# 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 di 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 instalalto);
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<Asction> 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, paralndo 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 dello statechart:

(NOTA: se slo 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

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. I

l secondo escludendo il promi 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 guarda uno zip in download 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.

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 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.

## Considerazioni sull’utilizzo di Evosuite

I problemi principali di Evosuite sono due:

1. Molto lento, 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. 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 ad entrambi i 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’iimplementazione 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 (anche se l’esecuzione è CycleBased o ci sono regioni ortogonali bisognerebbe verificare che chiamate di metodi nel metodo di test portano a dover eliminare il 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 nemmeno l’efficacia, visto che la copertura dallo statechart a volte è migliore per il caso standard, a volte per il simplified.

Queste immagini riguardano il caso “standard”, che fallisce a causa del test35, 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).

Ora, per verificare efficienza ed efficacia dei due casi, si reintroducono le linee di codice per solo il caso standard (così che non fallisca nessun test) 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:

**TODO**

**NOTA:** tutti gli esempi riportati sopra sono anche frutto del non determinismo di Evosuite, quindi le conclusioni tratte non valgono in generale per le singole esecuzioni.

**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.

## 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, in download del PC) <https://www.sciencedirect.com/science/article/abs/pii/S0167642323000242?casa_token=anj2D89-LEkAAAAA:XHerl4ayrIO3chDOPvMa86kocmg0r8REb6oq9IdZDwQaVUirw2G-HDL4BDfoGmh1u-lBB0hq> (in download del PC)

**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.
* 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 parsata 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> 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 ottenura). 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.