# 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.

### 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.

### 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 utilizzo

Un esempio di utilizzo dei tool è il seguente:

1. Creare un progetto Maven sull’app standalone Itemis CREATE;
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.

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**: java project creato per studiare sperimentalmente la struttura dei test JUnit ottenuti con la tool chain finora implementata. Visto che i test non si vogliono eseguire, ma solo visionare, è sufficiente un progetto java. 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.

**TODO**

* Mancano ancora 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>.
* **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" \l "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" \l "simu_simulating_statecharts) and the [SCTUnit testing framework](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/sctunit_the_sctunit_language" \l "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.