Report di Software Testing

Il report presentato è relativo all'attività di *software testing* svolta su due progetti open-source: *Apache Bookkeeper* e *Apache Avro*. Per svolgere tale attività sono stati messi in pratica criteri e metodi di *Category Partition*, *boundary-values*, adeguatezza strutturale e *mutation testing,* appresi durante il modulo di *software testing*. Eventuali riferimenti ai dati dei report di *mutation testing* e *adeguatezza strutturale*, possono essere consultati nei report che si trovano all'interno della cartella "ReportISW2-ModuloSWTesting/Report" nella directory principale del progetto *Bookkeeper*.

Bookkeeper

Bookkeeper è un servizio che permette la memorizzazione persistente di record di log detti *log entries*; un insieme di *log entries* va a costituire quello che viene chiamato *ledger*, ovvero l'unità base di memorizzazione di *Bookkeeper*. Tali *log entries* persistenti sono replicate su vari server chiamati *bookies*, i quali hanno il compito di gestire frammenti di *ledger* in modo distribuito. Le classi scelte per il software testing sono state selezionate a seguito di un compromesso tra le metriche del numero di revisioni e di linee di codice toccate, e una maggiore semplicità per effettuare il testing data la complessità del progetto *Bookkeeper*.

Classe BufferedChannel

E' una classe che ha la responsabilità di fornire un livello di buffering per effettuare le scritture delle *log entries* su file tramite un canale di tipo *File Channel* per renderle persistenti, e di fornire un livello di buffering per leggere tali *log entries* e di scriverle su un nuovo buffer. I metodi che si è deciso di testare per questa classe sono il metodo *write* ed il metodo *read*, proprio perché significativi per il compito che la classe svolge.

1. Category Partition e Boundary Values

1.1 Metodo *read(ByteBuf dst, long pos, int length)*

Dalle poche informazioni a disposizione su questa classe e dalle considerazioni fatte a partire dal significato del metodo e del contesto in cui viene utilizzato, i parametri di input sono sostanzialmente il buffer di destinazione su cui andare a scrivere dati letti dal buffer associato al *BufferedChannel* chiamato *writeBuffer* oppure letti dal file associato al *FileChannel*, la posizione a partire dalla quale si vuole iniziare a leggere e la lunghezza in numero di bytes di quanto si vuole leggere. Per fare la *Category Partition* si è ipotizzato che la capacità del *writeBuffer* da specificare al momento dell'instanziazione del *BufferedChannel*, sia importante per determinare il comportamento del SUT. Infatti, secondo le considerazioni fatte, dichiarare di voler iniziare a leggere dal *writeBuffer* da una posizione che sia maggiore, inferiore o uguale rispetto alla sua capacità, porterà probabilmente ad avere comportamenti del SUT differenti. Anche per il parametro *length* si può ipotizzare che voler leggere un numero di bytes che sia maggiore, inferiore o uguale alla capacità del buffer di destinazione ed al numero di bytes effettivamente leggibili (dato dalla capacità del *writeBuffer* meno la posizione), provocherà differenti comportamenti del SUT. A seguito di queste osservazioni è stata realizzata tale *Category Partition*:

- *writeBuffer_capacity:* {capacità ≥ 1}, {capacità = 0}

  per la creazione di un'istanza di *BufferedChannel* sono disponibili diversi costruttori che permettono anche di specificare per il *writeBuffer* una capacità di lettura e scrittura differenti. Ai fini dell'attività di testing si è deciso di utilizzare per il *writeBuffer* la stessa capacità sia per la scrittura che per la lettura

- *dst:* {istanza nulla}, {*dst_capacity* ≥ 1}, {*dst_capacity* = 0}
- *pos*: {*pos* = *writeBuffer_capacity*}, {*pos* < *writeBuffer_capacity*}, {*pos* > *writeBuffer_capacity*}
- *length*: {*length* ≤ *writeBuffer_capacity* − pos && *length* ≤ *dst_capacity* },
  {*length* > *writeBuffer*_capacity − *pos* || *length* > *dst_capacity* }

1.2 Metodo *write(ByteBuf src)*

Il parametro di input di questo metodo è un oggetto di tipo *ByteBuf* che rappresenta il buffer che contiene i byte dei dati da rendere persistenti e che vengono letti da tale buffer sorgente per essere scritti prima sul *writeBuffer* e poi per essere scritti su file.

La semplice *Category Partition* pensata per questo parametro è sostanzialmente identica a quella del buffer di destinazione del metodo *read*:

- *src:* {istanza nulla}, {*src_capacity* ≥ 1}, {*src_capacity* = 0}

Le istanze non valide di *BufferedChannel* e dei buffer in generale non sono state considerate in quanto non è possibile istanziare un *BufferedChannel* e dei buffer che non siano validi senza incappare nel lancio di qualche eccezione al momento dell'istanziazione; in particolare non è possibile istanziare dei buffer con capacità negativa.
A seguito dell'analisi *boundary-values* effettuata sui parametri per i quali è necessario scegliere dei valori numerici, si sono ottenuti i seguenti valori di input per i casi di test, in modo da coprire tutte le classi di equivalenza e rimanere con un insieme di casi di test minimale:

- *writeBuffer_capacity*: 0, 1, 2
- *pos*: *writeBuffer_capacity* − 1, *writeBuffer_capacity*, *writeBuffer_capacity* + 1
- *src_capacity, dst_capacity*: 0, 1, 2
- *length*: (*writeBuffer_capacity* − *pos*) − 1 && *dst_capacity* − 1
  (writeBuffer_capacity − pos) + 1 || dst_capacity + 1
  (writeBuffer_capacity − pos) && dst_capacity

Per poter eseguire i test, sono stati implementati dei metodi interni alla classe *TestBufferedChannel* che permettono di creare ed inizializzare il buffer *src*, il buffer *dst* ed il *file channel* necessario per l'istanziazione del *BufferedChannel*. E' stato previsto inoltre un metodo *setUpBufferedChannel* al quale è stata associata l'annotazione *@Before* in modo tale che prima dell'esecuzione di ogni test ci sia sempre una nuova istanza di *BufferedChannel*. Sono stati implementati due test: uno per il metodo *read* ed uno per il metodo *write*. Nel test del metodo *read* viene invocato prima il metodo *write* in modo tale che si possa effettuare la lettura a partire da un *writeBuffer* che contiene dati (in particolare gli stessi usati per effettuare la scrittura nel test del metodo *write)*. I test sono stati eseguiti in modo parametrico e si è deciso di illustrarli in modo congiunto per questione di praticità, dato che i casi di test per il metodo *write* coinvolgono un solo parametro.
Il criterio applicato per identificare i casi di test è stato quello unidimensionale, prevedendo quindi un caso di test per coprire ognuna delle classi di equivalenza individuate per i parametri di *write* e di *read*, in modo da ottenere un insieme di casi di test minimale.
L'insieme di casi di test minimale individuato è il seguente:

{*writeBuffer_capacity*: 1, *dst* con capacità: 1, *src* con capacità: 1, *pos:* 0 , *length*: 0 },
{*writeBuffer_capacity*: 0, *dst* con capacità: 0, *src* con capacità: 0, *pos:* 0 , *length*: 1 },
{*writeBuffer_capacity*: 2, *dst* con capacità: 2, *src* con capacità: 2, *pos:* 0 , *length*: 2 },
{*writeBuffer_capacity*: 2, *dst*: null, *src*: null, *pos:* 3 , *length*: 1 }

2.1 Adeguatezza – Criteri strutturali

Relativamente all'adeguatezza dell'insieme di casi di test minimale appena descritto, si è deciso di valutare e migliorare l'adeguatezza di tipo strutturale, andando ad aumentare due metriche di copertura: la *statement coverage* e la *condition coverage*.
Con l'insieme di casi di test minimale presentato prima, per la classe *BufferedChannel* si è ottenuta la seguente copertura:
• *statement coverage*: 68,1%
• *condition coverage*: 50%

Per i metodi invece la copertura ottenuta è la seguente:

- *read:* {*statement coverage*: 79%}, {*condition coverage*: 61%}
- *write:* {*statement coverage*: 74%}, {*condition coverage*: 50%}

A seguito dell'analisi condotta sulla copertura del metodo *write*, ci si è accorti che per aumentare la *coverage* è necessario aggiungere un parametro di input che serve per inizializzare il *BufferedChannel* (secondo uno dei suoi costruttori) e che viene utilizzato nel metodo *write* per gestire il *flush* dei dati dal *writeBuffer* al file. Tale parametro è chiamato *unpersistedBytesBound* ed in particolare è usato nel metodo *write* per rendere persistenti i dati rimasti all'interno del *writeBuffer,* quando è terminato il ciclo di lettura dal buffer sorgente. Se non specificato il valore di default di *unpersistedBytesBound* è pari a 0, perciò per poter avere una *coverage* migliore è stato necessario introdurre un valore almeno pari ad 1.

Sono stati aggiunti due casi di test: uno che forza il flush dei dati e svuota completamente il *writeBuffer* facendo si che la lettura venga effettuata dal file collegato al *FileChannel*, e l'altro che non forza il flush dei dati dal *writeBuffer* facendo si che la lettura dei dati venga effettuata a partire dallo stesso *writeBuffer.*
A seguito dell'introduzione dei nuovi casi di test, per la classe *BufferedChannel* si è ottenuta la seguente copertura:

- *statement coverage*: 81,9%
- *condition coverage*: 71,1%

Per i metodi invece la nuova copertura ottenuta è la seguente:

- *read:* {*statement coverage*: 91%}, {*condition coverage*: 66%}
- *write:* {*statement coverage*: 100%}, {*condition coverage*: 100%}

Notare come per il metodo *write* si è riusciti ad ottenere un'adeguatezza pari ad 1; quindi in questo caso l'insieme di casi di test risulta essere adeguato per il metodo *write*, relativamente alle metriche di copertura scelte. Gli *statement* e le condizioni del metodo *read* che non è stato possibile coprire sono dovute al fatto che ci sono due particolari condizioni che non possono mai sussistere in pratica, per come è stata implementata la classe. La prima condizione è *writeBuffer==null*; tale condizione non può mai verificarsi perché non è possibile istanziare un *BufferedChannel* con un *writeBuffer* nullo, ed inoltre non è neanche possibile impostarlo a *null* in un momento successivo in quanto la classe *BufferedChannel* non fornisce metodi di *set* per l'attributo *writeBuffer*. Dato che questa condizione semplice fa parte della condizione complessa *(writeBuffer == null && writeBufferStartPosition.get() <= pos)*, a causa del fatto che *writeBuffer == null* non può mai verificarsi ed a causa della *lazy evaluation,* anche se i casi *true* e *false* della seconda condizione sono coperti dai casi di test, in realtà risultano non coperti. La seconda condizione che da problemi per la copertura è *readBufferStartPosition > pos*, dove il *readBuffer* è un buffer di appoggio in cui vengono scritti i dati letti dal file associato al *FileChannel* prima di scriverli nel buffer di destinazione *dst*. L'attributo *readBufferStartPosition* è un attributo ereditato dalla classe *BufferedReadChannel* che viene inizializzato al valore *Long.MIN_VALUE,* pari a -9223372036854775808. Durante l'esecuzione del metodo *read* l'attributo *readBufferStartPosition* viene impostato pari al valore corrente di *pos*. Quindi la condizione *readBufferStartPosition > pos* non può mai verificarsi.
Anche la condizione *readBytes* <= 0 non può mai verificarsi. I *readBytes* sono i byte letti dal file e scritti nel *readBuffer*, ed assumono un valore negativo solo quando la *readBufferStartPosition* è maggiore della corrente dimensione del file. Al controllo della precedente condizione ci si arriva quando *pos* < *writeBufferStartPosition* e la *readBufferStartPosition* viene posta uguale al valore di *pos*. Dato che *writeBufferStartPosition* rappresenta la posizione del file pointer del file, e perciò ne rappresenta la corrente lunghezza, non potrà mai risultare che *readBufferStartPosition > writeBufferStartPosition* e quindi non potrà mai verificarsi la condizione *readBytes* <= 0.

2.2 Adeguatezza - Mutation Testing

Il framework PIT per *mutation testing* ha generato in automatico un insieme di 60 mutanti complessivi. Con l'insieme di casi di test ottenuto dai precedenti step di *Category Partition* e adeguatezza strutturale, sono stati uccisi 28 mutanti (i.e. *mutation coverage:* 47%). A seguito della fase di *mutation testing* sono stati uccisi ulteriori 8 mutanti, portando la copertura delle mutazioni al 60%. Queste ulteriori 8 mutazioni sono state rilevate aggiungendo altri 6 test case all'insieme di casi di test ottenuto dopo la fase di aumento dell'adeguatezza strutturale. Di tutti i mutanti sopravvissuti, 16 di questi non è stato possibile raggiungerli con i test case progettati oppure sono andati in *timeout*. Infatti ci sono diversi controlli che all'interno del metodo *read* non vengono effettuati e portano ad un loop infinito il metodo. Proprio a causa di questo motivo ci sono stati dei mutanti che non è stato possibile uccidere, perché per ucciderli sono necessari dei test case che però provocano un loop infinito del ciclo di lettura. Questo è stato ad esempio il caso del mutante generato a linea 266, che per poterlo uccidere sarebbe necessario un caso di test in cui si vada a leggere con più iterazioni dal file associato al *FileChannel*, ma per realizzare tale caso di test è necessario che: s*rc _capacity > dst _capacity*, il flush dei dati sul file sia abilitato esplicitamente oppure implicitamente imponendo che *writeBuffer_capacity < src_capacity*, *pos < writeBufferStartPosition*, *length = src_capacity*. Se queste condizioni sono vere, al primo ciclo di lettura il buffer di destinazione si riempie ed il parametro *length* non viene più decrementato nella seconda iterazione, impedendo così di uscire dal ciclo perché *length* non diventa mai 0. Sono presenti inoltre anche altri mutanti (linea 258 e 284) che non è possibile uccidere. Il mutante a linea 284 non è possibile ucciderlo perché agisce su una condizione che, come già detto in precedenza, non può verificarsi mai (*readBytes* < 0); il mutante a linea 258 invece effettua la mutazione *writeBuffer != null* su una condizione complessa che fa parte di un costrutto *if-else-if*, rendendo tale condizione complessa uguale al quella che viene eseguita precedentemente a linea 245 nella sequenza di

if-else-if. Questo comporta che quando la condizione a linea 245 risulta vera, gli altri rami alternativi non vengono percorsi e perciò la mutazione apportata a linea 258 viene ignorata; quando la condizione a linea 245 risulta falsa, si procede verificando la condizione con la mutazione a linea 258, ma essendo uguali le due condizioni, anche quella con la mutazione risulta essere falsa ed il relativo blocco non viene eseguito. Dato che la condizione a linea 258 risultava essere sempre falsa anche prima della mutazione perché, come già detto in precedenza, la condizione *writeBuffer* == *null* non può mai verificarsi, di conseguenza la mutazione apportata non cambia il comportamento e lo stato del SUT, quindi essa non può essere rilevata. Tutti i mutanti che sono stati raggiunti con i casi di test e che sono rimasti vivi, hanno attuato cambiamenti nel SUT che però non hanno avuto un impatto sul suo stato che sia risultato visibile all'esterno. Questi mutanti ancora vivi sono da considerarsi tutti equivalenti al SUT secondo *strong mutation*, tranne quelli a linea 258 e 284 che sono equivalenti al SUT secondo *weak mutation* in quanto il path di esecuzione e lo stato delle variabili è lo stesso che nel SUT. I mutanti generati che è stato possibile raggiungere sono stati 44 e con l'insieme di casi di test progettato si è riusciti a rivelarne 28, quindi circa il 63%.

Classe DigestManager

Questa classe, relativamente al lato di invio, ha la responsabilità di prendere una *entry*, allegarle un *digest* ed impacchettarla insieme ad altre informazioni, cosicché il pacchetto completo possa essere consegnato ad un *bookie*. Per quanto riguarda il lato ricezione la classe va a controllare se il *digest* ricevuto corrisponde a quello precedentemente inviato e poi preleva la *entry* dal pacchetto. I metodi che si è deciso di testare per questa classe sono *computeDigestAndPackageForSending, verifyDigestAndReturnLac, verifyDigestAndReturnData, verifyDigestAndReturnLastConfirmed*.

1.1 Category Partition e Boundary Values

In generale l'istanza del *DigestManager* viene creata a partire dal metodo *instantiate (long ledgerId, byte[ ] passwd, DigestType digestType, ByteBufAllocator allocator, boolean useV2Protocol)*; i parametri in input a tale metodo (tranne *psswd*) sono importanti per i metodi che si è scelto di testare, quindi anche per essi viene fatta la *Category Partition*. In base alle informazioni ricavate dalla documentazione la *Category Partition* per i parametri di input del metodo *instantiate* è la seguente:

- *ledgerId*: {*ledgerId* ≥ 0}, {*ledgerId* < 0}
- *allocator*: {istanza valida}, {istanza nulla} (un'istanza non valida non è possibile crearla)
- *useV2Protocol*: {true}, {false}
- *digestType*: {HMAC}, {CRC32}, {CRC32C}, {DUMMY}, {istanza nulla}
  dove *DigestType* è definita come un'enumerazione.

1.1 Metodo *computeDigestAndPackageForSending (long entryId, long lac, long length, ByteBuf data)*

Tale metodo computa un *digest* da allegare alla *entry* da inviare, contenuta nel *ByteBuf data*. Dalle informazioni raccolte dalla documentazione si sa che il *lac (last add confirmed*) è l'id dell'ultima *log entry* registrata sul *ledger*, per la cui registrazione è stato inviato un *ack.* Di conseguenza non dovrebbe essere possibile che nuove *entry* abbiano un *id* maggiore al *lac*.
In seguito alle considerazioni fatte la *Category Partition* effettuata è la seguente:

- lac: {lac < 0}, {lac ≥ 0}
- *entryId*: {*entryId* ≤ *lac*}, {*entryId* > *lac*}
- *length*: {*length* < 0}, {*length* > 0}, {*length* = 0}
- *data*: {*data* con capacità ≥ 1}, {*data* con capacità = 0}, {istanza nulla}

1.2 Metodo *verifyDigestAndReturnLac (ByteBuf dataReceived)*

Tale metodo verifica che il *digest* contenuto nel *ByteBuf* ricevuto sia lo stesso inviato precedentemente insieme al *lac*, e ritorna il *lac* estratto dal *ByteBuf* eliminando il *digest*. Il pacchetto ricevuto e contenuto nel buffer può sostanzialmente essere costruito in maniera corretta ed essere quindi valido oppure essere costruito in maniera errata e risultare non valido.
La *Category Partition* effettuata è perciò la seguente:

- *dataReceived*: {istanza valida}, {istanza non valida}, {istanza nulla}

1.3 Metodo *verifyDigestAndReturnData (long entryId, ByteBuf dataReceived)*

Tale metodo verifica che il *digest* contenuto nel *ByteBuf* ricevuto sia lo stesso inviato precedentemente insieme alla *entry* e ad altre informazioni, e ritorna la *entry* più queste informazioni estraendole dal *ByteBuf* eliminando il *digest*. Il pacchetto ricevuto e contenuto nel buffer può sostanzialmente essere costruito in maniera corretta ed essere quindi valido oppure essere costruito in maniera errata e risultare perciò non valido. In questo caso la *Category Partition* è identica a quelle viste precedentemente per il *lac* e per *dataReceive (*nel metodo *verifyDigestAndReturnLac)*.

1.4 Metodo *verifyDigestAndReturnLastConfirmed (ByteBuf dataReceived)*

Tale metodo fa la stessa cosa di *verifyDigestAndReturnLac*, ma usando un meccanismo differente per verificare il *matching* del *digest*. La *Category Partition* è uguale a quella vista precedentemente per il metodo *verifyDigestAndReturnLac*.
A seguito dell'analisi *boundary-values* effettuata sui parametri per i quali è necessario scegliere dei valori numerici, si sono ottenuti i seguenti valori di input per i casi di test, in modo da coprire tutte le classi di equivalenza:

- *lac, ledgerId, length*: −1, 0, 1
- *entryId*: *lac* − 1, *lac*, *lac* + 1
- *data_capacity*: 0, 1, 2

Anche se non contemplate nelle classi di equivalenza, esistono anche le capacità dei buffer di dati da costruire e da dare in input ai metodi che computano i *digest* per poter fare i test dei metodi che effettuano poi la verifica dei *digest*. Tali capacità vengono definite in modo parametrico all'interno dei casi di test, in modo da poter rendere più robusti i test creando buffer di dati con capacità critiche. Per le capacità di questi buffer sono quindi usati gli stessi *boundary-values* definiti per il parametro *data*.
Al fine di poter eseguire i test, sono stati implementati dei metodi interni alla classe *TestDigestManager* che permettono di creare ed inizializzare i buffer di dati da passare in input ai metodi da testare, così da poter effettuare i test in modo parametrico usando sia buffer con dati validi, che buffer con dati non validi. I metodi che costruiscono buffer con dati validi, lo fanno sfruttando i metodi *computeDigestAndPackageForSending* e *computeDigestAndPackageForSendingLac* offerti dalla classe stessa, in quanto non si è riusciti a trovare un altro modo per costruire dei pacchetti validi, se non usando le istruzioni messe a disposizione da tali metodi.
In particolare è possibile notare che esiste anche un metodo che computa *digest* per l'invio del *lac* e che non è stato testato. Questo perché non si è riusciti a trovare un modo per costruire un pacchetto di dati che fosse conforme a quello che viene restituito in output, per poter fare l'*assert* sull'uguaglianza tra i due. Questa difficoltà è stata riscontrata anche per il metodo *computeDigestAndPackageForSending*, per il quale infatti non si effettua l'*assert* sull'intera lista di buffer restituita (*digest* + dati), ma sono sul secondo elemento della lista, il quale deve corrispondere con il buffer di dati passato in input.
I test sono stati eseguiti in modo parametrico e si è deciso di illustrarli in modo congiunto per questione di praticità. Il criterio applicato per identificare i casi di test è stato quello unidimensionale, prevedendo quindi un caso di test per coprire ognuna delle classi di equivalenza individuate per i parametri dei quattro metodi, in modo da ottenere un insieme di casi di test minimale.
L'insieme di casi di test minimale individuato è il seguente:

1. {*ledgerId*: -1, *useV2Protocol*: true, *lac*: -1, *digestType*: HMAC, *entryId*: -2, *length*: -1, *allocator*: istanza valida, *data*: buffer con capacità=0, *dataLacReceived*: istanza valida, *dataReceived*: istanza valida, *dataLastConfirmedReceived:* istanza valida}
2. {*ledgerId*: 0, *useV2Protocol*: false, *lac*: 0, *digestType*: CRC32, *entryId*: 0, *length*: 0, *allocator*: istanza valida, *data*: buffer con capacità=1, *dataLacReceived*: istanza valida, *dataReceived*: istanza valida, *dataLastConfirmedReceived:* istanza valida}
3. {*ledgerId*: 1, *useV2Protocol*: false, *lac*: 1, *digestType*: CRC32C, *entryId*: 2, *length*: 1, *allocator*: istanza valida, *data*: buffer con capacità=2, *dataLacReceived*: istanza valida, *dataReceived*: istanza valida, *dataLastConfirmedReceived:* istanza valida}
4. {*ledgerId*: 0, *useV2Protocol*: false, *lac*: 0, *digestType*: DUMMY, *entryId*: 0, *length*: 0, *allocator*: istanza valida, *data*: buffer con capacità=0, *dataLacReceived*: istanza non valida, *dataReceived*: istanza non valida, *dataLastConfirmedReceived:* istanza non valida}

5. {*ledgerId*: 0, *useV2Protocol*: false, *lac*: 0, *digestType*: null, *entryId*: 0, *length*: 0, *allocator*: null, *data*: null, *dataLacReceived*: null, *dataReceived*: null, *dataLastConfirmedReceived:* null}

2.1 Adeguatezza – Criteri strutturali

Relativamente all'adeguatezza dell'insieme minimale di casi di test appena descritto, si è deciso di valutare e migliorare l'adeguatezza di tipo strutturale, andando ad aumentare due metriche di copertura: la *statement coverage* e la *condition coverage*.

Con l'insieme di casi di test minimale presentato prima, per la classe *DigestManager* si è ottenuta la seguente copertura:
• *statement coverage*: 83,9%
• *condition coverage*: 70,4%

Poiché i metodi *verifyDigestAndReturnData* e *verifyDigestAndReturnLastConfirmed* sono sostanzialmente coperti al 100% in quanto a loro volta invocano il metodo privato *verifyDigest*, per la *coverage* si andrà a considerare tale metodo privato.
Per i metodi la copertura ottenuta è la seguente:

- *computeDigestAndPackageForSending*: {*statement coverage*: 100%}, {*condition coverage*: 100%}
- *verifyDigestAndReturnLac*: {*statement coverage*: 63%}, {*condition coverage*: 66%}
- *verifyDigest*: {*statement coverage*: 58%}, {*condition coverage*: 70%}

Notare come per il metodo *computeDigestAndPackageForSending* si è riusciti ad ottenere un'adeguatezza pari ad 1; quindi in questo caso l'insieme di casi di test risulta essere adeguato per il metodo e relativamente alle metriche di copertura scelte.
Per aumentare la *coverage* sono stati introdotti ulteriori 3 casi di test in cui sostanzialmente si va a costruire dei pacchetti di dati, da passare in input ai metodi di verifica, che sono errati. Tali pacchetti risultano invalidi perché discrepanti rispetto al *ledgerId* e/o all'*entryId* e/o al tipo di *digest* usati per computare il pacchetto inviato.
A seguito dell'introduzione dei nuovi casi di test, per la classe *DigestManager* si è ottenuta la seguente copertura:
• *statement coverage*: 94,6%
• *condition coverage*: 88,9%

Per i metodi che non avevano già la *coverage* pari ad 1, la nuova copertura ottenuta è la seguente:

- *verifyDigestAndReturnLac*: {*statement coverage*: 100%}, {*condition coverage*: 100%}
- *verifyDigest*: {*statement coverage*: 100%}, {*condition coverage*: 100%}

Con solo 3 casi di test in più si è riusciti ad ottenere un'adeguatezza strutturale pari ad 1, dunque l'insieme di casi di test costruito è adeguato per i metodi testati e relativamente alle metriche di copertura scelte.
Durante l'analisi del codice per l'aumento della *coverage*, si è potuto constatare che le classi di equivalenza individuate nella fase di *Category Partition* non sono totalmente rispettate. Infatti molti controlli non vengono effettuati sulle variabili; ad esempio non ci sono controlli sui valori negativi degli *id* e non ci sono controlli relativi al valore dell'*entryId* rispetto al *lac*. Si potrebbe pensare che tali controlli vengano effettuati a monte da altri metodi che invocano quelli che sono stati testati.

2.2 Adeguatezza - Mutation Testing

Il framework PIT per *mutation testing* ha generato in automatico un insieme di 48 mutanti complessivi. Con l'insieme di casi di test ottenuto dai precedenti step di *Category Partition* ed aumento dell'adeguatezza strutturale, sono stati uccisi 38 mutanti (i.e. *mutation coverage:* 79%). Dei 10 mutanti sopravvissuti, 4 non è stato possibile raggiungerli con i casi di test generati. I mutanti totali che è stato possibile raggiungere sono dunque 44. Considerando solo le mutazioni che è stato possibile raggiungere, la percentuale delle mutazioni rilevate è dell'86,4% circa.
I restanti 6 mutanti raggiungibili e sopravvissuti non è stato possibile ucciderli per diversi motivi.
Le mutazioni a linea 107, 108 e 110 non possono essere rilevate in alcun modo perché effettuano dei cambiamenti che sono relativi al primo dei due buffer che vengono restituiti in output dal metodo *computeDigestAndPackageForSending*, il quale contiene la parte di *digest* e di informazioni aggiuntive che vengono allegate alla *entry* che si vuole inviare. Come già detto in precedenza, per problemi implementativi

l'*assert* del test viene effettuato solo sul secondo buffer di output, quindi le mutazioni applicate al primo buffer non influenzano in alcun modo l'esito del test, a prescindere dai valori dei casi di test posti in input.
La mutazione a linea 136 viene effettuata all'interno del metodo *computeDigestAndPackageForSendingLac,* il quale non viene testato, bensì viene solo usato per generare pacchetti di dati validi da dare in input al metodo *verifyDigestAndReturnLac*. Di conseguenza non è possibile costruire un caso di test che uccida un mutante che vive all'interno di un metodo che non è sotto test.
Le mutazioni a linea 161 e 204 sono identiche e sono relative ai metodi che effettuano la verifica del *digest*. Quando un pacchetto da inviare viene computato in maniera corretta, esso è costituito dalla parte di dati da inviare (*lac* oppure la *entry*) che ha una certa lunghezza *L1,* e da una parte che contiene il *digest* ed altre informazioni che ha una lunghezza $L2 = L_i + L_j$. La mutazione va ad agire su una condizione che controlla se $L_i + L_j > L1 + L2$, la quale provoca il lancio di un'eccezione se risulta vera. La mutazione che viene apportata consiste nella seguente modifica: $L_i - L_j > L1 +$ L2. Il problema è che anche se il pacchetto non contenesse dati da inviare ($L1 = 0$), avrebbe lunghezza pari ad *L2* e perciò la condizione rimarrebbe falsa. Quindi usando pacchetti validi non sarà mai possibile costruire un test che fallisca quando tale mutazione viene applicata al SUT. A seguito della mutazione quella condizione potrebbe effettivamente risultare vera, e provocare il fallimento di un test, quando in input viene dato un pacchetto non valido e mal costruito; il problema però è che, dato che il pacchetto non è valido, anche gli altri controlli effettuati nel metodo falliscono e lanciano lo stesso tipo di eccezione che lancerebbe $L_i - L_j > L1 + L2$ qualora risultasse vera. Quindi per far si che il test passi senza mutazione, si è costretti a mettere un *assert* sul lancio dell'eccezione; questo *assert* risulterebbe vero anche quando viene effettuata la mutazione ed il test continuerebbe a passare senza così rilevare la mutazione. Tutti i mutanti che sono stati raggiunti con i casi di test e che sono rimasti vivi, hanno attuato cambiamenti nel SUT che però non hanno avuto un impatto sul suo stato che sia risultato visibile all'esterno. Questi mutanti ancora vivi sono da considerarsi equivalenti al SUT secondo *strong mutation*.

Avro

Avro è una libreria open-source creata per la serializzazione dei dati in maniera indipendente dal linguaggio usato. Essa utilizza il formato JSON per definire tipi e protocolli di dati e serializza i dati in un formato binario compatto. *Avro* è fortemente basato sull'uso dei cosiddetti *Schema*, che sono delle stringhe in formato JSON per la definizione dei vari tipi di dato che possono essere serializzati. Per questo motivo, le classi scelte per il *software testing* sono state selezionate *a* seguito di un compromesso tra le metriche del numero di revisioni e di linee di codice toccate, e la significatività delle classi per il dominio del progetto. Di conseguenza le classi scelte sono state *SchemaCompatibility* e *GenericData*.

Classe SchemaCompatibility

E' la classe che ha la responsabilità di confrontare due *Schema* e sentenziare se questi sono compatibili tra loro oppure no. Per compatibili si intende il fatto che è possibile decodificare tutte le istanze di dato che compongono il secondo *Schema*, a partire dal primo *Schema*.
Per questa classe si è deciso di testare il metodo *checkReaderWriterCompatibilty*, che è il metodo principale con il quale si confrontano due *Schema* per determinarne la compatibilità. Tutti gli altri metodi della classe (o quasi) sono sostanzialmente invocati a partire da tale metodo.

1. Category Partition

Dalla documentazione di *Avro* è stato possibile ricavare le informazioni sugli *Schema* necessarie per effettuare la *Category Partition* per il metodo *checkReaderWriterCompatibilty(Schema reader, Schema writer)*. Tale metodo prende come parametri di input due oggetti di tipo *Schema*: l'oggetto *reader* e l'oggetto *writer*. Gli oggetti di tipo *Schema* hanno due attributi fondamentali necessari per la *Category Partion* e sono l'enumerazione *Type* che indica il tipo di *Schema* e la stringa s*chemaString* che contiene la stringa in formato JSON per la definizione dello *Schema*. Gli *Schema* possono essere di tipo complesso o semplice; per quelli semplici viene definito solo il tipo e la stringa rimane vuota, per quelli complessi viene anche definita la stringa JSON. A partire da queste informazioni, la *Category Partition* effettuata è stata la seguente:

- *reader*: {int}, {long}, {float}, {double}, {boolean}, {null}, {record}, {enum}, {array}, {map}, {union}, {fixed}, {string}, {bytes}, {istanza nulla}

  le classi di equivalenza sono sostanzialmente date dai tipi di *Schema*. Le istanze di *Schema* non valide non sono state considerate in quanto non è possibile istanziare *Schema* che non siano validi

- *writer*: {tipoSchemaReader=tipoSchemaWriter && stringaReader=stringaWriter},
  {tipoSchemaReader≠tipoSchemaWriter || stringaReader≠stringaWriter}, {istanza nulla}

  le classi di equivalenza sono state definite a partire dalla compatibilità dei tipi di *Schema* e delle stringhe che definiscono il formato JSON. Per i tipi semplici le stringhe sono sempre uguali in quanto sono vuote.

Al fine di poter eseguire i test, sono stati implementati dei metodi interni alla classe *TestSchemaCompatibility* ed una classe di utility per la generazione degli *Schema* chiamata *SchemaUtils.* In particolare con la classe di utilità si vanno a definire delle stringhe fissate in formato JSON e si vanno a generare gli *Schema* per poter eseguire i test in modo parametrico. Questi metodi permettono di generare due *Schema* semplici o complessi, con stringa uguale o diversa, in base al caso di test che si vuole effettuare.
Il criterio applicato per identificare i casi di test è stato quello unidimensionale, prevedendo quindi un caso di test per coprire ognuna delle classi di equivalenza in modo da ottenere un insieme di casi di test minimale. L'insieme di casi di test minimale individuato è il seguente:

{tipoSchemaReader: boolean, tipoSchemaWriter: boolean},
{tipoSchemaReader: bytes, tipoSchemaWriter: string},
{tipoSchemaReader: double, tipoSchemaWriter: double},
{tipoSchemaReader: int, tipoSchemaWriter: int},
{tipoSchemaReader: float, tipoSchemaWriter: float},
{tipoSchemaReader: long, tipoSchemaWriter: long},
{tipoSchemaReader: null, tipoSchemaWriter: null},
{tipoSchemaReader: string, tipoSchemaWriter: string},
{tipoSchemaReader: record, tipoSchemaWriter: record && stringReader = stringWriter},
{tipoSchemaReader: array, tipoSchemaWriter: array && stringReader ≠ stringWriter},
{tipoSchemaReader: enum, tipoSchemaWriter: array},
{tipoSchemaReader: fixed, tipoSchemaWriter: fixed && stringReader = stringWriter},
{tipoSchemaReader: map, tipoSchemaWriter: map && stringReader = stringWriter},
{tipoSchemaReader: union, tipoSchemaWriter: union && stringReader = stringWriter},
{SchemaReader: null, SchemaWriter: null}

2.1 Adeguatezza – Criteri strutturali

Relativamente all'adeguatezza dell'insieme minimale di casi di test appena descritto, si è deciso di valutare e migliorare l'adeguatezza di tipo strutturale, andando ad aumentare due metriche di copertura: la *statement coverage* e la *condition coverage*.
Con l'insieme di casi di test minimali presentato prima, per la classe *SchemaCompatibility* si è ottenuta la seguente copertura:

- *statement coverage: 49,8*%
- *condition coverage*: 21,4%

Per il metodo *checkReaderWriterCompatibilty* la copertura è la seguente: {*statement coverage*: 78%}, {*condition coverage*: 66%}.

Analizzando il codice ci si è resi conto che la *Category Partition* fatta inizialmente non rispecchia pienamente ciò che succede nel SUT. Infatti ci sono alcuni schemi che pur essendo di tipo differente, sono tra solo compatibili. A causa di questo, nel SUT vengono fatti molti più controlli sui tipi di *Schema* in diverse condizioni messe in OR tra loro, per poter determinare la compatibilità o meno dei due *Schema*. Inoltre la maggior parte dei casi in cui si hanno *Schema* di tipi differenti non sono stati coperti dall'insieme di casi di test minimale. Quindi per aumentare le suddette *coverage*, sono stati definiti 19 nuovi casi di test per considerare anche i confronti non già presenti nell'insieme di casi di test di partenza. Per aumentare ulteriormente la *coverage* sono state apportate anche delle modifiche ad alcune delle stringhe in formato JSON usate per creare gli *Schema*, in quanto con quelle usate inizialmente c'erano delle istruzioni e delle condizioni che non venivano coperte. Ad esempio è stato introdotto l'*alias* per i *fields* dello *Schema* di tipo *Record*.
A seguito dell'introduzione dei nuovi casi di test, per la classe *SchemaCompatibility* si è ottenuta la seguente copertura:

- *statement coverage: 63*%
- *condition coverage*: 42,8%

Per il metodo *checkReaderWriterCompatibilty* la copertura è la seguente: {*statement coverage*: 78%}, {*condition coverage*: 66%}

Notare che la *coverage* del metodo testato non è migliorata, mentre quella della classe si. Questo perché già in principio il metodo *checkReaderWriterCompatibilty* era quasi del tutto coperto, tranne che per un *branch* di uno *switch case* che viene percorso solo quando si verifica una certa eccezione, per la quale purtroppo non si è riusciti a trovare un caso di test che ne scaturisse il lancio. Avendo il metodo poche istruzioni, quel *branch* non coperto ha un peso maggiore sulla percentuale di copertura ottenuta. Dato che però a partire da tale metodo vengono invocati altri metodi privati, la *coverage* della classe è aumentata.

2.2 Adeguatezza - Mutation Testing

Il framework PIT per *mutation* testing ha generato in automatico un insieme di 12 mutanti complessivi. Con l'insieme di casi di test ottenuto dai precedenti step di *Category Partition* e adeguatezza strutturale, sono stati uccisi 7 mutanti (i.e. *mutation coverage:* 58%).
Le due mutazioni generate alla linea 102 del codice della classe vanno ad agire sulla condizione di controllo relativamente alla corrispondenza tra l'alias dello *Schema reader* ed il nome dello *Schema writer* per la compatibilità degli *Schema*, quando *reader* e *writer* risultano avere nomi diversi. Nelle stringhe considerate fino all'attività di adeguatezza strutturale non erano stati introdotti gli alias sui nomi degli *Schema*, e poiché le stringhe confrontate avevano sempre *fields* differenti, la presenza o meno di alias compatibili non cambiava l'esito del test: i due *Schema* risultavano sempre incompatibili. Quindi per uccidere le due mutazioni sono state introdotte altre due stringhe ad hoc per *Schema* di tipo *Record*; entrambe le stringhe sono uguali ad una delle stringhe *Record* già presente chiamata *record2*, ma hanno un differente nome per lo *Schema*, ed inoltre una delle due definisce un alias per il nome che è uguale al nome dello *Schema* definito dalla stringa *record2*. Usando queste due stringhe sono stati creati due nuovi casi di test che falliscono a fronte di queste due mutazioni. Con l'introduzione di questi due nuovi casi di test è stato possibile uccidere anche un terzo mutante generato a linea 901 e che va ad agire sulla condizione che controlla se due nomi di *Schema* sono uguali. Per tale condizione resa *true* dalla mutazione, tutti i test case hanno esito positivo perché con l'insieme di casi di test precedente anche se i nomi di due *Schema* risultavano uguali, nel caso in cui le stringhe considerate avevano *field* differenti, l'uguaglianza sul nome non cambiava l'esito dei test: i due *Schema* rimanevano non compatibili. Andando invece ad introdurre queste nuove stringhe e questi due nuovi casi di test, anche questa mutazione viene uccisa. Infatti nel caso in cui i due *Record* considerati sono identici, tranne che per il nome, il test si aspetta come risultato atteso l'incompatibilità degli *Schema*, ma se la mutazione introdotta rende *true* il confronto sui nomi, i due *Schema* risultano compatibili ed il test fallisce.
I restanti due mutanti sopravvissuti vanno ad introdurre delle mutazioni che non hanno un impatto sul comportamento del SUT relativamente ai test eseguiti in quanto vanno ad agire sulla lista delle incompatibilità che viene restituita quando due *Schema* sono incompatibili, ma i casi di test che sono stati progettati vanno esclusivamente a controllare se i due *Schema* sono compatibili o meno, senza analizzare la lista di incompatibilità eventualmente restituita. Quindi possiamo considerare questi mutanti come equivalenti al SUT secondo la *strong mutation*.

Classe GenericData

E' la classe che contiene utilities per trattare dati generici Java. In particolare permette di generare dati generici di tipo *Record, Mappe, Array*, etc. a partire da uno *Schema* del relativo tipo, riempiendo eventualmente i *field* con valori di default se sono presenti nella stringa che definisce lo *Schema*, altrimenti li lascia vuoti. Altre utilities importanti che mette a disposizione sono due: una permette di verificare che un dato oggetto abbia un matching con un certo *Schema*, l'altra permette di verificare che due oggetti, in accordo con un certo *Schema*, abbiano la stessa implementazione. Queste due ultime utilities sono state scelte per il software testing. La prima utility è implementata dal metodo *validate*, mentre la seconda è implementata dal metodo *compare*.

1. Category Partition

1.1 Metodo *validate(Schema schema, Object datum)*

I parametri di input di questo metodo sono un oggetto *schema* di tipo *Schema* ed un oggetto di tipo *Object* che chiamiamo *datum1*. Come visto per la classe *SchemaCompatibility*, per l'oggetto *Schema* le classi di equivalenza sono date dai vari tipi di schema e dall'istanza nulla; per quanto riguarda l'oggetto *Object*

invece, il comportamento del SUT si assume che vari in base al fatto che tale oggetto sia o meno conforme allo *Schema* di input. Perciò le classi di equivalenza individuate sono le seguenti:

- *schema*: {int}, {long}, {float}, {double}, {boolean}, {null}, {record}, {enum}, {array}, {map}, {union}, {fixed}, {string}, {bytes}, {istanza nulla}

- *datum1:* {conforme allo *schema*}, {non conforme allo *schema*}, {istanza nulla}

1.2 Metodo *compare(Object o1, Object o2, Schema s)*

Il metodo *compare* pubblico non fa altro che invocare il metodo *compare* protetto, quindi in realtà il metodo su cui i test vengono effettuati è quello protetto.
I parametri di input del metodo *compare* pubblico sono un oggetto *s* di tipo *Schema* che chiamiamo *schema,* e due oggetti di tipo *Object* che chiamiamo *datum1* e *datum2*. Le assunzioni sulle classi di equivalenza fatte per i parametri di input del metodo *validate* continuano ad essere valide anche per i parametri *schema* e *datum1* del metodo *compare*. Per quanto riguarda invece il parametro *datum2* si assume che il comportamento del SUT vari in base al fatto che tale oggetto sia o meno conforme allo *schema* e/o sia uguale al *datum1*.
Quindi le classi di equivalenza individuate per datum2 sono le seguenti:

- *datum2*: {conforme allo *schema &&* datum2=datum1}, {conforme allo *schema* && *datum2≠datum1*}, {non conforme allo *schema*}, {istanza nulla}

Al fine di poter eseguire i test, sono stati implementati dei metodi interni alla classe *TestGenericData* e due classi di utility: una per la generazione degli *Schema* chiamata *SchemaUtils* (la stessa usata per i test della classe *SchemaCompatibility*), ed un'altra per la generazione dei vari tipi di *datum* complessi (*record, union, fixed,..)* chiamata *CreateDatumUtils*.
I test sono stati eseguiti in modo parametrico e si è deciso di illustrarli in modo congiunto dato che i parametri di input del metodo *validate* costituiscono un input parziale per il metodo *compare.*
Il criterio applicato per identificare i casi di test è stato quello unidimensionale, prevedendo quindi un caso di test per coprire ognuna delle classi di equivalenza individuate per i parametri di *validate* e di *compare*, in modo da ottenere un insieme di casi di test minimale.
L'insieme di casi di test minimale individuato è il seguente:

{tipoSchema: int, datum1: 1, datum2: 1}, {tipoSchema: float, datum1: 1.0, datum2: 1.1},
{tipoSchema: long, datum1: 1, datum2: 1}, {tipoSchema: double, datum1: 1.0, datum2: 1.1},
{tipoSchema: string, datum1: "c", datum2: "c"},
{tipoSchema: bytes, datum1: ByteBuffer di capacità 1, datum2: ByteBuffer di capacità 2},
{tipoSchema: boolean, datum1: true, datum2: true},
{tipoSchema: fixed, datum1: array di byte di capacità 0, datum2: array di byte di capacità 1},
{tipoSchema: record, datum1: record di due stringhe {"joe", "black"}, datum2: simbolo enumerazione "ONE" },
{tipoSchema: mappa, datum1: 1, datum2: mappa <String, int> con 2 chiavi e 3 valori totali},
{tipoSchema: enum, datum1: simbolo enumerazione "ONE", datum2: simbolo enumerazione "TWO"},
{tipoSchema: array, datum1: array list con valori {1, 2, 3}, datum2: array list con valori {1, 2, 4 },
{tipoSchema: union, datum1: record di due campi {null, 23}, datum2: record di due campi {1, 23} },
{tipoSchema: null, datum1: null, datum2: null}, {Schema: null, datum1: 1, datum2: "c"}

2.1 Adeguatezza – Criteri strutturali

Relativamente all'adeguatezza dell'insieme minimale di casi di test appena descritto, si è deciso di valutare e migliorare l'adeguatezza di tipo strutturale, andando ad aumentare due metriche di copertura: la *statement coverage* e la *condition coverage*.
Con l'insieme di casi di test minimali presentato prima, per la classe *GenericData* si è ottenuta la seguente copertura:
• *statement coverage*: 18,9%
• *condition coverage*: 15,5%

Per i metodi invece la copertura ottenuta è la seguente:

- *validate:* {*statement coverage*: 75%}, {*condition coverage*: 65%}
- *compare* (metodo protetto)*:* {*statement coverage*: 65%}, {*condition coverage*: 47%}

A seguito dell'analisi del codice condotta sui due metodi *validate* e *compare*, sono stati introdotti 20 nuovi casi di test e si è modificato un caso di test appartenente all'insieme generato a partire dalla *Category Partition*. Con questi nuovi casi di test si è andati a coprire quasi tutti i possibili confronti tra tipi di *Schema* e gli oggetti che sono effettuati dai due metodi, molti dei quali non venivano considerati con il solo insieme di casi di test di partenza. Ci sono state però delle comparazioni che non si è riusciti a coprire. In particolare, relativamente al confronto tra *datum* di tipo *Mappa* all'interno del metodo *compare,* non è stato possibile coprire il caso in cui i due *datum* sono mappe identiche; questo perché il metodo *compare* pubblico che è stato testato, a sua volta invoca il metodo *compare* di tipo protetto a cui in input passa un valore booleano fissato a *false*. Quando questo parametro booleano è impostato al valore *false*, il confronto tra i due *datum* di tipo *mappa* non viene effettuato ed indietro viene sempre restituita un'eccezione.
Analizzando nel dettaglio il codice si è notato inoltre che le condizioni con le quali è stata definita la classe di equivalenza {conforme allo *schema* && *datum2≠datum1*}, relativa all'oggetto *datum2,* non sono valide per tutti i tipi di *datum*. Infatti per i *datum* che sono conformi ad uno *Schema* di tipo semplice (i.e. definito senza stringa JSON), non viene controllato se effettivamente questi sono conformi o meno allo *Schema* passato come parametro di input, bensì viene solo fatta tra la comparazione tra i due *datum*. Se i due *datum* sono uguali viene restituito *true*, anche se non sono conformi allo Schema, altrimenti *false.*
A seguito dell'introduzione dei nuovi casi di test, per la classe *GenericData* si è ottenuta la seguente copertura:
• *statement coverage*: 29,4%
• *condition coverage*: 28,3%

Per i metodi invece la nuova copertura ottenuta è la seguente:

- *validate:* {*statement coverage*: 98%}, {*condition coverage*: 97%}
- *compare* (metodo protetto)*:* {*statement coverage*: 88%}, {*condition coverage*: 84%}

Anche se è aumentata, la *coverage* complessiva della classe rimane comunque bassa in quanto sono stati testati solo due metodi, i quali sostanzialmente non invocano (o quasi) altri metodi all'interno della classe. Relativamente ai metodi testati invece la copertura è aumentata di molto ed si avvicina molto al 100%.

2.2 Adeguatezza - Mutation Testing

Il framework PIT per *mutation testing* ha generato in automatico un insieme di 346 mutanti complessivi. Con l'insieme di casi di test ottenuto dai precedenti step di *Category Partition* e adeguatezza strutturale, sono stati uccisi 100 mutanti (i.e. *mutation coverage:* 29%). La maggior parte delle mutazioni generate non sono state rilevate a causa del fatto che sono state generate su parti del SUT non raggiungibili dai test case progettati, in quanto essi coprono solo una piccolissima parte della classe essendo relativi a solo 2 metodi. I mutanti totali che è stato possibile raggiungere sono 137; 100 di questi mutanti sono stati uccisi (quindi il 73% circa).
A seguito della fase di *mutation testing* sono stati uccisi ulteriori 12 mutanti, portando la copertura delle mutazioni al 32%. Queste ulteriori 12 mutazioni sono state rilevate aggiungendo altri 7 test case all'insieme di casi di test ottenuto dopo la fase di aumento dell'adeguatezza strutturale. I nuovi test case vanno sostanzialmente ad introdurre ulteriori combinazioni di confronti tra i tipi di *Schema* ed i *datum* costruiti; tali confronti sono principalmente relativi a tipi di *Schema* semplici, per i quali precedentemente sono stati effettuati quasi solo test case base progettati a partire dalla *Category Partition*. In particolare i nuovi casi di test sono relativi al metodo *validate* e vanno ad effettuare ulteriori validazioni di conformità tra uno Schema ed un *datum*, quando quest'ultimo è sostanzialmente non conforme allo *Schema*.
Tutti i mutanti che sono stati raggiunti con i casi di test e che sono rimasti vivi, hanno attuato cambiamenti nel SUT che però non hanno avuto un impatto sul suo stato che sia risultato visibile all'esterno. Questi mutanti ancora vivi sono da considerarsi equivalenti al SUT secondo *strong mutation*.

Giorgia Francazi
mat. 0268098