# 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-junit-tester ws

* **JUnitTestGenerator**:

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.

* **JUnit2SCTUnit**:

maven project inizialmente scritto per pensare a come gestire la scrittura dinamica dei file .sctunit attraverso string template. Alla fine, il tentativo è riuscito e gli artefatti importanti sono:

* + **sct\_template.st**: stringtemplate contenente, per ora, la base dei test sctunit (per ora 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 raise, assert, proceed ed exit). Le singole istanze avranno valori diversi da null solo per le stringhe necessarie alla singola azione che descrivono (ad esempio l’azione è del tipo assert active state, le stringhe non a null sono solo state e not). Contiene solo il costruttore e i 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.

## 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?

**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.
* 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 patter 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.