Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Property-based Testing per la Programmazione Orientata ai Servizi

Relatore:
Chiar.mo Prof.
Saverio Giallorenzo

Presentata da:
Luca Gabellini

II Sessione
Anno Accademico 2023/24

Sommario

In considerazione del ruolo centrale delle architetture a microservizi nella progettazione dei moderni sistemi software, e la crescente necessità di soluzioni ad-hoc per la loro verifica sia a livello statico che dinamico, viene presentato un adattamento di un approccio automatizzato, già noto come *property-based testing*, per i microservizi. A partire dallo studio di alcuni casi di successo nei linguaggi di programmazione general-purpose, viene analizzato il problema nei suoi aspetti concettuali; segue una proposta implementativa nel contesto di un linguaggio *service-oriented* scelto.

Indice

Elenco delle figure

Elenco dei programmi

Elenco delle tabelle

Introduzione

La programmazione orientata ai servizi è, dalla diffusione di Internet, uno degli standard centrali alla realizzazione di sistemi software. Risponde alla necessità di costruire applicazioni non monolitiche ma fortemente distribuite, dove un insieme di componenti interconnessi debbono interagire attraverso opportuni protocolli di comunicazione. Denominiamo questi componenti *servizi*, e possiamo tracciarne le caratteristiche fondamentali nel *loose coupling* e nell'*incapsulamento* del proprio stato e logica interna; la combinazione di queste caratteristiche li rende sviluppabili e dislocabili in maniera indipendente (con il massimo grado di libertà di scelta in termini di tecnologie, linguaggi e hardware), tuttavia sempre sufficientemente flessibili per la composizione di architetture software complesse. Il principio secondo cui un servizio dovrebbe provvedere a risolvere un unico problema, e ridurre le proprie funzionalità all'essenziale a questo scopo, ha portato alla diffusione della terminologia *microservizi*, e di *micro-service architectures*, o MSA; la diffusione di queste è stata accompagnata da sviluppi nell'ambito dei linguaggi di programmazione, fornendo ai programmatori opportune astrazioni per esprimersi in maniera "service-oriented": illustriamo uno di questi linguaggi, denominato Jolie, nella sezione 1.1.1.

Una delle fasi dello sviluppo software più inficiate dallo stile service-oriented è il *testing*, proprio a causa della grande complessità logica e di coordinamento che un ipotetico test tra multipli servizi si trova a gestire: questo difetto ha motivato ricerca e sviluppo di framework di test (quali JoT, vedi sezione 1.1.2) che si facciano carico degli aspetti più "ingombranti"

del testing per MSA, e permettano invece al programmatore di focalizzarsi più precisamente sul dominio del problema.

Considerato quanto detto, dunque, il nostro scopo è raffinare ulteriormente questi strumenti nell'ottica di accorciare la distanza tra lo sviluppo service-oriented e le tecniche di verifica formale per il software, imbracciando una modalità di testing detta *property-based* (PBT). Ad alto livello, questo approccio è fondato sul mappare le specifiche tecniche di un programma direttamente nel codice, esprimendole sotto forma di test generali su ogni possibile input; nel concreto, questa generalizzazione è ottenuta generando in maniera automatizzata un vasto insieme di *test-cases*. Illustriamo il property-based testing nella sezione 1.2.1, tracciando il suo percorso evolutivo attraverso i linguaggi di programmazione, con riferimento particolare ad Haskell e Java.

Fatto ciò, cercheremo di correlare il paradigma service-oriented al PBT negli aspetti che lo interessano (sezione 2.1), per poi discutere alcuni percorsi percorribili per un'ipotetica implementazione in Jolie. Per completare l'argomentazione, nella sezione 2.2 forniamo – attraverso un catalogo di esempi – un suggerimento implementativo concreto basato su separazione tra la logica di test (espressa in Jolie) e la componente di feeding di dati (realizzata tramite codice Java prodotto da un compilatore).

Capitolo 1

Prerequisiti

In questo capitolo introduttivo vogliamo presentare al lettore alcuni concetti preliminari, quali il paradigma di programmazione orientato ai servizi – con particolare riferimento al linguaggio Jolie – e una metodologia di testing automatizzato per il software detta *property-based testing*.

Presentiamo inoltre, a cavallo tra questi due argomenti, il framework JoT per il testing in Jolie, su cui ci baseremo successivamente per la progettazione di un engine di test property-based per il linguaggio.

1.1 Il paradigma orientato ai servizi

La diffusione di Internet e delle economie digitali ha portato a una crescita d'utilizzo della programmazione distribuita, ovvero la realizzazione di programmi concorrenti pensati per l'esecuzione su macchine distribuite fisicamente. Emerge quindi l'esigenza di trovare meccanismi adeguati alla comunicazione tra processi in un contesto dove la memoria non è più condivisa, e dove i programmi devono essere in grado di operare flessibilmente tra di loro mantenendo però autonomia nelle loro caratteristiche interne (tecnologiche, linguistiche, etc...). Il paradigma orientato ai servizi risponde a questa esigenza, proponendo una astrazione – il *servizio* – pensato attorno a queste problematiche.

Un servizio è un processo indipendente che nasconde la sua rappresentazione interna, e offre un insieme di operazioni accessibili tramite *message passing*. L'aspetto esteriore di un servizio è quello della sua *interfaccia*; l'aspetto interiore è costituito dal comportamento che implementa e dallo stack tecnologico che utilizza [3]. La comunicazione è fondata sull'uso di adeguati standard di comunicazione (su tutti i livelli dello stack di rete, come TCP / IP, HTTP, JSON, etc...) e protocolli. Questo tipo di costruzione serve a garantire interoperabilità tra i servizi nonostante la debole associazione tra essi – in gergo, *loose coupling* – che li rende indipendenti per l'esecuzione o il deployment.

Nel contesto moderno di service-orientation, tipicamente ci si riferisce a *microservizi* e *microservice architectures*, o MSA. Il prefisso "micro" vuole enfatizzare la costruzione "a grana fine" dei servizi, riducendo all'essenziale il quantitativo di funzionalità a cui provvedono. Di qui in avanti, utilizzeremo la terminologia "servizio" sempre nell'ottica di un contesto di MSA.

Per illustrare al meglio gli aspetti linguistici del paradigma, introduciamo ora un linguaggio di programmazione orientato ai servizi, denominato Jolie.

1.1.1 Jolie: un linguaggio orientato ai servizi

Jolie è un linguaggio di programmazione interpretato studiato attorno alla programmazione di (micro)servizi; offre una serie di costrutti linguistici utili ad affrontare le principali problematiche riguardanti il paradigma. Di seguito, illustriamo gli aspetti di Jolie interessanti ai nostri scopi.

Elementi base

Un programma in Jolie ha questo aspetto:

```
1 // Some data types
2 type GreetRequest { name:string }
3 type GreetResponse { greeting:string }
4
```

```
// Define the API that we are going to publish
interface GreeterAPI {
    RequestResponse: greet( GreetRequest )( GreetResponse )
}

service Greeter {
    execution: concurrent // Handle clients concurrently

    // An input port publishes APIs to clients
    inputPort GreeterInput {
        location: "socket://localhost:8080" // Use TCP/IP
        protocol: http { format = "json" }    // Use HTTP
        interfaces: GreeterAPI     // Publish GreeterAPI
    }

    // Implementation (the behaviour)
    main {
        /*
        This statement receives a request for greet,
        runs the code in { ... }, and sends response
        back to the client.
        */
        greet( request )( response ) {
            response.greeting = "Hello, " + request.name
        }
    }
}
```

Programma 1.1: Un esempio di servizio in Jolie

L'esempio illustra chiaramente la netta distinzione linguistica in Jolie tra i principali elementi costitutivi di un programma orientato ai servizi, che classifichiamo tra:

- *Application Programming Interface* – o API – che specificano *quali* funzionalità il servizio offra. Il costrutto Jolie per le API è l'*interfaccia*: una collezione di operazioni caratterizzate da un nome, il pattern di

comunicazione che rispetta, che può essere:

- `RequestResponse`, dove il servizio invia una richiesta ad un altro e rimane in attesa di una risposta;
- `OneWay`, dove la comunicazione è di tipo asincrono e il servizio, al contrario, non attende risposta.

e dei tipi, che descrivono la struttura dei messaggi che l'operazione attende ed eventualmente restituisce.

- *Access Points* (o *porte*, in Jolie) che descrivono *dove* e *come* interagire con l'API; completano la descrizione del contratto pubblico del servizio con la sua locazione – sotto forma di URI –, e i protocolli e tecnologie associati (nell'esempio, JSON e HTTP).
- *comportamento*, ovvero la logica interna del servizio (visibile nell'esempio all'interno del blocco `main`).

Sistema di tipi

Jolie è un linguaggio tipato dinamicamente; le variabili non debbono essere dichiarate in precedenza e il loro tipo è inferito automaticamente dall'uso nella parte comportamentale dei servizi [9]. Nel verificare la conformità dei messaggi alle interfacce, invece, si esercita un controllo più rigido, facendo type-checking sia prima dell'esecuzione per i dati in entrata che dopo per quelli in uscita. Il type-checking segue un approccio strutturale chiuso, ovvero il sottotipaggio deve essere esplicito. La sintassi dei tipi scalari è la seguente:

```
1 T :: { void, bool, int, long, double, string, raw, any}
```

Programma 1.2: Sintassi dei tipi scalari in Jolie

Questi possono a loro volta essere raffinati per restringere il dominio dei valori validi:

```
type shortStrings: string( length[2, 4] )
```

Programma 1.3: Un tipo raffinato in Jolie

Tipi complessi possono essere espressi come composizione di altri tipi in una struttura ad albero:

```
type Coordinates: void {
    lat: string
    lng: string
}
```

Programma 1.4: Un tipo complesso in Jolie

In effetti, Jolie struttura ogni suo tipo di dato ad albero in modo che possano essere facilmente convertiti nei formati standard di comunicazione più comuni, quali JSON o XML. Ciascuno dei nodi dell'albero è un vettore dotato di cardinalità (se omessa, il vettore ha cardinalità a 1):

```
type ListOfInts {
    list[1, *]: int
}
```

Programma 1.5: Un tipo con cardinalità esplicita in Jolie

Embedding di servizi

L'*embedding* è un meccanismo utile al fine di creare composizioni di servizi; permette di lanciare un servizio, denominato *embedded*, all'interno di un altro, detto *embedder*. L'embedder quindi può sfruttare le operazioni dell'altro servizio come subroutine per le sue stesse operazioni. Un caso d'uso tipico è per l'invocazione dei servizi della Standard Library:

```
1 from console import Console
2
3 service HelloWorld {
4     embed Console as Console
5
6     main {
7         print@Console( "Hello World!" )()
8     }
9 }
```

Programma 1.6: Embedding di un servizio in Jolie

Alla riga 7 del programma 1.6 invochiamo l'operazione `print` al port di output `Console`, che fa riferimento al rispettivo servizio embedded. La sintassi di chiamata è la seguente:

```
1 operation_name@OutputPort_Name( request )( response )
```

Programma 1.7: Sintassi per un'invocazione di operazione a un servizio embedded in Jolie

Jolie supporta altri meccanismi che concernono la programmazione architetturale di servizi, – come aggregazioni, *courier*, etc... – che omettiamo in quanto non utili ai nostri fini. Per un approfondimento a riguardo si rimanda alla relativa sezione nella documentazione Jolie.

Integrazione con il linguaggio Java e JavaService

Jolie offre diversi costrutti di controllo sequenza utili all'implementazione del comportamento di servizi, quali operatori per l'esecuzione sequenziale, concorrente, o *input-guards*; tuttavia, permette anche di implementare servizi in Java estendendo la classe `JavaService` inclusa nelle distribuzioni Jolie. Molti servizi della Standard Library di Jolie sono realizzati proprio in questa maniera: il programma 1.8 mostra una versione semplificata dell'implementazione servizio `Console`.

```
1  import jolie.runtime.JavaService;
2
3  public interface ConsoleInterface {
4      void println( String s );
5  }
6
7  public class Console extends JavaService implements ConsoleInterface {
8      public void println( String s ) {
9          System.out.println( s );
10     }
11 }
```

Programma 1.8: Implementazione del servizio `Console` di Jolie come `JavaService`

Sessioni

Con la terminologia *sessione*, si intende in Jolie una conversazione aperta tra due o più servizi. Tipicamente, una sessione è avviata quando un messaggio viene ricevuto in una operazione request-response, e termina all'invio della risposta. Non forniamo una loro descrizione tecnica dettagliata; tuttavia, ci interessa accennare alla caratterizzazione *stateful* delle sessioni.

Normalmente, invocazioni successive di operazioni possono considerarsi indipendenti: nonostante ciò possono essere presenti *side-effect* nella manipolazione di certi oggetti, quali *variabili globali* e *correlation set*. Le prime sono uno stato condiviso tra i servizi; le seconde rappresentano una generalizzazione di Jolie dei *sessions identifiers*.

Ci focalizziamo sulla presenza della nozione di stato in Jolie in vista della presentazione dello *stateful testing*, un approccio al testing fortemente legato e spesso complementare a quello property-based.

1.1.2 JoT: un framework per il testing di MSA

Uno degli aspetti più penalizzati dallo stile a microservizi è il testing. Infatti, a un test che coinvolge più servizi spetta implementare una logica

di coordinazione potenzialmente molto complessa, tenendo in considerazione tecnologie differenti su diversi layer di comunicazione. Ancora più complicato è riconfigurare l'intero sistema di testing per il riutilizzo sotto altre configurazioni tecnologiche, come ad esempio per il deployment [4].

Il framework di test JoT – per Jolie Testing – mira a ridurre la quantità di lavoro necessario per il testing in Jolie, occupandosi autonomamente di gestire le problematiche elencate sopra. Un test in JoT è a sua volta un servizio (e quindi definito come un normale servizio in termini di interfacce, access points e comportamento), che agisce da orchestratore. L'uso di file di configurazione permette di parametrizzare la definizione degli *output port* sulle tecnologie utilizzate e gli indirizzi dei servizi da contattare, secondo il principio di *agnosticismo tecnologico*.

Un'operazione di test in JoT ha il seguente aspetto:

```
1 ///@Test
2 someTest()() throws AssertionError( string )
```

Programma 1.9: Un'operazione di test in JoT

Facciamo notare l'utilizzo del meccanismo delle *annotazioni* – analogamente a JUnit per Java, ad esempio – per la definizione dei test. L'annotazione `@Test`, in particolare, indica a JoT che l'operazione ha valenza di test. Esistono inoltre annotazioni per la definizione del *lifecycle* di test presenti normalmente anche in altri linguaggi, come `@BeforeAll`, `@BeforeEach`, etc... Per riportare un fallimento di test, si utilizza il meccanismo delle eccezioni, analogo (negli elementi essenziali) ad altri linguaggi.

1.2 Software testing e property-based testing

Lo scopo del *software testing* è verificare certe proprietà della semantica di un programma; in altri termini, valida una data implementazione rispetto alle sue specifiche [2]. Tradizionalmente i test sono definiti "basati su esempi", o *example-based*: si istanzia uno scenario concreto e si osserva il

comportamento del software su quello scenario. Il problema concettuale di questo approccio è che usa dei casi *particolari* per fare affermazioni *generali* sul comportamento del software [13]. Per questo, introduciamo un approccio alternativo che mira a correggere questa debolezza, detto property-based testing.

1.2.1 Elementi base del property-based testing

Nel *property-based testing*, o PBT, si definiscono proprietà generali che debbono valere per un programma. Queste vengono validate su un insieme di casi di test generati secondo un processo statistico; l'assunzione implicita in questo processo è che il campione scelto fornisca una buona approssimazione di un test "per ogni input", andando a coprire l'intero spettro di comportamenti del codice.

In questo modo, il PBT mira a colmare il divario tra metodi di testing tradizionali e metodi formali di verifica del software; il programmatore è incentivato ad essere più esplicito nella specifica di un programma, esprimendola direttamente nel codice sotto forma di proprietà, e in un test dal forte valore documentativo.

Con *proprietà* si intende un'invariante o post-condizione che ci aspettiamo valga per un programma, data una certa precondizione [12].

```
1 forall list l
2 reverse(reverse(l)) == l
```

Programma 1.10: Una proprietà su liste

La prima implementazione di nota del PBT è dovuta a *QuickCheck*, una libreria Haskell. Da allora ha visto diffusione su una vasta gamma di linguaggi, come nella libreria *jqwik* per Java e *Hypothesis* per Python.

Di qui in avanti, salvo specificato altrimenti, utilizziamo in maniera interscambiabile le terminologie *test* e *proprietà*, assunto che in un contesto PBT

una proprietà è codificata tramite un test e viceversa un test ha ovviamente valenza di proprietà.

1.2.2 Shrinking

Un problema insito nella generazione casuale di valori è la mancanza di controllo sulla scelta del valore falsificante. E' quindi possibile riscontrare scarsa correlazione tra l'errore verificatosi e il dato che lo causa: questo è fortemente dannoso per la comprensione del problema.

La soluzione standard a questo fenomeno è detta *shrinking*: in caso di fallimento del test, si "riduce" l'input responsabile ad un valore falsificante considerato più semplice o più piccolo. L'idea è che, in questa maniera, si ottenga un dato dalla struttura "essenziale" che perciò descriva meglio l'errore. La definizione di valore "più semplice", ovviamente, non è univoca, e la scelta spetta in ultimo luogo all'algoritmo usato per lo shrinking.

Shrinking type-based ed integrato

Come vedremo, in QuickCheck lo shrinking è definito esclusivamente sulla base del *tipo*: l'algoritmo opera ugualmente per ogni abitante dello stesso tipo, indipendentemente da come questo sia stato generato.

Ad oggi è considerata un'implementazione datata: nelle librerie moderne si preferisce un approccio detto shrinking *integrato nella generazione*. L'osservazione alla base è che il dominio di generazione e il dominio di shrinking in realtà coincidono, per cui è possibile restringere l'azione di shrinking solo ai valori generabili. Mostriamo la differenza con un esempio pratico:

```
1 even_numbers = integers().map(lambda x: x * 2)
2
3 @given( even_numbers )
4 def test_even_numbers_are_even( n ):
5     assert n % 2 == 0
6     assert n <= 4
```

Programma 1.11: Un esempio di test in Hypothesis

Il programma 1.11 illustra un esempio triviale di test scritto in Python, con l'ausilio della sua principale libreria di PBT, Hypothesis. Alla riga 6 è presente un'asserzione che, artificiosamente, causa il fallimento del test per ogni $n >= 5$; tuttavia, le due modalità di shrinking restituiscono valori "minimi" ben diversi:

- per lo shrinking integrato, $n = 6$: infatti, vengono considerati solo gli interi pari generati dalla funzione `even_numbers`;
- per lo shrinking type-based, $n = 1$: infatti, 1 è un valore dispari, e la prima asserzione viene invalidata.

Come si può vedere, in questo caso lo shrinking type-based svia dalla comprensione del problema, perchè riduce l'errore originale ad un errore di altro tipo.

1.2.3 Stateful testing

Storicamente, per le sue radici nel mondo della programmazione funzionale, il property-based testing è strettamente legato al testing di funzioni *pure*. Con funzioni pure si intendono funzioni deterministiche e prive di *side-effects*, ovvero non provocano alcuna mutazione di stato. Il vantaggio sta nel fatto che, data una funzione, due diverse chiamate sullo stesso input restituiranno lo stesso output. Questo rende più semplice ragionare su di esse e, di conseguenza, esprimerne proprietà [1].

Con la diffusione di librerie per il PBT in linguaggi imperativi o object-oriented, molte di queste hanno iniziato a proporre soluzioni per lo *stateful testing* di componenti software in maniera property-based.

Lo stato dell'arte dello stateful testing nelle librerie di PBT consiste nel modellare un sistema come un *automa a stati finiti*, con le dovute astrazioni per descrivere stati e transizioni [11]. Vedremo concretamente un esempio di realizzazione nella libreria jqwik per Java.

1.2.4 Approcci al property-based testing nei linguaggi

Di seguito presentiamo nei punti essenziali due librerie celebri di property-based testing; un'implementazione "storica" per il paradigma funzionale ed una, invece, più vicina al moderno stato dell'arte per il paradigma object-oriented. L'idea è quella di stabilire dei riferimenti pratici su cui basare la nostra proposta di PBT per Jolie.

Nel paradigma funzionale: QuickCheck

La libreria QuickCheck per Haskell ha introdotto gran parte dei concetti fondamentali per la progettazione di librerie per il property-based testing. Il suo design è stato fortemente influenzato dalla natura funzionale di Haskell.

Una proprietà, in QuickCheck, è una funzione Haskell, denotata dal prefisso `prop_`. Ad esempio, il programma 1.12, mostra il corrispondente del test 1.10 in QuickCheck, su liste di interi[1]:

```
1 prop_reverse :: [Int] -> Bool
2 prop_reverse l = reverse( reverse l ) == l
```

Programma 1.12: Una proprietà in QuickCheck

QuickCheck utilizza la typeclass `Arbitrary` per rappresentare un tipo di cui siamo in grado di generare abitanti:

```
1 class Arbitrary a where
2     arbitrary :: Gen a
3     shrink :: a -> [a]
```

Programma 1.13: La typeclass `Arbitrary` in QuickCheck

dove:

- `arbitrary` è un generatore per il tipo `a`;

[1] Il motivo per cui abbiamo specificato una istanza concreta di tipo per `l` (come liste di `Int`) è che la libreria non permette di specificare proprietà di tipo polimorfo, per poter risolvere l'overloading della funzione `quickCheck` [1].

- `shrink` è una funzione che, dato un valore di tipo `a`, calcola lo shrinking e restituisce una lista di possibili candidati (ricordiamo, shrinking type-based).

Ad esempio possiamo definire un'istanza di `Arbitrary` per la generazione di un semplice tipo `Colour` in questa maniera:

```
1 instance Arbitrary Colour where
2     arbitrary = oneof
3         [return Red, return Blue, return Green]
```

Programma 1.14: Un generatore per un tipo `Colour` in QuickCheck

dove `oneof` è una funzione QuickCheck che seleziona da una lista di generatori con distribuzione di probabilità uniforme. La maggior parte dei tipi base in Haskell sono già forniti di istanze di `Arbitrary` per la loro generazione.

Per lanciare il test al programma 1.12, è sufficiente passarlo come argomento alla funzione `quickCheck`, la quale ha il seguente tipo:

```
1 quickCheck :: Testable a => a -> IO()
```

Programma 1.15: Il tipo della funzione `quickCheck`

Data l'istanza `instance Testable Bool` già definita in QuickCheck, ogni proprietà scritta del tipo quella al programma 1.12 ha quindi valenza come valore `Testable`: si tratta infatti di una funzione da `Arbitrary` a `Testable`, che per QuickCheck è a sua volta `Testable`. Questo meccanismo permette di esprimere proprietà in maniera estremamente succinta.

Nel paradigma object-oriented: jqwik

Consideriamo ora jqwik, una libreria moderna di property-based testing per il linguaggio Java.

In jqwik, una proprietà è un qualsiasi metodo annotato con `@Property`, che ritorni un booleano. Ad esempio, proponendo un'implementazione in jqwik della proprietà 1.10:

```
1 @Property
2 boolean reverseTwiceIsOriginal(@ForAll List<Integer> l) {
3     return reverse(reverse(l)).equals(l);
4 }
```

Programma 1.16: Una proprietà in jqwik

I parametri devono essere preceduti dall'annotazione `@ForAll`, che indica a jqwik di generarne automaticamente i valori. In caso di fallimento di un test, Java lancia un `AssertionError`; per permettere la replicazione dell'errore ed assistere perciò il debugging, inoltre, le successive esecuzioni del test manterranno il medesimo *seed*.

Come QuickCheck, anche jqwik è in grado di generare naturalmente abitanti per pressappoco ogni tipo comune in Java, passando per la classe `Arbitrary`. Questa è definita in documentazione come: *"L'interfaccia generale per rappresentare oggetti che possono essere generati e sono soggetti a shrinking"* (analogamente all'omonima typeclass in QuickCheck) [8]. L'algoritmo di generazione è più raffinato; non è completamente casuale e i valori non sono distribuiti uniformemente:

- valori più piccoli (in termini numerici, di dimensione o lunghezza nel caso di strutture dati) vengono generati più frequentemente di quelli più grandi;
- all'aumentare dei tentativi di test, aumenta in media il numero di valori più grandi generati;
- occasionalmente, vengono "iniettati" valori speciali e casi limite.

A causa della complessità tecnica di questa implementazione, l'approccio raccomandato per definire un generatore di valori non è per sottotipaggio di

`Arbitrary`, ma bensì per tramite della classe `Arbitraries`, che fornisce una serie di metodi statici utili allo scopo. A titolo di esempio, il programma 1.17 mostra un generatore di liste di interi, di massimo 100 elementi e con lunghezza di lista a distribuzione uniforme.

```
Arbitraries.integers()
    .list().ofMaxSize(100)
    .withSizeDistribution(RandomDistribution.uniform());
```

Programma 1.17: Un generatore per liste di interi in jqwik

Buona parte degli `Arbitraries` memorizza un insieme di casi limite che, come già detto, vengono occasionalmente iniettati in fase di generazione. Ad esempio per il tipo `Float`:

```
EdgeCases[0.0, 1.0, -1.0, 0.01, -0.01, -3.4028235E38, 3.4028235E38]
```

Programma 1.18: Casi limite del tipo `Float` in jqwik

E' possibile introdurre vincoli di generazione in maniera simile a QuickCheck definendo un metodo provider:

```
@Property
void aProperty(@ForAll("emails") String email) { ... }

@Provide
Arbitrary<String> emails() { ... }
```

Programma 1.19: Generazione vincolata in jqwik

Come si può vedere nel programma 1.19, il metodo provider deve ritornare un'istanza della classe `Arbitrary`. Per semplificare la generazione di tipi complessi, jqwik mette a disposizione l'annotazione `@UseType`.

jqwik fornisce supporto per lo stateful testing; l'API è centrata sulla seguente interfaccia:

```
public interface Action<S> { ... }
```

Programma 1.20: Interfaccia per transizioni nello stateful testing in jqwik

Con `Action<S>` rappresentiamo una transizione di stato su un oggetto di tipo `S`. Ciascuna transizione può essere autonoma, o dipendere da una precisa precondizione. Una volta specificate le azioni su un oggetto, possiamo raggrupparle in una `ActionChain<T>`.

Una proprietà su oggetto stateful ha il seguente aspetto:

```
@Property
void exampleProperty(@ForAll("actions") ActionChain<T> chain) {
    chain.run().withInvariant( ... );
}

@Provide
Arbitrary<ActionChain<T>> actions() { ... }
```

Programma 1.21: Schema per un test stateful in jqwik

Il metodo `actions()` è un generatore di sequenze di azioni. L'idea è che, per ogni sequenza valida di azioni, l'esecuzione non riporti errori e rispetti eventuali invarianti specificate con il metodo `.withInvariant()`.

Capitolo 2

Property-based testing per il paradigma orientato ai servizi

In questo capitolo presentiamo un possibile approccio all'introduzione del property-based testing nel linguaggio Jolie.

Scegliamo di porre sul problema uno sguardo bipartito; prima studiandone i concetti da un punto di vista linguistico, e successivamente muovendoci su una vista "a basso livello", illustrando attraverso una serie di esempi un (possibile) approccio concreto di implementazione.

2.1 Vista ad alto livello

Cominciamo questo studio facendo alcune considerazioni sugli aspetti linguistici di Jolie (presi come rappresentativi delle caratteristiche generali del paradigma) che si interfacciano in un modo o nell'altro con i prerequisiti per un testing property-based.

Lo scopo è quindi quello di individuare quali degli elementi idiomatici di Jolie abbiano conseguenze interessanti sulle nostre future scelte implementative, ed approfondirne gli effetti.

2.1.1 Influenza del sistema di tipi di Jolie

Generazione per tipi scalari e tipi complessi

Come discusso nella sezione 1.2.1, requisito fondamentale per il property-based testing è la possibilità di generare abitanti per i tipi di dato interessati dal test. Per Jolie, in considerazione del suo sistema di tipi, possiamo definire la generazione in maniera induttiva, di modo che:

- per i tipi scalari, la generazione è definita caso per caso nel rispetto delle caratteristiche del tipo;
- per i tipi composti, utilizziamo un *algoritmo ricorsivo* di visita (in quanto strutture ad albero) e generazione, dove:
  - se visita un nodo di tipo semplice, genera un abitante del tipo secondo definizione;
  - se visita un nodo di tipo complesso, applica ricorsivamente l'algoritmo a tutti i suoi sottonodi (ovvero sottoalberi). [1]

Raffinamenti di tipo e vincoli di generazione

Il sistema di raffinamenti di tipo – caratteristico di Jolie – fornisce naturalmente supporto alla definizione di vincoli di generazione su tipi, analogamente a quanto fanno jqwik e QuickCheck con le rispettive classi `Arbitraries`.

Nel corso degli esempi riportati mostriamo dei raffinamenti non presenti attualmente in Jolie; tuttavia, la loro implementazione è facilmente immaginabile nel contesto del linguaggio.

[1] Al fine di concentrarci sugli aspetti essenziali, abbiamo omesso dalla procedura il caso di nodi a cardinalità multipla. Il caso è triviale perché è sufficiente applicare le due azioni *per ogni* elemento del nodo-vettore.

2.1.2 Proprietà a livello di interfaccia

Premessa

Nella sezione 1.1 abbiamo introdotto il principio di *loose coupling* come aspetto chiave nella progettazione di un'architettura a microservizi, in virtù del quale si ha disaccoppiamento tra la logica interna di un servizio – cioè, la sua implementazione – ed il suo aspetto esteriore – ovvero la sua interfaccia.

Nell'ottica di questa premessa, consideriamo un programmatore che voglia scrivere un API per servizi: non potendo prevedere tutte le possibili implementazioni che si facciano della sua interfaccia, potrebbe aver intenzione di accompagnare ad essa del codice di test, a indicazione di proprietà generali su di essa (o meglio, sulle sue operazioni) che debbano valere *per ogni* possibile implementazione. In un contesto di testing "classico" ciò non sarebbe possibile, perché le asserzioni prevedono (almeno) una chiamata di funzione, la cui sintassi fissa un'implementazione nel servizio a `OutputPort_Name` (vedi la sintassi al programma 1.7).

Specificare proprietà su interfacce

Per le motivazioni riportate sopra, introduciamo quindi una distinzione tra livelli di proprietà: espresse a livello di interfaccia – cioè generali, valide per ogni implementazione – ed espresse a livello di implementazione. Ricordando quanto detto nella sezione 1.2, possiamo vedere in questa distinzione tra test anche il vantaggio di una distinzione tra tipi di specifiche, e quindi una maggior chiarezza di codice.

Per fornire un esempio pratico, introduciamo una famiglia di servizi "contatore", che implementano ciascuno la seguente interfaccia:

```
1 interface CounterInterface {
2     RequestResponse:
3         incr( void )( int )
4 }
```

Programma 2.1: L'interfaccia per i servizi `Counter`

L'operazione `incr` incrementa un intero memorizzato come variabile condivisa e risponde con il valore aggiornato. Ciascun servizio decide per quale quantità incrementare l'intero. Ad esempio, un servizio `CountByOne` potrebbe fornire la seguente implementazione:

```
1 incr()( response ) {
2     global.counter = global.counter + 1;
3     response = global.counter
4 }
```

Programma 2.2: Un esempio di implementazione per l'operazione `incr`

La proprietà elementare che deve valere per `incr` è che il contatore sia a crescita monotona; nel caso di `incr@CountByOne`, invece, la richiesta è più precisa: l'incremento deve essere pari a uno.

I programmi Jolie 2.3 e 2.4 illustrano concettualmente questa distinzione tra livelli di test, sulla base di quanto detto sopra.

```
1 prev = global.counter;
2 incr@CountByOne()( response );
3 if ( response != prev + 1 ) {
4     throw AssertionError( "Increment should be exactly of one!" )
5 }
```

Programma 2.3: Proprietà a livello di implementazione

```
1 prev = global.counter;
2 ///@InterfaceTestAnnotation
3 incr()( response );
4 if ( !(response > prev) ) {
5     throw AssertionError( "Increment should be at least of one!" )
6 }
```

Programma 2.4: Proprietà a livello di interfaccia

L'istruzione alla riga 3 del programma 2.4 rappresenta una chiamata fittizia all'operazione `incr`, senza far riferimento ad alcuna implementazione.

É da intendersi come puramente finalizzata a fare asserzioni sul suo comportamento, senza alcuna esecuzione di codice. Questo tipo di indicazione può essere fornito all'interprete per via di un'*annotazione*. Vedremo successivamente come realizzare questo tipo di controllo; anticipando il discorso, possiamo figurarci un test su interfacce come un ulteriore "strato" da eseguire ogni qual volta si chiama la funzione interessata in un contesto di testing.

Vogliamo concludere sottolineando come questo meccanismo non introduca alcun potenziale problema di incoerenza tra i due livelli di test (ovvero nel caso in cui la validità di una proprietà comporta l'invalidità dell'altra, e viceversa): eventuali contraddizioni, infatti, portano al non superamento del test stesso, ed possono risultare persino benefiche in quanto indicano un errore di comprensione delle specifiche della funzione su uno dei due livelli.

2.1.3 Stateful testing e sessioni Jolie

Abbiamo precedentemente introdotto lo *stateful testing* (sezione 1.2.3) come feature avanzata presente in molte implementazioni moderne di property-based testing. Sebbene non sia da considerarsi un aspetto centrale di questa trattazione, e per questo motivo non ne presenteremo un'idea di implementazione, vogliamo comunque introdurlo quantomeno concettualmente per la sua relazione interessante con taluni aspetti del linguaggio Jolie.

Come già accennato, una sessione in Jolie è a tutti gli effetti un componente stateful al centro di una rete di microservizi. I dati caratteristici della sessione (come variabili `global` o *correlation sets*) possono essere manipolati per azione delle operazioni in gioco. Possiamo quindi modellare un'architettura a microservizi come una sorta di *automa a stati finiti*, dove ciascuna operazione rappresenta una transizione nella "macchina". Un'idea di sintassi potrebbe essere:

```
1 ///@Action( my_MSA )
2 operation()()
```

Programma 2.5: Sintassi Jolie per un'azione nel contesto dello stateful testing

Un'operazione che presenta l'annotazione `@Action` è considerata come un'azione utilizzabile durante il testing. Con il parametro `my_MSA` indichiamo un nome di una MSA sotto cui raggruppare le azioni.

E' possibile quindi esprimere delle invarianti sugli elementi stateful di una MSA in modo tale che, per ogni possibile combinazione di azioni, restino valide certe proprietà. Il ruolo del meccanismo delle precondizioni è parzialmente già adempito dal sistema di tipi di Jolie: nella costruzione di una sequenza di azioni, il tipaggio di una certa richiesta può rappresentare un vincolo sull'operazione che la precede, il cui tipo di risposta deve combaciare.

Facciamo notare come le annotazioni `@Action` vengano in questo caso disposte direttamente sul codice, e non su funzioni di test. Si tratta in questo senso di un approccio a carattere *dichiarativo*, dove il programmatore indica gli elementi di interesse per il test e questi vengono combinati a discrezione di un engine.

2.1.4 Verso un'implementazione di PBT per Jolie

Decidiamo di progettare la nostra proposta di implementazione attorno al framework JoT: questa scelta ci solleva dal dover trattare le problematiche già citate del testing su architetture a microservizi. Vogliamo quindi estendere opportunamente JoT in maniera tale da permettergli di supportare il property-based testing. In questa direzione, la facilità di integrazione di Jolie con altri linguaggi di programmazione, in particolare Java, risulta di grande utilità: ci permette di sviare da un'implementazione "nativa" e appoggiarci invece a librerie preesistenti di provata correttezza e efficienza.

Per restare consistenti con i principi di JoT, prevediamo che l'implementazione della logica di test venga fatta usando direttamente Jolie; il codice sarà poi opportunamente annotato dal programmatore per completare la de-

finizione della semantica di test. Queste annotazioni "oscurano" il ruolo di jqwik nel test, a cui – dietro alle quinte, nella modalità che vedremo successivamente – è delegato il compito di generazione automatica di *test case* e di shrinking.

Corrispondenza di tipo tra Jolie e Java

Se a jqwik è delegata la generazione di dati, significa che deve essere definita una corrispondenza tra i tipi di questi e quelli Jolie. Definiamo quindi una funzione di conversione su tipi che agisce come segue:

- ad ogni tipo scalare in Jolie associa un corrispondente tipo Java generabile da jqwik (vedi tabella 2.1);
- per ogni tipo composto in Jolie, costruisci una classe Java. Ad ogni sottonodo del tipo Jolie corrisponde un membro nella classe, il cui tipo è ottenuto per applicazione ricorsiva di questo algoritmo.

Il programma a seguire mostra un esempio di conversione per un generico tipo strutturato in Jolie. Per una realizzazione più completa di conversione di tipo tra Jolie e Java, rimandiamo al tool `jolie2java` incluso nel rilascio ufficiale di Jolie [10].

Java

```
public class T {
    public Integer rootValue;

    public String someNode;

    private class someComplexNodeType { ...
    ↪ }

    public someComplexNodeType
    ↪ someComplexNode;

    public ArrayList<Object> someNodeVector;

    // Constructor, Setters, etc...
}
```

Jolie

```
type T: int {
    someNode: string
    someComplexNode: { ... }
    someNodeVector[1,3]: any
}
```

Programma 2.6: Un esempio di conversione di tipo tra Jolie e Java

Concludiamo il discorso facendo notare come l'algoritmo applicato da jqwik per la generazione è del tutto simile a quello che abbiamo definito nella sezione 2.1.1; di fatto, quindi, il suo utilizzo non provoca alcuna discrepanza a livello concettuale con quanto detto sopra.

Tipo Jolie	Tipo Java
`void`	Non generabile
`bool`	`Boolean`
`int`	`Integer`
`long`	`Long`
`double`	`Double`
`string`	`String`
`raw`	`Byte`
`any`	`Object`

Tabella 2.1: Tabella per la conversione di tipo tra tipi scalari Jolie e Java

Un esempio introduttivo

Prima di passare alla sezione successiva, dove tratteremo gli aspetti implementativi, introduciamo un primo esempio di test property-based in Jolie. Con esso mostriamo un tipico pattern di applicazione del PBT: il testing di funzioni inverse [2]. La proprietà generale che ci aspettiamo valga è che, fissato un certo valore di input, l'applicazione successiva di una funzione e della sua inversa restituisca il valore originale. Presentiamo quindi come esempio una coppia di funzioni dalla Standard Library di Jolie per la codifica in Base64:

```
3 rawToBase64( raw )( string ),
4 base64ToRaw( string )( raw ) throws IOException( IOExceptionType )
```

Programma 2.7: Funzioni Jolie per la codifica e decodifica di dati in Base64

[2] Un caso di *proprietà metamorfica*, vedi [5].

Vogliamo esprimere con Jolie il primo verso della proprietà (ovvero in ordine: codifica → decodifica):

```
11 ///@Property, @ForAll( raw request )
12 decodeIsReverseEncode( request )() {
13     rawToBase64( request )( encoded );
14     base64ToRaw( encoded )( result );
15
16     if ( request != result ) {
17         throw AssertionError( "Decoding function does not act as reverse
           ↪  encoding!" )
18     }
19 }
```

Programma 2.8: Una proprietà elementare in Jolie

La sintassi delle annotazioni è pensata per essere familiare sia nei riguardi di JoT, che di jqwik. La semantica è la seguente:

- `@Property` annota un test property-based in modo tale che possa essere riconosciuto da JoT (analogamente alla già presente annotazione `@Test` per i test tradizionali);

- `@ForAll( T request )` indica che vogliamo generare abitanti del tipo `T` con cui inizializzare la variabile `request`.

Il test così scritto e lanciato viene eseguito per un certo numero di tentativi (in questo caso, il valore standard di jqwik). Se uno dei tentativi fallisce, si tenta lo shrinking ed infine l'errore viene riportato.

A differenza dell'approccio stateful descritto sopra, in questo caso il testing è fatto in maniera *imperativa*: l'ordine delle azioni è fissato dal programmatore proprio in fase di scrittura del test.

2.2 Vista a basso livello

In questa sezione introduciamo un'idea implementativa di quanto detto sopra, basata su un approccio statico di traduzione in Java. Mostriamo prima di tutto come estendere l'architettura di JoT per i nostri fini, a cui seguono alcuni esempi da cui eventualmente partire nella specifica di un compilatore. Al termine della sezione, inoltre, accenniamo brevemente e più in astratto una strada alternativa che si appoggia a un interprete.

2.2.1 Il modulo compilatore per JoT

Sulla base di e riprendendo quanto detto nella sezione 2.1.4, proponiamo di estendere JoT con:

- un insieme di *annotazioni* per il PBT, complementari alla logica dei test scritta in Jolie;
- un modulo *compilatore* che traduce test Jolie annotati in opportuni `JavaService` per il testing.

I `JavaService` agiscono concettualmente da "ponte" tra jqwik e i servizi di test: prima generano i dati del tipo richiesto, poi contattano il servizio originale passandoli nella richiesta. Questo approccio è preferibile ad una completa traduzione del sorgente Jolie se consideriamo l'eventualità di dover trattare istruzioni eseguite in parallelo, *input-guards*, od altri costrutti peculiari presenti in Jolie.

Una conseguenza importante delle nostre scelte implementative è la perdita del supporto di JUnit per i test in Java. jqwik, infatti, si appoggia ad esso per processare le sue annotazioni [8]; noi sostituiamo la sua presenza con JoT. Vedremo negli esempi che seguiranno come sostituire le annotazioni utili ai nostri scopi (ad es. `@ForAll`, `@UseType`) utilizzando direttamente l'API jqwik.

Descrizione della procedura di compilazione

Illustriamo la procedura di compilazione, e come va a collocarsi nell'architettura preesistente di JoT. La figura 2.1 illustra graficamente il procedimento.

1. JoT individua un servizio che offre come operazione un test property-based annotato, lo compila in un `JavaService` e ne fa l'embedding;
2. il medesimo servizio al punto 1. viene messo in embedding e lanciato; in questo modo, le operazioni di test saranno a disposizione per essere invocate;
3. il `JavaService` prodotto viene lanciato insieme ad altri test;
4. durante la propria esecuzione, il `JavaService` richiede l'operazione al suo stesso nome al servizio Jolie lasciato in esecuzione.

La procedura di compilazione ed embedding dei servizi avviene in contemporanea alla fase di raccolta test in JoT. Per il resto, l'esecuzione dei test property-based si colloca naturalmente all'interno del normale *lifecycle* di test (vedi sezione 1.1.2).

2.2.2 Esempi per l'approccio con compilatore

Di seguito offriamo, scandendo e commentando alcuni esempi già introdotti e di nuovi, un suggerimento operativo per un ipotetico compilatore, di cui già si è discusso; l'idea è che questi esempi possano essere utilizzati come linee guida nella sua realizzazione.

Compilazione di base

Cominciamo riprendendo la proprietà descritta nel programma 2.8, relativa a coppie di funzioni inverse come quelle per la codifica e decodifica in Base64. Eccezionalmente per questo caso, vogliamo mostrare una costruzione

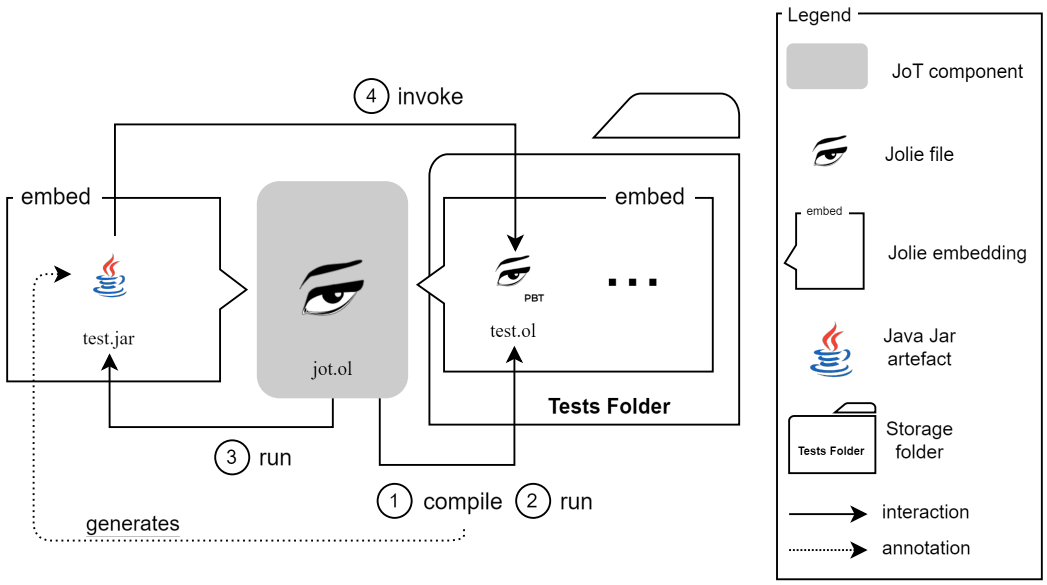

Figura 2.1: Schema d'esecuzione per un test property-based in JoT

incrementale del `JavaService` finale, passando prima per una versione con annotazioni JUnit (come se il test fosse stato nativamente scritto in Java), e solo successivamente il compilato. La versione annotata è quindi come segue:

```
1 @Property
2 void decodeIsReverseEncode(@ForAll Byte request) throws AssertionError {
3     TestServiceImpl.decodeIsReverseEncode(request);
4 }
```

Programma 2.9: Esempio di compilato (con annotazioni)

In questo esempio, come in tutti quelli a seguire, si assume che venga generata anche una classe con suffisso `-Impl` per contattare il servizio di interesse per il test; i suoi metodi pubblici non sono altro che funzioni *wrapper* per le rispettive operazioni Jolie.

L'annotazione `@Property` può essere rimossa senza conseguenza perché il suo ruolo è già adempito dalla corrispettiva annotazione lato Jolie. Possiamo invece replicare la semantica dell'annotazione `@ForAll` usando alcuni metodi della classe `Arbitraries`, come si può vedere nel programma 2.10.

```
1  void decodeIsReverseEncode() throws AssertionError {
2      Stream<Byte> requestStream = Arbitraries
3          .bytes()
4          .sampleStream()
5          .limit(JQWIK_DEFAULT_TRIES);
6      try {
7          requestStream.forEach(data ->
8              TestServiceImpl.decodeIsReverseEncode(data)
9          );
10     } catch(AssertionError err) {
11         Byte shrunkSample = err.failSample
12             .shrink().value();
13         Reporter.report(shrunkSample, err.failSample
14             /* Pass additional information */);
15     }
16 }
```

Programma 2.10: Esempio di compilato (senza annotazioni)

Approfondiamo rapidamente il funzionamento del codice:

- `.bytes()` instanzia un generatore per il tipo Byte;
- `.sampleStream()` crea uno stream di dati a partire dal generatore;
- `.limit()` restringe la dimensione dello stream al numero di test che jqwik deve eseguire (nell'esempio, una costante che indica un valore di default).

Infine, alle righe 7-9, contattiamo il servizio Jolie di test per ciascun elemento in `requestStream`. Se uno dei test fallisce, l'eccezione viene catturata e si esegue lo shrinking del valore falsificante; infine si riporta l'errore, fornendo entrambi i valori interessanti a `Reporter`.

Facciamo notare che la procedura `Reporter.report()` è una semplificazione dell'effettivo meccanismo di report di jqwik; in questo caso, ci siamo decisi per non includere dettagli implementativi superflui ma restare essenziali ai concetti.

Vincoli di generazione

Supponiamo di voler testare – come sarebbe buona pratica fare – la proprietà dell'esempio precedente anche nel senso opposto (ovvero, prima decodifica e poi codifica). Poiché l'operazione `base64ToRaw` è definita solo su un preciso sottoinsieme di `string`, ovvero quelle stringhe che sono valide in Base64, il test deve rispecchiare questa precondizione vincolando la generazione a valori sensati.

```
type Base64Strings: string(
    charRanges( ["a", "z"] ),
        ...
    withChars( "=" )
)

///@ForAll( Base64Strings request )
encodeIsReverseDecode( request )( response ) {
    base64ToRaw( request )( decoded );
    rawToBase64( decoded )( result);

    if ( request != result ) {
        throw AssertionError( "Encoding function does not act as reverse
        ↪  decoding!" )
    }
}
```

Programma 2.11: Un test PBT in Jolie con vincoli di generazione

Sulla base di quanto detto nella sezione 2.1.1, il vincolo viene specificato sulla base di raffinamenti di tipo [3].

Dato il programma 2.11, un compilatore produce il seguente metodo:

[3] Ai fini dell'esempio, abbiamo dato nel codice solo una specifica indicativa delle stringhe in codifica Base64. Per una definizione più precisa si rimanda a [7].

```
void encodeIsReverseDecode() {
    Stream<String> requestStream = Arbitraries
        .strings()
        .withCharRange("a", "z")
            ...
        .withChars("=")
        .sampleStream()
        .limit(JQWIK_DEFAULT_TRIES);
    // Usual operation call and shrinking / error reporting...
}
```

Programma 2.12: `JavaService` per un test con vincoli di generazione

Generazione su tipi complessi

Fino ad ora abbiamo mostrato solo la generazione per tipi scalari. Assumiamo di aver scritto in Jolie una proprietà per una qualche funzione su coordinate (che abbiamo introdotto nel programma 1.4); vediamo come deve comportarsi un `JavaService` per la generazione degli abitanti di tipo.

```
void aPropertyOnCoordinates() throws AssertionError {
    Stream<Coordinate> requestStream = Builders
        .withBuilder(() -> new Coordinate(null, null))
        .use(Arbitraries.strings()).inSetter(Coordinate::setLat)
        .use(Arbitraries.strings()).inSetter(Coordinate::setLng)
        .build()
        .sampleStream().limit(JQWIK_DEFAULT_TRIES); // Same as for
        ↪  scalar types

    // Usual operation call and shrinking / error reporting...
}
```

Programma 2.13: `JavaService` per un test su tipi complessi

Nel programma 2.13, sfruttiamo l'API `Builders` di jqwik per la composizione di `Arbitraries` al fine di costruire oggetti aggregati [9].

In questo caso, utilizziamo il metodo `Arbitraries.strings()` per impostare il *feeding* di dati ai metodi setter della classe. Se uno dei sottonodi fosse a sua volta di tipo complesso, sarebbe sufficiente passare un altro oggetto costruito con `Builders... .build()` a `Builders.use()` (in maniera ricorsiva, ricordando quanto detto nella sezione precedente).

Test a livello di interfaccia e di implementazione

Nella sezione 2.1.2 abbiamo introdotto un esempio ad alto livello di test a livello di interfaccia. In questa vogliamo essere più concreti sia nella definizione del test in Jolie che del compilato risultante.

```
1 ///@Invariant( operation_name ), @Track( global_var )
2 property_name( io )() { ... }
```

Programma 2.14: Sintassi di un test su interfacce in Jolie

Nel programma 2.14 mostriamo la sintassi di un test su interfacce in Jolie, dove:

- `operation_name` è l'operazione per cui si sta specificando la proprietà;
- `property_name` è il nome del test;
- `global_var` è una o più variabili globali di cui si vuole tracciare l'andamento pre- e post-chiamata di funzione;
- `io` è un oggetto con due sottonodi `input` e `output`, che registra i valori di input / output dell'operazione – inclusi i valori tracciati dall'annotazione `@Track` – in modo che si possa farne asserzioni.

Come abbiamo già accennato nella sezione 2.1.2, questo test viene "agganciato" a un normale test su operazione. La necessità di inizializzare la struttura dati `io` con le informazioni di input / output dell'operazione ci costringe ad introdurre un ulteriore *layer* all'architettura di test, lato Jolie.

Un compilatore quindi produce questo strato intermedio di codice combinando i due livelli di test: il servizio prodotto diventa quello effettivamente contattato dal `JavaService` risultante.

Ora mostriamo nella pratica un test di questo tipo, facendo riferimento all'esempio del "contatore", nel programma 2.15.

```
incrByOne_withInvariant()() {
    // io.input object setup
    with( io ) {
        .input.global_counter = global.counter
    };

    // Copy of implementation-level test code
    prev = global.counter;
    incr@CountByOne()( count );

    if ( count != prev + 1 ) {
        throw AssertionError( "Increment should be exactly of 1!" )
    }

    // io.output object setup
    with( io ) {
        .output = count
        .output.global_counter = global.counter
    };

    // Send request at interface-level test code
    incrAtLeastByOne@CounterTest( io )()
}
```

Programma 2.15: Implementazione di test su più livelli (lato Jolie)

Spieghiamo il codice appena presentato più nel dettaglio:

1. Costruiamo un oggetto `io` che memorizzi i valori di input / output (inclusi di eventuali variabili condivise) per ogni chiamata all'operazione. L'oggetto verrà poi fornito come richiesta al test su interfaccia.

```
3   with( io ) {
4       .input.global_counter = global.counter
5   };
```

Se l'operazione `incr` prendesse una richiesta di tipo non-`void`, verrebbe anche inizializzato `io.input`.

2. Alle righe 7-12 si trova una copia del corpo della funzione che testa l'implementazione di `incr`. Qui viene verificato il primo livello di test; in questo caso che l'incremento sia esattamente pari a 1.

```
8   prev = global.counter;
9   incr@CountByOne()( count );
10
11  if ( count != prev + 1 ) {
12      throw AssertionError( "Increment should be exactly of 1!" )
13  }
```

3. Si memorizzano i valori di output dell'operazione sotto test.

```
16  with( io ) {
17      .output = count
18      .output.global_counter = global.counter
19  };
```

4. Infine, viene lanciato il test a livello di interfaccia, con la richiesta correttamente inizializzata.

```
22  incrAtLeastByOne@CounterTest( io )()
```

Il `JavaService` prodotto è triviale:

```
1  void incrByOne_withInvariant()() throws AssertionError {
2      /* Data generation with Arbitraries
3         would go here. */
4      try {
5          CounterTestImpl.incrByOne_withInvariant();
```

```
6       } catch (AssertionError err) {
7           // etc...
8       }
9   }
```

Programma 2.16: Implementazione di test su più livelli (lato Java)

2.2.3 Cenni per un approccio via interprete

Ai fini di completezza del discorso, accenniamo in maniera più discorsiva a un'altra modalità di implementazione, basata su delle operazioni eseguite totalmente a runtime e senza alcuna produzione di codice compilato.

Mentre nell'approccio compilativo il codice lato Java comunicava direttamente con i servizi Jolie, in questo caso vogliamo che sia il motore di test di JoT ad agire come *orchestratore* tra le parti in gioco, prima recuperando da jqwik i dati generati e poi fornendo questi ai servizi di test. In questo caso, JoT può appoggiarsi liberamente a JUnit ed alle sue annotazioni, perché jqwik opera indipendentemente da Jolie. Se viene riportato un errore, JoT richiede a jqwik di calcolare lo shrinking.

Uno dei potenziali svantaggi previsti è la minor chiarezza del codice: infatti, l'interprete JoT agisce come una sorta di "scatola nera" che oscura la sua logica interna, in contrapposizione ad un compilato Java, il quale:

- è *leggibile* e *verificabile* da un programmatore, agendo anche come documentazione a livello di codice del funzionamento del modulo per il PBT di JoT. Questo presenta enormi vantaggi per il debugging;
- è *modificabile*; un programmatore che richiede maggiore flessibilità potrebbe voler applicare modifiche al test prodotto, sia in termini di funzionalità che di efficienza, senza dover sottostare alla disponibilità di annotazioni in Jot.

Conclusioni

Nel corso di questa trattazione, abbiamo illustrato – con particolare occhio per una visione linguistica, grazie al linguaggio Jolie – gli elementi essenziali del paradigma di programmazione service-oriented. Dopo uno studio del property-based testing a livello sia generale che specifico nelle sue implementazioni già esistenti, ne abbiamo trasposto i concetti su Jolie e la sua suite di testing JoT. A partire da questa discussione è stata presentato, attraverso una serie di esempi chiarificatori, un possibile approccio implementativo basato su compilazione in Java e sfruttamento delle primitive della sua libreria per il property-based testing, jqwik. Abbiamo concluso accennando in astratto ad un approccio interpretativo.

Abbiamo menzionato ad alto livello il ruolo dello stateful testing nella specifica di invarianti su architetture a servizi: una prosecuzione ovvia di questo lavoro consiste nella progettazione di un'implementazione per questa modalità di testing analoga a quelle mostrate nella sezione 2.2.2. Un'altra strada ovviamente percorribile consiste nel raffinamento delle annotazioni introdotte; ad es. per `@Property`: introduzione di parametri per la configurazione di test a livello di numero di tentativi, *seed* manuali, modalità di generazione e shrinking, etc...

Un ulteriore aspetto su cui lavorare, centrale al property-based testing in un contesto di utilizzo pratico, è l'analisi della distribuzione dei test-cases generati, attraverso l'impiego di adeguati tool per la visualizzazione statistica. Anche in questo caso ci figuriamo lo sfruttamento degli strumenti jqwik adibiti allo scopo, ipoteticamente "confezionati" in un `JavaService` apposito.

Infine, l'uso di *session types* [6] per lo studio della correttezza di programmi concorrenti suggerisce un'ulteriore direzione di miglioramento, dove sulla base di tipi che descrivono computazioni possiamo automaticamente generare catene di operazioni, similmente alla strategia dietro all'approccio di stateful testing.

Bibliografia

[1] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268 – 279, September 2000.

[2] George Fink and Matt Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74 – 80, 1997.

[3] Saverio Giallorenzo. Service-Oriented Programming Paradigm. In *Programming Languages: Principles and Paradigms*, pages 473 – 518. Springer, 2024.

[4] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Narongrit Unwerawattana. JoT: A Jolie Framework for Testing Microservices. *Science of Computer Programming*, 2024.

[5] John Hughes. How to specify it! In William J. Bowman and Ronald Garcia, editors, *Trends in Functional Programming*, pages 58–83, Cham, 2020. Springer International Publishing.

[6] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1), April 2016.

[7] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RCF 4648, RFC Editor, October 2006.

[8] The jqwik User Guide. `https://jqwik.net/docs/current/user-guide.html`.

[9] The Jolie programming language documentation. `https://docs.jolie-lang.org/v1.12.x/introduction/index.html`.

[10] Making a JavaService with Jolie2Java. `https://docs.jolie-lang.org/v1.13.x-git/language-tools-and-standard-library/technology-integration/java/index.html#making-a-java-service-with-jolie2java`.

[11] The sad state of property-based testing libraries. `https://stevana.github.io/the_sad_state_of_property-based_testing_libraries.html`, July 2019.

[12] Johannes Link. Property-based Testing in Java: From Examples to Properties. `https://blog.johanneslink.net/2018/03/26/from-examples-to-properties/`, March 2018.

[13] David R. MacIver. In praise of property-based testing. `https://increment.com/testing/in-praise-of-property-based-testing/#authors`, August 2019.