JML in a nutshell

Domenico Bianculli
bianculld@libero.it

Alessandro Monguzzi
alessandro.monguzzi@gmail.com

ABSTRACT

JML, Java Modeling Language, è un linguaggio usato per specificare il comportamento delle classi e delle interfacce Java attraverso opportune annotazioni del codice sorgente.
In questo documento forniamo, attraverso alcuni esempi, una panoramica delle caratteristiche e delle possibilità di specifica offerte dal linguaggio, ed una rassegna dei principali tool di supporto.

1. INTRODUZIONE

L'ingegneria del software ha, tra i suoi obiettivi, quello di fornire metodologie e tecniche per progettare software corretto e manutenibile. Sin dalla fine degli anni '60 si è cercato di proporre tecniche per verificare la correttezza dei programmi, basti citare i lavori di Floyd [13] e Hoare [17]; negli anni successivi si è visto lo sviluppo di diversi sistemi di aiuto nella dimostrazione, eseguita dalla macchina, della correttezza dei programmi.
I linguaggi di programmazione sono i principali strumenti usati nello sviluppo del software [14]. Diversi linguaggi sono stati sviluppati con l'obiettivo di garantire la correttezza dei programmi, attraverso le specifiche e la verifica del software; altri linguaggi sono stati sviluppati con l'obiettivo di tramutare le specifiche in controlli effettuati a runtime, verificando dinamicamente, per ogni esecuzione del programma, la sua correttezza.
Una delle proposte più rilevanti in tal senso è stata la *progettazione per contratto*, Design by Contract (DBC). DBC è un metodo usato nello sviluppo di software orientato agli oggetti, definito da Bertrand Meyer in [27]. Il concetto principale che sta alla base del DBC è la definizione di un accordo preciso e formale tra una classe e i suoi clienti, il "contratto", o più in generale, *specifica comportamentale dell'interfaccia*, Behavioural Interface Specification (BIS).
I contratti sono espressi usando asserzioni e hanno la forma di invarianti di classe e di pre- e postcondizioni di metodi. Questi costrutti costituiscono il nucleo dei cosiddetti *linguaggi per BIS*, BISL; essi forniscono due principali descrizioni relative ad un modulo di un progetto software: la sua *interfaccia*, cioè le operazioni che il modulo mette a disposizione dei suoi clienti, e il suo *comportamento*, cioè come si comporta il modulo quando viene usato. Attraverso questi costrutti il DBC può diventare una vera e propria *feature* del linguaggio di programmazione.
Un problema di fondo è che, purtroppo, ancora molte specifiche sono scritte in modo informale, facendo uso del linguaggio naturale, di per sè ambiguo; la stessa documentazione delle librerie dei moderni linguaggi di programmazione ha questo stile. Tutto questo fa sì che nello sviluppo di un modulo software molte assunzioni fatte dal programmatore siano lasciate non specificate, creando notevoli problemi di manutenibilità.
Nel 1999, per far fronte a questo problema nell'ambito del linguaggio Java, Gary Leavens della Iowa State University ha proposto JML, *Java Modeling Language* [20, 21], un linguaggio di specifica comportamentale dell'interfaccia per Java.
L'interfaccia sintattica di una classe o interfaccia Java è costituita dalle signature dei suoi metodi, i nomi, i tipi e la visibilità dei suoi campi; il comportamento è espresso attraverso le pre- e postcondizioni dei metodi così come gli invarianti di classe, seguendo l'orientamento del DBC.
JML permette di documentare il comportamento di un modulo software attraverso opportune annotazioni: per favorire la comprensibilità e diffonderne l'uso tra i programmatori Java, si è cercato di rendere la sintassi e la semantica di JML simili a Java, estendendole nelle parti riguardanti l'espressività della specifica.
Informazioni su JML, la documentazione e i tool di supporto sono disponibili all'indirizzo `http://www.jmlspecs.org`.

La struttura di questo documento è la seguente: la sezione 2 introduce la notazione e le espressioni di base del linguaggio; la sezione 3 illustra i tipi di specifica esprimibili in JML; la sezione 4 presenta alcuni aspetti avanzati del linguaggio. Segue, nella sezione 5, un esempio completo di specifica, mentre la sezione 6 offre una rassegna dei principali tool a corredo di JML. Infine la sezione 7 descrive i progetti affini a JML e la sezione 8 conclude questa panoramica di JML.

2. INTRODUZIONE A JML

Le specifiche JML, dette anche *annotazioni*, sono scritte come commenti Java, opportunamente marcati per essere riconosciuti dai tool di supporto. Una caratteristica fondamentale delle annotazioni JML è che sono *eseguibili*, nel senso che è possibile valutarle a runtime attraverso un *assertion checker*, come descritto nella sezione 6.

Per le annotazioni di una sola riga si utilizza
`//@ ...specifica...`
per quelle multiriga si utilizza
`/*@ ...specifica... @*/`.
Le annotazioni devono inoltre essere scritte o prima del metodo che si sta specificando, oppure alla fine del corpo del metodo, prima del `;` di chiusura [1].
Infine, in JML è anche possibile utilizzare specifiche informali, a solo scopo documentativo (non sono eseguibili): esse hanno la forma `(* specifica informale *)`.

2.1 Definizione di pre- e postcondizioni

In JML le precondizioni vengono espresse tramite la parola chiave `requires` e rappresentano ciò che deve essere vero al momento dell'invocazione del metodo.
Per le postcondizioni, bisogna distinguere ulteriormente tra postcondizioni normali e postcondizioni eccezionali. Le prime rappresentano ciò che è vero quando un metodo termina normalmente senza lanciare eccezioni e sono espresse con la parola chiave `ensures`; le seconde rappresentano le condizioni vere quando il metodo termina lanciando un'eccezione di un certo tipo, e sono espresse con la parola chiave `signals`. Ad esempio, se vogliamo dare la specifica di una funzione che calcola la radice quadrata a meno di un certo ϵ, indicato con `eps`, possiamo scrivere:

```
/*@ requires x >= 0.0;
  @ ensures x - \result * \result <= eps;
  @ signals (IllegalArgumentException ex)
  @ ex.getMessage() != null && !(x >= 0.0);
  @*/
public double sqrt(double x);
```

Questa specifica richiede che il metodo riceva in ingresso un valore non negativo e che, in caso di terminazione normale, valga la condizione specificata nella clausola `ensures`; mentre, nel caso venga lanciata un'eccezione di tipo `IllegalArgumentException`, è richiesto che il messaggio dell'oggetto eccezione `ex` non sia nullo e che il parametro in ingresso sia negativo. Precisiamo che nelle postcondizioni, sia normali che eccezionali, i parametri in ingresso sono sempre valutati con il valore che avevano al momento della chiamata.
L'esempio seguente mostra invece come sia possibile specificare che una certa eccezione deve essere lanciata in una determinata situazione: per farlo, è ncessario escludere questa situazione dalle altre clausole `signals` ed `ensures`, realizzando così una funzione totale sul dominio. Un altro modo per specificare lo stesso comportamento sarà descritto nella sezione 3.3.

```
/*@ requires true;
  @ ensures x>=0 && ...
  @ signals (Exception e) x<0 && ...
  @*/
public int foo(int x) throws Exception
```

2.1.1 Espressioni JML

Per definire formalmente pre- e postcondizioni in JML si utilizza una versione estesa delle espressioni Java (JLS 15 [15]). Nella tabella 1 sono riportate alcune delle principali estensioni. Oltre alle normali espressioni logiche, JML introduce una serie di operatori che permettono di gestire alcuni aspetti specifici dei linguaggi di programmazione: `\result` contiene il risultato della chiamata del metodo; `\old` permette di accedere ai valori di una variabile o di un oggetto prima dell'esecuzione del metodo. In generale, tutte le parole riservate in JML cominciano con il carattere `\`.

Espressione	Significato
`a ==> b`	a implica b
`a <== b`	a consegue da b (cioè b implica a)
`a <==> b`	a se e solo se b
`a <=!=> b`	non (a se e solo se b)
`\old(E)`	valore di E prima della chiamata
`\result`	risultato della chiamata del metodo

Tabella 1: Espressioni JML

A differenza di Java, JML impone che le espressioni utilizzate nelle specifiche non abbiano nessun side effect: non è possibile utilizzare espressioni di assegnamento (`=`, `+=`, ecc.), incremento (`++`) e decremento (`--`). Inoltre, nelle espressioni JML è possibile utilizzare solo metodi puri, cioè metodi che non alterino lo stato del sistema (per maggiori dettagli si veda il paragrafo 4.3). Questi metodi vengono indicati con la parola chiave `pure`, ad esempio `public \*@pure@*\ metodo`.

Nelle espressioni è possibile utilizzare diversi quantificatori: universale ed esistenziale (`\forall`, `\exists`), quantificatori generalizzati (`\sum`, `\product`, `\min`, `\max`) e un quantificatore numerico (`\num of`). Un esempio di utilizzo dei quantificatori in una espressione è il seguente, che stabilisce la proprietà di ordinamento crescente dei valori dell'array `a`:

```
/*@ \forall int x;
  @ x >= 0 && x < a.length-1;
  @ a[x] <= a[x+1];
  @*/
```

I componenti di un'espressione che utilizza quantificatori sono: una *dichiarazione* di una variabile locale al quantificatore (`\forall int x;`), seguita da un predicato opzionale di *range* che restringe il dominio a cui si applica il quantificatore (`x >= 0 && x < a.length-1;`); se questo predicato è omesso, tutti gli oggetti sono quantificati. L'ultimo componente, il *corpo* del quantificatore (`a[x] <= a[x+1];`), deve essere vero per tutti gli oggetti che soddisfano il predicato di range.
I quantificatori `\max`, `\min`, `\product` e `\sum` restituiscono rispettivamente il massimo, il minimo, il prodotto e la somma dei valori contenuti nei loro corpi quando le variabili quantificate soddisfano il predicato di range. Il quantificatore numerico `\num of` restituisce il numero dei valori delle variabili quantificate per cui il predicato di range e il corpo sono veri.
Ad esempio un'espressione come `\sum int x; 1 <= x && x <= 5; x` indica la somma dei valori compresi fra 1 e 5.

2.1.2 Clausole di specifica

Riportiamo le principali clausole che possono essere usate nella specifica di un metodo.

[1] Il carattere `@` non deve essere preceduto da spazi. Inoltre, per le annotazioni multiriga è possibile inserire caratteri `@` all'inizio di ogni riga.

requires: specifica la precondizione di un metodo.

ensures: specifica la postcondizione normale di un metodo, cioè la proprietà che è garantita valere alla fine del metodo qualora questo termini senza lanciare una eccezione.

signals: specifica la postcondizione eccezionale di un metodo, cioè la proprietà che è garantita valere alla fine del metodo qualora questo termini lanciando una certa eccezione.

signals_only: è una abbreviazione per una clausola `signals` che specifica quali eccezioni possono essere lanciate, e quindi implicitamente anche quali non possono essere lanciate.

diverge: se vera, indica che il metodo può non ritornare al chiamante, ad esempio perché è entrato in un ciclo infinito.

when: permette di specificare aspetti di concorrenza di un metodo [22]. Il chiamante viene messo in attesa fino a quando la condizione espressa nella clausola è vera. Ad esempio, permette di specificare l'attesa per ottenere un lock.

assignable: fornisce una condizione di frame [6], cioè elenca le variabili a cui il metodo può assegnare un valore e sotto quali condizioni lo può fare. Se non è specificata, si assume che non ci siano restrizioni sulla possibilità di scrittura in memoria. La sua presenza favorisce i controlli statici da parte dei tool di verifica.

capture: specifica quando un oggetto passato come parametro in una chiamata di metodo può apparire nella parte destra di un assegnamento nel corpo del metodo chiamato. È usata in una tecnica di controllo degli alias: ad esempio se un oggetto non deve avere nessun alias non può essere passato ad un metodo che può "catturarlo".

duration: specifica il massimo numero di cicli della macchina virtuale impiegati per eseguire la chiamata del metodo indicato come argomento. Per una certa JVM, un ciclo di macchina virtuale è definito come il minimo del massimo tempo, su tutte le istruzioni `i` della JVM, impiegabile per eseguire l'istruzione `i`.

accessible: specifica le locazioni a cui il metodo può avere accesso diretto durante l'esecuzione.

callable: indica i metodi che possono essere direttamente invocati dal metodo che si sta specificando.

2.2 Definizione degli invarianti

Un invariante è una proprietà che deve essere vera in tutti gli stati visibili al chiamante del metodo. Lo stato di un oggetto `obj` è visibile in uno dei seguenti momenti di esecuzione: dopo l'invocazione di un costruttore che ha inizializzato `obj` oppure prima dell'invocazione di un finalizzatore di `obj`; prima o dopo l'invocazione di un metodo non statico di `obj` o di un metodo statico della classe di `obj`; quando non è in esecuzione nessun metodo invocabile su `obj`. Ad esempio, con `//@ public invariant x != null;` si richiede che la variabile `x` non sia mai nulla in uno stato visibile. Questo invariante può essere sostituito annotando la variabile `x` con la clausola `non_null`[2].

In JML vi sono due tipi di invarianti: statici e di istanza; ogni invariante deve essere statico oppure di istanza ma non entrambi. È possibile specificare esplicitamente il tipo di invariante usando i modificatori `static` oppure `instance`. Un invariante dichiarato in una classe è di default di istanza, mentre uno dichiarato in un'interfaccia è di default statico, e in questo caso, può riferirsi solo agli attributi statici di un oggetto. Un invariante di istanza può invece fare riferimento sia ad attributi statici che non statici.

Un invariante statico dichiarato in un tipo T deve valere in ogni stato che è visibile per il tipo T, mentre un invariante di istanza dichiarato in un tipo T deve valere per ogni stato visibile di ogni oggetto di tipo T.

Come per i metodi statici, anche gli invarianti statici non possono fare riferimento all'istanza corrente `this` e non possono accedere agli attributi o ai metodi non statici.

Gli invarianti possono essere dichiarati con i consueti modificatori di visibilità Java, seguendo le regole standard del linguaggio.

Definiamo ora cosa significa assumere, stabilire e preservare un invariante: un metodo *assume* un invariante se l'invariante deve valere nelle precondizioni; un metodo *stabilisce* un invariante se esso deve valere nelle sue postcondizioni, e *preserva* un invariante se lo assume e lo stabilisce.

Normalmente un invariante vale per tutti i metodi di una classe o di una interfaccia; se si usano dei metodi privati di supporto, detti metodi *helper*, e si vuole che non debbano preservare gli invarianti, vanno annotati con il modificatore `helper`.

2.2.1 Invarianti ed Eccezioni

I metodi e i costruttori devono preservare gli invarianti sia nel caso di terminazione normale, sia nel caso di lancio di eccezioni: gli invarianti sono implicitamente inclusi sia nelle postcondizioni normali (nelle clausole `ensures`) che nelle postcondizioni eccezionali (nelle clausole `signals`).

2.2.2 Invarianti di ciclo

JML permette di annotare `while`, `do ... while` e `for` con degli invarianti di ciclo e delle funzioni varianti di ciclo. Un invariante di ciclo è un predicato dichiarato con le clausole `mantaining` oppure `loop-invariant`, prima del ciclo stesso. Un ciclo con invariante, come il seguente:

```
//@ maintaining J; while (B) { S }
```

è equivalente a:

```
while (true) {
  //@ assert J ;
  if (!(B)) { break; }
  S
}
```

Un invariante di ciclo viene usato nelle dimostrazioni di correttezza parziale del ciclo.

[2] `non null` può essere usato come modificatore in una dichiarazione di metodo per rappresentare una postcondizione relativa al comportamento normale del metodo, indicante che il valore di ritorno del metodo non è nullo. Può anche essere usato come modificatore di un parametro formale per indicare che il corrispondente parametro reale non può essere nullo.

Allo stesso modo per dimostrare la terminazione di un ciclo è possibile annotarlo con una funzione variante di ciclo; essa è un predicato, preceduto dalle clausole `decreasing` oppure `decreases`, costituito da una espressione di tipo `long` o `int` che deve essere non negativa e deve diminuire di almeno una unità ad ogni esecuzione. Il ciclo
`//@ decreasing E; while (B) { S }`
è equivalente a:

```
while (true) {
  int vf = E; // assuming vf is a fresh variable
  if (!(B)) { break; }
  S
  //@ assert 0 <= vf;
  //@ assert E < vf;
}
```

2.3 Asserzioni e vincoli

Un'asserzione è un predicato che deve sempre essere vero. La sintassi per le asserzioni è `//@ assert P`.
Il predicato `P` è un'espressione JML che viene valutata durante l'esecuzione del programma da parte del runtime assertion checker. A differenza dell'omonima istruzione inserita nel linguaggio Java a partire dalla versione 1.4, le asserzioni JML non possono avere effetti collaterali.

Un *history constraint* [25] è una relazione che deve valere per la combinazione di ogni stato visibile e di tutti gli stati visibili futuri in cui l'esecuzione del programma può trovarsi. È usato per descrivere come i valori possono cambiare prima e dopo uno stato visibile, ovvero come i valori possono cambiare nel tempo.
Ad esempio per descrivere che il valore del campo `a` non cambierà mai e la lunghezza di un array `b` potrà solo crescere, si può scrivere:

```
...
int a;
//@ constraint a == \old(a);
boolean[] b;
//@ constraint b.length >= \old(b.length);
```

3. CASI DI SPECIFICA

JML offre la possibilità di scrivere due tipi di specifiche, *heavyweight* e *lightweight*. Le prime sono usate tipicamente per scopi di verifica formale e sono introdotte da parole chiave nella forma `*behavior`; le secondo sono usate per scopi di documentazione e per il runtime assertion checking e non sono introdotte da nessuna parola chiave.

3.1 Lightweight

Una specifica lightweight è incompleta, poiché si assume che chi l'ha scritta non abbia predicato tutto sullo stato dell'oggetto, ma solo sugli aspetti che gli interessavano. Pertanto, questo tipo di specifiche hanno alcune clausole omesse e sottointese con dei valori di default diversi rispetto al caso heavyweight. Infatti, ad eccezione di `diverge`, che è `false`, e di `signals`, che permette di lanciare solo le eccezioni dichiarate nell'intestazione del metodo, tutte le altre hanno come valore predefinito `not_specified`.

3.2 Heavyweight

Distinguiamo tre *casi di specifica* introdotti dalle corrispondenti parole chiave `behavior` (generico), `normal_behavior` (normale) e `exceptional_behavior` (in presenza di eccezioni); in realtà gli ultimi due casi non sono altro che particolari varianti del primo.

3.2.1 Behavior

Una specifica generica ha la forma

```
behavior
  requires P;
  diverges D;
  assignable A;
  when W;
  ensures Q;
  signals (E1 e1) R1;
  ...
  signals (En en) Rn;
  accessible C;
  callable p();
```

La sua semantica è tale per cui se il metodo è chiamato in un pre-stato dove valgono la precondizione P e tutti gli invarianti, allora

- la JVM lancia un errore che eredita da `java.lang.Error` oppure
- l'esecuzione del metodo non termina (va in ciclo infinito oppure esce senza restituire un valore o lanciare un'eccezione) e il predicato è verificato nel pre-stato, oppure
- il metodo termina restituendo un valore o lanciando un'eccezione e:
  - durante l'esecuzione del metodo, vengono assegnati dei valori solo ai campi specificati nella clausola `assignable A`, oppure alle variabili locali del metodo, e
  - in tutti gli stati visibili valgono gli invarianti e gli history constraints, e
  - se l'esecuzione del metodo termina normalmente allora valgono la postcondizione normale `Q` e tutti gli invarianti e gli history constraints, e
  - se l'esecuzione del metodo termina lanciando un'eccezione di tipo `Ei` allora vale la postcondizione eccezionale `Ri` e tutti gli invarianti e gli history constraints, e
  - nel corpo del metodo si fa riferimento alle sole variabili elencate nella clausola `accessible C` e si possono invocare solo i metodi elencati nella clausola `callable p()`.

3.2.2 Normal behavior

Un caso di specifica `normal_behavior` è un `behavior` con la clausola `signals` definita implicitamente come
`signals (java.lang.Exception) false;`
in questo modo si specifica una precondizione che garantisce la terminazione normale, perché al metodo non è permesso lanciare eccezioni, neanche quelle unchecked.

3.2.3 Exceptional behavior

Un caso di specifica `exceptional_behavior` è un `behavior` con la clausola `ensures` definita implicitamente come `ensures false;`.
Una specifica di tipo eccezionale obbliga il metodo a terminare lanciando un'eccezione, tra quelle specificate nelle clausole `signals`, garantendo la postcondizione eccezionale corrispondente al tipo di eccezione lanciata. Ricordiamo la differenza tra la clausola `signals_only` e `signals`: la prima specifica *quali* eccezioni possono essere lanciate quando la precondizione del caso di specifica è verificata; la seconda indica solo un predicato che deve essere vero quando è lanciata una certa eccezione.

3.3 Casi di specifica multipli

Nella stessa specifica è possibile indicare contemporaneamente i casi di specifica normali ed eccezionali: quando viene soddisfatta la precondizione associata ad uno di essi deve essere verificata la postcondizione corrispondente. Nell'esempio seguente notiamo i due tipi di specifica e soprattutto le due clausole `requires` ad intersezione nulla:

```
/*@ public normal_behavior
  @ requires !theStack.isEmpty();
  @ assignable size, theStack;
  @ ensures theStack.
  @        equals(\old(theStack.trailer()));
  @ also
  @ public exceptional_behavior
  @ requires theStack.isEmpty();
  @ assignable \nothing;
  @ signals_only BoundedStackException;
  @*/
public void pop( ) throws BoundedStackException;
```

Questo tipo di specifica può essere trasformata in un'unica specifica di tipo `behavior` come descritto in [30].

4. ASPETTI AVANZATI

In questa sezione presentiamo alcuni aspetti avanzati di JML.

4.1 Model Fields

Un *model field* rappresenta un campo o una variabile che esiste soltanto a livello di specifica, usato per descrivere in modo astratto i valori degli oggetti; esso può essere mappato su uno o più campi concreti —ovvero appartenenti all'implementazione— a seconda delle esigenze. Poiché non fanno riferimento a nessuna area di memoria, i model fields non possono essere inizializzati; per descrivere un'inizializzazione astratta si può usare la clausola `initially`.
Ad esempio nella specifica seguente l'errore ammissibile della radice quadrata ϵ è un esempio di model field:
`//@ public model eps.`
La parola chiave `represents` permette di mapparlo su di un campo concreto:

```
public class MathFoo {
   //@ public model eps;
   private float error;
   //@ private represents eps <- error;
}
```

4.1.1 Model Methods and Types

Oltre ai campi, JML permette di specificare come `model` — ossia relativi solo alla specifica— anche metodi e tipi. A differenza dei model fields, i metodi e i tipi di modello non sono un'astrazione del loro equivalente nell'implementazione: sono semplicemente dei metodi e dei tipi che si possono pensare facenti parte del programma, ai soli fini della specifica.

4.1.2 Visibilità dei campi

A volte, sebbene sia una pratica discutibile, si può voler usare i campi concreti (tipicamente privati) di un modulo anche nelle specifiche; per non dover dichiarare un nuovo campo di modello da associare loro, in JML si possono usare i due modificatori `spec_public` e `spec_protected` per rendere visibile a livello pubblico o protetto, limitatamente alla specifica, il campo in questione.

```
public class Point2D {
  private /*@ spec_public @*/ double x = 0.0;
  private /*@ spec_public @*/ double y = 0.0;
  //@ public invariant !Double.isNaN(x)
                    && !Double.isNaN(y);

  /*@ requires !Double.isNaN(x+dx);
    @ assignable x;
    @ ensures JMLDouble.approximatelyEqualTo
                (x, \old(x+dx), 1e-10);
    @*/
  public void moveX(double dx) {
    x += dx;
  }
}
```

Nell'esempio i due campi concreti `x` e `y` sono resi pubblici a livello di specifica e possono quindi essere usati negli invarianti e nelle pre- e postcondizioni.

4.2 Modelli Astratti

JML ha a corredo un'ampia libreria di classi Java, che possono usate nelle specifiche per descrivere con un certo formalismo matematico il comportamento dell'interfaccia; esse sono contenute nel package `org.jmlspecs.models`. Tra di esse, ci sono quelle per la riflessione delle classi base di Java, vari tipi di collezioni come insiemi, sequenze e multi-insiemi per memorizzare oggetti, valori o identità di oggetti [3], relazioni e funzioni algebriche, classi ausiliarie come eccezioni ed iteratori.
Quasi tutte le classi rappresentano oggetti immutabili e hanno metodi puri; in questo modo, i metodi non cambiano lo stato originale dell'oggetto e il valore restituito dal metodo può essere usato come se si fosse in un contesto di programmazione funzionale. Ad esempio, per inserire un elemento `e` nell'insieme `s` si usa il metodo `insert`: la chiamata `s.insert(e)` non cambia lo stato di `s` e restituisce un nuovo

[3]La distinzione tra queste categorie è basata sul tipo di uguaglianza usata nel confronto tra oggetti e sul supporto alla clonazione: le collezioni di oggetti usano `==` e non clonano gli oggetti inseriti in esse; quelle di valori, usano `.equals` e clonano gli oggetti; le ultime contengono identità di oggetti appartenenti ad una stessa classe di equivalenza generata dalla relazione `.equals`: a seguito di una modifica all'oggetto l'appartenenza alla classe di equivalenza può non valere più e quindi la collezione si ritrova modificata.

oggetto che contiene tutti gli elementi di `s` più `e`.
Gli utenti possono creare i propri tipi con metodi puri per una modellizzazione matematica adatta alle proprie esigenze, implementando l'interfaccia
`org.jmlspecs.models.JMLType` e dichiarando la classe come `pure model class`, come nell'esempio:

```
//@ model import org.jmlspecs.models.JMLType;
/*@ public pure model class QueueEntry
  @  implements JMLType {

  @ ... specifiche dei metodi...
  @*/
```

4.3 Purezza

Un metodo si dice puro se è specificato con il modificatore `pure` oppure se è un metodo che appartiene alla specifica di una classe o interfaccia pura; una classe si dice pura se tutti i metodi non statici e tutti i costruttori dichiarati sono puri. Da notare come i metodi ereditati e non dichiarati puri rimangono non puri. Un metodo puro non può generare effetti collaterali: equivale a scrivere nella specifica queste annotazioni: `diverges false; assignable \nothing;`
Per un metodo puro, deve essere dimostrabile la terminazione: è sufficiente che il programmatore lo renda totale, ovvero faccia in modo che il metodo termini restituendo un valore o lanci una qualche eccezione. Poiché un metodo puro non può andare in loop infinito, se ha una precondizione non banale dovrebbe lanciare un'eccezione quando la sua precondizione normale non è verificata. Il comportamento eccezionale non deve essere specificato precisamente, l'importante è che venga verificata la richiesta di non entrare in loop.

4.4 Data Groups

La dichiarazione di un model field crea automaticamente un *data group*: un insieme di campi concreti e di modello a cui ci si riferisce attraverso un nome specifico, ad esempio il nome del campo di modello che lo ha creato. Lo scopo principale di un data group è di referenziare gli elementi di questo insieme senza esporre i dettagli della rappresentazione interna. Ad esempio, è conveniente aggiungere in un data group quei campi privati e protetti usati per determinare il valore di un campo di modello pubblico.
Un data group ha la stessa visibilità del model field che lo ha dichiarato: un campo non può essere membro di un gruppo con una visibilità inferiore alla sua.
Quando in una clausola `assignable` di un certo metodo si fa riferimento a un data group, tutti i campi di quel gruppo possono essere assegnati nel corpo del metodo.
Un campo può divenire membro di un data group attraverso le clausole `in` e `maps-into`, specificate al momento della sua dichiarazione. L'esempio seguente rende `theStack` membro del gruppo creato dalla dichiarazione di `size`: in questo modo `theStack` può cambiare valore quando cambia il valore di `size`.

```
/*@ public model int size;
  @ public model JMLObjectSequence theStack;
  @                                in size;
  @*/
```

La parola chiave `maps-into` invece, permette di aggiungere i campi di un oggetto in un data group esistente.

4.5 Ereditarietà delle specifiche

In JML, una sottoclasse eredita le specifiche, in termini di pre-, postcondizioni e invarianti, dalla sua superclasse e dalle interfacce che implementa; un'interfaccia eredita le specifiche delle interfacce che estende.
La semantica dell'ereditarietà delle specifiche riflette quella propria dell'ereditarietà in Java, basata sulla risoluzione statica dei tipi delle variabili e sul dynamic dispatching dei metodi. Questa semantica genera il concetto di sottotipo comportamentale, in accordo al principio di sostituzione di Liskov [26].
Nella specifica di una sottoclasse, le specifiche della superclasse vengono congiunte con quelle nuove usando la parola chiave `also`.

4.6 Refinement

Il linguaggio offre la possibilità di separare la specifica di un modulo dalla sua implementazione. Ad esempio si può speficare la classe `Foo` nel file `Foo.java-refined` e scrivere l'implementazione di `Foo` in un file `Foo.java` che include al suo interno la clausola `//@ refine "Foo.java-refine";`.
In questo modo si possono anche aggiungere le specifiche a del codice già scritto senza modificarlo, inserendo solo la clausola, che è un semplice commento.

5. ESEMPIO DI SPECIFICA

In questa sezione mostriamo un esempio completo di specifica di un tipo di dato astratto con JML. Quest'ultimo si presta a supportare l'approccio *abstraction by specification* suggerito in [24]: per ogni classe/interfaccia che rappresenta un ADT è possibile specificare le pre- e postcondizioni, la funzione di astrazione e gli invarianti astratti e di rappresentazione.
In particolare, nelle specifiche dei metodi, e soprattutto della funzione di astrazione e degli invarianti, l'uso dei campi di modello assume una notevole importanza. Essendo circoscritti solo alle specifiche, permettono di non esporre la rappresentazione interna di una classe, garantendo così l'information hiding; inoltre favoriscono la manutenibilità.
Una classe che implementa una specifica con campi di modello può specificare *funzioni di astrazione*, che indicano come associare i valori dei campi concreti (relativi all'implementazione) con i "valori astratti" dei campi di modello [24]. Se una funzione di astrazione è eseguibile, le asserzioni scritte usando i campi di modello diventano eseguibili e possono essere usate nella verifica delle asserzioni a runtime. In [8] viene suggerito di specificare le funzioni di astrazione utilizzando metodi di modello, i quali, se dotati di un corpo, diventano a loro volta eseguibili.

Nell'esempio seguente, mostriamo un'interfaccia Java `SortedIntListType` che rappresenta una sequenza ordinata di interi.
Il campo di modello `theList` è l'astrazione di una lista ordinata che conterrà una sequenza di valori interi ordinati in modo crescente. L'invariante specifica tutto ciò: tutti gli oggetti sono di tipo `Integer` e sono ordinati in modo crescente.
Questo campo di modello è usato per scrivere le specifiche dei metodi `size()`, `get()` e `contains()`, dichiarati puri per essere usati nelle specifiche come metodi observer. Il campo `theList` è usato anche nella specifica del metodo `add()` che ne richiama il metodo `isInsertionInto()`, fornito dal tipo

immutabile `JMLEqualsSequence`.
Infine notiamo che la postcondizione non specifica che la lista risultante dall'inserimento di un nuovo elemento sia ordinata poiché già specificato nell'invariante.

```
//@ model import
//@   org.jmlspecs.models.JMLEqualsSequence;

public interface SortedIntListType {
/*@ public model instance non_null
  @   JMLEqualsSequence theList;
  @ public initially theList.isEmpty();
  @ public instance invariant
  @ (\forall int i; 0 <= i && i < theList.size();
  @ theList.itemAt(i) instanceof Integer)
  @ && (\forall int i; 0 < i && i < theList.size();
  @ ((Integer)theList.itemAt((int)(i - 1)))
  @ .compareTo(theList.itemAt(i)) <= 0);
  @*/

//@ ensures \result == theList.length();
/*@ pure @*/ int size();

/*@ requires 0 <= index && index < size();
  @ ensures \result
  @ == ((Integer)theList.itemAt(index)).intValue();
  @*/
/*@ pure @*/ int get(int index);

//@ ensures \result ==
//@      theList.has(new Integer(elem));
/*@ pure @*/ boolean contains(int elem);

/*@ ensures theList.isInsertionInto
  @    (\old(theList), new Integer(elem));
  @*/
void add(int elem);
}
```

L'esempio seguente mostra invece la classe `SortedIntList` che implementa l'interfaccia descritta sopra, attraverso alberi di ricerca binari.
Il campo `isEmpty` indica se gli altri campi sono definiti; rispettando l'invariante, `isEmpty` è vero se e solo se i sottoalberi sinistro e destro sono nulli.
La funzione di astrazione è specificata usando la clausola `represents`: il valore del campo di modello `theList` è determinato dall'espressione `abstractValue()`. Essa è un metodo di modello il cui corpo può quindi essere eseguito dal runtime assertion checker. Ciò consente a quest'ultimo di determinare un valore per il campo `theList`, che può essere usato nella verifica delle altre asserzioni, ad esempio nell'invariante e nella postcondizione del metodo `add()`.

```
//@ model import
//@   org.jmlspecs.models.JMLEqualsSequence;
public class SortedIntList implements
                  SortedIntListType {
  private boolean isEmpty;
  private int val;
  private SortedIntList left, right;
//@ private invariant isEmpty ==
//@         (left == null && right == null);
//@ private represents theList <- abstractValue();
/*@ private model pure
  @         JMLEqualsSequence abstractValue() {
  @ JMLEqualsSequence ret = new JMLEqualsSequence();
  @ if (!isEmpty) {
  @   ret = left.abstractValue()
  @     .insertBack(new Integer(val))
  @     .concat(right.abstractValue());
  @ }
  @ return ret;
  @ }
@*/
//@ ensures theList.isEmpty();
public SortedIntList() { isEmpty = true; }
/* ... */
public void add(int elem) {
  if (isEmpty) {
    isEmpty = false; val = elem;
    left = new SortedInList();
    right = new SortedInList();
  } else {
    if (elem <= val) { left.add(elem); }
   else { right.add(elem); }
  }
}
}
```

6. TOOL

In questa sezione presentiamo alcuni tool a supporto di JML. I primi tre, `jmlc`, `jmlunit` e `jmldoc` fanno parte della distribuzione standard di JML, sono open source e sono reperibili sul sito ufficiale di JML. I restanti due, ESC/Java e LOOP, sono invece dei tool realizzati in ambito accademico: il primo è disponibile sul sito `http://secure.ucd.ie/products/opensource/ESCJava2/` mentre l'altro non è disponibile pubblicamente.

6.1 jmlc

Il compilatore JML (`jmlc`) [7], sviluppato alla Iowa State University, è un'estensione del tradizionale `javac` e può compilare programmi Java che contengono specifiche JML in bytecode; esso include istruzioni di controllo a runtime che verificano le specifiche JML come precondizioni, postcondizioni normali ed eccezionali, e invarianti. L'esecuzione di questi controlli è trasparente, nel senso che, a meno di violazioni delle asserzioni o di perdita di performance, il comportamento è identico a quello del programma originale senza annotazioni JML. La trasparenza del controllo a runtime delle asserzioni è garantita, poiché esse non possono avere effetti collaterali.
In generale, il compilatore JML fornisce "benefici programmativi" alla specifica formale delle interfacce, permettendo ai programmatori Java di usare le specifiche JML come pratici ed efficaci tool per debugging, testing e DBC.

6.1.1 Static Checker

Qualora non si voglia compilare il codice e le sue specifiche ma solo effettuare un controllo di tipo statico su di esse, si può utilizzare il tool `jml` che effettua solo il controllo di tipo sulle specifiche.

6.2 jmlunit

Una specifica formale può essere vista come un oracolo di test, ed è possibile utilizzare un runtime assertion checker

come procedura decisionale per l'oracolo. A tal fine, è stato sviluppato un tool [9] che inserisce il supporto JML all'interno di JUnit [5]: esso utilizza le specifiche, processate precedentemente da `jmlc`, per decidere se il codice testato funziona correttamente; in questo modo, il programmatore non deve più scrivere il codice che verifica il successo o il fallimento del test.
`jmlunit` genera classi di test JUnit che utilizzano il runtime assertion checker di JML: le classi di test generano messaggi per gli oggetti delle classi Java che vengono testate; il codice di test cattura le violazioni delle asserzioni per decidere se i dati violano le precondizioni del metodo testato (queste violazioni non sono considerate fallimenti del test). Quando il metodo rispetta le precondizioni, ma ha comunque una violazione delle asserzioni, significa che l'implementazione non rispetta la specifica, e quindi vi è un fallimento.

6.3 jmldoc

Poiché le specifiche JML sono pensate per essere utilizzate dai programmatori Java, è importante che questi ultimi possano creare e utilizzare questa documentazione come il tradizionale Javadoc. Il tool `jmldoc` (creato da David Cok) genera pagine HTML che contengono sia le API che le specifiche JML per il codice Java, esattamente come le pagine create con Javadoc.
`jmldoc` riutilizza il parsing e il checking forniti dal JML checker e le doclet API di Javadoc. Il tool combina e mostra in un unico file tutte le specifiche che riguardano una classe, un metodo o un campo. Sono supportate anche la gestione dell'ereditarietà e del polimorfismo delle classi, indicando le superclassi e le interfacce implementate.

6.4 ESC java

ESC/Java [12] è uno strumento, inizialmente sviluppato al Compaq Systems Research Center, che esegue un *extended static control* [11], un controllo statico esteso. Rispetto alle verifiche, prevalentemente di tipo, eseguite da un normale compilatore, ESC, a partire da codice Java annotato con specifiche JML, può trovare potenziali accessi a riferimenti nulli, sconfinamenti dell'indice di un array o cast malformati. I controlli sono eseguiti staticamente e automaticamente, senza nessuna interazione con l'utente.
Una caratteristica particolare è che non è corretto, ovvero non rileva tutti gli errori presenti, e non è completo, nel senso che può rilevare errori che in realtà sono impossibili; i progettisti giustificano questa scelta dicendo che è "cost-effective".
Alla base di ESC/Java si trova una semantica dettagliata del linguaggio Java e un theorem prover non interattivo. Esso si è tramutato di recente in ESC/Java2 [10]: questa nuova versione supporta Java 1.4 e le nuove feature introdotte dalle ultime versioni di JML.

6.5 LOOP

Il progetto LOOP (Logic of Object-Oriented Programming) [18] dell'università di Nijmegen ha come obiettivo quello di introdurre l'uso dei metodi formali per la specifica e la verifica delle proprietà di una classe in linguaggi orientati agli oggetti, in particolare per Java. Il cuore del progetto è rappresentato dalla formalizzazione, basata sulla teoria delle coalgebre, di una semantica denotazionale, prima di Java, e poi di JML, nel linguaggio del theorem prover PVS [28]. Il compilatore LOOP, descritto in [32], traduce il codice Java annotato con specifiche JML in *proof obligations* per PVS; queste sono espresse come particolari formule di correttezza relative ai metodi, in una logica alla Hoare e nel calcolo delle precondizioni più deboli, entrambi formalizzati in PVS. Le *proof obligations* possono poi essere dimostrate interattivamente con PVS stesso, verificando la correttezza dell'implementazione Java rispetto alle specifiche.
In confronto a ESC/Java, da un punto di vista operativo, LOOP permette la verifica di proprietà più complicate, nonostante ciò richieda una certa esperienza dell'utente con il theorem proving; per questo motivo è consigliato utilizzare LOOP a valle della verifica con ESC/Java, avendo così già eliminato una parte degli eventuali errori presenti nella specifica.
Questo tool è stato utilizzato per verificare le specifiche scritte in JML relative a *JavaCard*, il sottoinsieme dell'API di Java a supporto delle smart card, come descritto in [29].

7. ARGOMENTI CORRELATI

Diversi linguaggi di programmazione sono stati progettati con l'obiettivo di garantire la correttezza o la verifica del software; tra questi citiamo gli approcci pioneristici di Gypsy [2], Alphard [34], Euclid [19], e CLU [23], ognuno con un differente grado di formalità.
Gypsy è stato il primo ad ammettere le specifiche come parte del linguaggio di programmazione; esse venivano integrate nel codice sorgente del programma, supportando direttamente la verifica eseguita con un theorem prover interattivo.
Alphard fu pensato in accordo ad una metodologia di programmazione per il progetto e la verifica di strutture dati object-like, ma le dimostrazioni erano fatte manualmente.
In Euclid, le specifiche scritte usando le espressioni booleane del linguaggio di programmazione, erano verificate a run-time, con l'idea che quelle più complesse, scritte in forma di commento, fossero usate da strumenti di verifica dei programmi esterni.
La metodologia di programmazione di CLU includeva in modo massiccio le specifiche, ma queste erano scritte solo in forma di commento.
I tre moderni sistemi basati sul DBC che hanno avuto un effetto diretto sullo sviluppo di programmi concreti, sono Eiffel [27], Spark [3] e B [1].
Il primo è un linguaggio orientato agli oggetti progettato da Bertrand Meyer. La libreria standard a corredo è ben documentata attraverso i contratti, che quindi diventano di uso comune per i programmatori.
SPARK —acronimo per Spade Ada Kernel— è un sottoinsieme limitato del linguaggio Ada 95, senza alcune caratteristiche come i riferimenti, l'allocazione dinamica della memoria e le sottoclassi, ma arricchito con annotazioni, scritte come commenti, per supportare diversi tool. Tra di essi ricordiamo strumenti per l'analisi di tipo data- e information-flow e per la verifica del codice, in particolare per garantire l'assenza di eccezioni a runtime.
L'approccio seguito da B usa una differente metodologia per lo sviluppo del software basata sul concetto di *refinement*: il punto di partenza è una specifica completa del sistema; successivamente, con l'ausilio di strumenti automatici, si procede a raffinamenti delle specifiche fino ad ottenere programmi compilabili. Un aspetto negativo è che B richiede notevoli *skills* da parte dei programmatori, affinché questi facciano proprio il processo basato sul raffinamento.

Negli ultimi anni lo standard UML [31] si è affermato come linguaggio di modellizzazione per le fasi di analisi e progetto dei sistemi software; esso propone come linguaggio di specifica Object Constraint Language OCL [33], che come JML permette di specificare gli invarianti e le pre- e postcondizioni. Una differenza tra JML e OCL è che mentre il primo si rifà a Java, il secondo è indipendente dal linguaggio di programmazione: questa caratteristica di JML può però essere vista, in un certo senso, come un vantaggio, visto che permette a tutti i programmatori Java di usare la stessa sintassi ed espressività del linguaggio di programmazione nella stesura della specifica. Inoltre, ciò garantisce di avere una semantica ben definita per tutte le espressioni presenti nella specifica, cosa che non sempre è vera per OCL. Una traduzione da OCL a JML è descritta in [16].
Infine, i laboratori di ricerca Microsoft hanno proposto Spec# [4], un superset del linguaggio C#, a cui sono stati aggiunti il supporto per distinguere tra riferimenti ad oggetti non nulli e riferimenti ad oggetti possibilmente nulli, la specifica dei metodi in termini di pre- e postcondizioni, una gestione "disciplinata" delle eccezioni e il supporto per imporre vincoli sui campi di un oggetto. È presente un compilatore, integrato nella suite MS Visual Studio, che introduce a runtime dei controlli per la specifica dichiarata, e un verificatore di programmi, Boogie, che cerca di dimostrare staticamente le specifiche usando un theorem prover automatico. Nonostante le molte somiglianze con JML, sia in termini di espressività del linguaggio che di tool a disposizione, ci sono anche molte differenze, a partire dalla metodologia adottata per gli invarianti di un oggetto (Spec# supporta nativamente gli invarianti per le callback) e dalle modifiche apportate al linguaggio base (in JML Java non è stato modificato mentre in Spec# sono stati introdotti, ad esempio, gli inizializzatori di campo e i blocchi `expose`).

8. CONCLUSIONI

A conclusione di questo lavoro, vogliamo mettere in luce i principali punti di forza di JML:

- è facile da imparare per i programmatori Java poiché le sintassi e le semantiche dei due linguaggi sono molto simili.
- non è necessario sviluppare a priori una formalizzazione completa del sistema, poiché il punto di partenza è lo stesso codice sorgente. Ciò permette ad esempio di specificare il comportamento di codice già esistente, migliorandone la documentazione e la manutenibilità.
- permette uno sviluppo incrementale sia del codice che delle specifiche.
- garantisce un perfetto allineamento tra specifica formale e codice di implementazione.
- sono disponibili numerosi tool di supporto.

Riteniamo che queste caratteristiche siano un valido motivo per introdutte l'uso di un linguaggio di specifica formale come JML all'interno del processo industriale di sviluppo del software.

La versione attuale presenta ancora alcune limitazioni, sebbene gli sviluppatori di JML dichiarino che verranno aggiunte a breve; esse sono

- è ammessa la specifica del solo comportamento sequenziale di un modulo Java: ad esempio non è supportata la specifica di proprietà temporali come l'accesso coordinato a variabili condivise oppure l'assenza di deadlock.
- gli invarianti di oggetto non sono supportati in presenza di callbacks.
- non è possibile specificare vincoli sulle relazioni di alias tra oggetti.
- la definizione di metodo puro è molto forte: questo pone molti vincoli nella scrittura delle annotazioni.

9. BIBLIOGRAFIA

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells. Gypsy: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, 1977.

[3] J. Barnes and J. G. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices CASSIS 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[5] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[6] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transaction on Software Engineering*, 21(10):785–798, 1995.

[7] Y. Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Apr. 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.

[8] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software Practice and Experience*, 35(6):583–599, May 2005.

[9] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255, London, UK, 2002. Springer-Verlag.

[10] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices CASSIS 2004*, Lecture Notes in Computer Science, pages 108–128. Springer-Verlag, 2004.

[11] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report #159, Compaq System Research Center, Palo Alto, USA, 1998.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[13] R. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society, 1967.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice–Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, The*. Java Series. Addison-Wesley Professional, 3rd edition, July 2005.

[16] A. Hamie. Translating the Object Constraint Language into the Java Modeling Language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1531–1535, New York, NY, USA, 2004. ACM Press.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10):576–580, 1969.

[18] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security - Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003*, volume 3233, pages 134–153. Springer, 2004.

[19] B. Lampson, J. Horning, R. London, J. Mitchell, and G. Popek. Report on the programming language Euclid. *SIGPLAN Notices*, 12(2), 1977.

[20] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[21] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, 2003.

[22] R. A. Lerner. *Specifying Objects of Concurrent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991. TR CMU–CS–91–131.

[23] B. Liskov. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.

[24] B. Liskov and B. Liskov. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[25] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 16–28, New York, NY, USA, 1993. ACM Press.

[26] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transaction on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[27] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[28] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[29] E. Poll, J. van den Berg, and B. Jacobs. Specification of the Javacard API in JML. In *Proceedings of the 4th working conference on smart card research and advanced applications on Smart card research and advanced applications*, pages 135–154, Norwell, MA, USA, 2001. Kluwer Academic Publishers.

[30] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003.

[31] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[32] J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312, London, UK, 2001. Springer-Verlag.

[33] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[34] W. A. Wulf, R. L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 390, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.