SAPIENZA – Università di Roma

Tesina del corso di

Metodi Formali nell'Ingegneria del Software

a.a. 2007-2008

Progetto CASE:

Rappresentazione delle specifiche concettuali e delle specifiche realizzative

e

Integrazione dei tool Daikon e Jass per l'annotazione automatica del codice

Autori:

Cecchetti Terzilio

Draghi Giovanni

Indice

Capitolo 1

Introduzione

1 Il progetto

Il lavoro svolto consiste in un'estensione del progetto ReDiA-VeriFInt (Realizzatore Diagrammi Automatico con Verifica Formale Integrata) degli studenti Fabio D'Aprano e Claudio Di Ciccio, realizzato nell'ambito del corso di Metodi Formali nell'Ingegneria del Software, tenuto dal Professor Toni Mancini.

ReDiA-VeriFInt è un progetto pilota di CASE, un IDE che permette di assistere il lavoro di analisti, progettisti e programmatori nello sviluppo di applicazioni, tramite la generazione e la gestione di diagrammi UML e codice lungo tutte le fasi, e con annessa la verifica automatica di opportune proprietà formali, sulla base di quanto realizzato.

In particolare l'estensione realizzata si concentra sul concetto di operazione, attraverso i seguenti punti:

1. modellazione di classi UML per la rappresentazione delle specifiche concettuali delle operazioni (precondizioni, postcondizioni, parametri formali, tipo di ritorno);

2. sviluppo di classi UML (e Java) per la rappresentazione delle specifiche realizzative delle operazioni (segnatura Java + precondizioni + implementazione Java);

3. annotazione automatica del codice Java(con annotazioni in JML, o in altri formati) per permettere la verifica di precondizioni e postcondizioni, invarianti ed altre proprietà;

4. scrittura automatica di codice Java per permettere la gestione di precondizioni e postcondizioni.

Capitolo 2

ReDiA-VeriFInt

2.1 Introduzione

2.1.1 Architettura iniziale

Il sistema ReDiA-VeriFInt proposto dagli studenti D'Aprano e Di Ciccio in [DapDiC08] è stato sviluppato rispettando il paradigma architetturale MVC (**M**odel **V**iew **C**ontroller): il lavoro da noi svolto si è concentrato in particolare nel layer Model dove, nel package
it.uniroma1.dis.mfis.redia.verfint.model.analysis.classDiagram,
era già presente la realizzazione del diagramma delle classi concettuale.

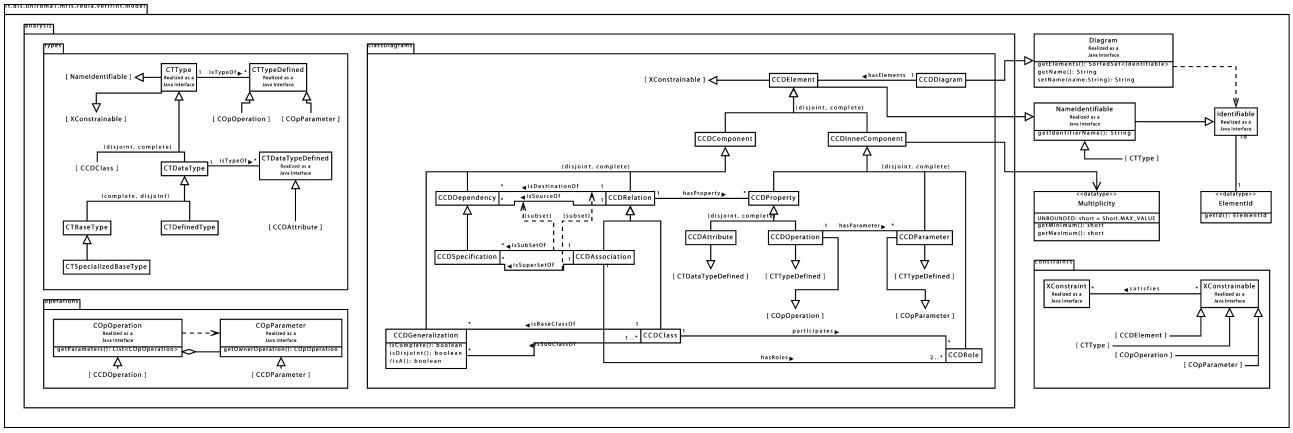


Figura 2.1 il package it.uniroma1.dis.mfis.redia.verifint.model

Il punto di aggancio per l'estensione realizzata è costituito proprio da questo diagramma nel quale in particolare è presente il concetto di Operazione (classe **CCDOperation**), oggetto principale del nostro lavoro.
Per un'esauriente comprensione del diagramma rimandiamo a [DApDiC08][ppar 2.2.2– 2.2.7].

2.1.2 Studi preliminari

Come già riportato nel paragrafo 2.2.1 di [DApDiC08], prima dell'avvio del progetto ReDiA VeriFInt, Michele Proni, studente di Ingegneria Gestionale de La Sapienza, in [Pro07], aveva operato un'analisi approfondita della modellazione dei dati necessari alla rappresentazione di varie tipologie di diagramma UML. In particolare nel corso del progetto sono state sfruttate le analisi da lui condotte, relative al Diagramma delle Classi Concettuale , alle Specifiche Concettuali, al Diagramma delle Classi

Realizzativo ed alle Specifiche Realizzative.

2.2 .Package analysis

Nel package *it.dis.uniroma1.mfis.redia.verifint.model.analysis* sono presenti tutte le entità necessarie alla rappresentazione della fase di analisi di un progetto. In particolare, di nostro interesse, si sono rivelate le classi presenti nel sotto-package *it.uniroma1.dis.mfis.redia.verifint.model.analysis.classDiagrams* (che rappresentano il Diagramma delle Classi Concettuale) e nel sotto-package *it.uniroma1.dis.mfis.redia.verifint.model.analysis.operation* (qui sono presenti alcune interfacce per la gestione delle operazioni).
Le modifiche da noi apportate in *model.analysis.classDiagram* si sono limitate all'inserimento di classi per la rappresentazione di nuove associazioni e alla modifica di classi già esistenti per tenere conto di tali nuove associazioni in cui sono coinvolte.

2.2.1 Package conceptualSpecifications

Poiché il primo obiettivo dell'estensione consisteva nel permettere la rappresentazione delle Specifiche Concettuali,è stato aggiunto in *it.dis.uniroma1.mfis.redia.verifint.model.analysis* il package "conceptualSpecifications", per contenere le classi necessarie alla loro rappresentazione.
Scopo delle Specifiche Concettuali è contenere le specifiche relative alle operazioni presenti nei vari elementi, ossia Classi, Use-Case o Tipi di Dato Definiti; tramite l'utilizzo di un linguaggio che è un ibrido tra il linguaggio naturale e quello logico, una specifica di operazione descrive quale sarà il comportamento dell'operazione e, più in dettaglio: l'insieme delle condizioni che devono valere al termine dell'esecuzione (le postcondizioni); sotto quali condizioni viene eseguita (le precondizioni); gli input di cui ha bisogno per farlo (i parametri concettuali) e il tipo di output che fornisce (il tipo di ritorno)[Pro07 p.16].

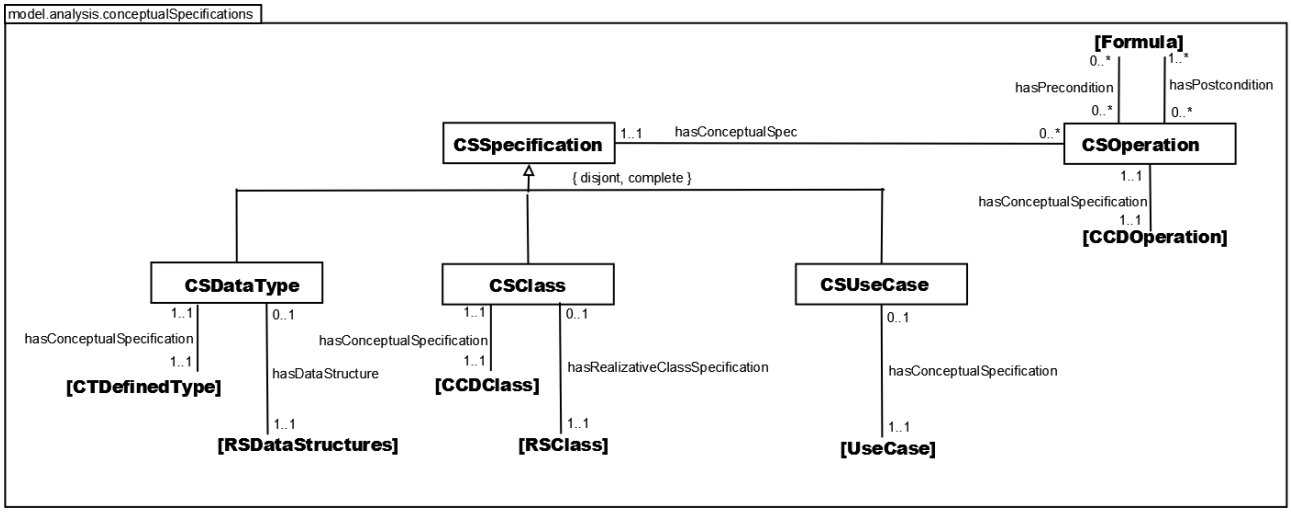


Figura 2.2: package it.uniroma1.dis.mfis.redia.verifint.model.analysis.conceptualSpecifications

Nella realizzazione di tale package si è fatto quasi del tutto riferimento alla rappresentazione delle specifiche concettuali proposta in [Pro07-par 2.1.5], salvo l'apporto di alcune modifiche, tra cui ridenominazioni orientate ad una maggiore leggibilità (ad esempio il comune prefisso **CS** che sta per Conceptual Specification). Di seguito pertanto è data una breve spiegazione delle varie componenti. Per una spiegazione più approfondita si rimanda a [Pro07 pp 25-26].
La classe astratta **CSSpecification**, rappresenta il concetto di Specifica Concettuale, che può essere, come detto in precedenza, di uno Use-case (classe **CSUseCase**), di una classe (classe **CSClass**) o di un tipo di dato (classe **CSDataType**).
Poiché lo scopo di una Specifica Concettuale è appunto quello di tenere traccia delle Specifiche di Operazione, è presente la classe **CSOperation**, associata alla Specifica Concettuale in cui si trova.
Essendo di interesse, per una Specifica di Operazione, le precondizioni e le postcondizioni cui deve sottostare, è presente una classe astratta **Formula** (collocata in *it.uniroma1.dis.mfis.redia.verifint.model.analysis.utilities*) per la rappresentazione di una generica formula in un'opportuna logica formale.
La classe **CSOperation** è inoltre collegata alla classe **CCDOperation**, del package *it.uniroma1.dis.mfis.redia.verifint.model.analysis.classDiagrams*: tramite questo collegamento è possibile risalire a concetti quali tipo di ritorno di un'operazione (**CTTypeDefined**) e parametri di un'operazione (**CCDParameter**) che sono di interesse per una Specifica di Operazione.
Inoltre le classi **CSClass** e **CSDataType** sono collegate alle relative classi di cui rappresentano la specifica concettuale (rispettivamente **CCDClass** e **CTDefinedType**).
Infine, è stata aggiunta nel *package it.uniroma1.dis.mfis.redia.verifint.model* una classe astratta **Specification** che è estesa da **CSSpecification** (vedremo in seguito (par 2.3.2) che è estesa anche da **RSSpecification**, la classe che rappresenta le Specifiche Realizzative).

2.2.2 Package utilities

In questo package sono presenti le classi **Formula** (classe astratta) e **FormulaFOL**. La prima è stata introdotta in quanto non è sembrato corretto ai fini del progetto indicare una precondizione ed una postcondizione come attributi di tipo Stringa (come era in [Pro07 fig 2.8 p26]); si è optato pertanto per la realizzazione di una Formula come entità a sè stante. In questo modo sarà agevole estendere il concetto di Formula, in base alle varie logiche che si vorranno usare, non note a priori.

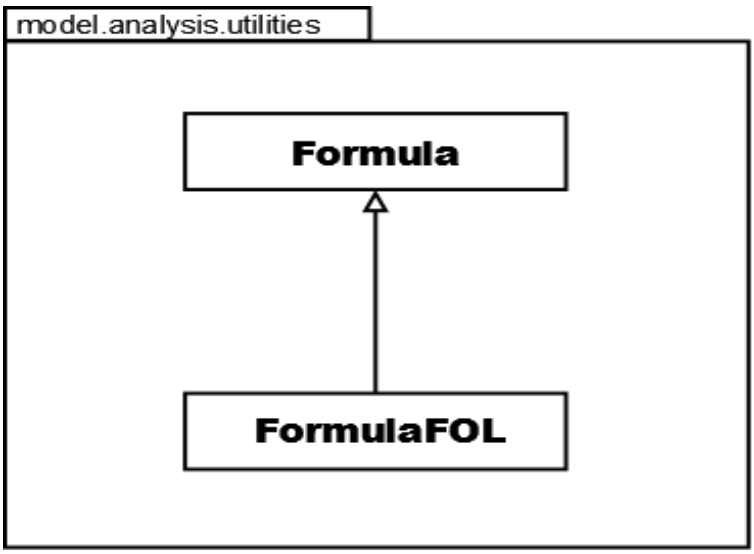


Figura 2.3 package it.uniroma1.dis.mfis.redia.verifint.model.analisys.utilities

2.3 Package realization

Secondo obiettivo del lavoro, dopo la rappresentazione delle Specifiche Concettuali, è consistito nel fornire all'utente la possibilità di scrivere codice Java per il corpo dei metodi. A questo scopo è stato realizzato nel layer "Model" il package *it.uniroma1.dis.mfis.redia.verifint.model.realization*; qui, nei sottopackage classDiagrams e realizativeSpecifications, sono presenti le classi necessarie alla rappresentazione delle Operazioni e delle loro Specifiche Realizzative.
Anche in questo caso, è stata rispettata la rappresentazione proposta in [Pro07-fig.2.14 p.33] e [Pro07-fig2.9 p.27], salvo alcune ridenominazioni; in particolare sono stati usati il prefisso **RCD** (Realizative Class Diagram) per indicare gli elementi del Diagramma delle Classi Realizzativo) ed il prefisso **RS** (Realizative Specification) per indicare gli elementi relativi alle Specifiche Realizzative.

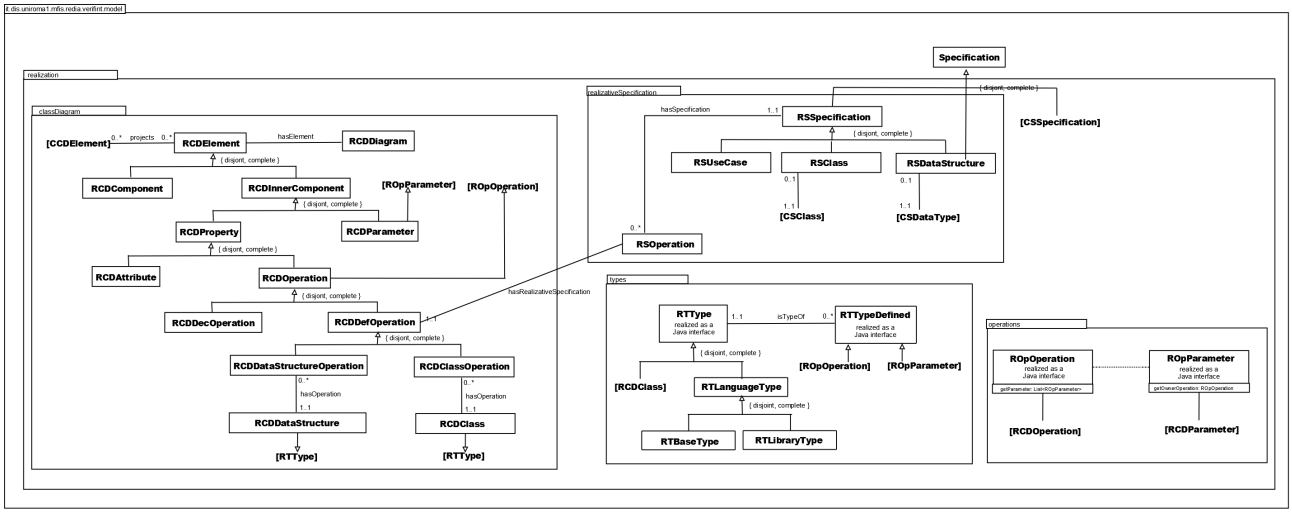


Fig. 2.4 package it.uniroma1.dis.mfis.redia.verifint.model.realization

2.3.1 Package classDiagrams

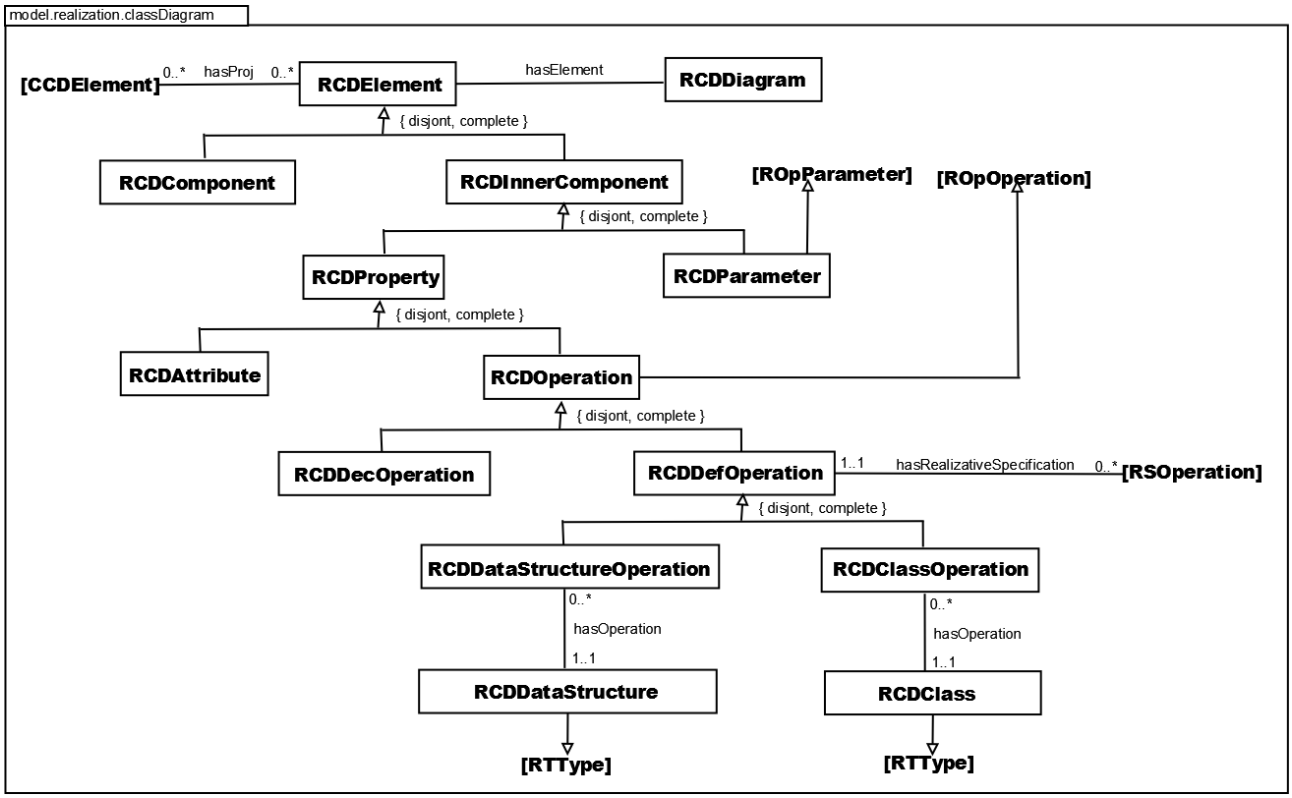


Fig. 2.5 package it.uniroma1.dis.mfis.redia.verifint.model.realization.classDiagram

La classe **RCDDiagram** che estende la classe astratta **Diagram** (così come la estende la classe **CCDDiagram**) contiene al suo interno oggetti di tipo **RCDElement**.
Tale classe è la classe madre di una gerarchia, il cui primo livello vede la distinzione fra **RCDComponent** e **RCDInnerComponent**. Questa, così come accade nel Diagramma delle Classi Concettuale, riflette la differenza che intercorre fra i componenti auto-sufficienti (come una classe) e quelli che devono necessariamente

riferirsi ai primi per la loro identificazione. Poiché ai fini del nostro lavoro era di interesse il concetto di Operazione, è stata considerata solo la gerarchia che discende da **RCDInnerComponent**, trascurando quella discendente da **RCDComponent**.
Le classi figlie di **RCDInnerComponent** sono:
- RCDProperty classe che indica le proprietà, ossia l'insieme unione di attributi (**RCDAttribute**) ed operazioni (**RCDOperation**);
- RCDParameter per la rappresentazione dei parametri di input alle operazioni.

Le classi più interessanti presenti in questo package sono quelle rappresentanti il concetto di operazione. Seguendo lo schema [Pro07 fig 2.11 p.29], è stata realizzata una classe astratta **RCDOperation**, per la rappresentazione di una operazione realizzativa. Per la visibilità dell'operazione, espressa in [Pro07] come attributo, si è preferita invece un'associazione con il tipo di dato **Visibility**, posto in *it.uniroma1.dis.mfis.redia.verifint.model._data Types*. Ci è sembrato opportuno rappresentare il concetto di "visibilità" in quanto, per eventuali estensioni del progetto, potrebbe essere riferito anche a altri concetti, oltre a quello di operazione.
La classe **RCDOperation** è estesa dalle classi **RCDDecOperation** e **RCDDefOperation** per rappresentare rispettivamente i concetti di operazione dichiarata e operazione definita. In particolare, di nostro interesse sono le operazioni definite, dato che sono proprio queste ultime ad avere una Specifica Realizzativa. Un'operazione definita può essere a sua volta riferita ad uno use-case (**RCDUseCaseOperation**, non implementata) ad una struttura dati (**RCDDataStructureOperation**) o ad una classe (**RCDClassOperation**).

2.3.2 Package realizativeSpecifications

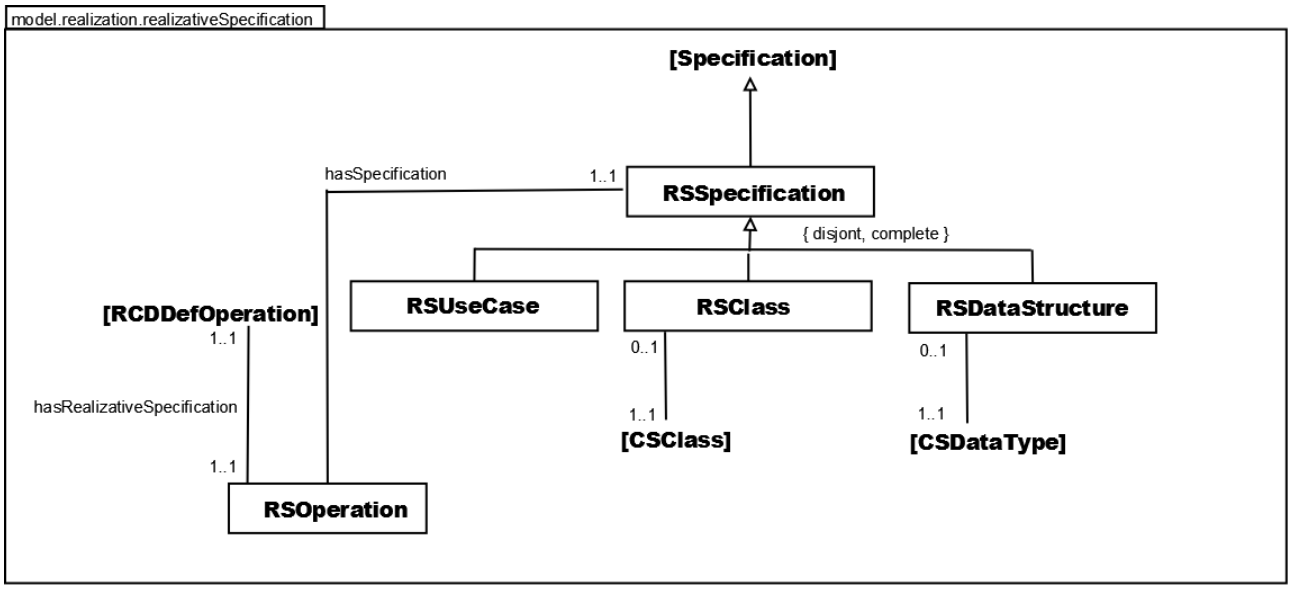


Fig. 2.6 package it.uniroma1.dis.mfis.redia.verifint.model.realization.realizativeSpecifications

In questo package sono presenti le classi necessarie alla rappresentazione delle Specifiche Realizzative.

Le specifiche realizzative sono documenti simili, per la forma con cui si presentano, alle specifiche previste in sede di analisi, ma posseggono una funzione diversa. Hanno infatti lo scopo di descrivere puntualmente gli algoritmi con cui svolgere le operazioni previste per ciascun elemento (use case, struttura dati, classe), in maniera tale da sollevare il realizzatore dal problema di elaborare lui stesso una possibile procedura per lo svolgimento di tali operazioni [Pro07-pp3-4].
Il concetto di Specifica Realizzativa è rappresentato dalla classe astratta **RSSpecification**, e può essere relativa a uno Use-Case (**RSUseCase**) ad una classe (**RSClass**) o a una struttura dati (**RSDataStructure**).
Poiché lo scopo di una Specifica Realizzativa è tenere traccia delle Specifiche di Operazione, è presente la classe **RSOperation**, associata alla Specifica Realizzativa in cui si trova.
Essendo di interesse, per una Specifica di Operazione, le precondizioni e l'algoritmo che implementa l'operazione, sono presenti due classi astratte, collocate in *it.uniroma1.dis.mfis.redia.verifint.model.realization.utilities*: **OperationAlgorithm** ed **OperationPrecondition**. Si ricorda a questo proposito che, a differenza di quanto avviene nella fase di analisi, la specifica realizzativa di un'operazione non dichiara quali sono le postcondizioni, bensì l'algoritmo da usare per rispettarle, descritto generalmente in uno pseudo-codice.
La classe **RSOperation** è inoltre collegata alla classe **RCDDefOperation** (classe figlia di **RCDOperation**), del package *it.uniroma1.dis.mfis.redia.verifint.model.realization.classDiagrams*: tramite questo collegamento è possibile risalire a concetti quali tipo di ritorno di un'operazione (**RTTypeDefined**) e parametri di un'operazione (**RCDParameter**) che sono di interesse per una Specifica di Operazione.
Inoltre le classi **RSClass** e **RSDataStructure** sono collegate alle relative classi di cui rappresentano la specifica realizzativa (rispettivamente **RCDClass** e **RCDDataStructure**).

2.3.3 Package types

Il prefisso di tutte le classi e le interfacce di questo package, denominato *it.uniroma1.dis.mfis.redia.verifint.model.realization.types*, è **RT** (Realization Type). Madre della gerarchia nel pacchetto è **RTType**, ossia il generico tipo, unione di classi (**RCDClass)**, Strutture Dati (**RCDDataStructure**) e tipi di dato propri del Linguaggio (**RTLanguageType**). Questi ultimi a loro volta possono essere tipi base (**RTBaseType**) o tipi di libreria (**RTLibraryType**).

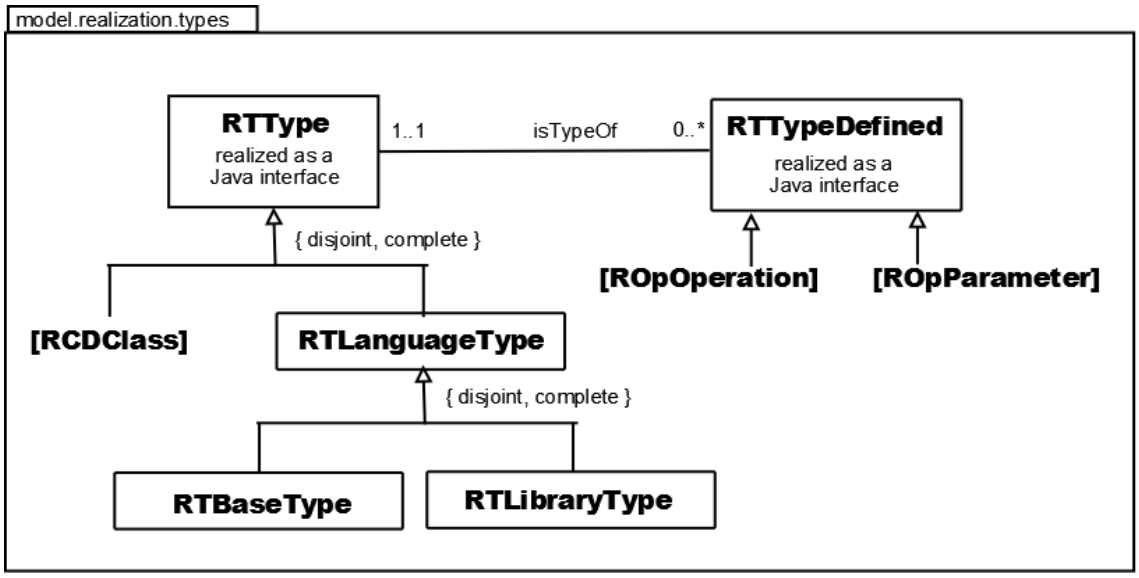


Fig 2.7 package it.uniroma1.dis.mfis.redia.verifint.model.realization.types

2.3.4 Package operations

Il prefisso di tutte le classi e le interfacce del package inerente questo aspetto della modellazione, che si riferisce alla semantica delle operazioni realizzative, è **ROp** (Realizative Operation).
ROpOperation indica il concetto di operazione realizzativa, e **ROpParameter** il concetto di parametro di operazione realizzativa.

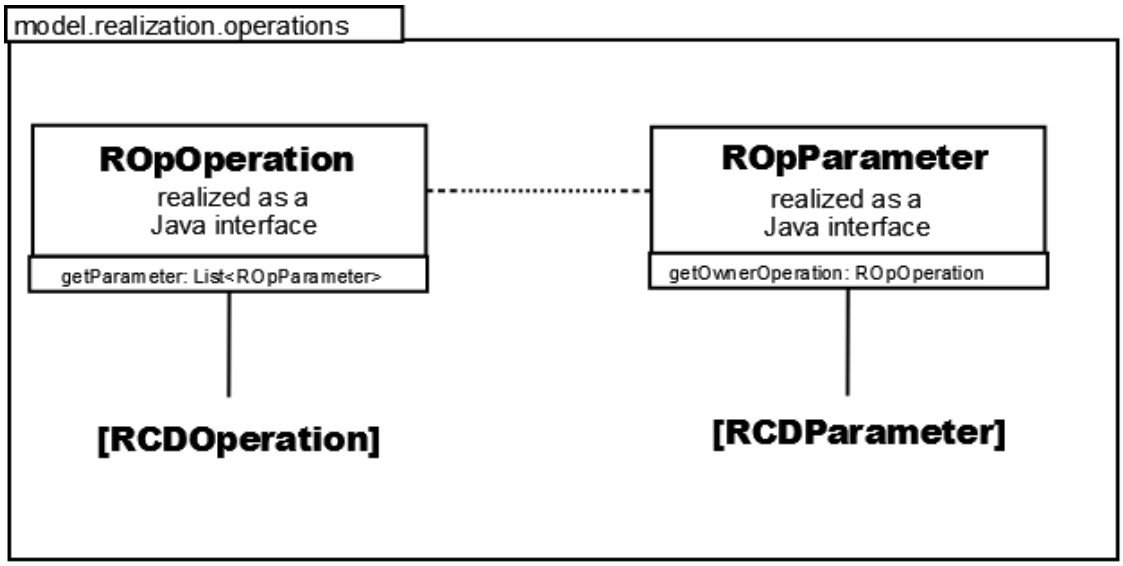


Fig 2.8 package it.uniroma1.dis.mfis.redia.verifint.model.realization.operations

2.3.5 Package utilities

In questo package sono state inserite le classi utili a rappresentare i concetti di algoritmo e precondizione. Questi concetti, che in [Pro07-fig.2.9-p.27] venivano rappresentati come attributi di tipo Stringa, nel lavoro da noi svolto sono stati modellati come classi, in quanto concetti a sé stanti. Inoltre, anche qui si fa riferimento al concetto di estendibilità: dato che una precondizione e un algoritmo possono essere rappresentati in più modi, abbiamo definito le corrispondenti classi astratte, in modo da poter essere estese da opportune classi, ove fosse necessario. Poiché ai fini del progetto era di interesse rappresentare il concetto di precondizione espressa in JML (Java Modeling Language), è stata realizzata la classe **JMLPrecondition**. E' stata realizzata anche la classe **AlgorithmJavaImplementation**, per rappresentare un algoritmo espresso con sintassi Java. Sia la classe **JMLPrecondition** che la classe **AlgorithmJavaImplementation** consentono la memorizzazione automatica rispettivamente delle precondizioni JML e dell'implementazione dell'algoritmo. Nella classe **JMLPrecondition** inoltre è presente il metodo: *toJML()* per la traduzione automatica delle precondizioni espresse in logica del primo ordine, in formule JML. Infine è presente una classe **JASSPrecondition** speculare alla classe **JMLPrecondition,** per la gestione delle precondizioni scritte in logica Jass (una logica quasi del tutto simile alla logica JML); tale classe contiene un metodo *toJass()*, per la traduzione di precondizioni scritte in logica del primo ordine, in precondizioni jass.

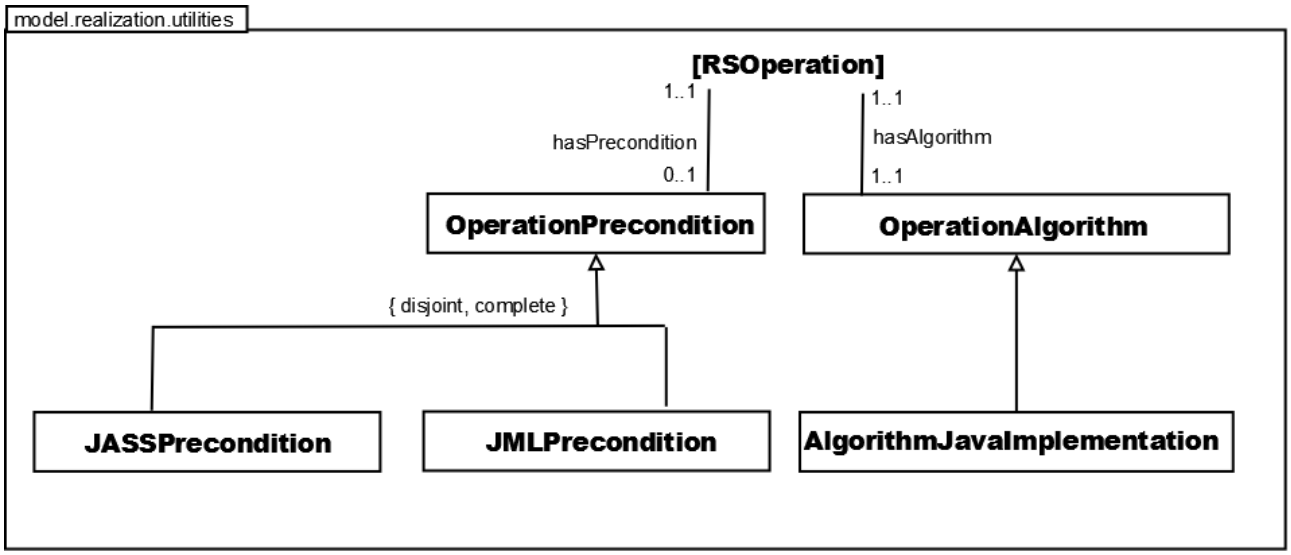


Figura 2.9 package it.uniroma1.dis.mfis.redia.verifint.model.realization.utilities

2.4 Package control

2.4.1 Collegamento con i tool Daikon e Jass

Altro obiettivo del progetto realizzato era permettere l'annotazione automatica di precondizioni e postcondizioni per le operazioni implementate. Per rendere tale compito il più possibile trasparente al programmatore, nel package *it.uniroma1.dis.mfis.redia.verifint.control* è presente una classe **ExecutionTools**, la cui esecuzione permette di lanciare i tool necessari (Daikon e Jass) a tale funzionalità.

Capitolo 3

JASS

3.1 Introduzione

Jass sta per **J**ava with **ass**ertions (pronunciato jazz, come lo stile musicale, da cui l'icona) ed è un precompilatore che supporta le asserzioni in Java. In pratica, si è trasferito il concetto di progettazione per contratto a Java, estendendolo con nuove funzionalità. Il precompilatore Jass è scritto interamente in Java e utilizza una sua logica per esprimere le asserzioni. Purtroppo, la versione da noi utilizzata, non utilizza JML, che invece è utilizzata nella versione successiva (Jass3). Abbiamo scelto comunque la versione 2 di Jass, perché al momento della realizzazione di questo lavoro, era rilasciata come funzionante, mentre la versione 3 era rilasciata come una release. Comunque, la logica utilizzata da Jass è molto simile alla logica JML (le differenze non sono sostanziali, ad esempio si usa *require* al posto di *requires*).

3.2 Progettazione per contratto

Due parole sulla progettazione per contratto sono doverose. Tale tecnica è stata sviluppata da Bertrand Meyer, come caratteristica del linguaggio di programmazione sviluppato proprio da Meyer, l'Eiffel. É possibile usare tale tecnica con pressoché qualsiasi linguaggio. Un'asserzione è un'istruzione che permette di testare eventuali comportamenti che un'applicazione deve avere. La progettazione per contratto si basa su tre tipologie di asserzioni: precondizioni, postcondizioni e invarianti.
Una precondizione esprime una proprietà che un'operazione deve avere quando viene invocata; una postcondizione specifica una proprietà che l'applicazione deve avere nel momento in cui l'operazione viene completata (dice cosa fare, non come, utile per separare interfaccia da implementazione interna); un invariante permette di specificare un vincolo per l'oggetto istanziato. La progettazione per contratto è appunto una tecnica di progettazione, non di programmazione. Essa permette ad esempio di testare anche la consistenza dell'ereditarietà. Una sottoclasse infatti, potrebbe indebolire le precondizione e fortificare le postcondizioni, al fine di convalidare l'estensione. Dal momento che si tende ad utilizzare le asserzioni in fase di test e debbugging, non si deve confondere l'utilizzo delle asserzioni con quello della gestione delle eccezioni.

3.3 Asserzioni in Jass

Le asserzioni devono apparire in commenti formali e sono introdotte da parole chiave, specifiche per ogni tipo di asserzione. Nella maggior parte dei casi, sono una lista di espressioni booleane che descrivono gli stati permessi. Le espressioni booleane possono contenere variabili e chiamate a metodi. I metodi chiamati non devono effettuare side effect.
Esempio di una semplice precondizione:

```
/** require !isFull(); o != null **/
```

Nota: le asserzioni sono comprese tra "/**" e "**/", per essere accettate dal precompilatore. L'uso del ";" equivale a un *and* logico e le condizioni possono essere divise in condizioni più piccole e chiare. Inoltre, possono essere introdotte da etichette, in modo da essere identificate più chiaramente, anche in caso di messaggi d'errore. Ad esempio, la condizione precedente può essere riscritta:

```
/** require    [buffer_not_full] !isFull();
           [object_is_valid] o!= null; **/
```

I tipi di asserzioni permessi sono:

- precondizioni;
- postcondizioni;
- class invariant;
- loop invariant;
- loop variant;
- check.

É permesso l'uso dei costrutti *forall* e *exists* per esprimere proprietà che devono essere valide per tutti gli elementi di un insieme finito di asserzioni, o per un elemento di quell'insieme.
Esempio: l'espressione che fa uso del costrutto *forall:*

```
(forall i: { 0 .. buf.length - 1 } # buf[i] != null);
```

esprime la condizione per cui tutti gli elementi dell'array `buf` devono essere diversi da *null*.

3.3.1 Precondizioni e postcondizioni

Una precondizione va dichiarata all'inizio del corpo del metodo, mentre la postcondizione alla fine.
Con una precondizione, il programmatore può specificare gli stati in cui un metodo può essere invocato. Soddisfare il vincolo è compito del chiamante. Una

precondizione è introdotta attraverso la parola chiave *require*.
Esempio di una precondizione:

```
public void addElement (Object o) {
/** require !isFull(); o != null; **/
...
}
```

Il metodo `addElement` può essere invocato solo se il buffer è non pieno e i parametri formali contengono riferimenti a oggetti validi (non *null*).
I metodi e le variabili usati nella precondizione devono essere visibili nel metodo in cui appaiono. Questo assicura che il chiamante può capire le condizioni sotto cui il metodo può essere invocato. Ciò implica che le variabili locali non possono essere usate. Nell'esempio precedente, il metodo
`isFull` deve essere dichiarato *public* per soddisfare questa condizione.

Una postcondizione specifica gli stati in cui al metodo è permesso ritornare. Soddisfare la postcondizione è compito dello sviluppatore del metodo. Una postcondizione è introdotta attraverso la parola chiave *ensure*.
Esempio di una postcondizione:

```
public void addElement (Object o) {
...
/** ensure !isEmpty() && contains(o); **/
}
```

Dopo aver invocato il metodo `addElement`, il buffer non può essere vuoto e l'oggetto inserito deve essere presente nel buffer. La postcondizione può contenere dei costrutti speciali, quali *Old*, *changeonly* e *Result*.

Gli stati degli oggetti all'invocazione di un metodo, sono memorizzati nella variabile speciale *Old*. Con questa variabile, lo sviluppatore può specificare la relazione tra gli stati di ingresso e uscita. Per esempio, la monotonia di un contatore:

```
public void addElement (Object o) {
...
/** ensure !isEmpty() && contains(o); Old.count == count-1; **/
}
```

Per memorizzare una copia di un oggetto, lo sviluppatore deve implementare il metodo *clone* senza lanciare eccezioni. Così lo sviluppatore decide in che modo deve essere generata una copia di un oggetto (dato che un oggetto può far riferimento a strutture dati, solo lo sviluppatore può decidere quali parti devono essere copiate e quali possono essere condivise).

Un altro costrutto speciale è *changeonly* seguito da una lista di attributi. Se tale lista di attributi è specificata in una postcondizione, solo agli attributi dichiarati è consentito cambiare valore; quelli non presenti nella lista devono rimanere costanti. Una lista vuota sta per *change nothing*. Gli attributi saranno paragonati con il metodo *equals*, che dovrà essere sovrascritto dallo sviluppatore. Esempio di uso di *changeonly*:

```
public void addElement (Object o) {
...
/** ensure !isEmpty() &&  contains(o);
           Old.count == count-1;
           changeonly{count,buffer}; **/

}
```

3.3.2 Invarianti

In ogni classe può essere definito un class invariant. La parola chiave che contraddistingue gli invarianti è *invariant.* Si suppone che un class invariant includa tutte asserzioni che contengano condizioni vere per l'intera classe, e non per un singolo metodo. Un class invariant può anche essere esplicitamente aggiunto a precondizioni e postcondizioni. Un cambiamento su un invariante quindi comporterà cambiamenti in tutte le pre e postcondizioni.
Esempio di class invariant:

```
public class Buffer {
...
/** invariant 0 <= in - out && in - out <= buffer.length; **/
}
```

Nell'uso di invarianti, non possono essere usate né variabili locali, né parametri formali. Un class invariant è controllato ogni volta che si chiama o termina un metodo della classe. Se un invariante contiene variabili che possono essere cambiate da altre istanze, Jass avvertirà lo sviluppatore con un messaggio. In questo caso, l'invariante può essere invalidato senza che venga prodotto nessun messaggio di errore.

3.3.3 Loop invariant e loop variant

Alcuni tipi di errori sono tipici dei cicli, come ad esempio:

- il ciclo non termina;
- il corpo del ciclo è eseguito una volta in più o in meno;
- casi speciali come zero iterazioni, non sono tenuti in considerazione.

Per tenere in considerazione questi casi, sono stati introdotti i *loop invariant* e i *loop variant*. Entrambi sono dichiarati all'inizio del ciclo, prima del corpo. Questi

invarianti definiscono un'asserzione che deve essere influenzata dall'esecuzione del loop, mentre i varianti denotano un'espressione di tipo integer. Due condizioni sono collegate al variante: il suo valore deve essere sempre positivo e deve decrescere ogni volta che il ciclo si ripete. Il precompilatore Jass controlla queste condizioni: dato che entrambe le condizioni non possono essere mantenute all'infinito, il ciclo deve terminare.

Esempio d'uso di loop invariant e variant

```
public boolean contains(Object o) {
 /** require o != null; **/
    for (int i = 0; i < buffer.length; i++)
    /** invariant 0 <= i && i <= buffer.length; **/
    /** variant buffer.length - i **/
       if (buffer[i].equals(o)) return true;
    return false;
/** ensure changeonly{}; **/
}
```

Se entrambi i costrutti sono utilizzati, il *loop invariant* deve essere dichiarato prima del *loop variant.* L'espressione del *variant* deve essere di tipo *int* e non può essere etichettata.

3.3.4 Il check statement

Un check statement è usato per indicare un'asserzione in qualsiasi parte del programma. Indica che in quella posizione, la validità della condizione è rispettata (anche se a prima vista, non ne è chiaro il motivo, guardando solo il codice). Per esempio, assumiamo di avere, in un certo punto del programma, un'assegnazione del tipo *x=y/z*. Il programmatore è sicuro che in quella posizione, *z* non sarà mai uguale a 0. La ragione per cui la variabile non sarà uguale a 0, potrebbe essere in un altro punto del programma, o anche in un altro metodo. L'uso di un check statement non deve far pensare a un errore del programmatore: questo potrebbe non essere a conoscenza di ogni singolo dettaglio dell'intero sistema o semplicemente potrebbe voler ricordare una proprietà in uno specifico punto. Usando un check-statement, si formula esplicitamente `check z!=0`: naturalmente il check-statement può essere anche controllato a runtime.
Esempio di un check-statement:

```
/** check x >= 0; **/
```

3.3.5 Rescue e retry

Introducendo un blocco di sicurezza, l'integrità delle rispettive istanze può essere assicurata in caso di un'eccezione (es. dovuta a chiusura file). Consiste in una sequenza di catch, che contengono uno statement di tipo *retry.*
Esempio:

```
public void add(Object o) {
```

```
   /** require [valid_object]    o != null;
               [buffer_not_full] !full();   **/  ...
   /** ensure ... **/
   /** rescue catch (PreconditionException e) {
                  if (e.label.equals("valid_object")) {
                      o = new DefaultObject(); retry;}
                  else throw e;
               }
  }
```

qui si mostra che vengono catturate le eccezioni delle precondizioni e si reinizializza il metodo chiamante con un riassegnamento dei parametri formali. Se i parametri passati sono *null*, viene creato un oggetto di default e inserito nel buffer. L'etichetta è usata per distinguere fra più eccezioni possibili.

3.3.6 Commenti nelle asserzioni

I commenti nelle asserzioni ne rendono più facile la lettura. Per inserire dei commenti, si usano i marcatori */#* e *#/*.
Esempio di commento inserito in un'asserzione:

```
/** require
/# The buffer must contain space to store the object: #/ !
isFull();
/# Only 'valid' objects can be used: #/ o != null;
**/
```

3.4 Raffinamenti

Java ha alcuni vincoli per l'ereditarietà. Per esempio, se una sottoclasse sovrascrive un metodo di una superclasse, il nuovo metodo deve: essere visibile come il metodo sovrascritto; avere lo stesso tipo di ritorno; dichiarare gli stessi parametri formali. Chiamiamo i metodi della supercalsse *astratti* e quelli della stottoclasse *concreti*. I vincoli garantiscono che una sottoclasse può apparire dove invece ci si aspetta una superclasse. Questa proprietà è tipica della programmazione orientata agli oggetti. Un assegnamento del tipo

```
A x = B;
```

è corretto se il tipo dell'espressione B è sottoclasse del tipo A. Nella progettazione per contratto, i vincoli di Java non sono sufficienti. La classe B che estende A può fornire una nuova definizione per un certo metodo di A: in questo modo è libera di cambiarne la semantica. Questa conseguenza è particolarmente problematica nel caso del polimorfismo perché la chiamata:

```
A a;
...
```

```
a.method();
```

potrebbe causare l'invocazione dell'implementazione di *method()* nella classe A oppure nella classe B in base al dynamic binding. Questo caso, per la progettazione per contratto, è un subappalto, ovvero la classe B, ridefinendo il comportamento di *method()*, deve rispettarne anche le precondizioni. Quindi, la precondizione del metodo concreto deve essere più debole di quella del metodo astratto e la postcondizione di un metodo concreto deve essere più forte di quella del metodo astratto.
Facciamo un esempio. Il seguente metodo astratto aggiunge un elemento in un buffer limitato.
Il metodo può essere invocato se il buffer non è pieno e il parametro passato non è *null*. Dopo l'esecuzione, il contatore viene incrementato.

```
public void addElement (Object o) {
  /** require !isFull(); o != null; **/
  buffer[in % buffer.length] = o;
  in++;
  /** ensure changeonly{in,buffer}; Old.in == in - 1; **/
}
```

La versione concreta premette anche il passaggio di parametri *null:* in questo caso il metodo genera un elemento di default e lo aggiunge al buffer.

```
public void addElement (Object o) {
  /** require !isFull(); **/
  if (o==null)
    buffer[in % buffer.length] = new Default();
  else
    buffer[in % buffer.length] = o;
  in++;
  /** ensure changeonly{in,buffer};
            Old.in == in - 1;
            o!=null ? contains(o) : true;
   **/
}
```

La precondizione del metodo concreto è più debole di quello astratto: la precondizione astratta implica quella concreta. La nuova postcondizione è più forte di quella vecchia: questo garantisce che un elemento che dovrebbe essere inserito sia contenuto dopo l'esecuzione del metodo. Naturalmente ci si aspetta questo, ma non è formalmente specificato nel primo esempio.

3.4.1 Raffinamenti in Jass

Decidere se un programmatore, estendendo una classe, debba usare i raffinamenti o

farne a meno, è un'importante decisione di progetto. In Jass si è preferito adottare la seconda possibilità. Il programmatore può usare i raffinamenti o no: se vuole che una sottoclasse adotti i raffinamenti, deve implementare l'interfaccia `jass.runtime.refinement`**.** Il controllo per le pre/ postcondizioni e invarianti viene eseguito a runtime. Per esempio, il precompilatore aggiunge un controllo del tipo

```
pre_a && !pre_c
```

all'inizio del metodo, che è equivalente a dire: la precondizione astratta non implica quella concreta. Se il controllo ha successo (il che non implica un successo per il programmatore!) viene lanciata un'eccezione del tipo `jass.runtime.RefinementException.` Questo è un nuovo tipo di errore nella progettazione per contratto. Fin qui abbiamo avuto un errore lato server se la postcondizione non fosse stata rispettata, e un errore lato client se la precondizione non fosse stata rispettata. Questo è un errore di progettazione. Il programmatore deve implementare un metodo astratto del tipo `jassGetSuperState` che deve ritornare un riferimento al tipo della superclasse. L'istanza di ritorno deve essere in uno stato astratto che è mappato dallo stato concreto attraverso la funzione di astrazione. Per esempio, se la funzione astratta è la funzione identità, il metodo potrebbe essere il seguente:

```
private theAbstractClass jassGetSuperState() {
  return (theAbstractClass)this;
}
```

La funzione astratta è la funzione identità se tutti i campi sono visibili nella sottoclasse e se questa mantiene la loro semantica. Un esempio più complesso è `UnlimitedBuffer`, nel package di esempi Jass. Un buffer illimitato è una sottoclasse di un buffer che può contenere una quantità illimitata di elementi. La struttura dati della sottoclasse non è più lunga di un array. Il buffer è implementato attraverso una `java.util.Vector`.

```
private Buffer jassGetSuperState() {
    Buffer b = new Buffer(v.size()+1);
    b.in = v.size();
    b.out = 0;
    for (int i = 0; i < b.buffer.length-1; i++)
        b.buffer[i] = v.elementAt(i);
    return b;
}
```

Questo metodo costruisce un buffer astratto che rappresenta lo stato corrente, ovvero un buffer che può sempre aggiungere un nuovo elemento. Ricordiamo che se si vuole che l'ereditarietà sia un raffinamento si deve implementare:

- l'interfaccia `jass.runtime.Refinement`;

- la funzione `jassGetSuperState`.

Jass quindi genera codice per il controllo dei raffinamenti come codice per le altre asserzioni.

Capitolo 4

Daikon

4.1 Introduzione

Daikon è uno strumento che, attraverso l'individuazione di invarianti, fornisce assistenza nella creazione di specifiche: fornendo in input al tool un programma (scritto in codice Java, C, C++ o Perl) questo individua le proprietà dello stesso e restituisce in output tali proprietà espresse in sintassi JML (o anche in altri formati) ed è in grado di inserirle automaticamente nel programma, nella corretta posizione all'interno del codice.
Daikon individua gli invarianti dinamicamente, a runtime: dato un programma, ne effettua una serie di esecuzioni, e riporta le proprietà risultate vere in tali esecuzioni, definendo in questo modo gli invarianti del programma.
Poiché si tratta di un'analisi di tipo dinamico, l'accuratezza degli invarianti individuati dipende dalla qualità e dalla completezza dei casi di test, dato che altre esecuzioni potrebbero dimostrare false alcune delle proprietà riportate come vere.
È utile a tal proposito l'utilizzo combinato di Daikon con un verificatore statico come ESC/Java: con il primo si individuano dinamicamente le proprietà del programma e si annotano automaticamente nel programma stesso; con il secondo si verifica staticamente il programma annotato, cioè si verifica se tali proprietà risultano effettivamente vere. Analisi statistiche hanno dimostrato che ESC/Java dimostra essere vere circa il 90% delle proprietà riportate da Daikon.

4.2 Individuare gli invarianti

L'individuazione degli invarianti consiste in due passi:

1. Ottenere uno o più "data trace" file attraverso l'esecuzione del programma sotto il controllo di un "front end" che registra una serie di informazioni sui valori assunti dalle variabili. Per evitare output eccessivamente lunghi e per migliorare le performace, è possibile analizzare anche solo una parte del programma.
Es.: l'esecuzione di `java daikon.Chicory package.Class arg` (dove Chicory è il "front end") produce in output un "data trace" file;
2. Eseguire Daikon prendendo come input i "data trace" file prodotti al punto precedente. Gli invarianti individuati potranno ora essere analizzati in forma testuale o essere forniti in input ad altri tool per essere processati.
Es.: l'esecuzione di `java daikon.Daikon Class.dtrace.gz` produce in output gli invarianti individuati da Daikon.

É possibile eseguire i due passi con un unico comando (si chiama il front end con l'opzione `--daikon`); con l'opzione `--daikon-online` inoltre, si evita la creazione di "data trace file", dato che le informazioni raccolte sotto il controllo del "front end" vengono inviate direttamente a Daikon.
Es.: l'esecuzione di
`java daikon.Chicory --daikon-online package.Class arg`
produce in output gli invarianti individuati da Daikon. Gli invarianti individuati, possono essere analizzati direttamente su terminale oppure possono essere "inseriti" come commenti all'interno del codice sorgente Java attraverso l'uso del tool Annotate.
Es.: l'esecuzione di
`java daikon.tools.jtb.Annotate Class.inv.gz package.Class.java`
produce in output il file java annotato:
`package.Class.java-escannotated.`

4.3 Come eseguire Daikon

In realtà per l'esecuzione di Daikon si possono fornire in input altri tipi di file, oltre ai "data trace" file di cui si è detto in precedenza; il comando generale è del tipo:

```
    java daikon.Daikon [flags] dtrace-files [decl-files] [spinfo-files]
```

a) Come detto i "dtrace-files" contengono i valori che le variabili hanno assunto in esecuzioni del programma;

b) i "decl-files" contengono dichiarazioni su punti del programma ("program point"). Non tutti i front end usati con daikon li producono, ad esempio per Java, il front end Chicory produce solo dtrace-files che contengono al loro interno anche le dichiarazioni sui program point;

c) gli "spinfo-files" servono per consentire l'individuazione di invarianti condizionali (vedi 4.5.2); possono essere creati automaticamente o a mano.

d) i "flags" servono per utilizzare ulteriori funzionalità di Daikon. Tra i più significativi citiamo:

- `--format formatName`. Produce l'output nel formato richiesto (vedi par. 4.1). Es.: `--format JML;`
- `--omit_from_output` [0rs]. Serve per omettere dall'output informazioni ridondanti o che non sono ritenute necessarie per l'uso che se ne vuole fare. Il flag deve essere seguito da uno o più dei seguenti caratteri:

'0' -> Omette informazioni su "program point" dichiarati in un "decl-file" ma per i quali non sono forniti esempi in nessun "dtrace-file"

'r' -> Omette invarianti "riflessivi", cioè quelli in cui una variabile compare più di una volta (es "y = z + z". In genere, gli invarianti di questo tipo, non sono di particolare interesse, poiché sono "duplicati" nella loro semantica da invarianti con meno variabili (es "y = 2 * z)

's' -> Omette invarianti che sono "soppressi" da altri invarianti; "soppressi" nel senso di logicamente implicati.

`--ppt-select-pattern=ppt_regexp` Serve per processare solo i program point il cui nome è uguale a ppt_regexp

`--ppt-omit-pattern=ppt_regexp` Serve per non processare i program point il cui nome è uguale a ppt_regexp

`--var-select-pattern=var_regexp` Serve per processare solo le variabili il cui nome è uguale a var_regexp

`--var-omit-pattern=var_regexp` Serve per non processare le variabili il cui nome è uguale a var_regexp

4.4 Analizzare l'output di Daikon

In questo paragrafo analizziamo i risultati prodotti dall'esecuzione del tool, cioè gli invarianti.
Un invariante è una proprietà che deve essere sempre mantenuta in ogni stato di una classe e deve essere vera quando il controllo non è interno ai metodi dell'oggetto. Perciò, un invariante è una proprietà che deve essere garantita quando termina l'esecuzione del costruttore di una classe Java ed all'inizio ed alla fine di un metodo. [Cer08-par.2.5-p.14].

4.4.1 Sintassi degli invarianti

Come detto in precedenza (par.3d) Daikon può produrre il proprio output in diversi formati:

a) Daikon format: un insieme di Java e logica matematica:

b) DBC format: è il formato di tipo "Design By Contract" richiesto dal tool Parasoft's Jtest tool (http://www.parasoft.com);

c) ESC/Java format: da usare con il tool ESC/Java originale; per la versione ESC/Java2 si usa il formato JML:

d) Java format: ogni invariante è un'espressione Java:

e) JML format;

f) Simplify format: è il formato atteso dal dimostratore di teoremi automatico "Simplify";

4.4.2 Program Points

Gli invarianti, output del tool, sono organizzati per program point: un punto del programma è uno specifico punto del codice sorgente, posizionato immediatamente prima di una particolare linea di codice. E' fornita di seguito una descrizione dei diversi tipi di program point che è possibile incontrare:

a) `method():::ENTER` è il punto che precede il metodo *method()*; gli invarianti qui presenti sono le precondizioni al metodo, cioè le proprietà che devono essere sempre verificate quando *method()* viene invocato;

b) `method():::EXIT` è il punto posto al termine di *method()*; gli invarianti qui presenti sono le postcondizioni al metodo, cioè le proprietà che devono essere sempre verificate quando *method()* termina. Quando il metodo può terminare in diversi modi, ci sono più program point di tipo EXIT, per rappresentare le postcondizioni che devono essere garantite nei diversi punti di uscita; ciascuno di questi punti EXIT è identificato dal numero di linea del punto di uscita dal metodo cui fa riferimento (es. `method():::EXIT20 method()::EXIT30`);

c) `:::OBJECT` è un program point cui seguono invarianti che devono essere validi su tutti i campi della classe. Queste proprietà devono essere vere per ogni oggetto istanza della classe; sono sempre vere ad eccezione del punto :::ENTRY del costruttore, dove i campi ancora non sono inizializzati.

d) `:::CLASS` è simile a :::OBJECT, ma coinvolge solo variabili statiche che hanno solo un valore per tutti gli oggetti.

4.4.3 Variabili

Come detto un front end produce un dtrace-file che associa variabili "di traccia" con determinati valori: i nomi delle variabili di traccia non devono essere necessariamente gli stessi di quelli delle variabili del programma; infatti la traccia può contenere valori che non sono assunti da nessuna delle variabili del programma, e quindi, in questi casi, il front end crea nomi che esprimano tali valori. Daikon ignora i nomi delle variabili durante l'individuazione degli invarianti; li usa solo per la produzione dell'output.
In aggiunta alle variabili che appaiono nel file dtrace, Daikon crea variabili aggiuntive (variabili derivate), dalla combinazione di variabili di traccia. Per esempio per ogni array di cardinalità *n*, Daikon crea *n* variabili *a[0], a[1], ... ,a[n-1]*.

E' fornito di seguito un elenco di convenzioni sulla denominazione delle variabili:

a[i] è l'elemento i-esimo dell'array *a*

a[-1] è l'ultimo elemento dell'array *a*

a[] è la sequenza degli elementi contenuti nell'array *a*, il suo contenuto. L'espressione *a*, indica invece solo l'indirizzo dell'array *a*; allora, dati due array *a* e *b* vale *a=b -> a[]=b[]* mentre non vale *a[]=b[] -> a=b*

x.y, x->y è il campo *y* della classe *x*. se *x* è un array, non ha senso scrivere *x.y*; ha senso invece *x[].y*

orig(x) si riferisce al valore della variabile *x* immediatamente prima dell'inizio di un metodo (il metodo potrebbe modificare il valore di *x*). Questo tipo di variabile può comparire solo in program point di tipo :::EXIT. Se il formato richiesto per l'output è JML, questa variabile viene riportata come *\old(x)*.

post(x) si riferisce al valore della variabile *x* immediatamente dopo il termine di un metodo.

I front end attuali, tra cui Chicory, non producono output per le variabili locali ai metodi, ma solo per quelle visibili al loro esterno. Più in generale i front end, producono output all'inizio e al termine delle procedure (precondizioni e postcondizioni), non al loro interno: Daikon dunque fornisce specifiche dal punto di vista del cliente della procedura.

4.5 Migliorare l'output di Daikon

4.5.1 Opzioni di configurazione

Molti aspetti del comportamento di Daikon possono essere controllati attraverso il settaggio di alcuni parametri di configurazione. Questi parametri di configurazione controllano quali invarianti devono essere controllati e riportati, quali variabili derivate vanno create, ed altro. Tali parametri si settano in un file di configurazione che viene passato in input a Daikon tramite l'opzione `--config`. Alternativamente un parametro di configurazione può essere settato direttamente da linea di comando usando l'opzione `--config_option name=value.`

4.5.2 Invarianti condizionali ed implicazioni

Gli invarianti condizionali sono invarianti che non sono sempre veri. Consideriamo ad esempio la postcondizione per il metodo che calcola il valore assoluto di un numero intero:

```
if num < 0
  then return == -num
  else return == num
```

L'invariante *return == -num* è un invariante condizionale perché dipende dal predicato *num < 0*.
Un'implicazione è un invariante composto da un predicato e da un invariante condizionale.
Daikon possiede alcuni predicati nativi che usa per trovare invarianti condizionali; inoltre può leggere predicati da uno *.spinfo* file e trovare implicazioni basate su tali predicati. Un file del genere può essere prodotto sia manualmente che automaticamente. Il termine "Splitter", da cui "splitter info file", deriva dalla tecnica con cui Daikon individua implicazioni ed invarianti condizionali: per ogni predicato nel file, Daikon crea due program point "condizionali": uno per le esecuzioni del programma che soddisfano il predicato ed uno per quelle che non lo soddisfano, e divide la traccia (fornita nel dtrace-file) in due parti. L'individuazione degli invarianti è dunque eseguita separatamente sui due punti, ed ogni invariante trovato è riportato come invariante condizionale all'interno di un'implicazione.

4.5.2.1 Gli Splitter Info File

Uno splitter info file contiene le condizioni che Daikon dovrebbe usare per la creazione degli invarianti condizionali. Ogni sezione del file contiene una sequenza di linee non vuote. Ci sono due tipi possibili di sezioni:

a) Program Point Section:

```
PT_NAME pptname
condition1
condition2
    DAIKON_FORMAT output string
    ESC_FORMAT output string
condition3
...
```

Le "Program Point Section" presentano inizialmente una linea di codice che specifica il nome del punto di programma che rappresentano, seguita da altre linee di codice che specificano le Condizioni che ad esso sono associate (una condizione per linea); per ciascuna condizione, possono essere fornite anche informazioni addizionali quali ad esempio il formato dell'output.

b) Replacement Section:

```
REPLACE
procedure1
replacement1
procedure2
replacement2
```

In genere le condizioni presenti nello spinfo-file non possono invocare metodi definiti nel codice sorgente, poiché quando Daikon analizza i dtrace-files non ha accesso ai file .java. Una replacement section dunque specifica i corpi dei metodi, permettendo alle condizioni, di invocarli.
Nell'esempio, replacement *i* è un'espressione java che costituisce il corpo della procedura *i*. In ogni condizione dunque, quando viene chiamato un metodo, Daikon esegue il corrispondente "replacement".

4.5.2.2 Esempio di uno splitter info file

E' fornita di seguito l'implementazione di una classe Stack per interi positivi ed il corrispondente .spinfo file.

La classe

```
class simpleStack {

  private int[] myArray;
  private int currentSize;

  public simpleStack(int capacity) {
    myArray = new int[capacity];
    currentSize = 0;
  }

  /** Adds an element to the back of the stack, if the stack is
    * not full.
    * Returns true if this succeeds, false otherwise.  **/
  public String push(int x) {
    if ( !isFull() && x >= 0) {
      myArray[currentSize] = x;
      currentSize++;
      return true;
    } else {
      return false;
    }
  }

  /** Returns the most recently inserted stack element.
    * Returns -1 if the stack is empty. **/
  public int pop() {
    if ( !isEmpty() ) {
      currentSize--;
      return myArray[currentSize];
    } else {
      return -1;
    }
  }

 /** Returns true if the stack is empty, false otherwise. **/
  private boolean isEmpty() {
    return (currentSize == 0);
  }

  /** Returns true if the stack is full, false otherwise. **/
  private boolean isFull() {
    return (currentSize == myArray.length);
  }
}
```

Lo Splitter info file

```
REPLACE
isFull()
currentSize == myArray.length
isEmpty()
currentSize == 0

PPT_NAME  simpleStack.push
!isFull() && x >= 0
```

```
    DAIKON_FORMAT !isFull() and x >= 0
    SIMPLIFY_FORMAT (AND (NOT (isFull this)) (>= x 0))

PPT_NAME  simpleStack.pop
!isEmpty()

PPT_NAME  simpleStack.isFull
currentSize == myArray.length - 1

PPT_NAME  simpleStack.isEmpty
currentSize == 0
```

4.5.3 Migliorare l'individuazione di invarianti condizionali

Il meccanismo nativo di Daikon per l'individuazione degli invarianti condizionali, ha delle limitazioni. Fornendo uno spinfo-file è possibile trovarne un numero maggiore. Per creare automaticamente tali file esistono tre metodi:

a) Analisi statica per la creazione degli splitters: tutte le espressioni booleane presenti nel codice sono estratte ed usate come condizioni di splitting; questo perché si suppone che le condizioni testate esplicitamente nel programma ne influenzino il comportamento e possano portare all'individuazione di utili invarianti condizionali. Il tool CreateSpinfo serve a questo scopo: attraverso il comando

```
java daikon.tools.jtb.CreateSpinfo Class.java
```

viene creato il file Class.spinfo contenente tutte le espressioni booleane presenti nel codice sorgente.
b) Analisi basata su cluster per la creazione degli splitters.
L'analisi basata su cluster è un metodo statistico per trovare gruppi (cluster) di dati (invarianti condizionali) che rispettino una certa proprietà condizionale (predicato). Una proprietà condizionale in un punto del programma separa i dati tra quelli che soddisfano la proprietà e quelli che non la soddisfano (li separa in due cluster); ogni invariante dunque che è in un cluster e non nell'altro è un invariante condizionale. Il meccanismo funziona individuando nel data trace file i cluster, inferendo gli invarianti sui cluster e riportandoli in uno splitter info file.

c) Selezione random degli invarianti per la creazione degli splitters

Capitolo 5

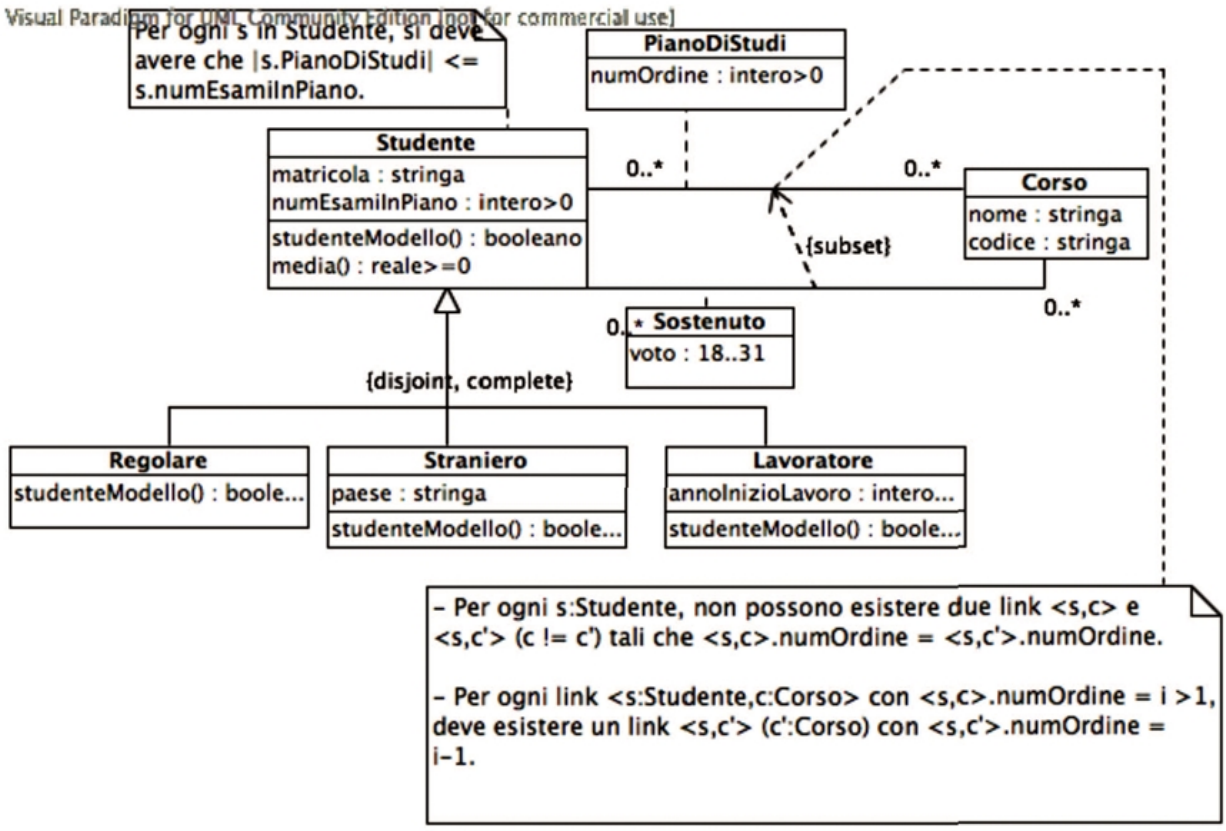
Caso d'uso dei tool Jass e Daikon: Segreteria Studenti

Attraverso il caso di studio che segue (tratto dal sito del corso di Progettazione del Software www.dis.uniroma1.it/~tmancini), ci si propone di mostrare attraverso un'applicazione concreta, le potenzialità dei due tool descritti in precedenza.

5.1 Requisiti

L'applicazione è relativa alla gestione di una segreteria studenti di un'università. Quando gli studenti si iscrivono, vengono loro assegnati una matricola ed un numero di esami da fare.
Successivamente decidono il proprio piano di studi, ovvero scelgono un corso (di cui interessa codice e nome) come loro i-mo esame ($1 \leq i \leq$ numEsami). La scelta del piano di studi può avvenire anche un solo esame alla volta. Gli studenti possono sostenere esami (solamente se sono nel loro piano di studi). Esistono tre categorie di studenti completamente disgiunte fra loro: regolari, stranieri e lavoratori. Degli stranieri interessa il paese di origine, mentre dei lavoratori interessa sapere da quanti anni lavorano.
Infine, se uno studente ha sostenuto almeno una certa percentuale del numero di esami, relativamente al numero di quelli previsti in quel momento dal suo piano di studi, viene classificato come "studente modello" e può concorrere a prendere una borsa di studio.
Suddetta percentuale dipende dalla tipologia dello studente: 100% per gli studenti regolari, 50% per quelli lavoratori, 70% per gli stranieri.
La segreteria studenti, come cliente della nostra applicazione, deve poter effettuare delle operazioni sugli studenti. In particolare si faccia riferimento al seguente use-case: dato un insieme *S* di studenti, al fine di scegliere i destinatari delle borse di studio, si vuole ottenere l'insieme di studenti in *S* tali che: (i) sono studenti modello e (ii) hanno la media più alta tra tutti gli altri studenti modello in *S* della loro categoria (regolare, straniero, lavoratore). Si noti che, per ogni categoria, vi possono essere diversi studenti con la media più alta (ovviamente la stessa).

5.2 Diagramma UML delle classi



5.3Specifica concettuale delle classi

InizioSpecificaClasse *Studente*

studenteModello *(): booleano*

pre: nessuna

post: *result* è pari a true se e solo se |*this.sostenuto*|/|*this.pianoDiStudi*| ≥ x (con *this.sostenuto* e *this.pianoDiStudi* l'insieme dei link di tipo, rispettivamente, *sostenuto* e *pianoDiStudi* relativi allo studente *this*). Il valore x dipende dalla sottoclasse di Studente alla quale appartiene *this*.

media *(): reale >= 0*

pre: |*this.sostenuto*| ≥ 1;

post: *result* è pari a *l*∈*this.sostenuto*(*l.voto*) / |*this.sostenuto*|.

FineSpecifica

InizioSpecificaClasse *Regolare is-a Studente*

***studenteModello** (): booleano*

pre: nessuna

post: Quelle di *Studente.studenteModello()* con x = 1.

FineSpecifica

InizioSpecificaClasse *Straniero is-a Studente*

***studenteModello** (): boolean*

pre: nessuna

post: Quelle di *Studente.studenteModello()* con x = 0.7.

FineSpecifica

InizioSpecificaClasse *Lavoratore is-a Studente*

***studenteModello** (): boolean*

pre: nessuna

post: Quelle di *Studente.studenteModello()* con x = 0.5.

FineSpecifica

5.4 Implementazione dei metodi:

In *Studente.java*

studenteModello(double d):

```
protected boolean studenteModello(double x) {
    return (double)sostenuto.size() >=
    ((double)pianoDiStudi.size())*x;
}
```

media():

```
public double media() throws EccezionePrecondizioni {
    if (sostenuto.size() == 0)
     throw new EccezionePrecondizioni("Impossibile
     calcolare la media. " + "Non ci sono esami
     sostenuti");
```

```
        int somma = 0;
        Iterator it = sostenuto.iterator();
        while (it.hasNext()) {
            TipoLinkSostenuto t =
            (TipoLinkSostenuto)it.next();
            somma += Math.min(30, t.getVoto());
            // In questo esempio "30 e lode" vale 30 ai
            //fini della media
      }
      return ((double)somma) / sostenuto.size();
    }
```

Per quento riguarda il metodo *studenteModello()* riportiamo come esempio valido per tutte le sottoclassi (*Regolare.java*; *Straniero.java; Lavoratore.java*), quello presente in *Straniero.java*.

In *Straniero.java*

studenteModello():

```
    public boolean studenteModello() {
        return super.studenteModello(0.7);
    }
```

Riportiamo inoltre il codice di un metodo *main(),* usato come esempio per l'esecuzione dei tool sull'applicazione:

```
    public static void main(String args[]) throws
        EccezionePrecondizioni {
        Studente s3 = new Straniero("003", 27,
        "Romania");

        Corso c1 = new Corso("FondInf", "AAA");
        Corso c2 = new Corso("ProSw", "AAB");
        Corso c3 = new Corso("Analisi1", "AAC");
        Corso c4 = new Corso("Geom1", "AAD");

        s3.inserisciLinkPianoDiStudi(c1);
        s3.inserisciLinkPianoDiStudi(c2);
        s3.inserisciLinkSostenuto(c1, 28);
        s3.inserisciLinkSostenuto(c2, 27);

        Set studenti = new
                InsiemeArrayOmogeneo(Studente.class);
```

```
        studenti.add(s3);

        boolean modello = s3.studenteModello();

        double d = s3.media();

  }
```

5.5 Jass

Per il suo funzionamento, il tool Jass necessita di un file *.jass:* un file *.java* con annotazioni scritte, come detto in precedenza, in logica jass. Nell'esempio tale file è *Studente.jass*. Per la sua creazione è stato fatto uso di una semplice applicazione, creata da noi per tale scopo, che, ricevendo in input un file *.java* e un file di testo contenente le formule in logica jass, produce automaticamente in output un file *.jass*. Nel progetto è stato previsto il metodo *toJass()* che traduce semplici formule scritte in logica del primo ordine in formule jass.

File *media.txt*

```
    /** require [non_ci_sono_esami_sostenuti]
        sostenuto.size() > 0; **/
```

Il metodo *media()* annotato nel file *Studente.jass*

```
public double media() throws EccezionePrecondizioni {
      /** require [non_ci_sono_esami_sostenuti]
          sostenuto.size() > 0; **/
      int somma = 0;
      Iterator it = sostenuto.iterator();
      while (it.hasNext()) {
              TipoLinkSostenuto t =
                  (TipoLinkSostenuto)it.next();
              somma += Math.min(30, t.getVoto());
      }
      return ((double)somma) / sostenuto.size();
}
```

A questo punto la nostra applicazione lancia automaticamente il tool Jass che

prendendo in input il file *Studente.jass*, produce in output un nuovo file *Studente.java*, ora in grado di gestire le precondizioni attraverso controlli.

Il metodo *media()* nel nuovo file *Studente.java*

```
    public double media() throws EccezionePrecondizioni {
        double jassResult;

      /* precondition */
      if (!(sostenuto.size()>0)) throw new
  jass.runtime.PreconditionException("Studente","media()"
  ,41,"non_ci_sono_esami_sostenuti");

      int somma = 0;
      Iterator it = sostenuto.iterator();
      while (it.hasNext()) {
            TipoLinkSostenuto t =
           (TipoLinkSostenuto)it.next()
      somma += Math.min(30, t.getVoto());
      }
      jassResult = ( ((double)somma) / sostenuto.size());
      return jassResult;

    }
```

Jass è in grado di gestire automaticamente anche le postcondizioni di un metodo; riportiamo di seguito l'esempio del metodo *studenteModello(double d)* annotato con la postcondizione nel file *Studente.jass* e lo stesso metodo nel nuovo file *Studente.java* prodotto da Jass sulla base del file *Studente.jass*.

Metodo studenteModello(double d) del file Studente.jass:

```
  protected boolean studenteModello(double x) {
    return (double)sostenuto.size() >=
          ((double)pianoDiStudi.size())*x;
      /** ensure this.sostenuto.size()  /
  this.pianoDiStudi.size()  >= x; **/
   }
```

Metodo *studenteModello(double d)* nel file *Studente.java*:

```
protected boolean studenteModello(double x) {
    boolean jassResult;
```

```
    jassResult = ( (double)sostenuto.size() >=
    ((double)pianoDiStudi.size())*x);
    /* postcondition */
    if (!
    (this.sostenuto.size()/this.pianoDiStudi.size()>=x))
    throw new
  jass.runtime.PostconditionException("Studente","student
  eModello(double)",36,null);

  return jassResult;
}
```

5.6 Daikon

Con l'esecuzione dei comandi

```
java daikon.tools.jtb.CreateSpinfo Straniero.java

java daikon.tools.jtb.CreateSpinfo Studente.java
```

vengono creati i file *Straniero.spinfo* e *Studente.spinfo*, dove sono riportate le condizioni utili all'individuazione di invarianti condizionali. In particolare riportiamo le condizioni trovate relative ai metodi *media()* e *studenteModello(double d)* di *Studente.java* e *studenteModello()* di *Straniero.java*:

In *Studente.spinfo*:

```
PPT_NAME Studente.media
it.hasNext()
orig(it.hasNext())
orig(sostenuto.size()== 0)
sostenuto.size()== 0

PPT_NAME Studente.studenteModello
(double)sostenuto.size()>=((double)pianoDiStudi.size())*x
orig((double)sostenuto.size()>=((double)pianoDiStudi.size
())*x)
```

In *Straniero.spinfo*:

```
PPT_NAME Straniero.studenteModello
orig(super.studenteModello(0.7))
super.studenteModello(0.7)
```

Con l'esecuzione in sequenza dei comandi

```
java daikon.Chicory Straniero

java daikon.Daikon --format jml Straniero.dtrace.gz
    Straniero.spinfo     Studente.spinfo
```

viene creato il file *Straniero.inv.gz*, contenente in forma serializzata, gli invarianti individuati.
Riportiamo gli invarianti individuati relativi alle precondizioni e alle postcondizioni dei metodi *media()* e *studenteModello(double d)* di *Studente.java* e al metodo *studenteModello()* presente in *Straniero.java*.

```
Studente.media():::ENTER
this.numEsamiInPiano has only one value
Straniero.studenteModello():::ENTER
==========================================================
==================
Straniero.studenteModello():::EXIT
this.paese == orig(this.paese)
return == true
this.paese.toString == orig(this.paese.toString)
==========================================================
==================
Straniero.studenteModello():::EXIT;condition="return ==
true"

==========================================================
==================
Studente.media():::EXIT
this.matricola == orig(this.matricola)
this.numEsamiInPiano == orig(this.numEsamiInPiano)
this.pianoDiStudi == orig(this.pianoDiStudi)
this.sostenuto == orig(this.sostenuto)
this.numEsamiInPiano has only one value
return == 27.5
this.matricola.toString == orig(this.matricola.toString)
this.pianoDiStudi.getClass() ==
orig(this.pianoDiStudi.getClass())
this.pianoDiStudi.getClass() ==
```

```
orig(this.sostenuto.getClass())

Studente.studenteModello(double):::ENTER
this.numEsamiInPiano has only one value
arg0 == 0.7
=========================================================
=================
Studente.studenteModello(double):::EXIT
this.matricola == orig(this.matricola)
this.numEsamiInPiano == orig(this.numEsamiInPiano)
this.pianoDiStudi == orig(this.pianoDiStudi)
this.sostenuto == orig(this.sostenuto)
this.numEsamiInPiano has only one value
return == true
this.matricola.toString == orig(this.matricola.toString)
this.pianoDiStudi.getClass() ==
orig(this.pianoDiStudi.getClass())
this.pianoDiStudi.getClass() ==
orig(this.sostenuto.getClass())
=========================================================
=================
Studente.studenteModello(double):::EXIT;condition="return
== true"
```

Infine con l'esecuzione dei comandi

```
java daikon.tools.jtb.Annotate Straniero.inv.gz
    Studente.java

java daikon.tools.jtb.Annotate Straniero.inv.gz
    Straniero.java
```

gli invarianti trovati vengono annotati automaticamente nei file.java, in particolare per i metodi in analisi:

In *Studente.java:*

studenteModello(double d):

```
/*@ requires this.numEsamiInPiano != null; */
/*@ requires arg0 == 0.7; */
/*@ ensures this.numEsamiInPiano != null; */
/*@ ensures \result == true; */
/*@ ensures \typeof(this.pianoDiStudi) ==
```

```
    \old(\typeof(this.pianoDiStudi)); */
  /*@ ensures \typeof(this.pianoDiStudi) ==
    \old(\typeof(this.sostenuto)); */
  protected boolean studenteModello(double x) {
    return (double)sostenuto.size() >=
    ((double)pianoDiStudi.size())*x;
  }
```

media():

```
  /*@ requires this.numEsamiInPiano != null; */
  /*@ ensures this.numEsamiInPiano != null; */
  /*@ ensures \result == 27.5; */
  /*@ ensures \typeof(this.pianoDiStudi) ==
    \old(\typeof(this.pianoDiStudi)); */
  /*@ ensures \typeof(this.pianoDiStudi) ==
    \old(\typeof(this.sostenuto)); */
  public double media() throws EccezionePrecondizioni {
      if (sostenuto.size() == 0)
          throw new EccezionePrecondizioni("Impossibile
           calcolare la media. " +
          "Non ci sono esami sostenuti");
      int somma = 0;
      Iterator it = sostenuto.iterator();
      while (it.hasNext()) {
          TipoLinkSostenuto t =
          (TipoLinkSostenuto)it.next();
           somma += Math.min(30, t.getVoto());
          // In questo esempio "30 e lode" vale 30 ai
        //fini della media
      }
      return ((double)somma) / sostenuto.size();
  }
```

In *Straniero.java*:

studenteModello():

```
    /*@ also_ensures \result == true; */
      public boolean studenteModello() {
        return super.studenteModello(0.7);
     }
```

5.7 Considerazioni sul caso d'uso

Riportiamo alcune considerazioni doverose sul caso d'uso preso come test.
Per quanto riguarda Jass, il fatto che usi una propria logica per esprimere le condizioni, invece di JML, può essere visto come una limitazione (anche se, come detto in precedenza, la logica è molto simile). Ad esempio, usando JML, si potrebbe prevedere l'integrazione con altri tool che usino questo standard. Sarebbe interessante testare anche la versione successiva di Jass, per vedere se questo "problema" sia stato risolto.
Nel nostro esempio, la funzionalità più interessante di Jass, è stata quella di produrre codice java, a partire da precondizioni, per gestirle. Questo compito dovrebbe essere a carico del programmatore, quindi si ha una facilitazione nella stesura del codice. Anche in questo caso, una limitazione è data dal fatto che, per far funzionare il precompilatore, le precondizioni devono essere elencate subito dopo la chiamata del metodo, prima di qualsiasi altra dichiarazione. Questo può essere un problema se per esempio, nella dichiarazione della precondizione si deve far rifermento a qualche oggetto da "creare". Riportiamo un esempio per essere più chiari, prendendo spunto dal caso d'uso.

```
public void eliminaLinkPianoDiStudi(TipoLinkPianoDiStudi
l)  {
      if (l==null) throw new EccezionePrecondizioni("Il
link da cancellare non puo' essere null");
      // Controllo che i vincoli sull'evoluzione della
proprieta' non vengano violati dalla cancellazione del
link
      // L'esame non deve essere stato gia' sostenuto
      Iterator it = sostenuto.iterator();
      while (it.hasNext()) {
        Corso c =
((TipoLinkSostenuto)it.next()).getCorso();
        if (l.getCorso() == c)
          throw new EccezionePrecondizioni("Non possono
eliminare questo esame dal piano di studi: e' stato gia'
sostenuto");
}
```

Per quanto riguarda la prima condizione, non ci sono problemi, può essere espressa tramite la formula

```
    /** require
            [il_link_non_può_essere_null] l!=null;
    **/
```

Per la seconda condizione, si presenta un problema: l'oggetto *c* di tipo *Corso*, viene

creato, usando un iteratore e quindi non può essere noto subito dopo l'invocazione del metodo. Quindi non si può esprimere un'eventuale precondizione in jass.
L'inconveniente può essere aggirato, usando una clausola di tipo *check*, nel caso particolare, si può esprimere:

```
/**
check
[Esame_già_Sostenuto_impossibile_eliminarlo_dal_piano_di_
studi]   l.getCorso() !=c;
**/
```

Per permettere al precompilatore di gestire anche le clausole *check*, si deve passare un comando diverso. Non abbiamo riportato il comando specifico per far gestire le precondizioni a Jass, in quanto è "nascosto" al programmatore, invocando il nostro *main* in *ExecutionTools*. Per completezza, riportiamo qui i comandi, dato che nella nostra applicazione non si fa riferimento ai *check*.
Mentre per far gestire le precondizioni, si deve invocare un comando del tipo:

```
java jass.Jass -contract[pre] nomeFile.jass
```

dove *-contract[pre]* indica appunto che si stanno considerando le precondizioni, per far gestire anche i *check*, si deve passare un comando del tipo:

```
java jass.Jass -contract[check] nomeFile.jass
```

Aggiungiamo che si possono anche considerare entrambi i casi insieme, passando:

```
java jass.Jass -contract[pre,check] nomeFile.jass
```

Per quanto riguarda Daikon la limitazione più grande è data dal fatto che il tool può essere eseguito solo su casi "eseguibili", in pratica, che hanno il metodo *main*. Introducendo questo vincolo spesso vengono trovate proprietà vere solo per quel caso di test specifico, il più delle volte anche banali. Per essere chiari, riportiamo un esempio preso dal caso d'uso. Prendendo il *main* da noi scritto

```
    public static void main(String args[]) throws
        EccezionePrecondizioni {
        Studente s3 = new Straniero("003", 27,
        "Romania");

        Corso c1 = new Corso("FondInf", "AAA");
        Corso c2 = new Corso("ProSw", "AAB");
        Corso c3 = new Corso("Analisi1", "AAC");
        Corso c4 = new Corso("Geom1", "AAD");
```

```
        s3.inserisciLinkPianoDiStudi(c1);
        s3.inserisciLinkPianoDiStudi(c2);
        s3.inserisciLinkSostenuto(c1, 28);
        s3.inserisciLinkSostenuto(c2, 27);

        Set studenti = new
                InsiemeArrayOmogeneo(Studente.class);

        studenti.add(s3);

        boolean modello = s3.studenteModello();

        double d = s3.media();

}
```

il tool, come proprietà, trova ad esempio

```
/*@ ensures \result == 27.5; */
```

cioè indica che come postcondizione si deve avere un risultato per la *media*, uguale a 27.5 : vero per il caso di test specifico, ma non in generale. Inoltre, è stato nostro compito aggiungere il metodo *main* nel caso d'uso.

Biblografia:

1. [Pro07] Michele Proni. OOPS: un'applicazione per il supporto alla progettazione e alla realizzazione di software object oriented. Tesi di Laurea in Ingegneria Gestionale a.a. 2006-2007, presso SAPIENZA – Universit`a di Roma, 2007.

2. [DapDiC08] Fabio D'Aprano – Claudio Di Ciccio ReDiA-VeriFInt. Realizzatore Diagrammi Automatico conVerifica Formale Integrata. tesina di Metodi Formali nell'Ingegneria del Software, SAPIENZA – Università di Roma

3. [Cer08] Alberto Cerullo Java Modeling Language (JML)- Tesina del corso "Metodi Formali nell'Ingegneria del Software" - SAPIENZA – Universit`a di Roma – 2008

4. [MS08] Toni Mancini and Monica Scannapieco. Slides del corso di Progettazione del Software per il Corso di Laurea in Ingegneria Gestionale.

   SAPIENZA - Dipartimento di Informatica e Sistemistica, 2008. http://www.dis.uniroma1.it/~tmancini

5. [CM07a] Marco Cadoli and Toni Mancini. Slides del corso di Metodi Formali nell'Ingegneria del Software. SAPIENZA - Dipartimento di Informatica e Sistemistica, 2007. http://www.dis.uniroma1.it/~tmancini

6. Jass sito web ufficiale http://csd.informatik.uni-oldenburg.de/~jass/

7. Daikon sito web ufficiale http://groups.csail.mit.edu/pag/daikon/

8. Java, sito web ufficiale http://www.sun.com/

9. The Java Modeling Language (JML), http://www.eecs.ucf.edu/~leavens/JML/index.shtml

10. An overview of JML tools and applications, Lilian Burdy1, Yoonsik Cheon2, David R. Cok3, Michael D. Ernst4, Joseph R. Kiniry5, Gary T. Leavens6?, K. Rustan M. Leino7, Erik Poll5. http://research.microsoft.com/en-us/um/people/leino/papers/jml-sttt.pdf