UNIVERSITÀ DEGLI STUDI DI TORINO

Dipartimento di Informatica

Corso di Laurea in Informatica

Tesi di Laurea in Informatica

Logica dinamica per la verifica di programmi in Java

Relatore:
Prof. Ugo de'Liguoro

Candidato:
Jacopo Tancredi

a.a. 2016/2017

"Testing shows the presence, not the absence of bugs."

Edsger W. Dijkstra.

Ringraziamenti

Ringrazio il Prof. Ugo de'Liguoro per l'aiuto e il supporto datomi durante lo svolgimento dello stage. Inoltre ringrazio la mia famiglia per avermi dato la possiblità di intraprendere questo percorso. Infine un ringraziamento particolare va a tutti i miei amici che mi hanno aiutato a superare le difficoltà incontrate e, cosa più importante, mi hanno sopportato durante questi anni.

Indice

Listings

Glossario

DL	Dynamic Logic. 7
JavaDL	Java Dynamic Logic. 7
JFOL	Java First-Order Logic. 2
JML	Java Modelling Language. 11

Introduzione

Il crescente aumento della complessità dei sistemi e delle architetture software richiede strumenti sempre più sofisticati per poter giudicare la correttezza di un programma. In particolare la fase di testing risulta esser diventata troppo laboriosa ed espansiva, in quanto con essa non siamo in grado di mostrare la totale assenza di errori, ma solamente verificarne una probabile presenza. Per poter ovviare a queste problematiche vengono utilizzati opportuni tool di verifica che a loro volta applicano i metodi formali come mezzo per la verifica dei sistemi software e hardware.
Gli strumenti esistenti sono molteplici ed ognuno si differenzia in base alle caratteristiche del programma e del linguaggio di programmazione in cui è stato implementato.
La scelta è ricaduta su KeY poichè risulta essere, fino ad oggi, il tool più diffuso e semplice nel verificare programmi Java.
KeY non solo si occupa di verificare formalmente programmi in Java utilizzando il linguaggio di specifca JML e della logica dinamica ma supporta anche prove di correttezza semi-automatiche. Esso è stato sviluppato congiuntamente dalle Università di Karlsruhe (Germania), Darmstadt (Germania) e Gothenburg (Svezia).
Il lavoro su cui si basa questa tesi consiste nel verificare la correttezza di una personale implementazione del BubbleSort tramite il solo utilizzo di KeY in modo da non dover generare alcun test JUnit. Per ottenere tale risultato sono stati usati vari strumenti che spiegheremo più nel dettaglio nei capitoli successivi. In particolare ci soffermeremo: sulla descrizione delle logiche usate all'interno di KeY; sull'uso del Java Modelling Language per la parte riguardante la specifica delle proprietà fondamentali dell'algoritmo; sui passaggi che compie KeY per poter valutare simbolicamente l'esecuzione del programma.

Capitolo 1

Metodi logici per la verifica di programmi

1.1 Introduzione

In questo capitolo verranno trattate le logiche utilizzate da KeY per poter effettuare la verifica di programmi in Java. Le logiche presenti si suddividono in: *logica del primo ordine*, *logica di Hoare*, *logica dinamica*; sarà discussa in modo generale ogni logica, omettendo il calcolo e la verifica di correttezza e completezza.
Infine introdurremo il concetto di *updates* all'interno della logica dinamica.

1.2 Logica del primo ordine

La logica del primo ordine è una raccolta di sistemi formali usati in matematica, filosofia, linguistica e informatica che utilizza variabili quantificate su oggetti non logici, distinguendosi così dalla logica proposizionale.
La sintassi che mostreremo avrà al suo interno la definizione di tipo generico. Questo perchè KeY utilizza un'estensione della logica del primo ordine, specifica per programmi in Java, chiamata **JFOL**.

1.2.1 Sintassi

La sintassi che verrà mostrata è la base su cui si appoggia la logica dinamica per Java.

1.2.1.1 Tipo generico

Definizione 1.1. Un tipo generico è una coppia τ = (TSym, $\sqsubseteq$), dove

- TSym è un insieme di simboli.
- $\sqsubseteq$ è una relazione riflessiva, transitiva su TSym, chiamata relazione di sottotipo.
- Ci sono due diversi simboli, **vuoto** $\perp \in$ TSym e quello **universale** T $\in$ TSym con $\perp \sqsubseteq A \sqsubseteq T$ per tutti gli A $\in$ TSym.

Inoltre due tipi A, B in τ vengono chiamati **incomparabili** se nè A $\sqsubseteq$ B e nè B $\sqsubseteq$ A.
Abbiamo introdotto la nozione di *tipo generico* perchè si vuole estendere la normale logica del primo ordine al linguaggio di programmazione Java.

1.2.1.2 Signature

Definizione 1.2. Una **signature** $\sum$ = (FSym, PSym, VSym) per un dato tipo generico τ è composta da:

- un'insieme FSym funzioni, f:A_1x...xA_n $\rightarrow$A
- un'insieme PSym di predicati, p(A_1,...,A_1)
- un'insieme VSym di variabili, v : A for v $\in$ VSym

In questa definizione il tipo di ogni A deve essere diverso da $\perp$. Una funzione con arietà 0, c:$\rightarrow$A, è chiamata **costante**. Un predicato p() con arietà 0 viene chiamato **variabile proposizione** oppure **atomo proposizionale**. Gli stessi simboli, con tipi diversi, possono non essere contenuti in FSym $\bigcup$ PSym $\bigcup$ VSym.
Con le prossime due definizioni si precisano i termini e le formule della logica del primo ordine.

1.2.1.3 Termini

Definizione 1.3. Sia τ un tipo generico, e $\sum$ una signature per τ. L'insieme Trm_A di termini di tipo A, per A $\neq\perp$, è definito da:

1. v $\in Trm_A$ per ogni variabile v : A $\in$ VSym di A.
2. f(t_1,...,t_n) $\in Trm_A$ per ogni f : A_1x...xA_n $\rightarrow$A $\in$ FSym e tutti i termini $t_i \in Trm_B i$ con $B_i \sqsubseteq A_i$ per $1 \leq$ i $\leq$ n.
3. (if ϕ then t1 else t2) $\in Trm_A$ per $\phi \in$ Fml e $ti \in Trm_A i$ tale che $A_2 \sqsubseteq A_1 =$ A o $A_1 \sqsubseteq A_2 =$ A

I Termini delle forma definiti in (3) sono chiamati termini condizionali. Per ogni formula contentente i termini condizionali ve n'è una equivalente priva di essi.

1.3 Logica di Hoare

La logica di Hoare è un sistema formale che rientra tra le semantiche assiomatiche. Definendo un'insieme iniziale di assiomi e le regole che agiscono su di essi, essa si prefigge di valutare la correttezza di programmi utilizzando il rigore dei formalismi matematici.

1.3.1 Tripla di Hoare

Una tripla di Hoare descrive il modo in cui l'esecuzione di un pezzo di codice possa cambiare lo stato della computazione. Una tripla di Hoare viene definita:

$$\{A\}C\{B\}$$

Dove A e B sono **asserzioni** e C è un **comando**.
Più formalmente si può affermare che per ogni stato σ che soddisfi A, se l'esecuzione di C dallo stato σ termina nello stato σ', allora σ' soddisfa B.

1.3.2 Asserzioni di correttezza parziale

Le triple della forma $\{A\}C\{B\}$ vengono chiamate **asserzioni di correttezza parziale**. A è chiamata **PreCondizione** e B **PostCondizione**. La tripla viene chiamata in questo modo perchè non dice nulla sulla terminazione del comando C.
Difatti la tripla

$$\{true\}C\{false\}$$

con

$$\text{C} \equiv \text{while true do skip}$$

viene considerata valida anche se l'esecuzione di C, partendo da qualsiasi stato, non termina. Per cui diremo che se nel comando C vi sarà presente un ciclo, allora qualsiasi asserzione di correttezza parziale della forma $\{A\}C\{B\}$ è valida.
Possiamo descrivere il significato di $\{A\}C\{B\}$ con:

$$\forall\sigma \in \sum(\sigma \models A \rightarrow C[\![c]\!]\sigma \models B)$$

Concludiamo accennando alle **asserzioni di correttezza totale**,

$$[A]C[B]$$

in cui viene richiesto che l'esecuzione del comando, partendo da qualsiasi stato che soddisfi A, **termini** in uno stato che soddisfi B.

1.3.2.1 Regole di Hoare

La logica di Hoare viene definita come un insieme di regole di inferenza per derivare asserzioni di correttezza valide. Esse sono:

Skip

$$\{A\}skip\{A\}$$

Assignament

$$\{B[a/x]\}X := a\{B\}$$

Sequencing

$$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}}$$

Conditionals

$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A\neg b\}c_1\{B\}}{\{A\}\ if\ b\ then\ c_0\ else\ c_1\{B\}}$$

While Loops

$$\frac{\{A \wedge B\}c\{A\}}{\{A\}\ while\ b\ do\ c\{A \wedge \neg b\}}$$

Consequence

$$\frac{\models (A \to A') \quad \{A'\}c\{B'\} \quad \models (B' \to B)}{\{A\}c\{B\}}$$

1.3.3 KeY e la logica di Hoare

KeY utilizza la logica di Hoare estendendo la sua tripla ad una quadrupla, inserendo un **update** prima di un qualsiasi comando. La nuova quadrupla di Hoare assume la seguente forma:

$$\{A\}[U]C\{B\}$$

In cui se S è un qualunque stato che soddisfi la PreCondizione e C inizia in S_U, allora se C termina finiremo in uno stato finale che soddisfi la PostCondizione B. La tripla di Hoare con l'aggiunta degli updates è esattamente equivalente ad una formula della logica dinamica.

$$A \rightarrow \{U\}[C]B$$

1.4 Logica dinamica

La logica dinamica (DL) è un'estensione della **logica modale** originariamente definita per il ragionamento di programmi e in seguito applicata a compiti più generali e complessi derivati dalla linguistica, dalla filosofia, dall'intelligenza artificiale e da altri campi. In KeY verrà usata un'istanziazione della logica dinamica, chiamata **JavaDL**.
Il principio della DL è la formulazione di asserzioni sul comportamento del programma utilizzando a sua volta programmi e formule. A tal fine, le **modalità** $\langle p \rangle$ e [p] possono essere usate nelle formule, dove p può essere una qualsiasi sequenza di istruzioni (DL è una **logica multi-modale**).
Questi operatori si riferiscono allo stato finale di p e possono essere collocati prima di ogni formula. La formula $\langle p \rangle \phi$ esprime che il programma p termina in uno stato in cui ϕ persiste, a sua volta la formula [p]ϕ non richiede che il programma p termini ma esprime solamente il fatto che se p dovesse terminare allora ϕ persiste nello stato finale.

1.4.1 Logica dinamica e logica di Hoare

Vi è un legame tra la logica dinamica e la logica di Hoare, in cui la prima può anche essere vista come un'estensione della seconda. La formula $\phi \rightarrow [p]\psi$ è simile alla tripla di Hoare $\{\phi\}p\{\psi\}$. La differenza si trova nel tipo di formule utilizzate. In Hoare, le formule ϕ e ψ sono **formule del primo ordine**, mentre nella logica dinamica possono contenere **programmi**. Usando un programma in ϕ, per esempio, viene più semplice specificare che una struttura dati in input non è ciclica, cosa impossibile usando la logica del primo ordine.

1.4.2 Sintassi

La sintassi che mostreremo si basa sulla **JavaDL** (Java Dynamic Logic), un'estensione della logica dinamica per i programmi Java.

1.4.2.1 Signature

In JavaDL i simboli possono essere sia *rigidi* che *non rigidi*. I primi mantengono la loro interpretazione durante l'esecuzione del programma, invece i secondi possono variare la propria interpretazione.

Definizione 1.4. Sia τ un tipo generico per un programma Java Prg. Una signature τ è una tupla:

$$\sum = (\text{FSym, PSym, VSym, ProgVSym})$$

dove

- (FSym, PSym, VSym) sono signature della JFOL.
- l'insieme ProgVSym di simboli di funzione non rigidi e non nulli, che chiameremo **variabili del programma**, contengono tutte le variabili locali a dichiarate in Prg, dove il tipo della variabile a:A∈ProgVSym è dato da quello della variabile T.

  - A = T se T è un tipo di riferimento,
  - A = boolean se T = boolean,
  - A = int se T $\in \{byte, short, int, long, char\}$

- ProgVSym contiene un infinito numero di simboli per ogni tipo.
- ProgVSym contiene la variabile "speciale"

$$\text{heap:Heap} \in \text{ProgVSym}.$$

Vi è un'importante differenza tra le *varibili logiche* in VSym e le *variabili del programma* in ProgVSym: le prime possono essere universalmente o esistenzialmente quantificate ma non occorrono in nessun programma, mentre le seconde possono occorrere nei programmi ma non possono essere quantificate.

1.4.2.2 Termini e formule

I *termini* e le *formule* della JavaDL sono gli stessi della logica del primo ordine, con la differenza che le formule possono contenere gli operatori modali $\langle p \rangle$ e [p].

Definizione 1.5. Sia Prg un programma Java, τ un tipo generico per Prg e $\sum$ una signature per τ.
L'insieme $DLTrm_A$ dei termini di tipo A, per A $\neq \perp$, e l'insieme DLFml delle formule sono definiti esattamente come quelli della logica del primo ordine tranne per:

- La signature $\sum$ ora riferisce alla signature JavaDL.
- Trm_X e Fml ora sono rispettivamente $DLTrm_X$ e DLFml.
- Le seguenti quattro clausole sono un'aggiunta alla definizione delle formule:

$$\langle p \rangle \phi,\ [\text{p}]\phi \in \text{DLFml per ogni frammento di programma p.}$$

Un termine di una formula viene chiamato *rigido* se non contiene nessuna variabile del programma.

1.4.3 Updates

JavaDL estende la classica logica con un'ulteriore categoria sintattica basata sugli operatori modali in aggiunta a frammenti del programma, chiamati **updates**. Come quest'ultimi, gli updates denotano i cambiamenti di stato. La differenza tra i due consiste nel fatto che gli updates sono più semplici e meno restrittivi.
Per esempio, questi terminano sempre e le espressioni che occorrono al loro interno non hanno mai effetti collaterali.

Definizione 1.6. Sia Prg un programma Java, τ un tipo generico per Prg e $\sum$ una signature per τ. L'insieme Upd degli update è definito come:

- (a := t) $\in$ Upd per ogni variabile a : A $\in$ ProgVSym per ogni termine t $\in DLTrm'_A$, tale per cui A' $\sqsubseteq$ A.
- *skip* $\in$ Upd.
- $(u_1 \parallel u_2) \in$ Upd per ogni update u_1, $u_2 \in$ Upd.
- $(\{u_1\}\ u_2) \in$ Upd per ogni update u_1, $u_2 \in$ Upd.

Un'espressione della forma $\{u\}$, dove u $\in$ Upd, viene chiamata **update application**.
Intuitivamente, un *update elementare* a:= t assegna il valore del termine t alla variabile a. L'*update vuoto* che non produce cambiamenti viene denotato da *skip*. Un *update parallelo* $u_1 \parallel u_2$ permette di eseguire in parallelo u_1 e u_2.

Definizione 1.7 (**Termini e formule con gli updates**)**.** La definizione dei termini viene estesa come:

- $\{u\}$ t $\in DLTrm_A$ per ogni update u $\in$ Upd e tutti i termini t $\in DLTrm_A$.

La definizione per le formule viene estesa come:

- $\{u\}\phi \in$ DLFml per ogni formula $\phi \in$ DLFml e per ogni update u $\in$ Upd.

Gli update permettono di trasformare uno stato in un altro. Il significato di $\{u\}t$, dove u è un update e t è un termine/formula/update, consiste nel fatto che t è valutato in uno stato prodotto partendo da u.

Capitolo 2

Specifica in JML

2.1 Introduzione

Il *Java Modelling Language*, JML, è un linguaggio di specifica per programmi in Java basato sul paradigma della programmazione per contratti.
JML è stato progettato per la specifica di moduli. Che in Java si dividono in:

1. metodi, dove JML specifica il risultato dell'invocazione di un singolo metodo;
2. classi, dove JML specifica solo i vincoli sulla struttura interna di un oggetto;
3. interfacce, dove JML specifica il comportamento esterno di un oggetto.

Utilizzando questi moduli JML permette di specificare il comportamento dei programmi in Java e di definire una loro implementazione.
In questo capitolo definiremo la sintassi del JML e come scrivere: PreCondizioni, Post-Condizioni e Invarianti di ciclo.

2.2 Sintassi

JML permette di definire asserzioni, inserite dentro determinati commenti, all'interno del codice Java di una classe.
Abbiamo due tipologie di scritture.
- La prima corrisponde al commento linea per linea:

//@ statement JML
//@ statement JML

- La seconda corrisponde al commento multilinea:

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

Ogni statement JML è formato da:

- una clausola
- un'asserzione oppure una parola chiave, preceduta da un blackslash(\)

Mostriamo, di seguito, un esempio per facilitare la compresione

```
1  /*@
2     @ requires a.length > 0;
3     @ ensures (\forall int x; 0 <= x && x < a.length;
4     @          (\forall int y; 0 <= y && y < a.length;
5     @          x < y ==> a[x] <= a[y]));
6   @*/
```

Listing 2.1: Esempio codice JML

Come si può notare troviamo **requires** ed **ensures**, l'asserzione ***a*.length > 0** e la parola chiave **\forall**.
Infine JML aggiunge dei nuovi operatori rispetto a quelli già presenti in Java, che consistono in:

- a ==> b, a implica b
- a <== b, 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 un sottotipo di b (l'operatore è anche riflessivo)

2.3 Quantificatori

JML offre la possiblità di definire diversi tipi di quantificatori all'interno delle asserzioni. Per fare ciò, la sintassi usata è la seguente:

(\quantificatore tipo nome_variabile; [range]; corpo)

I quantificatori si differenziano in:

- **\forall**, universale
- **\exist**, esistenziale
- **\max**, **\min**, **\product**, generalizzati
- **\num_of**, numerico

Dopo aver scelto il tipo di quantificatore da utilizzare, bisogna definire: il tipo ed il nome della variabile da quantificare, importante precisare che si può definire più di una singola variabile; il range, che può anche essere omesso, che consiste in una o più espressioni che

vincolano l'uso delle variabili; ed infine il corpo che è un'espressione booleana in grado di soddisfare i valori delle variabili nel range definito.
Mostriamo un esempio utilizzando il quantificatore \forall:

```
1    (\forall int k; 0 <= k && k < j; a[k] <= a[j]);
```

Listing 2.2: Esempio quantificatore forall

Questa espressione descrive l'ordinamento di un vettore ad un certo indice *j*, cioè tutti gli elementi in (a[o], a[j-1]) dovranno essere minori dell'elemento in posizione a[j]. Il suo significato letterale sarà: "Per ogni intero k compreso tra 0 e j-1, implica che a[k] deve essere minore uguale ad a[j].

2.4 PreCondizioni

Una PreCondizione è una condizione che deve essere vera immediatamente prima dell'invocazione di un metodo. Si possono avere più precondizioni in base alla difficoltà di specifica del metodo.
Per scrivere una PreCondizione in JML si usa la clausola **requires**, con la seguente sintassi:

\requires pred;

dove *pred* esprime qualsiasi predicato. Nel caso in cui avessimo diversi predicati, gli uni diversi dagli altri, possiamo scriverli in due diversi modi però equivalenti tra loro.

\requires P; equivale a \requires P && Q;
\requires Q;

Nella figura 2.1 alla linea 2 troviamo un esempio di PreCondizione associata ad un metodo.

2.5 PostCondizioni

Una PostCondizione è una condizione che deve essere vera subito dopo l'invocazione di un metodo, senza che esso lanci un'eccezione. Si utilizza la clausola **\ensures** per poter scrivere una PostCondizione in JML:

\ensures pred;

Se una PostCondizione avesse più di un predicato, si utilizza lo stesso metodo spiegato in precedenza. Anche per le postcondizioni possiamo averne più di una, sempre in base alla difficoltà di specifica del metodo.
Si rimanda alle linee 3-5 della figura 2.1 per avere un esempio di come si possono scrivere delle postcondizioni.

2.6 Invarianti

Uno degli elementi di specifica più importanti e ampiamente utilizzati nella programmazione ad oggetti sono gli invarianti [1], chiamati anche invarianti di classe o interfaccia, a seconda di dove vengono definiti. Un invariante è un'espressione booleana (JML) definita sullo stato dell'oggetto e può essere visto come una condizione per vincolare un'istanza ad uno stato.
In JML viene definito usando la clausola **invariant**, nel seguento modo:

invariant bool_exp

Dove *bool_exp* è un'espressione booleana che deve essere vera.
Nella figura 2.3 viene mostrato un esempio, molto semplice, dell'utilizzo di un invariante. Sarebbe stato anche valido aggiungere queste specifiche alle precondizioni. Tuttavia ogni volta che aggiungiamo un metodo alla classe dovremmo riscrivere, per ognuno di essi, le stesse precondizioni. Questo viene evitato utilizzando l'invariante di classe o interfaccia; nel primo caso viene ereditato da tutte le sottoclassi, nel secondo tramite l'implementazione dell'interfaccia erediteremo anche lo stesso invariante.

[1] Da non confondere con gli invarianti di ciclo.

```
1    /*@ public invariant (\forall int i; 0 < i && i < size;
2      @                              a[i-1] <= a[i]);
3      @*/
```

Listing 2.3: Esempio invariante di classe

2.7 Invarianti di ciclo

Un invariante di ciclo è una proprietà che deve essere vera all'inizio, durante e alla terminazione di un ciclo. In JML, un invariante di ciclo è composto da:

1. **loop_invariant** bool_exp
2. **assignable** var
3. **decreasing** expr

Con (1) specifichiamo l'invariante del ciclo attraverso un'espressione booleana. Questo deve essere posto prima del ciclo interessato. La clausola **maintaining** assume lo stesso significato e struttura di loop_invariant.
Con (2) specifichiamo tutte le posizioni il cui valore può venire modificato durante l'esecuzione del ciclo. Viene utilizzato maggiormente in presenza di array, difatti se per esempio il valore di ogni singolo elemento di un array a viene modificato scriveremo assignable $a[*]$.
Con (3) specifichiamo la condizione di terminazione del ciclo mediante l'utilizzo di un'operazione di sottrazione tra l'indice e la variabile contente il numero totale di passi. È necessario che il valore dell'espressione sia sempre positivo e si decrementi ad ogni passo del ciclo.
Le specifiche dell'invariante devono essere scritte prima della dichiarazione del ciclo.

La figura 2.4 mostra l'invariante del ciclo for definito alle linee 8-10.

```
1         /*@ loop_invariant 0 <= i && i <= n;
2           @ loop_invariant (\forall int j; 0 <= j && j < i;
3           @                         b[n-j-1] == a[j]);
4           @ assignable b[*];
5           @ decreases n - i;
6           @
7           @*/
8             for(int i = 0; i != n; i++) {
9                         b[n-i-1] = a[i];
10                }
```

LISTING 2.4: Esempio invariante di ciclo

Capitolo 3

KeY

3.1 Introduzione

In questo capitolo parleremo di come KeY valuti un programma Java attraverso l'*esecuzione simbolica*. Sarà scomposto in due parti: nella prima mostreremo un metodo con una semplice condizione, ma sprovvisto di cicli, ed esamineremo il processo di valutazione; nella seconda spiegheremo come vengono trattati i cicli durante una verifica.
Infine verrà trattato il concetto di *taclet*, della sua composizione, dichiarazione e dell'uso delle *schema variables* al suo interno.

3.2 Esecuzione simbolica

L'esecuzione simbolica esegue un programma usando valori simbolici al posto di valori concreti. Per spiegare meglio questo concetto usiamo come esempio il seguente programma Java:

```
1    public static int main(int x, int y){
2        if(x < y){
3            return x;
4        }
5        else{
6            return y;
7        }
8    }
```

Listing 3.1: Minimo tra due interi

Questo semplice programma prende in input due variabili intere, le confronta e restituisce la variabile con il valore minore.
Quando viene richiamato il metodo *min*, vengono assegnati dei valori simboli x e y alle due variabili corrispondenti. Fintanto che non si sa nulla della relazione tra x e y non si potrà scegliere quale ramo del **if** intraprendere; per questo motivo l'esecuzione simbolica dividerà in due branch l'esecuzione creando così un **albero simbolico di esecuzione**. Un ramo continuerà l'esecuzione nel caso in cui $x < y$ sia soddisfatta, mentre l'altro ramo espanderà il caso *!(x < y)*. Lo stato padre dei due rami viene chiamato *path condition* e definisce un vincolo sui valori delle variabili in modo da assicurare di intraprendere il giusto percorso. Inoltre viene utilizzata la conoscenza acquisita nelle successive fasi dell'esecuzione simbolica, in modo da poter potare percorsi non attuabili; difatti se si invocasse il metodo *min* per una seconda volta con gli stessi valori simbolici per x e y non si dovrà dividre in due il risultato della condizione padre.

Possiamo affermare che l'esecuzione simbolica scopre tutti i possibili percorsi di esecuzione ed ognuno di essi può rappresentare infinite rappresentazioni concrete.
Di seguito viene mostrato l'albero di esecuzione per il metodo *min*.

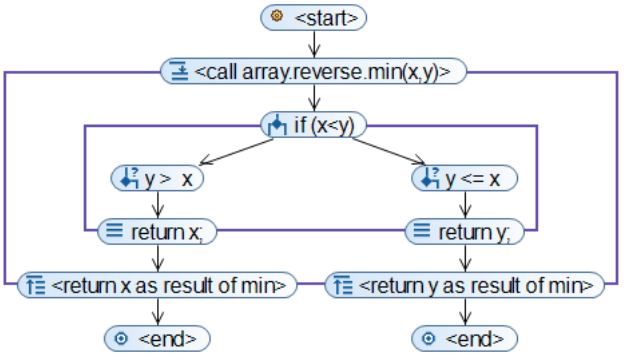

FIGURA 3.1: Esecuzione Simbolica metodo min

3.3 Esecuzione simbolica degli invarianti di ciclo

Tutto risulta molto più complesso se all'interno di un programma introduciamo un ciclo (assumeremo che si tratti di un ciclo *while*). Questo perchè l'esecuzione simbolica ora cercherà di srotolare il ciclo e verificarlo ad ogni passo, per poterlo fare userà la seguente formula della JavaDL:

$$\text{loopUnwind}\ \frac{\Longrightarrow \langle \pi \text{ if (e) \{ p while (e) p \}} \omega \rangle \phi}{\Longrightarrow \langle \pi \text{ while (e) p } \omega \rangle \phi}$$

Se la guardia del ciclo viene valutata come vera nel corrente stato simbolico, allora il corpo del ciclo p viene eseguito una volta ed in seguito il puntatore del programma viene riportato, ancora una volta, all'inizio del ciclo. Altrimenti l'esecuzione simbolica prosegue con il codice ω.
Questo però funziona solamente quando il numero di iterazioni del ciclo è limitato a piccole costanti. Una guardia valutabile potrebbe essere espressa come $i \quad a$.length, dove a è un

array arbitrario di una lunghezza sconosciuta.
Per ragionare su loop infiniti oppure su quelli il cui corpo viene eseguito molto spesso (ad esempio, $0 \leq$ i && i <Integer.MAX_VALUE), è necessario un qualche tipo di principio di induzione che permetta di dimostrare le proprietà di strutture illimitate in un modo finito.
Nel precedente capitolo abbiamo parlato degli *invarianti di ciclo* nel JML, ora vedremo come vengono trattati all'interno di KeY.
In KeY un invariante di ciclo è una formula $inv \in$ DLFml che si mantiene all'inizio dello stato iniziale del ciclo e si mantiene nello stato successivo ad ogni esecuzione del corpo del ciclo. Se il ciclo dovesse terminare significa che l'invariante si mantiene anche nello stato del programma successivo ad esso.
Di conseguenza, se riuscissimo a dimostrare che una formula *inv* è un invariante di ciclo, allora potremmo utilizzarla durante l'esecuzione simbolica del codice che segue il nostro ciclo. Grazie agli invarianti di ciclo ora possiamo iniziare anche a ragionare anche su quei programmi che contengono cicli illimitati.
Mostriamo la regola per trattare gli invarianti di ciclo all'interno di KeY, per semplicità assumiamo che il programma all'interno della guarda del ciclo e del corpo non abbiamo accesso all'heap.

$$\text{loopInvariant1}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}inv, \Delta \text{ (initially valid)} \\ \text{inv,g = TRUE} \Longrightarrow \text{[p]inv (preserved by body)} \\ \text{inv,g = FALSE} \Longrightarrow [\pi\omega]\phi \text{ (use case)}\end{array}}{\Gamma \Longrightarrow \{u\}[\pi \text{ while(g) p; } \omega]\phi, \Delta}$$

La prima premessa afferma che *inv* si mantiene all'inizio del ciclo, la seconda che se *inv* si mantiene in uno stato che valuta vera la guardia del ciclo allora anche nello stato finale dell'esecuzione simbolica del corpo del ciclo verrà mantenuto, provando la terminazione. Infine la terza premessa permette di usare l'invariante, con l'aggiunta della guardia negata, per provare la correttezza della continuazione.
La regola viene verificata attraverso i seguenti passi:

- *Ipotesi Induttiva*: per ogni $n \geq 1$ l'invariante *inv* è valido nello stato all'inizio della n-esima esecuzione del corpo del ciclo.

- *Passo Base*: L'invariante *inv* è valido nello stato iniziale della prima esecuzione del ciclo, esattamente nello stato in cui l'esecuzione simbolica del ciclo inizia. Questo è esattamente ciò che viene detto nella prima premessa.
- *Passo induttivo*: Se *inv* è valido nello stato iniziale dell'n-esima esecuzione, e se il ciclo viene eseguito almeno una volta, allora *inv* è valido anche nello stato iniziale della $n + 1$ esecuzione del ciclo.

KeY svolge questi passi generando tre diversi branch all'interno dell'albero di prova:

- Invariant initially valid
- Body preserved invariant
- Use case

3.4 Regole come taclet

I *taclet* furono concepiti sotto il nome di "Regole specifiche della teoria schematica", il cui intento iniziale era catturare assiomi e specifiche algebriche sotto forma di regole. Il loro compito consiste nel descrivere schematicamente un insieme di regole del calcolo dei sequenti.
Vengono usati in KeY per molti scopi, tra cui:

- la definizione delle regole per il calcolo della logica del primo ordine;
- per la definizione delle regole per il calcolo della JavaDL;
- per introdurre tipi di dati e procedure decisionali;
- per poter dare all'utente la possibilità di definire nuove regole.

Per *calcolo* di un logica viene intesa una collezione di operazioni sintattiche che permettono di definire se una formula sia valida o meno.

3.4.1 Sezione del taclet

$\langle taclets \rangle$:: = \rules {($\langle taclets \rangle$)∗}
| \axioms {($\langle taclets \rangle$|$\langle axiom \rangle$)∗}

Esattamente come i file di input in KeY sono divisi in diverse sezioni che definiscono varie parti del linguaggio sintattico che può essere usato (funzioni, predicati, tipi, ecc...), anche i taclet sono divisi in due sezioni: regole e assiomi.
Un taclet può essere composto da una singola regola, da più regole, da un solo assioma, da più assiomi oppure sia da regole che da assiomi.

3.4.2 Dichiarazione di un taclet

$\langle taclets \rangle$:: =
$\langle identifier \rangle$ {
$\langle localSchemaVarDecl \rangle$*
$\langle contexttAssumptions \rangle$? $\langle findPattern \rangle$?
$\langle applicationRestriction \rangle$? $\langle variableConditions \rangle$?
($\langle goalTemplateList \rangle$ | \closegoal)
$\langle ruleSetMemberships \rangle$?
}

Ogni taclet ha un nome univoco e un corpo contenente elementi che descrivono come deve essere abbinato il taclet ad un sequente, seguito da una descrizione di quale azione avverrà. Inoltre il sistema non accetta definizioni che non rispecchiano l'ordine, precedentemente definito, degli elementi.

3.4.3 Schema Variables

Le *schema variables* sono dei "segnaposto" per diversi tipi di entità sintattiche che possono essere usate in un taclet. Nonostante contengano al loro interno il nome "variabile",

hanno un grande utilizzo all'interno di KeY; difatti possono contenere al loro interno diverse istanze, come variabili (logiche o di programma), termini, formule, programmi o concetti più astratti come gli operatori modali.
Sono usate prevalentemente nella definizione di un taclet. Quando un taclet viene applicato, il contenuto delle schema variables viene rimpiazzato da un'entità sintattica concreta. Questo processo viene chiamato *istanziazione* ed assicurano che lo schema variables non occorra mai nei sequenti della prova. Alcune schema variables sono instanziate tramite l'accoppiamento di espressioni schematiche con espressioni concrete sul sequente del goal, invece altre istanziazioni avvengono durante l'applicazione del taclet (attraverso un'iterazione dell'utente oppure come consequenza di una strategia automatica).

Mostriamo alcuni tipi di schema variables nel contesto di un tipo generico (TSym, $\sqsubseteq$)

- \variable A variabile logica di tipo A $\in$ TSym
- \term A termine di tipo B $\sqsubseteq A(conA\in$ TSym)
- \formula una formula
- \skolemTerm A una costante/funzione di Skolem di tipo A $\in$ TSym
- \program t un'entità programma di tipo t

Capitolo 4

Caso di studio: Bubble Sort

4.1 Introduzione

In questo capitolo verrà mostrata la verifica della correttezza del Bubble Sort.
Il Bubble Sort è un algoritmo di ordinamento iterativo basato sui confronti, deve il suo nome al modo in cui vengono ordinati gli elementi.
Per poter verificare l'algoritmo tramite KeY, è stata utilizzata una sua implementazione in Java, come linguaggio di programmazione, e JML.

4.2 L'algoritmo

Per l'implementazione del Bubble Sort è stato utilizzato l'array come struttura dati. Questo viene scorso dal fondo verso l'inizio controllando gli elementi a due a due e, nel caso in cui questi siano disordinati, li scambia. La figura 4.1 mostra infatti la sua implementazione. L'algoritmo ha una complessità computazionale O(n^2), con n indicante gli elementi del vettore a.

```
1  void sort(int[] a) {
2      for (int n = a.length; 0 <= --n;) {
3                  for (int j = 0; j < n; j++) {
4                          if (a[j+1] < a[j]) {
5                                  int tmp = a[j];
6                                  a[j] = a[j+1];
7                                  a[j+1] = tmp;
8                          }
9                  }
10         }
11 }
```

Listing 4.1: Codifica Java BubbleSort

Il funzionamento dell'algoritmo è il seguente: il for più esterno compie un ciclo dall'ultima posizione dell'array, ne consegue che ad un generico passo i in ($a[i]$, $a[a.\text{length-}1]$) gli elementi saranno ordinati, invece in ($a[0]$, $a[i\text{-}1]$) si troveranno ancora quelli da ordinare; il secondo for (linee 3-8) compie un ciclo da 0 fino all'iteratore n del primo ciclo, dove si confrontano a due a due gli elementi. Se l'elemento in $a[j]$ è maggiore di quello in $a[j+1]$ si scambiano tra loro, si procede in questo modo fino alla conclusione del ciclo.

4.3 Specifiche JML

Per poter passare alla verifica del Bubble Sort, è necessario specificare quattro componenti utilizzando il JML:

- PreCondizioni
- PostCondizioni
- Invariante del primo ciclo
- Invariante del secondo ciclo

4.3.1 PreCondizioni e PostCondizioni

```
1 /*@ public normal_behavior
2     @ requires a.length > 0;
3     @ ensures (\forall int x; 0 <= x && x < a.length;
4     @          (\forall int y; 0 <= y && y < a.length;
5     @          x < y ==> a[x] <= a[y]));
6     @ diverges true;
7     @*/
8         void sort(int[] a) {
```

Listing 4.2: Codifica Java+JML PreCondizioni e PostCondizioni

In questo caso abbiamo un'unica PreCondizione (linea 2) in cui viene specificata che la lunghezza iniziale dell'array deve essere maggiore di zero, quindi questo deve contenere al minimo un elemento.
La PostCondizione (linee 3-6) specifica l'ordinamento dell'array a alla terminazione del metodo che in questo caso deve essere crescente. Questo tipo di ordinamento prevede che per un qualsiasi elemento in posizione i, esso sia sempre minore o uguale ad un altro elemento in posizione j.
La clausola **diverges true** specifica che verrà eseguita una verifica parziale della correttezza dell'algoritmo, omettendola il sistema esegue una verifica della correttezza totale per default.

4.3.2 Primo Invariante

```
1 /*@ loop_invariant 0 <= n && n <= a.length;
2   @ loop_invariant (\forall int i, k;
3                     n <= i && i < a.length && 0 <= k && k < i;
4                     a[k] <= a[i]);
5   @ assignable a[*];
6   @ decreases a.length - n;
7   @*/
8  for (int n = a.length; 0 <= --n;) {
```

Listing 4.3: Codifica Java+JML primo invariante

L'invariante del ciclo specifica che alla fine di ogni iterazione avremo nell'ultima posizione l'elemento più grande dell'array. Quindi, ad un passo generico, gli elementi in a[0,...,n-1] dovranno essere ancora da ordinare e dovranno essere minori o uguali di quelli in a[n,...,a.length-1], che saranno a loro volta già ordinati.
Dividendo l'invariante si ottengono due formule: la prima, (linea 1), permette di definire il range di n; la seconda, invece, consente di specificare quanto detto in precedenza.
Infine, per permettere al sistema di provare la terminazione del ciclo, viene inserito il termine a.length-n dentro la clausola **decrease**.

4.3.3 Secondo Invariante

```
1   /*@ loop_invariant  0 <= j && j <= n;
2     @ loop_invariant (\forall int i, k;
3                       n < i && i < a.length && 0 <= k && k < i;
4                       a[k] <= a[i]);
5     @ loop_invariant (\forall int k; 0 <= k && k < j; a[k] <= a[j]);
6     @ assignable a[*];
7     @ decreases n - j;
8     @*/
9     for (int j = 0; j < n; j++) {
10            if (a[j+1] < a[j]) {
11                    int tmp = a[j];
12                    a[j] = a[j+1];
13                    a[j+1] = tmp;
14            }
15    }
```

Listing 4.4: Codifica Java+JML secondo invariante

Il secondo invariante specifica che al passo j-esimo ogni elemento in posizione a[0,...,j-1] deve essere minore uguale all'elemento in posizione $a[j]$.
A differenza di quello precedente, questo verrà diviso in tre formule: la prima (linea 1) definisce il range di appartenenza di j; la seconda (linee 2-4) corrisponde alla seconda formula del primo invariante. È necessario, però, che debba essere nuovamente riscritta perchè senza di essa non si avrebbe un invariante consistente, ed infine la terza (linea 5) consiste nella specifica formale della definizione dell'invariante data inizialmente.
Le linee 5-6 svolgono esattamente la stessa funzione di quelle nel ciclo precedente, con la sola differenza della modifica del termine nella clausola **decreases** che in questo caso corrisponde a n-j.

```
1  public class BubbleSort {
2    /*@ public normal_behavior
3      @ requires a.length > 0;
4      @ ensures (\forall int x; 0 <= x && x < a.length;
5      @        (\forall int y; 0 <= y && y < a.length;
6      @          x < y ==> a[x] <= a[y]));
7      @ diverges true;
8      @*/
9          void sort(int[] a) {
10                 /*@ loop_invariant 0 <= n && n <= a.length;
11         @ loop_invariant (\forall int i, k;
12                             n <= i && i < a.length && 0 <= k
13                             && k < i;
14                             a[k] <= a[i]);
15         @ assignable a[*];
16         @ decreases a.length - n;
17         @*/
18                 for (int n = a.length; 0 <= --n;) {
19                         /*@ loop_invariant  0 <= j && j <= n;
20             @ loop_invariant (\forall int i, k;
21                                 n < i && i < a.length && 0 <= k
22                                 && k < i;
23                                 a[k] <= a[i]);
24             @ loop_invariant (\forall int k; 0 <= k && k < j;
25                                 a[k] <= a[j]);
26             @ assignable a[*];
27             @ decreases n - j;
28             @*/
29                         for (int j = 0; j < n; j++) {
30                             if (a[j+1] < a[j]) {
31                                     int tmp = a[j];
32                                     a[j] = a[j+1];
33                                     a[j+1] = tmp;
34                             }
35                         }
36                 }
37         }
38 }
```

Listing 4.5: Codifica Java+JML BubbleSort

4.4 Verifica tramite KeY

L'ultimo passo consiste nel verificare l'algoritmo, precedentemente implementato, utilizzando il software di verifica KeY.
Si è provato a verificare sia la correttezza parziale sia quella totale, ottenendo così due risultati diversi. Entrambe le prove sono state eseguite su un Dell Venue 11 Pro 7140, 1.40 GHz Intel Core M-5Y71, 8GB RAM. La versione di KeY utilizzata corrisponde alla 2.6.2.

4.4.1 Correttezza parziale

In questa prova, KeY verifica l'algoritmo chiudendo automaticamente tutti i goal, senza l'interazione da parte dell'utente. I risultati ottenuti sono mostrati nella seguente tabella.

Numero Nodi	5,012
Branches	37
Interazioni Utente	0
Tempo	12.4s
Tempo Medio per Step	2.486ms

4.4.2 Correttezza totale

In questo caso KeY non riesce a verificare autonomamente la correttezza totale dell'algoritmo, si rende perciò necessaria un'interazione da parte dell'utente per poter aiutare il software a chiudere tutti i goal.
Ad ogni modo, KeY riesce a verificare in modo autonomo 33 goals su 34, il problema che viene riscontrato risiede nella sezione **Use Case** all'interno dell'albero di verifica.

```
1  wellFormed(heap),
2   self.<created>  =TRUE,
3   sort.BubbleSort::exactInstance(self)  =TRUE,
4   a.<created>  =TRUE,
5  measuredByEmpty,
6   a.length >= 1 ,
7  wellFormed(anon_heap_LOOP<<anonHeapFunction>>),
8   a.length >=  1  + j_0 ,
9    \forall int k;
10       \forall int i;
11           (    i   <= j_0
12             | i   >= a.length
13             | k   <= -1
14             | k   >= i
15             | a[k]@heap[anon(a.*, anon_heap_LOOP<<anonHeapFunction>>)]
16            <= a[i]@heap[anon(a.*, anon_heap_LOOP<<anonHeapFunction>>)]),
17  wellFormed(anon_heap_LOOP_0<<anonHeapFunction>>),
18   j_0  >= 0 ,
19   n_0  =  1  + j_0 ,
20   \forall int k;
21       \forall int i;
22           (    i   <= j_0
23             | i   >= a.length
24             | k   <= -1
25             | k   >= i
26             |        a[k]@heap[anon(a.*, anon_heap_LOOP<<anonHeapFunction>>)
27             ][anon(a.*, anon_heap_LOOP_0<<anonHeapFunction>>)]
28                <= a[i]@heap[anon(a.*, anon_heap_LOOP<<anonHeapFunction>>)
29                 ][anon(a.*, anon_heap_LOOP_0<<anonHeapFunction>>)]),
30   \forall int k;
31       (    k   <= -1
32         | k   >= j_0
33         |        a[k]@heap[anon(a.*, anon_heap_LOOP<<anonHeapFunction>>)]
34         [anon(a.*, anon_heap_LOOP_0<<anonHeapFunction>>)]
35            <= a[j_0]@anon_heap_LOOP_0<<anonHeapFunction>>)
36 ==>
37   self = null,
38   a  = null
```

Listing 4.6: Codice KeY del nodo non verificato

Conclusioni

Per concludere, possiamo confermare che KeY risulta essere un tool molto potente in grado di riuscire a verificare i programmi e le loro specifiche in modo automatico. Esso, infatti, può considerarsi un ottimo software in grado di aiutare i programmatori in modo da: evitare la scrittura di innumerevoli test che a loro volta potrebbero non bastare a soddisfare la correttezza del programma; rilevare eventuali errori dovuti alla sintassi oppure ad una scorretta implementazione a fronte delle specifiche richieste; per semplificare il debugging.
Tuttavia, abbiamo potuto constatare che non sempre KeY riesce a verificare la correttezza di un programma in modo automatico. Difatti, come mostrato in precedenza, viene richiesta un'interazione da parte dell'utente per completare la verifica dell'algoritmo.
La causa principale risiede nella difficoltà di verificare se un ciclo all'interno dell'algoritmo termini, in quanto la scelta di quali variabili istanziare e di quali regole logiche applicare risulta non essere banale.

Bibliografia

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, Mattias Ulbrich. *The KeY Book: Deductive Software Verication in Practice*. Springer, 2016, ISBN: 978-3-319-49811-9.

[2] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Huahnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, Mattias Ulbrich. *The KeY Platform for Verication and Analysis of Java Programs*. Springer, 2014.

[3] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, Peter H. Schmitt. *Verifying Object-Oriented Programs with KeY: A Tutorial*. Springer, 2007.

[4] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bube, Reiner Hähnle. *OpenJDKs java.utils.Collection.sort() is broken: The good, the bad and the worst case*. 2015.

[5] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[6] Gary T. Leavens, Yoonsik Cheon. *Design by Contract with JML*. 2006.