifiche vanno definiti come puri attraverso il costrutto */*@ pure @*/.*
- Se un metodo lancia una o più eccezioni va usata nelle specifiche la clausola *signals_only* che segnala la possibilità che tali eccezioni possano essere lanciate. Qualora oltre a segnalare la possibilità del lancio dell'eccezione bisogna garantire che vengano mantenute alcune proprietà, va usata la clausola "*signals*".
- Le variabili modificate da un metodo vanno segnalate nelle specifiche attraverso la clausola "assignable".
- Se un attributo deve garantire particolari proprietà, ad esempio un valore intero compreso in un particolare intervallo, va usata la clausola "*invariant"* per definire un invariante.

Tali aspetti possono essere assunti come una guida generale nella definizione di una qualsiasi specifica JML di una o più classi Java.

4.5.4 Fashion District: Fase di Realizzazione

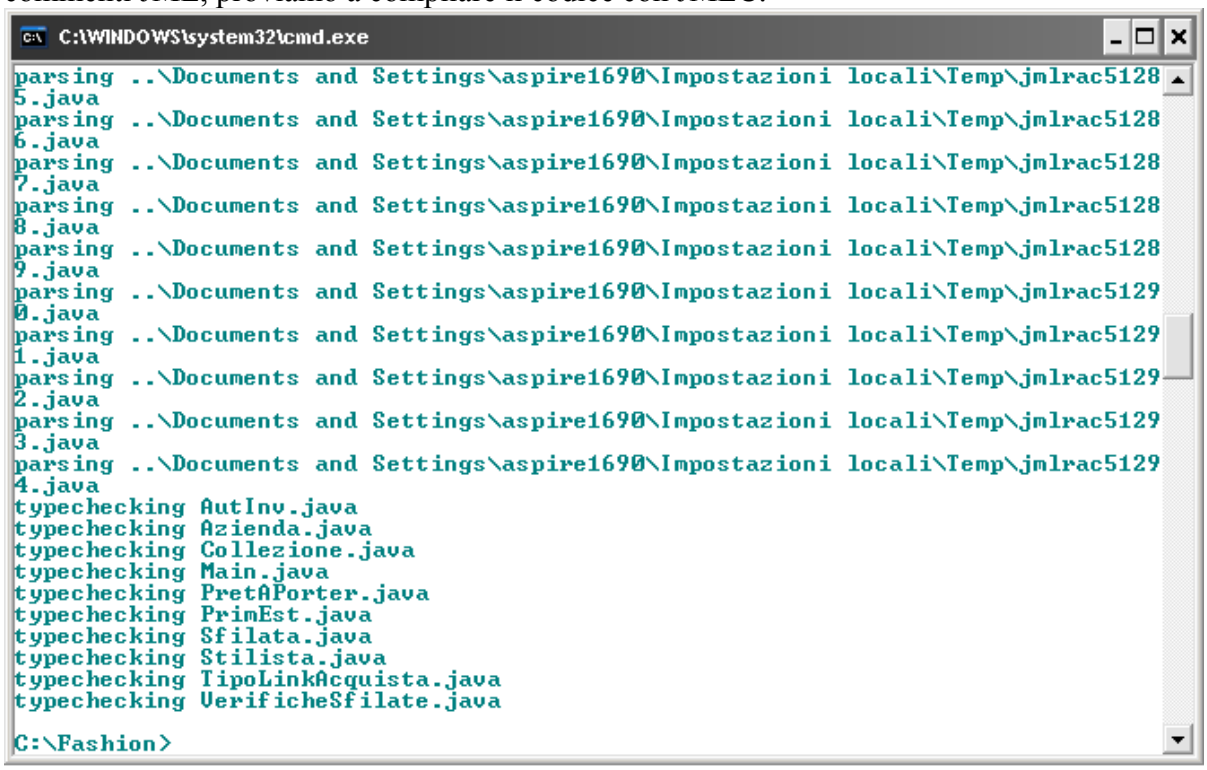
Terminata la fase di progetto, si passa alla fase di realizzazione in cui si realizzano le classi Java, conformi in maniera corretta e completa alle specifiche dei requisiti e alle scelte fatte nei passi precedenti della progettazione ed in più corredate dalle specifiche JML.

A titolo d'esempio vedremo una delle classi Java realizzate, che presenta diverse specifiche JML interessanti.. Per tutte le altre classi realizzate, rimandiamo alle classi allegate a questo lavoro. La classe che vedremo è Stilista.java che realizza le specifiche relative al concetto Stilista.

```
public class Stilista {
  private /*@ spec_public non_null @*/ final String nome;
  private /*@ spec_public @*/ int anniEsp;
  //@ invariant anniEsp>=0;

  /*@     requires n!=null;
          assignable nome,anniEsp;
          ensures (nome == n) && (anniEsp == a);
          signals_only EccezionePrecondizioni;
   @*/
  public Stilista(String n, int a) throws EccezionePrecondizioni {
          if (n==null) throw new EccezionePrecondizioni("Il nome non puo' essere
null");
          nome=n;
          setAnniEsperienza(a);
  }

  /*@
          requires a>=0;
          assignable anniEsp;
          ensures anniEsp == a;
          signals_only EccezionePrecondizioni;
   @*/
  public void setAnniEsperienza(int a) throws EccezionePrecondizioni {
          if (a<0) throw new EccezionePrecondizioni("a deve essere positivo o nullo");
          anniEsp=a;
  }

  public /*@ pure @*/ String getNome() { return nome; }

  public /*@ pure @*/ int getAnniEsperienza() { return anniEsp; }

  public /*@ pure @*/ String toString() {
          return nome + " (" + anniEsp + " anni di esperienza)";
  }
```

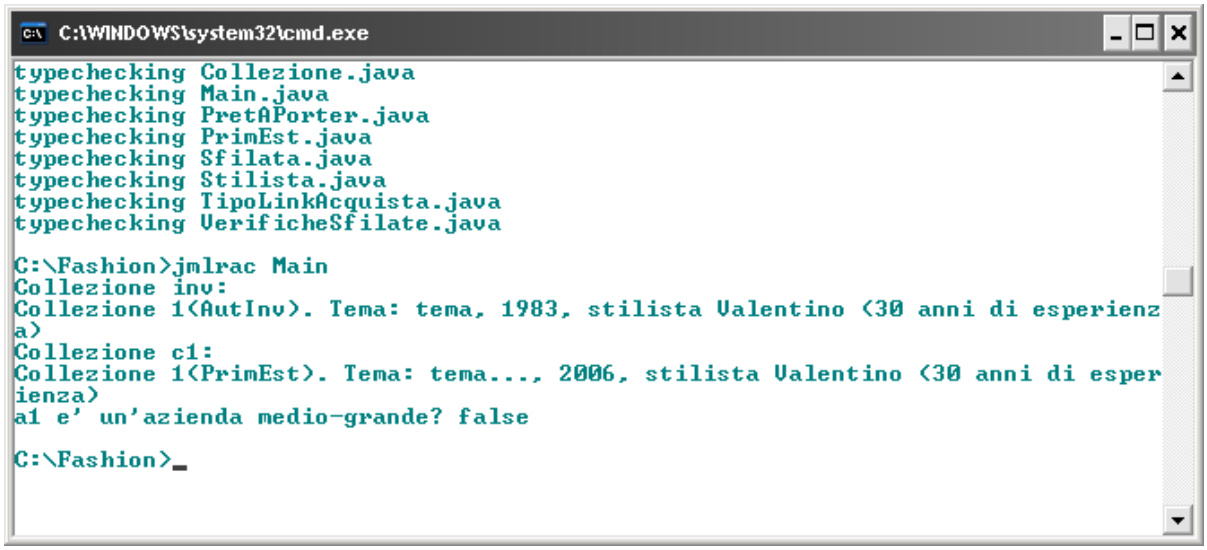
Come possiamo vedere nel codice le variabili private “nome” e “anniEsp”, che sono usate nelle specifiche vengono dichiarate “spec_public”, perché usate nelle specifiche di alcuni metodi ed in particolare la variabile “nome” che, nella fase di progetto abbiamo deciso che sia immutabile, viene definita “non_null”. La variabile anniEsp presenta un invariante perché il suo valore deve essere, coerentemente con le specifiche, sempre >=0.
Il costruttore ha come precondizione la verifica che la variabile “n” passata in ingresso sia diverso da null e garantisce come postcondizione la definizione delle variabili “nome” e “anniEsp”. Inoltre viene segnalata l’eccezione che potrebbe essere lanciata nell’invocazione del metodo che è del tipo EccezionePrecondizioni. In maniera simile è definita la specifica per il metodo setAnniEsperienza(int a).

Per verificare la correttezza di Stilista.java e delle altre classi Java realizzate con i relativi commenti JML, proviamo a compilare il codice con JMLC.



La compilazione, come è chiaramente visibile dall'immagine sovrastante, è andata a buon fine.

Per concludere eseguiamo una classe Main di prova. Il codice è ben formato e non produce violazioni delle specifiche JML, come possiamo vedere nell'immagine sottostante.



Capitolo 5: Altri tool che usano JML

In questo capitolo passeremo in rapida rassegna altri tool, attualmente disponibili, che fanno uso di JML e tentano approcci diversi nell'ambito della correttezza dei programmi.
Lo scopo di questo capitolo è di fare un quadro generale dello stato dell'arte nel momento in cui viene svolto questo lavoro, utile non solo per descrivere le potenzialità del prodotto sotto esame, ma anche come possibile fonte di spunto per futuri lavori degli studenti del corso di Metodi Formali nell'Ingegneria del Software.

5.1 Static checking and verification

In questa sezione presenteremo alcuni tool che permettono di effettuare il controllo statico dei programmi.

5.1.1 LOOP

5.1.1.1 Lo static checker LOOP

Il termine LOOP deriva da "Logic of Object-Oriented Programming" ed appartiene alla famiglia degli static checker, ossia quella di ESC/Java2.
Verifica proprietà definite attraverso le asserzioni JML che vengono opportunamente trasformate e date in pasto ad un prover di teoremi utile a controllare la correttezza dell'implementazione. Uno dei prover usati è PVS. Per effettuare la verifica usa una semantica preposta a lavorare con PVS e per ogni metodo del codice da verificare genera una "proof obligation".
L'output che si ottiene, tuttavia, è di difficile interpretazione.

La novità che introduce questo prodotto rispetto ad ESC/Java e alla sua estensione ESC/Java2 consiste nella possibilità di interazione con l'utente.
Una possibile strategia per una verifica quanto più completa e precisa di una specifica potrebbe consistere, ad esempio, nell'usare ESC/Java2 in una prima fase e poi nell'applicazione di LOOP per successivi raffinamenti dei risultati ottenuti.
Solitamente,infatti, questo tool è usato solo usato nei progetti in cui lo sforzo applicativo richiesto è alto.

5.1.1.2 PVS, il prover usato da LOOP

Il termine PVS sta per Prototipe Verification System ed è un sistema di verifica per la realizzazione di specifiche formali e di teoremi. PVS è caratterizzato da un linguaggio di specifica basato sulla **logica higher-order** integrato con tool di supporto e un theorem prover. La scelta è caduta su tale logica perché, a detta dei realizzatori di PVS, promuove e permette la costruzione di specifiche compatte e facilmente comprensibili.

L'obiettivo principale di PVS, infatti, consiste nel fornire un supporto formale per la concettualizzazione e la definizione di specifiche già nelle prime fasi del ciclo di vita della progettazione di un software.
L'idea alla base di PVS consiste nella definizione di "proof" o teoremi che vanno verificati attraverso il theorem prover.
Il prover mantiene un "proof tree" e si pone l'obiettivo di costruire un albero completo, nel senso che tutte le foglie dell'albero vengono riconosciute come vere. Ogni nodo dell'albero viene denominato "proof goal" ed è un *sequente* che consiste di una sequenza di formule chiamate *antecedenti* e una sequenza di formule chiamate *conseguenti*.
In PVS un sequente è descritto così:

```
{ -1 } A1
{ -2 } A2
[ -3 ]  A3
          .
          .
          .
----------
{  1 }  B1
[  2 ]   B2
{  3 }  B3
          .
          .
          .
```

dove Ai e Bj sono formule PVS che insieme formano una *formula sequente*. Gli Ai sono antecedenti, mentre i Bj sono i conseguenti, mentre la riga serve per separare antecedenti e conseguenti. I numeri nelle parentesi prima delle formule sono usati per denominare la corrispondente formula. La parentesi quadra indica una formula che non è cambiata in un sotto-goal ossia in un nodo figlio, mentre le parentesi graffe indicano una formula nuova o che è stata modificata. Un altro modo per rappresentare la formula è il seguente:

$$A1,A2,A3,.. \vdash B1,B2,B3,\ldots$$

La sequenza di antecedenti o dei conseguenti può essere vuota. L'interpretazione di tale formula, cioè di un sequente, è che la congiunzione degli antecedenti implica la disgiunzione dei conseguenti.
Il proof tree inizia con un nodo radice della forma $\vdash A$ con A il teorema o proof che deve essere dimostrato. PVS costruisce passo passo un albero. Un sequente è vero (true) se ogni antecedente è falso o se ogni conseguente è vero oppure se ogni antecedente assume lo stesso valore del conseguente.
Altri sequenti possono essere verificati come true facendo leva su meccanismi di inferenza più complessi.
Quando un sequente è riconosciuto come vero, il ramo dell'albero della dimostrazione in cui si trova è concluso. L'obiettivo è costruire un albero in cui tutti i rami terminino con sequenti veri.

PVS, quindi, usa il *calcolo dei sequenti* per la dimostrazione dei proof. Usando Δ e Γ per rappresentare una sequenza di formule, le regole di inferenze vengono indicate con

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ \mathrm{R.}$$

che afferma che data una foglia di un proof tree del tipo $\Gamma \vdash \Delta$, applicando la regola R, si possono ottenere un albero con n nuove foglie.
Le regole di inferenza applicabili sono di due tipi: strutturali e logiche, alle quali si aggiungono delle regole definite per l'uso dei quantificatori. Per maggiori dettagli sul calcolo dei sequenti, che in questa circostanza esulano dallo scopo del presente lavoro, rimandiamo a [11].

5.1.2 JACK

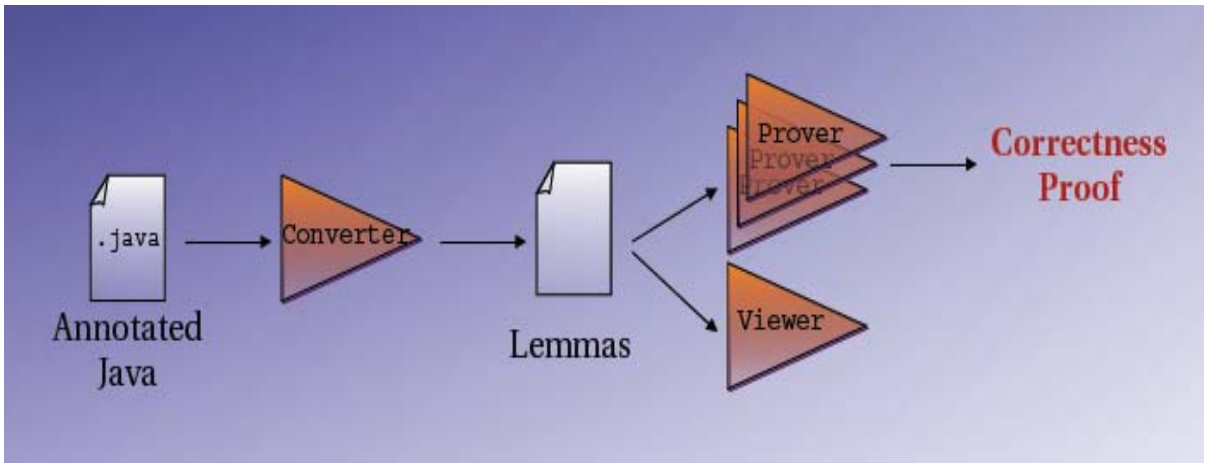
Alla stessa famiglia del precedente tool appartiene anche JACK: Java Applet Correctness Kit. Questo tool è disponibile come plugin per Eclipse, uno dei più famosi ed usati IDE per lo sviluppo in diversi linguaggi d'applicazione, tra cui anche Java.
L'approccio usato in JACK può essere sintetizzato come una via di mezzo tra ESC/Java e LOOP, ma più vicino a quest'ultimo che al primo. JACK è molto più ambizioso di ESC/Java cercando di essere un vero programma di verifica, piuttosto che solo un checker statico. D'altra parte facilita l'utente non richiedendo un particolare sforzo per la comprensione dell'utilizzo del theorem prover e dei suoi risultati come accade con LOOP.

JACK, partendo dalle annotazioni JML, implementa un calcolo automatizzato di precondizioni più deboli. Tali "weakest precondition" sono poi dimostrate attraverso l'uso di prover automatici.
Una delle operazioni consentite all'utente è quella di cambiare il prover con cui fare la verifica (ci sono circa quattro prover che possono essere scelti) o cambiare il numero dei task da provare in parallelo.
Tra i prover disponibili segnaliamo la presenza di Simplify, il prover usato anche da ESC/Java2, per maggiori dettagli sulla tipologia di formule generate rimandiamo al paragrafo relativo a Simplify (3.2.1).

Gli obiettivi perseguiti dagli ideatori del tool nella sua realizzazione sono sintetizzabili in tre punti:

- mantenere Java come linguaggio di validazione. (per questo hanno usato JML come linguaggio di specifica).
- non cambiare ambiente di lavoro (perciò è un plugin di eclipse)
- permettere differenti livelli di validazione (infatti JACK dispone di diversi provers automatici ed interattivi).

La struttura del lavoro che compie JACK è la seguente:



Come possiamo vedere dalla figura i passi che compie sono:

- Conversione di file Java annotati con specifiche JML in lemmi
- Uso di Prover automatici/interattivi per verificare la correttezza dei lemmi
- Uso di viewer dei lemmi che nascondono la complessa struttura sottostante

JACK è un tool corretto e completo; infatti, genera solo giudizi la cui formalizzazione è rispettata dall'applicazione.

Forniamo, infine, qualche nota per chi fosse interessato all'uso del plugin.
Dopo l'installazione del prodotto, va creato un progetto Java ed una classe Java sulla quale può essere fatta la verifica con JACK usando il bottone "c" o selezionando nel Menù contestuale Jack-> Verify Source.
Questa fase permette di generare il risultato. Gli eventuali lemmi possono essere visualizzati tramite il bottone "e" o selezionando Jack-> Edit

5.2 Generazione di specifiche

In questa sezione tratteremo alcuni tool che permettono di inferire automaticamente le specifiche che un codice deve avere e quindi soddisfare.

5.2.1 Daikon

La maggior parte dei tool per JML assumono l'esistenza di specifiche JML e verificano, di conseguenza, il codice in base a tali specifiche. La definizione delle specifiche, quindi, è a carico del programmatore. Il tool che prenderemo ora sotto esame, Daikon, realizzato presso i laboratori del MIT, permette di generare in maniera automatica specifiche per il codice sorgente che gli viene passato in input.

L'idea di fondo, che ha portato alla realizzazione del tool, risiede nella necessaria richiesta di tempo e alla presenza di errori che un task, come la definizione di specifiche, solitamente comporta.
Il tool Daikon è progettato per trovare in maniera dinamica invarianti in punti specifici del codice.
Inoltre crea dei profili di esecuzione del programma, riporta le proprietà che risultano vere in queste esecuzioni ed infine osserva i valori che il programma computa a runtime, generalizzandoli e restituendo proprietà come risultato finale. Come tutte le analisi dinamiche, la tecnica può presentare delle imprecisioni. Ad esempio, altre esecuzioni del programma, non previste dal tool, potrebbero falsificare alcune delle proprietà che sono state trovate in precedenza.
Per questo quando vengono osservati i valori calcolati durante i test, ne viene controllata anche l'accuratezza. Tale accuratezza dipende dalla qualità e dalla completezza dei test che vengono eseguiti. Per ridurre la probabilità d'errore viene fatta una analisi statica ed usati test statici ed altre tecniche per cercare di eliminare false proprietà.
Studi statistici sul prodotto hanno dimostrato che circa il 90% delle proprietà trovate da Daikon sono verificabili da ESC/Java, le altre pur essendo vere vanno oltre le capacità di dimostrazione di ESC/Java. Inoltre, circa il 90% delle proprietà che ESC/Java ritiene necessarie per completare le verifiche relative ad un programma sono individuate da Daikon.

5.2.2 Houdini

Questo tool deduce annotazioni JML per un dato programma. Come il precedente, cerca di trovare annotazioni candidate per il programma passato in input. Per fare ciò usa un algoritmo particolare che si compone dei seguenti passi:

- Vengono generate annotazioni JML candidate
- Viene eseguito ESC/Java più volte per eliminare tutte quelle annotazioni che non sono consistenti con il codice, non considerando, però i potenziali errori a runtime.
- Quando tutte le rimanenti annotazioni sono consistenti, viene eseguita per un'ultima volta ESC/Java, il cui output viene restituito con gli eventuali warnings.

I benefici delle annotazioni JML sono le stesse sia per Houdini che per ESC/Java, ma a differenza di quest'ultimo Houdini trova gli errori di un programma fra un insieme più piccolo delle annotazioni JML.

Nonostante i vantaggi, questo tool presenta anche alcuni problemi che lo rendono meno efficace. Infatti, ad esempio proprio la strategia di ricerca delle asserzioni, che è piuttosto semplice, limita il numero di possibili warnings che possono essere individuati.

5.3 Runtime Assertion Checker

5.3.1 Jass

Jass è l'acronimo di "Java with assertions"(pronunciato "jazz", da qui il simbolo del tool). E' un precompilatore che supporta le asserzioni Java.
Nasce per rispondere a due esigenze delle richieste fatte dagli utilizzatori di Java, ossia le asserzioni e la Progettazione a contratto.
Il precompilatore è scritto completamente in Java.
Una classe annotata con asserzioni Jass deve essere salvata in un file con estensione .jass o .java.
Il precompilatore produce un classe valida Java con estensione .java. Tale file deve essere compilato con un compilatore Java per ottenere il comportamento a runtime specificato, ossia viene usato il tradizionale comando *javac*.
Sono previsti, inoltre, una serie di controlli a livello di compilazione utili per gestire diversi aspetti.

I principali sono:

- Pre: controlla le precondizioni
- Post: controlla le postcondizioni
- Inv: controlla gli invarianti
- Check: verifica tutte le asserzioni
- Forall: controlla le espressioni quantificate (sia forall che exists)
- Trace: vengono controllate le asserzioni che si stanno tracciando (per ulteriori dettagli vedere in seguito)
- Opt: il codice java rilasciato in output viene ottimizzato

Jass supporta differenti tipi di asserzioni usate per specificare differenti proprietà delle classi Java. Tutte le asserzioni devono essere poste in maniera formale all'interno di commenti. La sintassi è molto simile a quella di JML. I costrutti hanno nomi simili, ad esempio require al posto di requires, ensure al posto di ensures. Inoltre, i commenti siano essi su una linea o su più linee non necessitano del simbolo speciale "@" .
Dalla versione 3 Jass non usa più il suo proprio linguaggio di input, ma usa JML per la definizione delle asserzioni.

Ciò che lo diversifica da JML e dagli altri linguaggi di specifica risiede principalmente nell'offerta addizionale che propone all'utente. Infatti, non solo permette la gestione di pre-condizioni, post-condizioni ed invarianti, ma anche il tracciamento delle asserzioni e controlli di raffinamento. In particolare, il tracciamento delle asserzioni è usato, ad esempio, per monitorare il comportamento dinamico di un oggetto, le chiamate e l'ordine d'invocazione di metodi. Questa peculiarità lo rende molto utile soprattutto nella progettazione di sistemi concorrenti.

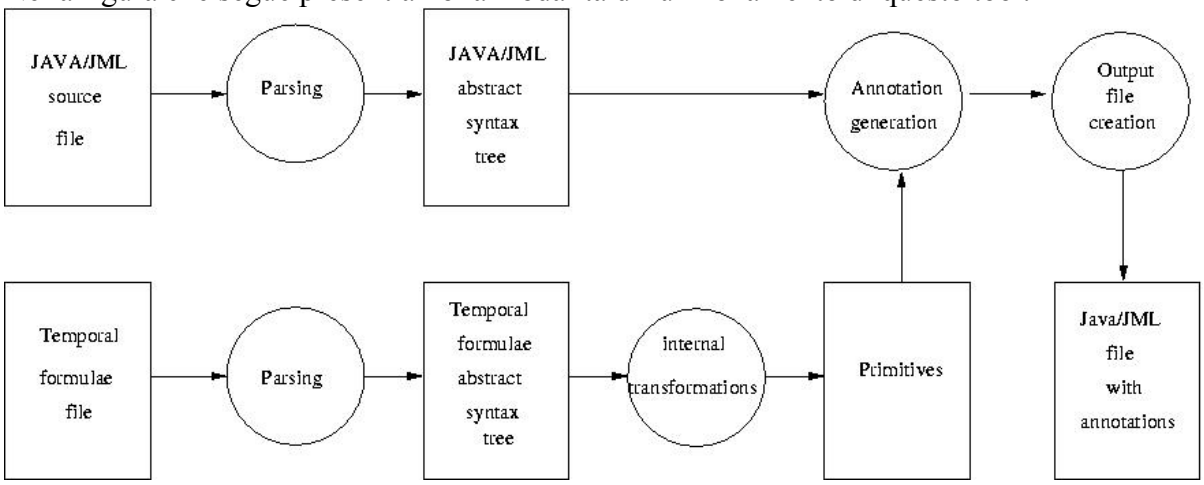
5.4 JML + Proprietà temporali

5.4.1 JAG

JAG è un generatore di annotazioni JML utile per verificare proprietà temporali in classi Java. Consiste in più traduttori che permettono di trasformare proprietà dinamiche in annotazioni standard JML, che assicurano che le proprietà della classe siano soddisfatte.
Nelle primissime versioni di JAG veniva usato come linguaggio di input JTPL (Java Temporal Pattern Language), una sorta di adattamento a Java di un frammento della Logica Temporale Lineare (LTL). Questa logica permetteva di gestire terminazioni dei metodi ad esempio a causa di eccezioni e permetteva di esprimere sia proprietà di safety (proprietà che devono essere soddisfatte, affinché non avvenga qualcosa che possa nuocere all'applicazione) che di liveness (proprietà, che sotto certe condizioni, prima o poi devono verificarsi).

Per la verifica della correttezza del codice Java può essere usato un qualsiasi meccanismo visto nei precedenti paragrafi, come ad esempio JACK.

Nella figura che segue presentiamo la modalità di funzionamento di questo tool.



Il tool JAG prende in input un file Java, preferibilmente già annotato, e un file contenente una o più formule temporali ed effettua le seguenti operazioni:

- Le formule temporali vengono tradotte in primitive intermedie. Ogni proprietà temporale viene ridotta in una o più primitive intermedie che sono semanticamente equivalenti alle proprietà. Esempi di primitive sono *Inv* che rappresenta una proprietà di tipo safety, *Loop* che rappresenta un proprietà di liveness e *Witness* che è uno speciale marcatore per la classe e che può essere usato, ad esempio, per verificare se un metodo è stato già chiamato durante l'esecuzione.

- Le primitive intermedie vengono tradotte in annotazioni standard JML. Ogni primitiva ha il suo corrispettivo valore in JML. Ad esempio Inv viene tradotta in un invariante (in JML invariant); Loop diventa un insieme di invarianti e di vincoli utili per descrivere l'assenza di deadlock del sistema. Il tool restituisce un file di output formato dal file iniziale con le annotazioni JML appena generate, in modo che possa essere sfruttato da altri tool che lavorano con JML.
- Le annotazioni generate vengono preservate. Per ogni annotazione è sempre possibile risalire alla primitiva intermedia e alla proprietà temporale che l'ha generata.

Questo tool è ancora in fase di sviluppo, ma si sta avviando verso una migliore integrazione con altri tool che lavorano con JML e a verso una maggiore capacità di input, attraverso l'uso di altre logiche temporali.

Capitolo 6 Conclusioni

A conclusione di questo lavoro tracciamo le fila di quanto visto relativamente a JML e ai diversi tool che ne sfruttano le potenzialità.
Dovendo sintetizzare i benefici di questo linguaggio di specifiche formali possiamo enuclearli in tre grandi tematiche:

- JML risulta semplice da imparare per programmatori Java: la sua sintassi e semantica, infatti, è molto simile a quella di Java. Ciò è un valido aiuto per l'apprendimento, a differenza di altri linguaggi di specifica più generali, ma che richiedono un maggiore sforzo. Scegliendo JML si invoglia, inoltre, il programmatore a definire formalmente le specifiche del proprio codice, superando i limiti dovuti alla mancanza di tempo e/o volontà che solitamente si presentano nell'acquisizione di nuovi formalismi e meccanismi.
- JML "non necessita" della costruzione di un modello formale a priori per essere utilizzato. Questo perché il modello è il codice stesso della classe o interfaccia che si sta realizzando e di cui si stanno definendo le specifiche formali. Questo comporta sostanzialmente due vantaggi:
  - JML può essere usato per codice già esistente, elemento da non trascurare quando si parla di sistemi legacy.
  - È semplice introdurre l'uso di JML in maniera graduale, aggiungendo nuove asserzioni al codice Java
- JML si avvale del crescente interesse intorno a sé. Infatti, sono molti i tool che sono stati sviluppati e che coprono un ampio spettro di aspetti, come abbiamo visto nei capitoli precedenti. Questa caratteristica la si deve anche alla scelta di JML di non imporre una particolare metodologia di programmazione ai suoi utenti e perché è stato realizzato per rispondere sia alle esigenze dell'interfaccia sintattica del codice Java che al suo comportamento.

La scelta di JML, quindi, rappresenta un valido contributo nell'ambito della programmazione in Java, quando si desidera realizzare un prodotto ben formalizzato ed attinente alle specifiche.

Bibliografia

[1] G. T.Leavens, A.L. Baker, C. Ruby, *JML: A notation for Detailed Design*
[2] G. T.Leavens, K. Leino C. Ruby, E. Poll *JML: notations and tools supporting detailed design in Java*
[3] Gary T.Leavens, Yoonsik Cheon , *Design by Contract with JML*, Settembre 2006
[4] Betrand Meyer, *Applying "design by contract"*, Ottobre 1992
[5] Burdy ed Altri, *An overview of JML tools and applications*, 2002
[6]Giorgetti, Groslambert, *JAG: JML Annotation Generation for Verifying Temporal properties*
[7] Detlef Bartetzko, *Jass – Java with Assertions*
[8] Detlef, Nelson, Saxe, *Simplify:A Theorem Prover for Pragram Checking,* luglio 2003
[9] Shankar, Owre, Rushby, *PVS Prover Guide,* novembre 2001
[10] Shankar, Owre, Rushby, Stringer, *PVS Language Reference,* novembre 2001
[11] Sakharov, *Sequent Calulus Primer*