# Appunti generali

## Disponibilità materiale

Tutto il codice prodotto per la tesi si trova alla repository pubblica di GitHub:

<https://github.com/PellegrinelliNico/test-generator-for-yakindu>

La tesi stessa e ulteriori documenti come il report dei vari incontri si trova al progetto Overleaf:

<https://it.overleaf.com/project/6528e34894fee1ad0fc2dc72>

Dubbi e informazioni generali si trovano su una nota sullo smartphone.

## Utilizzo dei tool

### Itemis CREATE (5.1.1)

A causa di diversi problemi nell'installare Itemis CREATE (ma ance Yakindu) come plug-in di Eclipse, si è optato per l'utilizzo del tool come app standalone (che in fin dei conti è Eclipse con Itemis CREATE già scaricato). La documentazione e i tutorial (disponibili al link https://www.itemis.com/en/products/itemis-create/documentation/) non sono completamente up-to-date con l'ultima versione del tool, quindi ci sono alcune differenze, ad esempio nella generazione di codice (ad esempio il metodo runCycle() non è public ma private, c’è però il metodo triggerWithoutEvent(), in realtà è probabilmente up-to-date, runCycle è private solo nel caso EventDriven). Itemis CREATE necessita di una licenza. In realtà itemis cerate funziona anche su Eclipse (provato con nuova installazione di eclipse), probabilmente c’era qualche conflitto.

### EvoSuite (1.0.6)

EvoSuite come plugin non funziona (né se installato da update site né da marketplace), si è optato quindi per l’approccio progetto maven con test generati ed eseguiti da linea di comando. Per la generazione è obbligatorio visto che i plugin non funzionano. Per l’esecuzione, il comando mvn test funziona, da Eclipse invece no, lancia un’eccezione. Con il nuovo eclipse i test si eseguono anche senza maven ma direttamente dall’ide. EvoSuite 1.2.0 genera il codice ma nel farlo lancia delle eccezioni.

### Ambiente di sviluppo finale

Per scrivere statechart Itemis CREATE, generare automaticamente del codice java che li implementi e successivamente generare automaticamente dei casi di test JUnit per il codice servono:

* **Java**: JDK versione 1.8 installata sul PC;
* **Maven**: versione 3.8.4 installata sul PC (la versione è probabilmente irrilevante);
* **EvoSuite**: file eseguibile evosuite-1.0.6.jar, disponibile a https://www.evosuite.org/downloads/;
* **App** **standalone Itemis CREATE**: IDE con **m2e** (plugin per l’utilizzo di Maven) installato e una **licenza valida**;
* **Dependency**: **EvoSuite** 1.0.6, **JUnit** 4.12 (dependency del progetto Maven).

Esempio di POM:

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>Prova</groupId>

    <artifactId>prova</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <properties>

        <maven.compiler.target>1.8</maven.compiler.target>

        <maven.compiler.source>1.8</maven.compiler.source>

    </properties>

    <dependencies>

        <dependency>

            <groupId>com.sun</groupId>

            <artifactId>tools</artifactId>

            <version>1.8</version>

            <scope>system</scope>

            <systemPath>C:\Program Files\Java\jdk1.8.0\_202\lib\tools.jar</systemPath>

        </dependency>

        <!-- https://mvnrepository.com/artifact/junit/junit -->

        <dependency>

            <groupId>junit</groupId>

            <artifactId>junit</artifactId>

            <version>4.12</version>

            <scope>test</scope>

        </dependency>

        <!-- https://mvnrepository.com/artifact/org.evosuite/evosuite-runtime -->

        <dependency>

            <groupId>org.evosuite</groupId>

            <artifactId>evosuite-runtime</artifactId>

            <version>1.0.6</version>

        </dependency>

    </dependencies>

</project>

In realtà basta un eclipse “pulito” con itemis installato da update site, evosuite-1.0.6.jar da usare da linea di comando e da importare nel progetto per eseguire i test.

### Esempio di utilizzo

Un esempio di utilizzo dei tool è il seguente:

1. Creare un progetto Maven (o java) sull’app standalone Itemis CREATE (o eclipse con itemis installato);
2. Sempre su Itemis CREATE, creare un modello, simularlo e generare il codice Java come illustrato nella documentazione del tool;
3. Generare da linea di comando i test;
4. Se si vogliono eseguite i test, spostarli nella source folder test ed eseguirli da linea di comando (o da ide eclipse).

Esempio di comandi windows:

C:\Users\lenovo\Desktop\ProgettiGit\test-generator-for-yakindu\tools-use-example\Calculator>  
java -jar C:\Users\lenovo\Downloads\evosuite-1.0.6.jar -class calc.Calculator -projectCP .\target\classes

C:\Users\lenovo\Desktop\ProgettiGit\test-generator-for-yakindu\tools-use-example\Calculator> mvn test

## Contenuto della repo

### tool-use-example ws

* **Calculator**: maven project creato per provare ad utilizzare EvoSuite su una semplice classe non ottenuta con itemis.
* **CallHandling**: java project inizialmente ottenuto seguendo il tutorial a <https://www.itemis.com/en/products/itemis-create/documentation/tutorials/tutorial_code-generation#tutorial_code-generation> e successivamente utilizzato per generare i test con evosuite partendo dal codice generato da itemis.
* **CallHandlingTestExecution**: maven project ottenuto da CallHandling con l’obiettivo di eseguire i test da linea di comando con maven visto che utilizzando l’IDE viene lanciata una eccezione.
* **LightSwitch**: general project implementato seguendo il tutorial a <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sctunit_sctunit_by_example#sctunit_sctunit_by_example>.

### junit-test-generation-example ws

* **StatechartExamples**: maven project creato per studiare sperimentalmente la struttura dei test JUnit ottenuti con la tool chain finora implementata. I modelli sono:
  + **SimpleLoop**: semplice macchina che implementa un loop infinito, con una interfaccia con nome light. Le interfacce con nome diventano classe statiche (ossia interne alla classe della macchina), per chiamare l’evento bisogna prima accedere alla classe statica con il metodo pubblico nome\_classe\_statica(), dove nome\_classe\_statica è il nome dell’interfaccia, in questo caso quindi IstanzaMacchina.light().raiseToggle(). Nota che isFinal() restituisce sempre false perché non si raggiunge mai un Final State;
  + **SimpleSeries**: semplice macchina che implementa tre stati collegati in serie tra loro ed infine ad un exit node, in aggiunta alla serie c’è una transizione per verificare la differenza tra EventDriven e CycleBased;
  + **SimpleSeriesCycleBased**: identica alla precedente ma CycleBased(200). La gestione dei tempi del ciclo deve essere implementata dal client, il 200 non è presente nel codice generato;
  + **SimpleChoice**: macchina che utilizza l’elemento choice. In aggiunta si usano le variabili, le guardie e le reaction (sia nelle transizioni che negli stati);
  + **SimpleCompositeState**: macchina con stato composto, con ingresso alternativo. Nel codice generato, quando bisogna entrare in uno stato composto (o uscire), ogni metodo di ingresso chiama il metodo per entrare nell’elemento successivo, ad esempio se si entra in StateB con ev2, viene chiamato il metodo per entrare in StateB, che chiama il metodo per entrare nella region r1, che chiama il metodo per “entrare” in alt\_entry, che infine chiama il metodo per entrare nello stato StateB2. Gli enumerativi che rappresentano gli stati hanno un nome che riflette questa “gerarchia”;
  + **SimpleParallel**: semplice macchina per simulare il “parallelismo” (in realtà implementato serialmente) negli stati composti. Il vettore StateVector contiene due elementi anziché uno, perché in uno stesso momento possono essere attivi fino a due stati. Quindi il metodo microStep() deve gestire switch case.
* **Temp**: java project per testare il progetto descritto al punto successivo

### ysc-to-sctunit-translator ws

* **JUnit2SCTUnit**:

vecchio maven project inizialmente scritto per pensare a come gestire la scrittura dinamica dei file .sctunit attraverso string template e come leggere i metodi di test Junit con javaparser. Alla fine, il tentativo è riuscito e il codice è stato portato nel progetto descritto sotto. Gli artefatti importanti sono:

* + **sct\_template.st**: stringtemplate contenente la base dei test sctunit (no variabili, no eventi temporali…).
  + **TestCase**: classe java contenente le informazioni di un test JUnit, con metodi molto semplici e autoesplicativi basati sulla classe Action. Contiene il nome del test (String name) e la lista di azioni che deve eseguire (List<Action> actions).
  + **Action**: classe java contenete tutti le stringe necessarie a definire una qualsiasi azione (in realtà per ora solo enter, raise, assert, proceed, exit e poco altro). Le singole istanze avranno, parlando dei fields, valori diversi da null solo per le stringhe necessarie alla singola azione che descrivono (ad esempio se l’azione è del tipo assert active state, le stringhe non a null sono solo state e not). Contiene solo il costruttore e i vari getter. Questa struttura un po’ particolare è resa necessaria da come lavora Stringtemplate, che non permette alcun tipo di computazione nel template se non scorrere una lista. La creazione di queste azioni è nei metodi della classe TestCase, che “mascherano” questa struttura.
  + **Junitreading**: workspace contenente le classi che usano JavaParser per ottenere le informazioni dalla classe di test generata da evosuite:
    - **JavaParserTest**: Chiama le varie classi Collector per ottenere il nome della classe e una lista di TestCase da passare al template.
    - **TestCollector**: il collector più importante, con l’ausilio di altri collector ottiene le informazioni dai metodi di test junit e le struttura in una lista di TestCase.
    - Le altre classi sono semplici e ovvie.
* **Ysc2SCTUnit**:

inizialmente maven project per generare automaticamente casi di test Junit partendo da un progetto java di questo tipo (anche in un altro ws):

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Le librerie di JUnit 4 ed Evosuite servono solo se si vogliono eseguire i test o compilare le classi di test.

Gli argomenti da passare sono: path dello workspace, nome del progetto, package dove mettere l’implementazione java dello stetachart e nome del file .ysc contenente lo statechart (NON il nome dello statechart) senza l’estensione:

(NOTA: se lo statechart ha elementi temporali, bisogna aggiungere un quinto argomento “time”)

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Il risultato finale è il seguente:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Ora tutto è stato gestito con CLI, guarda la sezione CLI a proposito.

Sono stati gestiti statechart con eventi temporali, ma potrebbero esserci ulteriori casistiche da gestire.

Ora l’esecuzione del main da due warning, che però non ne compromettono la corretta esecuzione:

warning: Implicitly compiled files were not subject to annotation processing.

Use -proc:none to disable annotation processing or -implicit to specify a policy for implicit compilation.

1 warning

SLF4J: Class path contains multiple SLF4J bindings.

SLF4J: Found binding in [jar:file:/C:/Users/lenovo/.m2/repository/ch/qos/logback/logback-classic/1.1.3/logback-classic-1.1.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: Found binding in [jar:file:/C:/Users/lenovo/.m2/repository/org/evosuite/evosuite-master/1.0.6/evosuite-master-1.0.6.jar!/org/slf4j/impl/StaticLoggerBinder.class]

SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.

SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]

Il primo l’ho risolto inserendo -implicit:class come argomento della compilazione (javac), che è l’opzione di default ma mettendola esplicita non da errore.

Il secondo escludendo il secondo binding nel POM:

<dependency>

<groupId>org.evosuite.plugins</groupId>

<artifactId>evosuite-maven-plugin</artifactId>

<version>1.0.6</version>

<exclusions>

<exclusion>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

</exclusion>

</exclusions>

</dependency>

È stato modificato il path della vm in headless.ini in C:\Program Files (x86)\itemis\_CREATE, per vedere com’era all’inizio scarica un nuovo zip di itemis create, da plugins/org.eclipse.justj.openjdk.hotspot.jre.full.stripped.win32.x86\_64\_17.0.5.v20221102-0933/jre/bin, a C:\Program Files\Java\jdk-17.0.2\bin  
Questo perché nell’eseguire TestGenerator.java diceva:

A Java Runtime Environment (JRE) or Java Development Kit (JDK)

must be available in order to run ItemisCREATEc. No Java virtual machine

was found after searching the following locations:

plugins/org.eclipse.justj.openjdk.hotspot.jre.full.stripped.win32.x86\_64\_17.0.5.v20221102-0933/jre/bin

Perché? Inoltre, significa che per eseguire correttamente l’eventuale .jar che produrrò, bisognerà prima modificare questo file (non mi sembra molto bella come cosa). Inoltre, il file scc.bat viene preso dalla mia macchina, stesso discorso.

**NOTA: ad un certo punto la mia installazione di itemis create standalone ha smesso di visualizzare correttamente gli statechart, al che ne ho installato una nuova e in questa non è stato necessario modificare in alcun modo il file headless.ini.**

Il progetto è stato trasformato in un progetto maven => c’è il problema di specificare la posizione del jdk.

Inoltre, è stato inserito il codice del progetto precedente e con l’aggiunta di altre classi per compilare è in grado di implementare tutta la tool-chain (chiamando il metodo main di **TestGenerator**). Inoltre, non genera un solo file di test ma due. Il secondo si ottiene modificando il codice dello statechart generato trasformando i metodi da protected a private (fatto con Javaparser) per cercare di migliorare l’output di evosuite. **Tutto il Progetto è ben commentato, guarda quello come documentazione**.

## Ipotesi sullo statechart in ingresso

Di seguito vengono fatte delle ipotesi sullo statechart ricevuto in ingresso (i.e. delle condizioni che l’utente deve rispettare perché la toolchain operi correttamente) con le relative giustificazioni. Nota bene che tutti questi problemi sono dovuti alle decisioni fatte da Itemis Create nel generatore di codice Java:

1. **Il nome di regioni, eventi e stati deve essere tutto lower case**! Itemis Create è case sensitive, quindi teoricamente si possono avere due stati o eventi che differiscono solo nel case. Nel caso di eventi viene segnalato uno warning, nel caso degli stati no. Il problema è che nella traduzione in java (fatta da itemis create stesso) ciò si traduce in metodi o enumerativi con lo stesso nome:

Immagine che contiene testo, schermata, software, Icona del computer

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Inoltre, avendo perso l’informazione iniziale sulla reale stringa che rappresentava il nome o l’evento (soprattutto per gli stati, che diventano completamente upper case), nel tornare in ambiente Itemis Create (case sensitive) con SCTUnit non si può sapere quali caratteri erano upper case e quali lower. Il software è stato implementato ipotizzando che tutti le stringhe rappresentanti nomi di elementi dello statechart siano in lower case (comprese regioni ed eventuali altri elementi ancora non trattati, alcuni magari potrebbero essere gestiti correttamente anche se case sensitive).

1. **I nomi di stati e regioni NON possono contenere spazi, underscore, punti, trattino, #, @ etc.!** Gli stati vengono rappresentati con enumerativi, tutti upper, ottenuti a partire dai nomi delle regioni e degli state in cui lo stato corrente fa parte (in una sorta di gerarchia). Gli spazi e gli underscore (e tutti i caratteri che non gestisce altrimenti, compresi trattini, punti e simboli come # e ♫) vengono sostituiti entrambi con l’underscore e, ancora peggio, anche il carattere tra il nome di un elemento e un altro sono separati da un underscore. Il tutto porta all’impossibilità di ricostruire correttamente il nome dello stato così come SCTUnit si aspetta.  
   Le seguenti state machine, anche se presentano elementi con nomi chiaramente diversi, generano la stessa identica classe java, ma i test SCTUnit per verificare lo stato iniziale sono diversi:

**Immagine che contiene testo, schermata, diagramma, Rettangolo

Descrizione generata automaticamenteImmagine che contiene testo, schermata, diagramma, Rettangolo

Descrizione generata automaticamente**

**Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente**

**Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente**

**Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente**

Non è possibile sapere al momento della traduzione da Java a SCTUnit se l’underscore è dovuto a uno spazio (o underscore, ma non ci sarebbe problema se la differenza fosse solo questa, in SCTUnit gli spazi bisogna sostituirli con underscore, come si vede nell’esempio sopra) o al passaggio ad uno stato/regione più interno nella gerarchia. Il software è stato implementato ipotizzando che tutti le stringhe rappresentanti nomi di stati e regioni NON contengano spazi o underscore (per gli eventi non c’è questo problema, forse per altri elementi di Itemis Create sì).

Nota positiva: sembra che invece sia gestito il problema di utilizzo di keyword come nome degli eventi, ad esempio potrebbero esserci conflitti o difficoltà di traduzione in SCTUnit se gli eventi potessero chiamarsi “raise”, “enter”, “isFinal” o “isActive”, ma viene segnalato un errore nell’editor di Itemis Create:

Immagine che contiene testo, schermata, Carattere, schermo

Descrizione generata automaticamente

Il problema non è stato però indagato sufficientemente a fondo per stabilire con certezza se problematiche simili possano o meno presentarsi.

Potrebbero esserci altri problemi simili a questi non ancora individuati.

Tutto ciò rappresenta delle forti limitazioni per l’utente e scoraggiano l’uso del tool. Si può pensare di risolvere il problema andando ad analizzare non solo la classe di test generata da evosuite ma anche lo statechart stesso.

### Soluzione

Sono state valutate tre possibili strade:

1. utilizzare le librerie di itemis create per ottenere i nomi degli stati;
2. generare non solo il .java ma anche un .scxml, da cui ottenere i nomi degli stati;
3. leggere direttamente il file .ysc con un DOM parser per ottenere i nomi degli stati.

La prima soluzione non ha portato a risultati per la difficoltà nel capire il funzionamento delle librerie (non documentate e nemmeno commentate). La seconda soluzione portava a leggere un file molto semplice ma non completo: ad esempio, non conteneva alcun riferimento alle regioni degli stati composti. La terza idea è stata vincente.

Inoltre, leggendo il file .ysc si sono ottenuti non solo i nomi degli stati (pronti per essere utilizzati in ambiente SCTUnit), ma anche il nome dello statchart (problema finora non affrontato, visto che si considerava uguale a quello del file contenete lo statchart) e degli eventi.

Per ciascun nome di stato, la sua versione utilizzata in java (enum tutto maiuscolo e con solo underscore) è stata ottenuta, così da poter mappare facilmente l’enum con il corretto stato.

Per ciascun nome di evento, la sua versione utilizzata in java (metodo con prima lettera maiuscola e preceduto da “raise”) è stata ottenuta, così da poter mappare facilmente il metodo con il corretto evento.

Quindi, ricapitolando la situazione ottenuta (ciò che si riesce gestire oppure no) e aggiungendo ulteriori elementi emersi durante l’implementazione della soluzione:

* **Il nome dello statechart** deve essere composto da lettere (anche maiuscole) e numeri, volendo anche l’underscore e forse altri caratteri. In ogni caso, l’editor di itemis create stesso segnala un errore se il nome presenta spazi o caratteri strani, inoltre, se presenta lettere accentate o caratteri simili non può essere visto dai file .sgen. 🡺 **statechart name: solo lettere (maiuscole e minuscole), numeri e underscore.**
* **I** **nomi degli stati e delle regioni** possono contenere praticamente tutto eccetto caratteri particolari come le lettere accentate, che rendono lo stato non identificabile nei file .sctunit. Gli spazi sono ammessi, così come lettere maiuscole, minuscole, numeri e caratteri quali underscore, il punto, il cancelletto e così via. 🡺 **state name: meglio lettere (sia maiuscole che minuscole), numeri, underscore e spazio**, volendo anche altri caratteri ma non si ha la certezza di quali (esempio, no lettere accentate).
* **I nomi degli eventi** vengono gestiti abbastanza bene dall’editor stesso, se si inseriscono spazi, punti, caratteri strani (lettere accentate, cancelletti e così via ma non underscore) viene segnalato un errore. 🡺 **event name: lettere (maiuscole e minuscole), numeri e underscore.**

È ancora possibile che qualcosa non funzioni:

* Due stati (o regioni) o due eventi possono ancora portare ad **uno stesso elemento** (enum o variabile e metodo) **in Java**, in tal caso quando si compila viene generato un errore (dal compilaore, tipo variabile già definita). Quindi non si pone il dubbio sull’inserimento dei dizionari di due elementi con la stessa chiave.
* È possibile mettere in interface degli eventi (in event …) che non vengono utilizzati in nessuna transizione. In tal caso non vengono inseriti nel dizionario. **Prima di accedere ai dizionari si verifica che la chiave sia presente, se non lo è**, come nel caso, **il metodo** (raise o assert) **non viene considerato** (non viene aggiunta l’azione al test case) ma **viene stampato a video**: test55: problems encountered in the translation, the test may fail.

Per verificare che comunque non ci siano problemi con il software, visto che è stato pensato guardando ad una sola macchina, prima di procedere con la copertura di altri elementi dello statchart (come variabili, history ecc.) lo si vuole testare su diverse macchine. Anche eventi con guardie e azioni non sono stati gestiti.

### Ipotesi sull’xml

Il file .ysc è di tipo xmi (quindi un xml). Guardando all’xml di uno statechart di esempio con BaseX, si è implementato il codice per leggere i nomi dello statechart e di stati ed eventi. Sono state fatte quindi alcune ipotesi ragionevoli, visto che non è presente documentazione a riguardo:

* Nel file .ysc, le informazioni interessanti (i.e. la descrizione dello statechart, con nomi e dipendenze) si trova nel sotto albero con **radice l’unico nodo nel file con tag “sgraph:Statechart”**. Tale nodo ha un attributo “name” il cui valore è il nome dello statechart.
* Tra i figli del nodo radice (del nostro sottoalbero di interesse), ci sono le regioni (quindi anche più di una) in cui si sviluppa lo statechart.
* Le regioni sono ELEMENT\_NODE con tag “regions” e hanno un attributo “name” il cui valore è il nome della regione.
* Gli stati sono ELEMENT\_NODE con tag “vertices” e hanno un attributo “xsi:type” il cui valore è il tipo dello statechart, alcuni di questi valori sono:
  + "sgraph:State" per gli stati normali, in questo caso il nodo ha un attributo “name” il cui valore è il nome dello stato.
  + "sgraph:Entry".
  + "sgraph:FinalState".
  + "sgraph:Synchronization".
* Per ogni nodo si vistano solo i nodi figli di tipo ELEMENT\_NODE e con tag “regions” o “vertices”. Quindi, **gli unici nodi visitati sono regioni e stati**. Se il nodo è una regione, si verifica se sia contenuto uno stato finale (si assume che gli stati finali possano essere solo nelle regioni e uno per regione), per cui deve esserne ottenuto il nome. Se il nodo è uno stato (qualsiasi) si controllano le transizioni uscenti (si assume che solo i nodi “vertices”, ossia gli stati, possano avere transizioni in uscita), per cui il nome dell’evento deve essere ottenuto. Inoltre, se lo stato è “normale” (xsi:type uguale a sgraph:State) se ne ottiene il nome.
* Le transizioni sono ELEMENT\_NODE con tag “outgoingTransitions” e hanno l’attributo “specification”, il cui valore rappresenta il nome dell’evento che triggera la transizione (del tipo interfaccia.evento se relativo ad un’interfaccia con nome). Tale attributo sembra esserci sempre (anche se mi sembra che in alcuni casi non c’era, motivo per cui viene fatto un controllo) ma può avere valore pari alla stringa vuota “” (ad esempio, nelle transizioni in uscita da stati iniziali).
* Per ottenere il full name di uno stato “normale” o “finale”, si sale verso i padri. **Ogni padre sarà un altro stato normale o una regione**. In entrambi casi avrà un attributo name da usare. Nell’ottenimento di tale nome, tutti i caratteri NON alfanumerici vengono sostituiti da un underscore.

Guarda direttamente il codice nella classe statechart/statechart.java con l’aiuto di BaseX su uno statechart di esempio per capire di cosa si sta parlando.

## Considerazioni sull’utilizzo di Evosuite

I problemi principali di Evosuite sono tre:

1. Molto lento (**problema efficienza**), ovviamente è il collo di bottiglia della tool-chain e porta le esecuzioni a durare più di due minuti anche per case semplici. Si potrebbe imporre un time budget o dare la possibilità all’utente di imporlo, ma così genera con una copertura minore.
2. Per statechart complessi (si ipotizza, non sono stati ancora implementati casi molto complessi) oppure CycleBased (anche abbastanza semplici) la copertura finale non è molto alta (**problema efficacia**).
3. Molti metodi di test utilizzano metodi della classe java per cui non esiste un corrispettivo in Sctunit e che quindi non possono essere tradotti.

Una soluzione ai problemi è stata individuata nel fornire a Evosuite una classe “semplificata” (o meglio, modificata) che agevoli il suo lavoro. La prima idea (e per ora l’unica) è quella di rendere private tutti i metodi protected (con javaparser). Questo porta ad avere metodi di test che chiamano solo metodi pubblici dell’implementazione java, ossia quei metodi con un corrispettivo elemento in ambiente Statechart/SCTUnit. Quindi non è necessario inserire nella classe TestCaseCollector codice come il seguente:

// Discards methods in which the method .enter IS NOT called or the method .setIsExecuting IS called

**if** (!node.getBody().toString().contains(".enter") || node.getBody().toString().contains(".setIsExecuting")) {

**return**;

}

Che, prima di fare la modificata alla visibilità dei metodi, permetteva di rimuovere casi di test che fallivano perché ad esempio prima settava setIsExecuting a true e poi chiamava enter, che faceva subito il return. Comportamenti quindi particolari e non traducibili in SCTUnit o che portavano a test che falliscono. Però una gestione come questa può magari eliminare test utili (anche se magari non molto) e diventa difficile catturare tutti i comportamenti come questo (ad esempio, il metodo setStateConfVectorPosition () presente se si usano regioni ortogonali è protected, si può “semplicemente” ignorare o quando presente bisogna eliminare il test? Se si rende private il problema non si pone perché non viene mai chiamato nei metodi di test).

Ora viene proposto un confronto tra la tool-chain “standard” e quella che passa per il SimplifiedJava, mostrando che senza quelle linee di codice il caso “simplified” funziona ugualmente, quello “standard” no. Inoltre, si mostra che, inserendo le linee di codice riportate sopra al caso “standard”, non viene modificata l’efficienza di Evosuite e, parrebbe, nemmeno l’efficacia, visto che la copertura dallo statechart a volte è migliore per il caso standard, a volte per il simplified.

Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamente

Queste immagini riguardano un’esecuzione della toolchain nel caso “standard”, senza le righe di codice riportate sopra e con esecuzione dello statechart CycleBased; la test suite fallisce a causa del test35(), a causa del metodo setIsExecuting(), inoltre molti test sono vuoti:



Non fallisce nessun test invece per quanto riguarda il caso “simplified” e il numero di test vuoti è minore (2 anziché 7).

### Esecuzione test

Ora, per verificare efficienza ed efficacia dei due casi, si reintroducono le linee di codice per entrambi i casi (così che anche in quello standard non fallisca nessun test, in quello simplified i metodi senza enter non modificano la copertura e setIsExecuting non può essere chiamato) e si provano più esecuzioni, per cercare di tirar fuori dei trend non validi sul singolo caso ma che possono comunque giustificare o meno le azioni intraprese. Lo statechart utilizzato è il seguente:



Si procede con 20 esecuzione per il caso standard e 20 per il caso simplified, divise in 10 su statechart CycleBased e 10 EventDriven. Si calcolano media e deviazione standard di Coverage (di SCTUnit, di Evosuite, ossia di Junit e del mutation score) e tempo di esecuzione di evosuite (quindi **non** l’intera tool-chain, che comunque non aggiunge molto).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caso “standard” CycleBased** | | | | |
| Numero esecuzione | Coverage SCTUnit [%] | Coverage Evosuite [%] | Mutation score Evosuite [%] | Tempo esecuzione Evosuite [s] |
| 1 | 47 | 78 | 26 | 94.2958185 |
| 2 | 47 | 79 | 25 | 91.8659946 |
| 3 | 38 | 76 | 19 | 93.3244168 |
| 4 | 64 | 83 | 29 | 95.7783336 |
| 5 | 28 | 72 | 18 | 91.0857212 |
| 6 | 38 | 75 | 19 | 91.8576726 |
| 7 | 38 | 75 | 23 | 95.5228463 |
| 8 | 28 | 73 | 15 | 91.7171372 |
| 9 | 47 | 76 | 23 | 93.2535778 |
| 10 | 55 | 81 | 22 | 92.0725595 |
| **Media:** | **43** | **76.8** | **21.9** | **93.0774078** |
| **Deviazione standard:** | **11.34** | **3.46** | **4.20** | **1.654418031** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caso “standard” EventDriven** | | | | |
| Numero esecuzione | Coverage SCTUnit [%] | Coverage Evosuite [%] | Mutation score Evosuite [%] | Tempo esecuzione Evosuite [s] |
| 1 | 64 | 85 | 30 | 122.6090106 |
| 2 | 88 | 93 | 39 | 140.7569529 |
| 3 | 72 | 88 | 35 | 129.1156438 |
| 4 | 64 | 86 | 35 | 123.6148318 |
| 5 | 72 | 89 | 33 | 126.0127144 |
| 6 | 100 | 96 | 37 | 140.2612697 |
| 7 | 100 | 96 | 39 | 149.0567644 |
| 8 | 88 | 93 | 37 | 148.6037865 |
| 9 | 96 | 95 | 37 | 131.0020648 |
| 10 | 64 | 84 | 30 | 121.5287066 |
| **Media:** | **80.8** | **90.5** | **35.2** | **133.2561746** |
| **Deviazione standard:** | **15.18** | **4.65** | **3.29** | **10.58869776** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caso “simplified” CycleBased** | | | | |
| Numero esecuzione | Coverage SCTUnit [%] | Coverage Evosuite [%] | Mutation score Evosuite [%] | Tempo esecuzione Evosuite [s] |
| 1 | 64 | 81 | 25 | 92.3540567 |
| 2 | 64 | 81 | 23 | 94.5280364 |
| 3 | 64 | 85 | 26 | 95.9395932 |
| 4 | 38 | 74 | 21 | 91.8577934 |
| 5 | 47 | 78 | 22 | 92.6160457 |
| 6 | 47 | 79 | 24 | 90.0466194 |
| 7 | 38 | 74 | 18 | 85.9416636 |
| 8 | 47 | 77 | 24 | 89.9062989 |
| 9 | 47 | 75 | 21 | 98.1647117 |
| 10 | 55 | 77 | 19 | 92.5350898 |
| **Media:** | **51.1** | **78.1** | **22.3** | **92.3889909** |
| **Deviazione standard:** | **10.14** | **3.51** | **2.58** | **3.401242031** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caso “simplified” EventDriven** | | | | |
| Numero esecuzione | Coverage SCTUnit [%] | Coverage Evosuite [%] | Mutation score Evosuite [%] | Tempo esecuzione Evosuite [s] |
| 1 | 72 | 89 | 36 | 128.3539082 |
| 2 | 64 | 84 | 27 | 127.9233027 |
| 3 | 100 | 96 | 40 | 142.2338082 |
| 4 | 100 | 95 | 40 | 142.1095877 |
| 5 | 88 | 92 | 36 | 130.8958353 |
| 6 | 72 | 88 | 34 | 130.4133982 |
| 7 | 64 | 85 | 31 | 125.6036149 |
| 8 | 96 | 94 | 30 | 137.7541938 |
| 9 | 72 | 88 | 29 | 136.0723262 |
| 10 | 72 | 88 | 31 | 135.2305576 |
| **Media:** | **80** | **89.9** | **33.4** | **133.6590533** |
| **Deviazione standard:** | **14.48** | **4.15** | **4.53** | **5.907667135** |

Sempre tenendo in conto che si parla di sole 10 esecuzioni per ogni casistica, si nota che innanzitutto la copertura per statechart EventDriven è molto migliore di quella di statechart CycleBased, al prezzo di un maggiore tempo di esecuzione. La maggior difficoltà di Evosuite ad aumentare la copertura è forse dovuta alla necessità di chiamare il metodo runCycle() oltre a raiseEvent() per poter effettivamente attivare una transizione, mentre in EventDriven basta il raiseEvent().

Inoltre, per EventDriven è meglio il caso standard, ma la differenza è veramente piccola e quindi trascurabile e magari dovuta ai pochi test fatti (media migliore solo di 0.8%). Per CycleBased invece la miglioria introdotta da simplified sembra più interessante (media migliore di 8.1%) ma forse è solamente un caso.

Per quanto riguarda invece la deviazione standard, il caso simplified sembra ridurla (anche se di poco) sia nel caso CycleBased che nel caso EventDriven. Una minore deviazione standard è meglio perché significa un non-determinismo meno “evidente”. Anche in questo caso, forse è semplicemente dovuto alle poche esecuzioni.

I tempi di esecuzione invece sono molto simili nel caso CycleBased (meno di un secondo di differenza su 93 tra standard e simplified) e praticamente identici nel caso EventDriven (neanche mezzo secondo su 133).

Per ottenere una maggiore sicurezza sono stati integrati i dati precedenti con ulteriori 20 esecuzioni (sia sul caso CycleBased che EventDriven) con focus sulla sola coverage dei test SCTUnit:

|  |  |  |
| --- | --- | --- |
| CycleBased | | |
|  | **Caso “standard”** | **Caso “simplified”** |
| Min | 27 | 38 |
| Max | 64 | 96 |
| Media | 42.27 | 53.83 |
| Deviazione standard | 12.22 | 14.12 |

Sia il minimo che il massimo risultano maggiori e anche la media (persino maggiore di prima, più del 11%!). La deviazione standard invece è più alta (prima era minore), questo soprattutto per l’outlier Max, ossia il test che ha portato ad un 96% di coverage (uno solo su 30): se fosse stato 72% (secondo valore più alto, verificatosi 4 volte) la deviazione standard sarebbe stata molto simile al caso standard (un po’ più bassa, 12.20). Non si sa, però, come di preciso fornire ad Evosuite una classe in cui sono state rese più stringenti le visibilità di alcuni metodi possa portare a tali miglioramenti nella coverage.

|  |  |  |
| --- | --- | --- |
| EventDriven | | |
|  | **Caso “standard”** | **Caso “simplified”** |
| Min | 64 | 64 |
| Max | 100 | 100 |
| Media | 78.40 | 80.27 |
| Deviazione standard | 13.27 | 13.12 |

Sia il minimo che il massimo risultano identici, a rimarcare questa similitudine tra caso standard e simplified sono la media (a differenza dell’esperimento con soli 10 test per caso, è più alta quella del caso simplified, ma di poco. Niente che possa confermare una superiorità) e la deviazione standard, in cui i due casi sono praticamente identici.

Sul file di documentazione ConfrontoStandardSimplified.xlsx sono presenti tutte le prove (le prime 10 sono quelle di sopra), c’è la possibilità di inserirle nella tesi o meglio magari inserire dei grafici o qualcosa del genere.

Quindi, alla luce delle prove fatte, si evince che, in relazione ai problemi indicati all’inizio della sezione:

1. **Problema efficienza:** non risolto, i tempi di esecuzione di Evosuite rimangono sostanzialmente invariati.
2. **Problema efficacia:** sembra esserci un buon miglioramento nel caso CycleBased, sia in media che nel caso si voglia eseguire più volte con l’obiettivo di ottenere una buona copertura. EventDriven nessun miglioramento tangibile, forse a causa dell’alta copertura garantita anche dal caso standard. Magari in un caso complicato in cui anche EventDriven ha bassa copertura, passare per simplified migliora. Da investigare.
3. Problema chiamata metodi non traducibili: risolto.

Magari tutte queste considerazioni sono da rivalutare alla luce di uno studio approfondito sul funzionamento di Evosuite. Inoltre, si potrebbe fare un paired t-test o qualcosa del genere per valutare meglio la significatività dei risultati.

**NOTA:** ancora i nomi degli statechart e dei vari stati e regioni sono presi dalla classe di test generata da evosuite, quindi è necessario eliminare “Simplified” dove necessario al nome completo dello stato (quello con l’intera gerarchia) in TestCaseCollector e anche per ottenere il nome dello statechart nel metodo genarateSctunit() della classe Generators. Non è una soluzione elegante, e per niente robusta (se simplified è parte del nome di uno stato o qualcosa del genere?). Bisogna gestirla quando si penserà ad ottenere i nomi di state ecc.. da altre fonti. Risolto.

**NOTA:** nella creazione di queste 120 classi di test (ognuna con diversi metodi di test) un metodo non compila:



Era nel caso “standard” EventDriven, il problema poteva verificarsi anche negli altri tre casi. È sicuramente da gestire, ma visto che non è questo il lavoro svolto in questa sezione, l’intera classe di test è stata sostituita con il risultato di una nuova esecuzione (molto improbabile che si verifichi di nuovo proprio in quella nuova visto che è successo una volta su migliaia).

Inoltre, nell’esecuzione dei casi di test EventDriven Evosuite ha lanciato un eccezione, qualcosa simile a questo (non mio):



Poi l’esecuzione delle altre chiamate ad evosuite è andata correttamente e mi sono trovato due test identici (il 3 e il 4 del caso standard, generati circa in quel momento). Il problema è stato probabilmente che il PC era sconnesso da internet e in quel momento si è connesso. La cosa sembra aver in qualche modo disturbato Evosuite. Comunque, il test 4 è stato rigenerato.

## Ottenimento del jar

Il risultato finale deve essere un jar il più possibile auto contenuto e con una interfaccia decente. Per ottenere il jar si utilizza Maven.

### Maven

Per creare il jar bisogna fare tasto destro sul progetto, Run as, Maven build (quello che esegue direttamente) oppure il comando: mvn clean package. In target vengono prodotti due Jar, quello “original” e quello shaded (che ha il nome normale). Il tutto funziona grazie ai seguenti plugin nel pom:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>3.2.4</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

<mainClass>

generators.TestGenerator

</mainClass>

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<source>${java.version}</source>

<target>${java.version}</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

Il primo serve per buildare un uber jar (praticamente un jar auto contenuto con tutte le dipendenze) con shading (credo significhi spostare o rinominare degli elementi per evitare conflitti). Il secondo specifica che versione si java usare per la build; senza, da errori tipo l’operatore diamante (ossia new ArrayList<>()) non è previsto in java 1.5 o qualcosa del genere.

**Per risolvere il problema degli warning stampati da evosuite è stato utilizzato un altro plugin per generare il jar. Si faccia riferimento alla prossima sezione.**

Inoltre, è stato necessario rendere visibili i file .stg all’applicazione anche durante l’esecuzione del jar (innanzitutto includendoli nel jar). Per far ciò, sono stati spostati dalla cartella “template” a “src/main/resource” già esistente (creata automaticamente quando si crea il progetto maven). Inoltre, nel POM è stato necessario inserire:

<resources>

<resource>

<directory>src/main/resources</directory>

<includes>

<include>sgen\_java\_template.stg</include>

<include>sctunit\_template.stg</include>

</includes>

</resource>

</resources>

Ora per accedervi basta specificare come “path” il nome stesso del file, ad esempio:

STGroupFile group = **new** STGroupFile("sgen\_java\_template.stg");

In ogni caso, guarda direttamente al POM per vederlo nella sua interezza e aggiornato.

### Evosuite

Innanzitutto, è stato utilizzato direttamente evosuite-1.0.6.jar scaricato in locale e non la dipendenza maven che si trova su internet. Per fare ciò è necessario (anche da terminale di eclipse) eseguire il seguente comando (che mette nella cartella .m2 tutto ciò che serve):

mvn install:install-file -Dfile="C:\Users\lenovo\Desktop\evosuite-1.0.6.jar" -DgroupId="evosuite" -DartifactId="evosuite" -Dversion=1.0.6 -Dpackaging=jar

E inserire come unica dependency nel POM:

<dependency>

<groupId>evosuite</groupId>

<artifactId>evosuite</artifactId>

<version>1.0.6</version>

</dependency>

Che va a cercare la dipendenza nella cartella .m2.

In questo modo le Maven Dependencies risultano molto ridotte, il jar creato sembra più leggero e non è necessario specificare due dependencies come nel caso:

<dependency>

<groupId>com.sun</groupId>

<artifactId>tools</artifactId>

<version>1.8</version>

<scope>system</scope>

<systemPath>${env.JAVA\_HOME}\lib\tools.jar</systemPath>

</dependency>

<!--

https://mvnrepository.com/artifact/org.evosuite.plugins/evosuite-maven-plugin -->

<dependency>

<groupId>org.evosuite.plugins</groupId>

<artifactId>evosuite-maven-plugin</artifactId>

<version>1.0.6</version>

<exclusions>

<exclusion>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

</exclusion>

</exclusions>

</dependency>

Dove la prima serve per evitare un errore nel POM che non ho ben capito.

Fatto ciò, tutto funziona ma nell’esecuzione vengono stampati molti warning come il seguente:

[MASTER] 10:48:25.123 [logback-1] WARN JUnitAnalyzer - Found unstable test named initializationError -> class java.lang.RuntimeException: Cannot find either the compilation target folder nor the EvoSuite jar in classpath: C:\Users\lenovo\AppData\Local\Temp\EvoSuite\_pathingJar4591789224065816540.jar

[MASTER] 10:48:25.188 [logback-1] WARN AgentLoader - Failed to find EvoSuite jar in current classloader. URLs of classloader:  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/jaccess.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/sunec.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/../lib/tools.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/access-bridge-64.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/sunjce\_provider.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/nashorn.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/dnsns.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/localedata.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/jfxrt.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/sunpkcs11.jar  
file:/C:/Users/lenovo/AppData/Local/Temp/EvoSuite\_pathingJar4591789224065816540.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/sunmscapi.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/zipfs.jar  
file:/C:/Program%20Files/Java/jdk1.8.0\_202/jre/lib/ext/cldrdata.jar

È possibile rimuovere questo warning andando a specificare l’opzione per evosuite -evosuiteCP, che si aspetta il path del file del jar di evosuite. Tale jar non è presente nel jar della mia applicazione e passare direttamente il jar della mia applicazione non funziona.

Il problema è stato risolto generando non uno shaded jar ma sia un normale jar (senza le dipendenze necessarie all’interno) che una cartella libs contenente le dipendenze richieste. Per far ciò, anziché maven-shade-plugin, si sono usati i plugin:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<version>3.6.1</version>

<executions>

<execution>

<id>copy-dependencies</id>

<phase>prepare-package</phase>

<goals>

<goal>copy-dependencies</goal>

</goals>

<configuration>

<outputDirectory>

${project.build.directory}/libs

</outputDirectory>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>3.3.0</version>

<configuration>

<archive>

<manifest>

<addClasspath>true</addClasspath>

<classpathPrefix>libs/</classpathPrefix>

<mainClass>

generators.TestGenerator

</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

Il primo serve per creare una cartella chiamata libs contenente tutte le Maven dependencies del progetto e di cui il jar avrà bisogno. Il secondo plugin è quello che genera il .jar vero e proprio, in cui viene specificato anche il classpath (ossia la cartella libs), che non sarà più uber e quindi quando eseguito deve trovarsi in una cartella contenente la cartella libs.

Non essendo un uber jar, il jar così ottenuto è molto più compatto, da 32MB a 28KB, ovviamente tutti quei dati ora si trovano nella cartella libs, che pesa infatti 31.6MB.

Si fa notare, inoltre, che se non si fosse usato lo stratagemma di usare evosuite-1.0.6.jar installato sulla macchina come dipendenza, usando invece le dipendenze evosuite che si trovano sul sito mvnrepostory.com, oltre ad avere la necessità di specificare una dependency in più, che tra l’altro richiedeva di specificare il path di un’installazione java, la cartella libs sarebbe stata molto più grande (85MB) e piena di jar:

Immagine che contiene testo, menu, schermata

Descrizione generata automaticamente

Anziché:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Tutti quei jar in più sono probabilmente inutili visto che tra essi c’è evosuite-master-1.0.6.jar che da solo pesa quanto evosuite-1.0.6.jar e sembrano lo stesso artefatto (cambia solo il nome). Quindi, secondo me, sono entrambi uber jar (o qualcosa di simile), visto che si possono eseguire da soli, e quindi basta la dipendenza al solo evosuite-1.0.6.jar.

La presenza del jar antlr-runtime-3.3.jar è dovuta a stringtemplate.

Un ulteriore problema legato ad Evosuite: è possibile che ci metta troppo tempo a minimizzare i test, finora mi è successo solo quando eseguivo il jar con il portatile non collegato alla corrente:

[MASTER] 08:02:38.904 [logback-1] WARN TestSuiteMinimizer - Minimization timeout. Roll back to original test suite

A quel punto smette di provarci e i risultanti test sono più difficili da leggere, meno sensati e con molte operazioni inutili:



Presenta anche statements e asserzioni che il mio codice non riesce a coprire, ad esempio:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Il tutto risulta in un sctunit con alcuni test che falliscono:

Immagine che contiene testo, schermata, software, Icona del computer

Descrizione generata automaticamenteImmagine che contiene schermata, testo, modello, bianco e nero

Descrizione generata automaticamente

E ugualmente poco sensati e difficili da leggere:

Immagine che contiene testo, schermata

Descrizione generata automaticamente

Aggiungendo Dassertion\_minimization\_fallback\_time=1.0 e -Dminimization\_timeout=600 alla chiamata di Evosuite non viene fatto fallback (credo sia un controllo che viene fatto dopo x secondi, se non sono stati minimizzati abbastanza test smette di minimizzare) e viene impostato il tempo massimo per la fase di minimizzazione a 10min. Anche con il portatile non collegato alla corrente e sottoposto a grande stress la minimizzazione termina correttamente, anche se magari impiegando tanto tempo, e i test generati sono quelli “soliti”.

### Itemis create

Come detto, si vuole rendere l’applicazione il più auto contenuta possibile. Il che vorrebbe dire avere tutto ciò che serve per generare il codice java nel jar. Il problema è che per adesso viene chiamato il file scc.bat contenuto nell’installazione di itemis create. Tale file ha bisogno e chiama il file headless.ini ma anche itemieCREATEc.exe, quindi, non sembra utilizzabile così com’è ma dovrei utilizzare direttamente le librerie. Il problema è che le librerie sono già state analizzate e non si riesce a capire come utilizzarle.

Un’opzione (non molto bella) è quella di richiedere in ingresso il path del file scc.bat. Ciò richiede all’utente di preoccuparsi di un input in più e di doversi occupare anche dell’installazione di itemis create standalone (certo è che se vuole generare test case per uno statechart .ysc è già familiare con itemis stesso). Però sorge un problema: e se le nuove versioni di itemis create apportano modifiche al generatore di codie java? **Bisognerebbe imporre all’utente di scaricare una versione di itemis create compatibile**, limitandolo ancora di più.

Alla fine, si è optato per la prima opzione, anche perché l’obiettivo di rendere l’applicazione il più auto-contenuta possibile me lo sono imposto io e non so quanto sia sensato. Inoltre, non so se si può includere Itemis Create nel jar, visto che non è open source ed ha bisogno di licenza. L’utente del mio applicativo dovrà quindi scaricare Itemis Create con una versione che sia compatibile (magari funziona anche con altre versioni, ma non è garantito o magari “funziona peggio”, generando test peggiori) e indicare il path del file scc.bat come opzione della CLI.

Inoltre, nel testare il file .jar mi sono accorto che chiamare scc.bat in una cartella con tanti file .ysc e .sgen, genera il codice per tutti. Per dire al generatore di guardare solo ai due file che mi interessano (il .ysc o .sct e il .sgen) si utilizza l’opzione -m quando si chiama il file scc.bat, ad esempio:

C:\Users\lenovo\Desktop\itemis\_CREATE\scc.bat -m Statechart.ysc,ExamplStc.sgen

Il file scc.bat viene scaricato anche quando si installa il plugin di itemis create per Eclipse, però questo non è direttamente utilizzabile. In particolare, dopo aver installato il plugin, nella cartella di eclipse la situazione è la seguente:

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

Dove scc.bat e headless.ini sono stati aggiunti installando il plugin. Il problema è che entrambi i file non vanno bene. Veniva dato l’errore:

"C:\Users\lenovo\Desktop\eclipse\itemisCREATEc.exe"" non ? riconosciuto come comando interno o esterno, un programma eseguibile o un file batch.

Quindi, in scc.bat bisogna sostituire da itemisCREATEc.exe e eclipsec.exe:

@SET wd=%~dp0  
@"%wd%eclipsec.exe" --launcher.ini "%wd%headless.ini" %\*

Questo non è sufficiente, perché viene dato errore:

A Java Runtime Environment (JRE) or Java Development Kit (JDK)

must be available in order to run ItemisCREATEc. No Java virtual machine

was found after searching the following locations:

plugins/org.eclipse.justj.openjdk.hotspot.jre.full.stripped.win32.x86\_64\_17.0.5.v20221102-0933/jre/bin

L’errore non è esattamente questo, per lo meno il path forse non è lo stesso, comunque molto simile, almeno fino a stripped.win32.x86.

Per risolverlo, ho modificato il file headless.ini, in particolare è stato modificato in

-vm  
plugins/org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_17.0.10.v20240120-1143/jre/bin

Mettendo il path che ho trovato nel file eclipse.ini.

In ogni caso, non ho indagato molto approfonditamente e i problemi sembrano essere dovuti a Itemis Create, che crea dei file (scc.bat e headless.ini) come se si trovasse nella directory di Itemis Create standalone. In ogni caso la soluzione ha senso e sembra funzionare perfettamente.

### User experience e CLI

È stata implementata un’interfaccia a linea di comando (CLI) con apache commons cli. L’interfaccia gestisce tutti i possibili input errati e presenta il comando -help per imparare ad usarla. L’unico problema rimasto è che non è possibile bloccare l’esecuzione con Ctrl+c quando il “comando” passa ad evosuite, visto che il mio security manager in checkExit() semplicemente lancia un eccezione. Se si preme Ctrl+c due volte o più l’esecuzione continua normalmente e in più viene stampato a video:

Exception in thread "SIGINT handler" java.lang.SecurityException  
 at support.MySecurityManager.checkExit(MySecurityManager.java:39)  
 at java.lang.Runtime.exit(Runtime.java:107)  
 at java.lang.System.exit(System.java:971)  
 at org.evosuite.utils.ExternalProcessHandler$6.handle(ExternalProcessHandler.java:572)  
 at sun.misc.Signal$1.run(Signal.java:212)  
 at java.lang.Thread.run(Thread.java:748)

Nel tentativo di risolvere la questione, si è modificata la gestione della terminazione di Evosuite: anziché utilizzare il mio security manager durante l’esecuzione di Evosuite, che, come visto, dava problemi, si è deciso di copiare il contenuto del metodo Evosuite.main() anziché chiamarlo, tralasciando la problematica chiamata System.exit(). In realtà è stato eliminato anche il try catch, passando da:

**try** {

EvoSuite evosuite = **new** EvoSuite();

evosuite.parseCommandLine(args);

} **catch** (Throwable t) {

logger.error("Fatal crash on main EvoSuite process. Class "

+ Properties.TARGET\_CLASS + " using seed " + Randomness.getSeed()

+ ". Configuration id : " + Properties.CONFIGURATION\_ID, t);

System.*exit*(-1);

}

System.*exit*(0);

Alle sole istruzioni:

org.evosuite.EvoSuite evosuite = **new** org.evosuite.EvoSuite();

evosuite.parseCommandLine(evoArgs.toArray(**new** String[evoArgs.size()]));

Il try cacth non dovrebbe essere utile visto che il mio codice si preoccupa di gestire correttamente il setup generale dell’ambiente di esecuzione (i path specificati esistono e contengono i file necessari) e passa degli argomenti (almeno teoricamente) formattati correttamente.

Quindi, a questo punto non è più necessario gestire System.exit(), e la classe MySecurityManager è inutile. Parte del problema è però rimasta: durante l’esecuzione di evosuite, premendo Ctrl+c, non viene generata più nessuna eccezione, però l’esecuzione continua e termina alla fine dell’esecuzione di Evosuite (quindi, l’esecuzione del mio jar termina, ma non subito, bensì solo quando Evosuite ha finito, questo è un po’ problematico visto che è proprio evosuite la parte corposa dell’esecuzione). A seguito di qualche tentativo fallito, si è deciso di lasciar la situazione così, tenuto anche in conto che eseguendo direttamente java -jar evosuite-1.0.6.jar, evosuite si comporta nella stessa identica maniera.

**Ora gli argomenti sono gestiti con apache commons CLI** per avere un’interfaccia a linea di comando migliore. Guarda direttamente l’help della CLI per capire che opzioni sono necessarie e quali no. Quelle non obbligatorie servono ad ottenere un output migliore (ad esempio -evoTestDir e -targetDir, per cui se non specificate viene usato un valore di default). C’è anche -evoSearchBudget per specificare il budget di evosuite. Quindi, per ora, da una situazione iniziale del genere:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Con un comando come:

-scc C:\Users\lenovo\Desktop\itemis\_CREATE\scc.bat   
-projectPath C:\Users\lenovo\Desktop\ProgettiGit\test-generator-for-yakindu\junit-test-generation-example\Temp -sourceDir models\sct -sourceFile Statechart -targetPackage statechart -evoTestDir test  
-evoSearchBudget 10

Si ottiene alla fine:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Nota che bisogna specifica la posizione dello script di itemis (scc.bat) e che anche senza tutti i possibili argomenti è già molto pesante. Anche con solo quelli obbligatori risulta molto pesante. Inoltre, se non ci fosse stato la source directory test prima dell’esecuzione, sarebbe stata creata una normale cartella test da cui non si sarebbero potuti eseguire i test.

## Test dell’applicazione

Si vuole verificare come si comporta la mia applicazione su statechart non fatti da me ma reperiti su internet. Questa sezione riguarda la versione 0.0.1 del mio eseguibile, ossia quella relativa all’ottenimento del primo jar corretto e con CLI. In tale versione, non sono supportate interfacce con nome e le variabili sono “ignorate”. Questa non è la sezione riguardante i test d’unità (che ancora devono essere fatti). Purtroppo, si sono trovati solo gli esempi forniti da Itemis Create stesso, quelli scelti si trovano ai seguenti link (per i primi tre bisogna trascinare il pulsante di download su Eclipse):

* <https://www.itemis.com/en/products/itemis-create/documentation/examples/com-yakindu-sct-examples-trafficlight-multism?hsLang=de>
* <https://www.itemis.com/en/products/itemis-create/documentation/examples/com-yakindu-sct-examples-coffeemachine-c>
* <https://www.itemis.com/en/products/itemis-create/documentation/examples/com-yakindu-sct-examples-codegen-multism-java?hsLang=de>
* <https://github.com/itemisCREATE/example-turn-signal-control/tree/main/turn-signal-control-bluepill/Model>

I modelli si trovavano nel Progetto ExaplesFromItemis (workspace junit-test-generation-example). Il path sulla macchina è: C:\Users\lenovo\Desktop\ProgettiGit\test-generator-for-yakindu\junit-test-generation-example\ExamplesFromItemis\model. Per ognuno, viene riportato di seguito il risultato:

* traffic-light: non controllato.
* coffe-machine: lo statechart CoffeMachine.ysc è pieno di errori visto che è un esempio di deep C, quindi non ci si aspetta niente di buono. Infatti, la chiamata del generatore di Itemis Create lancia delle eccezioni perché: This generator can not be used for domain com.yakindu.domain.c. Valid domains are [com.yakindu.domain.java, com.yakindu.domain.default]. Anche da Eclipse il file .sgen generato mostra lo stesso errore, dalla finestra di aiuto di quest’ultimo si può modificare il domain in com.yakindu.domain.java, però il generatore continua a non generare niente. A questo punto il file .ysc è accessibile in sola lettura e viene mostrato l’errore: “domain 'com.yakindu.domain.java' is not available. – editor is in read olny mode”. Dovevo probabilmente cambiare il domain in default, visto che quando creo un nuovo statechart questo è lo wizard:

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

Nella documentazione viene detto: “In the wizard, select a statechart domain. Which domains are available depend on your itemis CREATE license. The Default domain for “normal” statecharts without any language integration is always available”. Non c’è java ma C/C++ sì perché si sta parlando di: “itemis CREATE 'C/C++ Domain' is a domain tailored to work perfectly together with your C and C++ projects. Choose this domain if you want to use the power of statecharts in your existing C/C++ project and want to use custom types and call existing functions from your statechart. Only the C/C++ Domain allows you to use the Deep C/C++ Integration of itemis CREATE. The 'C/C++ Domain' requires a 'Professional License'”. Questa cosa, ad oggi, per java non sembra ancora disponibile (però lasciano la possibilità di modificare il dominio in com.yakindu.domain.java attraverso la finestra di errore, magari ci stanno lavorando).

* simple-light: il jar crea problemi solo con light\_controller.ysc perché contiene delle interfaccie con nome, quindi il .java simplified che creo presenta errori di compilazione. Per light.ysc viene generato un sctunit con copertura scarsa (50%) visto i soli due stati, questo perché non gestisco le variabili**. Nota: anche se mi sono dimenticato di indicare l’opzione -t, sono comunque state generate le interfacce ITimed.java e ITimerService.java, da indagare meglio.**
* turn-signal-control: non controllato.

A volte, come nel generare i test per queste prove, il mio jar smette di funzionare (anche l’esecuzione direttamente da Eclipse) correttamente: il generatore di itemis Create da un errore su input su cui prima funzionava correttamente e dà lo stesso errore su input diversi. Per sistemare, basta riavviare il computer, forse invece basta chiudere itemis create. L’errore è il seguente:

ERROR com.yakindu.sct.generator.application.GeneratorApplication - Error  
java.lang.NoClassDefFoundError: org/eclipse/core/resources/IResource

Non so se è sempre questo o a volte altro. La situazione sarebbe da indagare meglio, però non so come riottenere l’errore.

Comunque, visto che tutti i file provati hanno delle criticità, non ha senso provare il mio software su tutti e documentare, ho smesso quasi subito per procedere con l’implementazione del codice necessario a coprire interfacce con nome e variabili (per fare la versione 0.0.2). Il progetto qui descritto è stato eliminato.

## Versione 0.0.2

Il principale obiettivo della versione 0.0.2 è quello di coprire statechart più complessi, in particolare statechart che contengono interfacce con nome e variabili. Di seguito vengono illustrate le considerazioni emerse durante lo sviluppo. Si faccia riferimento alla seguente immagine.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Guarda il link <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/quick_ref_definition_section#quick_ref_definition_section> per avere un’idea più chiara su a cosa si vuol mettere mano. Ci sono ancora molte cose che non conosco, come il payload degli eventi (ossia eventi con tipo), la possibilità di indicare più eventi sulla stessa transizione ecc.

**Nota:** nello sviluppo della versione 0.0.2 si sono sviluppate diverse macchine. Nel modificare una di queste, itemis create si è bloccato e ho dovuto chiuderlo forzatamente. Nel riaprire lo workspace che ha dato errore mi viene stampato un errore: *an error has occurred, see the .log file ecc.*. Nel log file (si tova nei .metadata dello workspace) ci sono un po’ di messaggi tipo *!MESSAGE The workspace exited with unsaved changes in the previous session; refreshing workspace to recover changes.* seguiti da diverse centinaia di righe di errori ed eccezioni. Forse non è la prima volta che mi capita. Ho eliminato i .metadata dello workspace e riaperto itemis create, a questo punto è necessario importare i progetti attraverso *import -> Existing projects into workspace*. Non mi segna più che sono legati ad una repo GitHub ma sembra comunque che le modifiche vengano viste (e quindi si possono committare) su GitHub Desktop.

**Nota 2:** Ad un certo punto, ho deciso di eliminare tutti i metodi di test in junit che contengano una istruzione try catch, visto che in sctunit le eccezioni non ci sono. Inoltre, risultavano sempre (o quasi, credo) in test sctunit vuoti.

**Nota 3:** il codice ottenuto alla fine di questa “Versione 0.0.2” funziona con Itemis CREATE versione 5.2.1.202312211636. Una qualsiasi modifica in future versioni ad itemis create e a come gestisce sctunit (es. sintassi delle istruzioni) o al codice generato (es. nome di metodi come setTimer o setTimerService) potrebbe rendere il mio tool inutile (errori durante l’esecuzione) o meno performante (copertura più scarsa o test che falliscono o ancora peggio test che non compilano).

### Interfacce con nome

Per indagare questo aspetto si è utilizzato lo statechart Light.ysc.

È stato necessario modificare il modo in cui veniva ottenuto il .java “Simplified”, visto che la presenza di interfacce con nome implica la presenza di inner class (nested) con nome e costruttore a cui non si vuole aggiungere “Simplified” nel nome. Inoltre, queste nested class hanno dei riferimenti alla classe genitore (nel parametro del costruttore e in un campo), per cui il tipo deve essere modificato. Inoltre, la classe principale ha, per ogni nested class (dovute alle interfacce con nome), un campo di tipo la nested class, questi campi devono essere resi privati per facilitare la vita ad evosuite, che altrimenti nei test inizia ad impostarli a null o assegnargli oggetti creati da lui (quando invece devono puntare agli oggetti creati nel costruttore). Nota positiva, se si usa come nome per l’interfaccia una parola chiave come switch, itemis create la gestisce bene inserendo \_ID dove necessario. Il nome delle interfacce si comporta esattamente come il nome degli eventi, quindi, caratteri non supportati vengono segnalati con errore, nomi che differiscono solo per il case vengono segnalati con warning ed in ogni caso la prima lettera in java sarà maiuscola, quindi è meglio ottenerli dal file dello statechart direttamente (i nomi degli eventi in outgoingTransition sono del tipo interfaccia.nome se relativi ad un’interfaccia con nome). Quindi, sono permesse **lettere (maiuscole e minuscole), numeri e underscore.**

### Variabili e costanti

Per indagare questo aspetto si sono utilizzati gli statechart Light.ysc e EasyChoice.ysc.

Le variabili e costanti possono essere integer, real, boolean o string. Possono essere nell’internal scope, nell’interfaccia default o nelle interfacce con nome. Le variabili possono essere readonly.

È emerso un problema importante: il modo in cui ottengo il nome di interfacce e eventi non va bene, se ci sono guardie o azioni nella transizione, vengono riportati in Specification nella outgoingTransition insieme all’evento, ad esempio (&quot; sono i doppi apici):

specification="color.change / color.value = &quot;RED&quot;"

oppure

specification="switch.increaseBrightness &#xD;&#xA;[brigthness &lt; maxBrightness]&#xD;&#xA; / brigthness+=10"

ma anche se semplicemente si mettono degli white space (ad esempio a capo o spazio) dopo l’evento senza mettere guardie o azioni:

specification="switch.toggle&#xD;&#xA;&#xD;&#xA;&#xD;&#xA;&#xD;&#xA;"

specification="switch.toggle "

Soluzione, ottenere la sottostringa da 0 alla posizione del primo carattere che non sia una lettera (maiuscola o minuscola), un numero o il punto (questa soluzione non va bene, è stata modificata, guarda la sezione trigger complicati).

Inoltre, quando ci sono delle variabili nelle interfacce (sia con che senza nome), vengono generati due metodi pubblici get e set che vengono usati da Evosuite (**bene**) e una variabile, anch’essa pubblica (**male**, non voglio che sia usata da evosuite). Anche le costanti sono pubbliche (**male**) ed hanno un metodo get pubblico (**bene**). Passando a internal, né le variabili né le costanti possono essere utilizzate in sctunit; nel codice generato, le variabili sono private (**bene**) e i metodi get e set protected (e quindi private nel simplified, **bene**), la costante invece è pubblica (**male**, sarebbe da rendere privata), mentre il suo metodo get è protected (e quindi private nella versione simpilified, **bene**).

L’idea è quindi rendere tutte le variabili private, non si vuole che Evosuite acceda direttamente alla variabile, e mantenere public i metodi get e set delle interfacce, così che l’utilizzo delle variabili da parte di evosuite avvenga solamente mediante metodo get e set, questa è una mia scelta che dovrebbe facilitare la traduzione dei test case in junit. A livello implementativo, la gestione della visibilità dei metodi è già sufficiente, bisogna lavorare invece sulle variabili pubbliche da rendere private. Semplicemente nella classe che modifica la visibilità dei campi da protected a private, si aggiunge la modifica da public a protected.

Sembra però che il più delle volte, quando lavora con le variabili, Evosuite prima utilizzi il set e poi faccia un assertEquals sul get della stessa variabile e con il valore del set, non molto interessante. Si potrebbe pensare di rendere private il metodo set e lasciare solo i get, così che i valori delle variabili vengano modificati solo da transizioni e azioni degli stati. In alcuni casi riesce a fare dei controlli interessanti sulle variabili, ma a volte, gli unici controlli che fa sembrano essere quelli sul loro valore iniziale. Un’ulteriore ipotesi è mascherare completamente le variabili ad evosuite, riduce sicuramente il lavoro da fare (ad esempio, come ottengo il nome corretto delle variabili?) e sembra quasi produrre test migliori (si potrebbe verificare confrontando con il caso non Simplified). Il dubbio è quale sia la scelta migliore. Inizialmente, si è deciso di nascondere tutto (anche perché gestire o meno le variabili non cambia la copertura finale).

### History ed entry/exit point references

Per indagare sugli history state si è utilizzato lo statechart History.ysc.

Il software implementato era già sufficiente a coprire gli history state (è emerso un piccolo bug nel testarlo, se gli eventi avevano un underscore, il nome ottenuto si fermava prima dell’underscore). Non sembra creare test molto significativi riguardo agli history state, li copre ma non sembra verificare lo stato subito dopo averli attivati, il miglior test case visto a riguardo è il seguente:

@Test

**operation** test46 () {

**assert** !**is\_active**

**enter**

**raise** shallow\_continue

**assert** **is\_active**

**assert** **active** (History.main\_region.Answering\_Questions.r1.q1)

}

Per indagare su entry ed exit point multipli con nome si è utilizzato lo statechart EntryExit.ysc.

Il software implementato era già sufficiente a coprire entry ed exit point multipli con references. Attenzione che comunque non so come effettivamente il codice java implementi questi history states ed entry ed exit point references, si potrebbe, volendo, dargli un’occhiata.

### Subdiagrams

Per indagare sui subdiagrams si sono utilizzati gli statechart SimpleLight.ysc, Controller.ysc e Controller2.ysc (nella cartella sub-machines).

Il problema è legato a quello delle variabili, visto che i subdiagrams non sono altro che variabili il cui tipo è la classe della macchina da importare. La classe java non compila se non è possibile importare la classe del subdiagram (quindi eccezione del mio codice in fase di compilazione), bisogna prima eseguire il mio tool sui subdiagrams o quantomeno generarne il solo codice.

Inoltre, adottando l’approccio deciso sopra per le variabili (nasconderle completamente a evosuite) tutto funziona, ma non so perché la copertura risulta molto bassa e la test suite scadente. Con le visibilità di default il numero di test è molto maggiore così come la copertura, però anche con le visibilità ridotte evosuite dovrebbe arrivare al 100% su una macchina semplice. Sembra che non riesca ad entrare in metodi in cui si facciano operazioni relative ai subdiagrams. Ad esempio, la copertura del codice spesso è 0% (perché non viene mai chiamato enter) in una situazione come la seguente:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Per avere una buona copertura bisogna mettere public i metodi set (anche i get se si vuole ma non cambia niente) delle variabili relative a subdiagrams. Il problema sarebbe identificare quali variabili riguardano subdiagrams e quali sono invece “normali”. Se si mettono pubblici i set di variabili nomali ma non si traducono in assegnamenti sctunit, i test potrebbero fallire (ad esempio se ci sono choice state). Soluzione: visto che i tipi definiti in itemis create sono sempre i soliti quattro (void è l’assenza di tipo):

* integer -> long in java
* real -> double in java
* boolean -> boolean in java
* string -> String in java
* void

Basterebbe controllare il tipo del parametro del metodo set per capire se si può lasciare pubblico oppure no! Questo rende ridondante il controllo fatto per verificare se il metodo è setOperationCallback (tipo del parametro: OperationCallback) o setTimerService (tipo del parametro: ITimerService), visto che il loro tipo non è uno di quelli riportati sopra. Viene lasciato comunque per sicurezza (programmazione difensiva). Mentre rende privato setIsExecuting (fortunatamente, ma potrei mettere anche qui un controllo aggiuntivo) perché il parametro è boolean.

Nota: se chiamo una variabile int o private o altro, me lo fa fare perché poi nel codice generato la variabile si chiama intVariable, privateVariable e così via. Avevo già riscontrato un comportamento simile con i nomi delle interfacce, a cui aggiungeva \_ID alla fine. Sembra proprio quindi che non c’è bisogno di preoccuparsi per nomi in itemis create che corrispondono a keyword in java.

NOTA: LA COPERTURA DI UNA STATEMACHINE CHE NE UTLIZZA ALTRE, DA LA COPERTURA ANCHE DELLE ALTRE

**Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente**

Dubbio: e se non fosse solo questo il caso in cui modificare la visibilità peggiora la test suite generata? Magari anche in altri casi con le variabili sarebbe meglio lasciarle visibili ad evosuite e magari poi ignorarle.

### Internal vs external events, incoming vs outgoing external events e typed events

Per indagare sui vari tipi di eventi (interni, out, typed…) si è utilizzato lo statechart Events.ysc.

Gli eventi internal non possono essere chiamati in sctunit, quindi il fatto che il mio software modifichi la visibilità del metodo associato a private è corretto.

Nota: in sctunit su eventi interni (in event) si possono fare solo operazioni di raise, su quelli esterni (out event) solo operazioni di assert:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Per la copertura dello statechart, la cosa più importante è coprire i casi di raise di in event con tipo (è anche l’unico caso che per ora il mio software non copre, si ferma a raise typedInEvent che da errore in compilazione). Pensavo di nascondere completamente gli out event (sia raise che get nel caso di typed event) e solo il get dei typed in event (in realtà questo viene già fatto perché sono tutti metodi protected). Altrimenti la cosa sembra complicarsi parecchio e magari per pochi casi di test non molto significativi (al solito, evosuite non sembra mai utilizzare intelligentemente i metodi get relativi agli out event).

Così facendo non si faranno mai verifiche sugli out events, così come non vengono fatte verifiche sulle variabili.

Nota: se l’outevent, sia typed che non, viene utilizzato in una guardia, nell’implementazione java vengono inseriti dei nuovi campi e metodi in cui viene utilizzato local\_ davanti al nome:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Inoltre, per gli out events (sia typed che non), vengono generati dei metodi come:



Sono pubblici, ma visto che iniziano con get vengono messi privati. Quindi una feature implementata per le variabili torna utile qui, ma se si vuole cambiare la gestione della visibilità delle variabili, questo potrebbe rimanere public, attenzione.

Quindi, tornando allo sviluppo del codice per gestire in events con tipo, ci sono 5 diversi tipi in itemis create: integer, real, boolean, string, void. Di seguito vengono riportati alcuni esempi da tradurre:

eventsSimplified0.raiseStr("OK");  
eventsSimplified0.raiseStr((String) **null**);  
eventsSimplified0.raiseB(**true**);  
eventsSimplified0.raiseI(76L);  
eventsSimplified0.raiseTypedInEvent((-403.327540894101));

Si notano alcuni problemi: primo di tutto il (String) null utilizzato al posto di una stringa, in sctunit non si può usare il null quando si aspetta una stringa; quindi, il metodo deve essere scartato (la chiamata di metodo non viene tradotta, non l’intero test case, ma non dovrebbe creare problema perché fare raise null non porta a nessuna modifica nello stato dello statechart). Si vede che i numeri negativi sono circondati da parentesi, che si rimuovono. I numeri interi sono seguiti da una L perché in java sono long, bisogna togliere la L. I booleani vanno bene così. Void, per sctunit ma probabilmente in generale, è come non avere tipo.

La copertura dello statechart risulta comunque molto bassa (comprensibile, visto quanto è complicato andare allo stato StateB, è impossibile che arrivi allo stato StateF).

### Operations

Per indagare sulle operations si è utilizzato lo statechart OperationTest.ysc.

Le operations possono essere messe solo nelle interfacce (in internal da uno warning: Internal declaration is not used in statechart.). In pratica per ogni interfaccia (anche quella “senza nome”), se ha almeno una operation, nel codica java viene inserita una interfaccia ed il metodo setOperationCallback() che si aspetta un oggetto la cui classe estenda quell’interfaccia. Ciò ha creato due problemi:

1. Il mio codice aggiungeva Simplified al nome dell’interfaccia, quindi ho modificato la classe che visita le classi per indicare che se è un’interfaccia non bisogna fare niente.
2. Il mio codice rende privati i metodi setOperationCallback(), il che rende impossibile ad evosuite generare dei casi di test interessanti (li crea piccoli ed ogni evento che riguarda anche una operation è circondato da try catch). Nella classe che visita i metodi, non viene fatto niente ai metodi che si chiamano “setOperationCallback”.

Risolti questi problemi, tutto funziona correttamente, evosuite usa i mock per gestire le interfacce OperationCallback e quindi l’utente non deve implementare niente (quindi i mock simulano l’eventuale classe utente e restituiscono un valore scelto da evosuite quando i metodi vengono chiamati). In sctunit le operation si possono utilizzare:

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

In ogni caso, sembra che evosuite non chiami mai i metodi dei mock direttamente ma li passi solamente come argomento a setOperationCallback(), quindi le operations non vengono “trattate”.

### Eventi temporali

Per indagare sugli elementi temporali si sono utilizzati gli statechart TemporalEvents.ysc e TemporalEvents2.ysc.

Se nello statechart c’è un evento temporale (che mi sembra siano solo after ed every), vengono generati automaticamente le interfacce ITimed e ITimerService e la classe VirtualTimer (che non so a cosa serva, non è specificata nella documentazione. Implementa ITimerService e viene anche usata da evosuite nei suoi test in setTimerService()). Inoltre, se si specifica nel .sgen:

feature GeneralFeatures {

TimerService = true

}

Viene generata anche la classe TimerService che implementa l’interfaccia ITimedService. L’idea è che l’utente deve implementare una classe che estenda tale interfaccia e fornirla all’implementazione della macchina con il metodo setTimerService() (arriva dall’interfaccia ITimed, dove è pubblico. La classe implementa tale interfaccia). TimerService viene fornito da Itemis Create ed è una implementazione di default che “in many cases it will be sufficient”. A differenza del metodo setOperationCallback, che se non pubblico rendeva la vita difficile ad evosuite, quando setTimerService viene reso privato c’è un errore di compilazione (non posso rendere la visibilità da pubblic, nell’interfaccia, a private nell’implementazione). Quindi è da lasciare pubblico come setOperationCallback.

Attenzione, in questa macchina ho creato una operation che si chiama setTime(). Il problema è che chiamandosi set, la rendo private pensando sia legata ad una variabile e il codice non compila (non posso avere un metodo privato in una interfaccia). Quindi se si è in una interfaccia, la visibilità dei metodi non viene modificata.

In questo caso, evosuite non sembra utilizzare i mock perché è disponibile VirtualTimer ed utilizza quello in setTimerService, mentre se l’implementazione TimerService è disponibile vengono usate tutte e due senza particolare logica, comunque funziona anche se TimerService non è disponibile; quindi, si può togliere dalla CLI l’opzione -t e rimuovere tutto ciò che ne consegue. Inizialmente, ho deciso di mantenere l’opzione ma cambiargli leggermente il nome e la descrizione,magari può tornare utile all’utente che se lo ritrova lì e può usare già anche l’implementazione java.

La copertura junit nel caso standard è molto inferiore di quella del caso Simplified ma non capisco il perché, sembra che utilizzi meno volte il metodo raiseTimeEvent. Il metodo raiseTimeEvent è quello utilizzato nell’implementazione java per inserire un evento temporale nell’array di boolean timeEvents; tale array ha un elemento per ogni utilizzo di un evento temporale (nel mio statechart uso due after e due every, quindi l’array ha quattro elementi). Esempio di come viene usato nel codice:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

E come viene usato in evosuite:

Immagine che contiene testo, Carattere, numero, schermata

Descrizione generata automaticamente

Quindi in java tutto ciò che è temporale viene trattato come un normale evento per cui bisogna fare raise. Ma, su sctunit si torna ad una gestione puramente temporale: la traduzione che dovrei fare io è qualcosa del tipo:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Da quello che ho capito, nell’implementazione java in ingresso a stati che utilizzano degli eventi temporali, si utilizza un’implementazione di ITimerService per fare il setup degli eventi temporali ed è tale implementazione a fare il “raise” di tali eventi dopo tot secondi (anche periodicamente).

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

Una strada per avere la corrispondenza tra *raiseTimeEvent(ID)* in junit e il *proceed t ms* da fare in sctunit potrebbe essere leggere questi timerService.setTime, il primo numero dovrebbe essere l’ID e il secondo il t in millisecondi. Nota che nel mio semplice esempio, utilizza praticamente solo il timeEvent con ID uguale a 0. Quindi in ogni caso la copertura non sarà il massimo anche se implemento questa strada. Secondo me sarebbe da mettere un proceed sia che ci sia true che false nell’ultimo parametro (indica se è periodico o no), ma sarebbe da testare su una macchina magari con soli eventi temporali. Potrei cercare ogni chiamata del metodo setTimer all’interno dell’intero file.

In altri modi non sembra fattibile passare da una situazione all’altra => il proceed 2 s non c’è e il metodo fallisce. Le possibilità rimanenti sono eliminare completamente il test per evitare che fallisca oppure rendere private raiseTimeEvent().

Di seguito viene descritto il tentativo di sfruttare le chiamate timerService.setTimer per ottenere un dizionario con ID come chiave con relativa durata in secondi come valore.

Innanzitutto, se gli eventi temporali sono dell’ordine dei microsecondi o nanosecondi, l’ambiente non sembra lavorare correttamente e i test potrebbero fallire:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Quindi i miei test potrebbero fallire, ma non per colpa mia. Infatti, sostituendo i ns con ms e i us con s, il test passa.

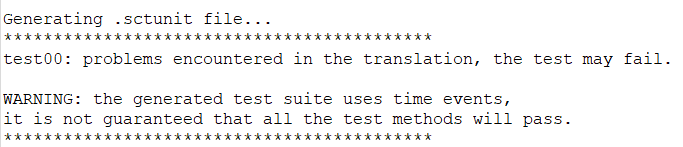
La soluzione funziona ma con qualche accortezza: se evosuite utilizza VirtualTimer o TimerService, genera test come:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Per far sì che l’unico elemento temporale utilizzato sia il raiseTimeEvent, bisogna far sì che non vengano utilizzate classi che implementano ITimerService. È stata quindi rimossa l’opzione che permette di generare TimerService; per quanto riguarda VirtualTimer, viene invece eliminato il file .class corrispondente prima di chiamare evosuite.

Se però l’environment dell’utente è diverso (**è già presente una classe che implementa ITimerService nel progetto**), è possibile che i test generati falliscano o che siano poco significativi. Questo, in aggiunta alla difficoltà di Itemis di utilizzare eventi temporali con ordini di grandezza piccoli, mi ha fatto stampare il seguente messaggio se nella mia test suite finale c’è almeno un proceed temporale:



La copertura risulta anche buona, ma gli assert sugli stati non sono molto interessanti: i metodi che alzano la copertura presentano solitamente assert isActive o cose simili.

Inoltre, ogni tanto quando apro il mio sctunit generato con eventi temporali, si apre una finestra di errore. Però il tutto viene eseguito correttamente. Non so come ottenere tale situazione.

### Trigger “complicati”

Ci sono probabilmente altre cose che vanno gestite. Di alcune mi sono accorte, altre magari mi sono sfuggite e mi sfuggiranno.

Per indagare sugli elementi temporali si è utilizzato lo statechart Triggers.ysc.

È possibile indicare più eventi (anche temporali) in un’unica transizione separandoli con una virgola. Si faccia riferimento a <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions>. Tutti quelli dopo la prima virgola non vengono trovati e quindi inseriti nel dizionario. Inoltre, se il primo evento è preceduto ad esempio dal tab, neanche lui viene trovato:

Immagine che contiene testo, schermata, schermo, diagramma

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Nota che, anche in una situazione limite come questa, l’evento temporale viene correttamente gestito (non viene ricavato dal .ysc ma dal .java). La nuova gestione prevede di sostituire tutte le sequenze di whitespace con un solo spazio e poi scorrere la stringa relativa a specification fintanto che ci sono virgole ottenendo il nome del trigger (è stato necessario gestire il caso di trigger “vuoto”, che può essere ottenuto anche inserendo solo whitespaces in itemis create); si potrebbe pensare di gestire le guardie e le azioni, ma per ora ci si limita ai trigger. Nota che tutte quelle sequenze di simboli strani in specification sono relative a whitespace e per catturarne sequenze con una regex si può fare semplicemente “\\s+” (sequenze di almeno un whitespace).

Nota che gli “special trigger” definiti da itemis:

* [after: execute after a given time](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions" \l "sclang_reaction_trigger_after)
* [every: execute periodically after a given time](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions" \l "sclang_reaction_trigger_every)
* [always: execute always](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions#sclang_reaction_trigger_always)
* [oncycle: same as always](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions#sclang_reaction_trigger_oncycle)
* [else: useful for choice states](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions#sclang_reaction_trigger_else)
* [default: same as else](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions#sclang_reaction_trigger_default)
* [entry: execute upon entering the state](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions" \l "sclang_reaction_trigger_entry)
* [exit: execute upon exiting the state](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_reactions" \l "sclang_reaction_trigger_exit)

Non sono di interesse, quindi è stato implementato codice per escluderli (eccetto entry ed exit, che non sono permessi nelle transizioni e quindi non sono da trattare).

### CycleBased e SuperSteps

Tutto il lavoro svolto è stato testato su macchine impostate EventDriven e senza super step. Quindi, è stato deciso di eseguire il codice su tutte le macchine utilizzate per la versione 0.0.2, ma impostate @CycleBased(200) e, dove possibile, @SuperSteps(yes). Dico dove possibile perché se si mettono i super step in macchine con delle transizioni che iniziano e finiscono sullo stesso stato (auto anelli), la macchina può finire in loop infiniti (dalla documentazione di itemis: You can think of the superstep semantic as a loop which performs a regular step until no new state is entered anymore.). In tal caso, la minimizzazione dei test di evosuite risulta molto lunga, l’intero processo richiede quasi 10 minuti e il test risultante non viene eseguito (si blocca senza dare errori).

Attenzione: nel caso di Operations.ysc e @CycleBased + @SuperStep(yes), la fase di minimizing di evosuite è molto lunga e alla fine i test sctunit si bloccano (come nel caso di loop infiniti):

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

Attenzione: alcuni miei test non passano e anzi non compilano, è il caso di @CycleBased + @SuperStep(yes) in TemporalEvents.ysc:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteImmagine che contiene testo, schermata, software, schermo

Descrizione generata automaticamente

Sembra verificarsi quando si usa @CycleBased, sia @SuperStep(yes) che @SuperStep(no) (ma non sono sicuro, e sembra più possibile con super step); quindi, dire che il mio codice non supporta l’esecuzione SuperStep, non è sufficiente. Può essere sensato per la ragione precedente. Rimane comunque il dubbio che possa accadere anche in altre situazioni. Visto che già viene stampato uno warning nel caso di macchine con eventi temporali, per ora si lascia la situazione così. Idea: una volta ottenuto l’oggetto TestCase, scorrerne le action e verificare se viene chiamato proceed dopo exit o prima di enter. Si è deciso di eliminare solo i casi di test che non compilerebbero, quindi con proceed a seguito di exit. Ora tutti i test dovrebbero compilare in tutti i casi, ma alcuni (quelli con warning) potrebbero non passare. Non è un problema perché sono casi particolari ed in ogni caso il tool non deve essere perfetto, va bene anche se rimangono aperte problematiche, da riscontrare nella fase di raccolta dati sperimentali, e da sistemare in futuro (anche non da me).

Non è stato fatto nessun controllo sull’efficacia (ed efficienza) del tool su macchine in cui cambia solo il tipo di esecuzione in questa sezione. Ma risulta abbastanza evidente che il caso CycleBased() riduca la copertura dei test sctunit.

### FunctionInline

Ho provato in velocità a settare a true tutte le opzioni di inline per lo statechart TemporalEvents.ysc (event driven e no super step):

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Il tutto ha funzionato e la copertura data era nella media. Si potrebbe indagare di più la possibilità di usare inline così da ottenere test di maggiore qualità, ma non è assolutamente una priorità. Guarda il link <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/codegen_configuring_a_generator> altrimenti.

## Refactoring con Structure 101 e JDepend

Si vuole migliorare la struttura del codice senza modificarne le funzionalità. Per farlo si utilizza il tool Structure101 (sia Studio che Workspace). Il focus di Structure101 è la complessità del codice, che viene misurata verificando se ci sono elementi fat e/o tangled. Io volevo anche utilizzare il grafico Instability vs Abstractness, quindi ho deciso di utilizzare anche JDepend, molto meno potente.

Si procede illustrando la situazione iniziale (riproducibile utilizzando la cartella compressa test-generator-for-yakindu-v0.0.2.zip nel desktop), le problematiche e le relative azioni intraprese (giustificandole) e, di conseguenza la situazione, finale.

Sembra comunque che Structure101 sia un po' overkill per il mio progetto.

### Situazione iniziale

Di seguito vengono riportate le infografiche di **Structure101**.

**Dipendenze tra package**

Immagine che contiene linea, diagramma, schermata, pendio

Descrizione generata automaticamente

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

**Dipendenze tra classi**

Immagine che contiene testo, diagramma, schermata, Piano

Descrizione generata automaticamente

Immagine che contiene testo, schermata, numero, diagramma

Descrizione generata automaticamente

**Complessità**

Immagine che contiene testo, schermata, software, Sistema operativo

Descrizione generata automaticamente

Immagine che contiene testo, schermata, linea, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Nota: ci sono molti modi su Structure101 per mostrare le dipendenze, qui sono stati riportati quelli che sembrano più significativi. Questa breve carrellata serve anche come punto di riferimento per sapere come muoversi nel momento di scrittura della tesi.

La metrica utilizzata da Structure101 è XS, che indica la complessità del progetto ed è data da elementi tangled e/o fat. Il problema principale sarebbe la presenza dei cicli (tangles). Fortunatamente (o meglio, grazie al fatto che ho fin da subito pensato alla struttura del codice, che comunque non è complesso), non ci sono cicli. Per quanto riguarda elementi fat, ho solamente due metodi fat. La fatness viene così calcolata da Structure101:

**Fat** is the degree to which an item exceeds a size threshold and is applied at every level of the hierarchy.  Fat is measured as [Cyclomatic Complexity](http://en.wikipedia.org/wiki/Cyclomatic_complexity" \t "_blank) at the method level, and the analogous measure of the number of edges in the dependency graph is used at all other levels.

Di seguito vengono riportate le infografiche di **JDepend**.

Nota: c’è un problema con i nomi dei package, se contengono java non vengono considerati da JDepend; quindi, è necessario modificare il nome (per ora, ma non riguarda la fase di refactoring che seguirà), ho rimosso la ‘a’ finale in java.

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, linea, diagramma

Descrizione generata automaticamente

### Problematiche e azioni intraprese

**Metodi fat**: l’analisi condotta con l’ausilio di Structure101 mette in luce una sola problematica: la presenza di due metodi fat (ossia con alta cyclomatic complexity), ossia TestCaseCollector.visit (fatness 32 su 15 di treshold) e CLIManager.parse (fatness 19 su 15 di treshold). Un interessante articolo sulla CC e su come ridurla è il seguente: <https://linearb.io/blog/reduce-cyclomatic-complexity>.

Per CLIManager.parse si è semplicemente diminuita la dimensione della funzione aggiungendo due nuove funzioni private (sono anche stati rinominati alcuni elementi):

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Oltre alla maggiore modularità e facilità di comprensione del codice, questa soluzione non sembra risolvere il problema alla radice, l’esecuzione segue gli stessi path di prima, la difficoltà nel testare la classe è identica a prima (la CC può essere vista come una metrica che indica quanto è difficile testare un metodo, però se un metodo viene scomposto in tre di cui due privati e chiamati solo nel primo, la CC diminuisce ma la difficoltà nel coprire il codice è la medesima). Mi sembra però che vista la natura del problema, non si possa fare di meglio.

Anche per TestCaseCollector.visit si è semplicemente diminuita la dimensione della funzione aggiungendo delle nuove funzioni private:

Immagine che contiene testo, software, Software multimediale, schermata

Descrizione generata automaticamente

Rimangono valide le perplessità emerse prima: è diminuita la CC ma la difficoltà nel testare rimane invariata, migliorano magari leggibilità e manutenibilità del codice. Forse qui si poteva agire in altri modi (strategy pattern?).

Nota che in entrambi i metodi la misura della fat è 15, proprio sul limite.

**Abstractness**: si è modificata la struttura del codice mantenendone le funzionalità, in particolare si è data una struttura più modulare (forse? Non mi piaceva molto il package generators) aggiungendo e rinominando package e si sono aggiunte delle interfacce (una per package). I metodi della classe Generators sono stati spostati in tutte le classi che estendono le interfacce (eccetto per quella riguardante la CLI). Nota che le interfacce non sono state commentate (né con JavaDoc né con normali commenti). Le classi che implementano le interfacce erano probabilmente da rendere singleton, ma è difficile avere un singleton che implementa un’interfaccia (il metodo getInstance dovrebbe restituire l’interfaccia? Come si fa a renderlo abstract senza che dia problemi con l’interfaccia?). Ovviamente, non ci sono metodi fat. Il refactoring non ha introdotto classi o package fat e non ci sono cicli (no elementi tangled). Dal punto di vista Abstractness vs Instability la situazione è migliorata di molto ma non è perfetta (i pallini sono segnati in verde da JDepend ma non sono tutti tra le due fasce) e viene riportata nel paragrafo successivo.

Nota: per facilitare il testing si possono rendere metodi private visibili con l’annotazione @VisibleForTesting, ma non so se è una buona idea.

### Situazioni finale

Si procede con le stesse analisi fatte prima di eseguire il refactoring per vedere se ci sono state migliorie.

Di seguito vengono riportate le infografiche di **Structure101**.

**Dipendenze tra package e Dipendenze tra classi**

Quelle viste sopra non sono facilmente riportabili qui (o troppo grandi o non riesco io a ottenere il taglio giusto), quindi ne ho messe di più semplici (senza il numero di dipendenze).

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Immagine che contiene linea, appendiabiti

Descrizione generata automaticamente

Guarda direttamente Structure101

**Complessità**

Immagine che contiene testo, schermata, software, Sistema operativo

Descrizione generata automaticamente

Immagine che contiene testo, schermata, software, Icona del computer

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Di seguito vengono riportate le infografiche di **JDepend**.

Nota: il problema con i nomi dei package non c’è più anche se alcuni hanno java nel nome

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, linea, diagramma

Descrizione generata automaticamente

Il pallino rosso non è un mio package. Nota che tre puntini verdi sono sovrapposti.

Conclusione: si è migliorata la struttura del codice mantenendone inalterate le funzionalità.

## Unit testing

È emersa, tra le altre cose, un problema: la gestione delle eccezioni, ad esempio nel caso di YscReader, ci sono questi tre test:

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

Che passano correttamente, ma prima che mettessi (expected = Exception.class) non passavano, vedendo che eccezioni lanciavano, l’ho aggiunto. Inoltre, ho messo throws NullPointerException al costruttore di YscReader ma non è necessario metterlo qui nei test. Perché? Stessa cosa per il test di SgenWriter, anche lì c’è NullPointerException ma non ho modificato la classe.

Non riesco a testare JavaWriter.callICGenerator perché ha bisogno dell’intero itemis create. Invece, per testare JavaWriter.writeSimplifiedVersion nel caso corretto ci sono due possibilità: utilizzare javaparser e praticamente ricopiare il contenuto dei vari metodi visit nelle classi visitor sostituendo le modifiche con degli assert o fare come nel caso di SgenWriter, ossia trasformare l’intero file in una stringa e vare delle verifiche su di essa. Il primo metodo è sicuramente più lungo e difficile da implementare, nel secondo è forse più chiaro cosa si vorrebbe avere. Ho scelto di utilizzare javaparser.

Non riesco a testare JunitWriter.callEvosuite perché dovrei avere a disposizione un progetto java con le classi che mi servono e compilate. Ho provato a mettere il tutto in “src/test/resources” ma attualmente senza successo. Sono poi riuscito, ho fatto lo statechart e generato il codice necessario, poi ho creato un NUOVO java project (che poi ho cancellato) e ho inserito ciò che mi serviva, ho copiato questo java project in src/test/resources.

Quindi, in generale, si potrebbe pensare di gestire meglio queste eccezioni, documentandosi un po’ su come farlo.

TODO: rileggi questa parte. Sistema la repo github in generale, cambia il nome ai progetti (tipo Temp) e cose così. Pensa a come testare la classe rimanente (quella con il main), sempre per quella classe, vedi se si possono togliere le linee di codice commentate (stesso per TestCaseCollector); secondo me si possono togliere, basta poi magari documentare qui cosa è stato tolto e dove. Crea il nuovo .jar e verifica che funzioni.

## Siti utilizzati finora

* Documentazione Itemis Create: <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_state_machines>
* Documentazione Evosuite: <https://www.evosuite.org/documentation/>
* Viatra: <https://dodin.ca/files/dp-report.pdf>
* Gamma: <https://inf.mit.bme.hu/sites/default/files/publications/icse18.pdf>
* Y2U: <http://www.cs.iit.edu/~code/software/Y2U/>
* UPPAAL: <https://uppaal.org/>
* StingTemplate: <https://github.com/antlr/stringtemplate4/blob/master/doc/index.md>
* TXL: <https://www.txl.ca/> (ci sono anche dei paper)
* JavaParser: <https://javaparser.org/getting-started.html> e il libro
* Eventuali paper su evosuite e itemis create/yakindu e altri (per darsi un tono con i paper) o meglio in generale sull’utilità deigli statechart e della generazione automatica di test? Ad esempio puoi guardare <https://www.evosuite.org/publications/>.
* EvoMBT: <https://github.com/iv4xr-project/iv4xr-mbt/wiki>, <https://ieeexplore.ieee.org/document/9810734>, <https://zenodo.org/records/7225016>, (sono lo stesso) <https://www.sciencedirect.com/science/article/abs/pii/S0167642323000242?casa_token=anj2D89-LEkAAAAA:XHerl4ayrIO3chDOPvMa86kocmg0r8REb6oq9IdZDwQaVUirw2G-HDL4BDfoGmh1u-lBB0hq>
* SCXML: <https://www.w3.org/TR/scxml/>;
* SCXML-commons: <https://commons.apache.org/proper/commons-scxml/>, <https://commons.apache.org/proper/commons-scxml/javadocs/0.9/overview-summary.html>;
* Apache CLI commons: <https://commons.apache.org/proper/commons-cli/index.html>;
* Structure101: <https://structure101.com/resources/#resources=1>;
* Guarda in generale tutte le librerie utilizzate.

# TODO

* Mancano ancora lo studio dei file generati partendo da tanti comportamenti modellabili con itemis create, ad esempio le history, le regioni ortogonali, gli out event e le operations, gli eventi temporali ecc. Per info sulle operations guarda <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/codegen_java_code_generator#codegen_java_operation_callback>. Si è deciso prima di iniziare a scrivere il tool per tradurre in sct i casi più semplici e comuni.
* **NOTA IMPORTANTE**: al link <https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sclang_definition_section#sclang_cyclebased> viene detto:

The @CycleBased annotation specifies that the [cycle-based execution scheme](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/codegen_general_concepts_of_the_state_machine_code#codegen_execution_schemes)is to be used.

Synopsis: @CycleBased( period)

The mandatory parameter period indicates the suggested period of time between two successive run-to-completion steps in milliseconds. Only the [statechart simulator](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/simu_simulating_statecharts#simu_simulating_statecharts) and the [SCTUnit testing framework](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sctunit_the_sctunit_language#sctunit_the_sctunit_language) take the period value into account, however. It is neither of significance to nor reflected in the generated code, and thus it remains the client code’s responsibility to explicitly call runCycle() – and to decide when to do so.

Quindi c’è già una possibile discrepanza tra quello che si vorrebbe ottenere e quello che effettivamente si può ottenere.

* Si potrebbe commentare, nella tesi, la struttura del codice java generato e conseguentemente dei casi di test generati da evosuite.
* Usare JaCoCo come maven plugin per la coverage dei test junit o guardare quella che dice evosuite quando genera i test. JaCoCo non funziona, così come codecover eclemma direttamente da eclipse, secondo me non supporta qualcosa utilizzato da evosuite nei test generati. Si può utilizzare il file di report generato da evosuite.
* È stato modificato il path della vm in headless.ini in C:\Program Files (x86)\itemis\_CREATE, sarebbe bello capire quale è il problema e come sistemarlo. Stesso discorso per il file scc.bat utilizzato (non modificato ma sulla macchina). Una soluzione potrebbe essere usare i Docker per creare un ambiente con tutto e solo quel che serve, però poi come fa il programma ad accedere al progetto dell’utente per cui deve generare il file? Un ulteriore problema: il -projectCP punta a \bin, ma se il progetto fosse maven (e magari altro?) dovrebbe puntare a \target\classes. Possibile soluzione, bisogna passare come argomento solo il file .ysc, poi è il mio tool a creare un nuovo progetto, gestire il tutto e poi restituire il file .sctunit.
* Commentare il codice scritto (anche i template), magari con JavaDoc. Quando commenti, riguarda i nomi dati (ad esempio, perché il parametro si chiama md nelle classi che estendono VoidVisitorAdapter?) e magari pure le scelte fatte, cerca di rendere il codice auto-esplicativo e con le stesse “convenzioni” nei nomi dati. Per quanto riguarda le scelte fatte intendo tipo:

List<String> classNameList = **new** ArrayList<String>();

VoidVisitor<List<String>> classCollector = **new** ClassDeclarationCollector();

classCollector.visit(cu, classNameList);

String statechartName = classNameList.get(0).replace("\_ESTest", "");

Ottiene il nome delle classi di test nel file parsato da javaparser sapendo che c’è solo una classe, evosuite quando genera le classi di test usa come nome il nome della classe da testare concatenato a “\_ESTest” e che le classi che Itemis Create genera hanno lo stesso nome dello statechart di partenza. Magari non è il modo migliore, anche perché se si vuole utilizzare versioni più aggiornate di itemis o evosuite, magari non si comportano più così. Si potrebbe prendere il nome dal file dello statechart stesso. Altrimenti, documenta bene queste scelte e perché funzionano.

Anche il modo in cui si ottengono i test case (testCase) nella classe TestCollector non mi entusiasma molto.

* Spostare la generazione dei file sctunit con stringtemplate in un metodo in una nuova classe (chiamata ad esempio TestSuite, contenente una lista di TestCase, come nel main di UseExample).
* **Capire il visitor pattern utilizzato in javaparser e come funziona in generale, perché sembra che i metodi visit vengano chiamati più volte?**
* Gestire le eccezioni, le lancio o le catturo e mando un messaggio? (Si parla di quelle tipo FileNotFoundException).
* Bisogna pensare a tutte le possibili semplificazioni fatte, ad esempio inizialmente ipotizzo che evosuite non utilizzi mai più di uno stato all’interno dello stesso metodo. È vero? Come faccio a esserne sicuro? E se non fosse vero? Tale ipotesi possono riflettersi sull’input accettato dall’utente, ad esempio non usare spazio o underscore nei nomi di regioni ed eventi.
* **Dovrei studiare meglio il linguaggio SCTUnit e le possibili configurazioni del file .sgen. Si potrebbe anche guardare un po’ la documentazione di evosuite e le pubblicazioni, capire in che senso è basato sulle mutation ecc. Al link** [**https://stackoverflow.com/questions/51951069/how-can-failing-test-cases-be-generated-via-evosuite**](https://stackoverflow.com/questions/51951069/how-can-failing-test-cases-be-generated-via-evosuite) **un utente spiega bene l’utilità di Evosuite.**
* Non ho pensato alla gestione dei possibili nomi dello statechart stesso (modificabile dall’interfaccia grafica di itemis). Bisognerebbe fare considerazioni simili a quelle fatte nella sezione “Ipotesi sullo statechart in ingresso”.
* Gestire non determinismo evosuite. Anche con specificando un seed l’output di evosuite non è sempre identico (anche se in qualche modo sembra più simile nel numero di test e nella copertura ottenuta). Magari provare con la versione 1.2.0, che utilizza un algoritmo diverso. Due esecuzioni diverse portano a risultati anche significativamente diversi:

Immagine che contiene testo, schermata, diagramma, software

Descrizione generata automaticamente

Immagine che contiene testo, schermata, software, diagramma

Descrizione generata automaticamente

* **Si potrebbe confrontare il mio tool con altri, tipo Viatra/Gamma e soprattutto EvoMBT o simili. Per EvoMBT, ad esempio, confronti sulla difficoltà di specificare una EFSM come classe Java (molto verboso) rispetto ad usare l’editor di Itemis Create, oppure la differenza tra EFSM e quelle possibili in Itemis (Harel FSM e altro). Forse anche un esempio di utilizzo di entrambi sullo stesso caso d’uso per confrontare i risultati. Per Viatra/Gamma l’obiettivo è diverso, magari pensare ad un utilizzo integrato di Viatra/Gamma con il mio tool.**
* **Gestire documentazione (i.e. diagrammi UML) e testing del mio software. Uno dei paper di EvoMBT nella sezione “Software description” può dare una mano.**
* Un po’ di refactoring e miglior ottenimento delle stringhe in input (come se fosse un jar serio con -m e cose del genere.
* Migliorare il file in uscita eliminando test inutili, ad esempio quelli vuoti.
* **Testare il jar anche su mac e linux con una virtual machine.**
* Gestire multi state machines (<https://www.itemis.com/en/products/itemis-create/documentation/user-guide/multi_state_machine_testing#multi_state_machine_testing>) se possibile.
* Provare a usare inline true nel code generator, dovrebbe rendere meno leggibile ma più efficiente il codice, come si comportano evosuite ed il mio software?
* Riguardare i commenti, forse sono un po’ troppo esagerati? Ad esempio per un semplice replace:

// Integers are subsituted with long in java, but the final L must be dropped to

// be compliant with sctunit,

// also the external parentesis are dropped to increase readability

value = value.replaceAll("[L()]", "");

* **PER RIPRESA DOPO SESSIONE:**
  + In una mail viene detto: “Intanto procedi con (1) trasformazione per aiutare/focalizzare evosuite e (2) lettura dei modelli per ricostruire l'sct corretto (ad esempio con i nomi maiuscoli/minuscoli ...)”. Il punto (1) è stato fatto, bisogna fare il punto (2), partendo dalle librerie di itemis create: 
  + Procedere con la traduzione di macchine più complesse e sistemare il problema indicato alla fine della sezione Considerazioni sull’utilizzo di Evosuite.
  + Documenta con degli esempi il modo in cui vengono generati i testCase, così che si capiscano le funzioni del codice e cosa contengono le variabili nei vari momenti.
  + Magari riguarda tutto questo file e la nota su smartphone per avere un’idea di cosa è stato fatto e cosa c’è ancora da fare.