

ANALISI DI DIAGNOSTICABILITA

ELABORATO DI ALGORITMI E STRUTTURE DATI





Docenti:

**PROF.SSA MARINA ZANELLA**

**ING. ALESSANDRO SAETTI**

Studenti:

**MATTEO MARIO 86015**

**MICHELE MASCIALE 85982**

**Università degli Studi di Brescia**

**Dipartimento di Ingegneria dell’Informazione**

**Corso di Laurea Magistrale in Ingegneria Informatica**

**Anno accademico 2014 - 2015**

# Indice

[Indice I](#_Toc458515512)

[Capitolo 1 - Introduzione 1](#_Toc458515513)

[1.1 Descrizione del problema 1](#_Toc458515514)

[1.2 Strumenti utilizzati 2](#_Toc458515515)

[Capitolo 2 - Definizione delle strutture dati 3](#_Toc458515516)

[2.1 Rappresentazione dell’input 3](#_Toc458515517)

[2.2 Classi Python utilizzate per rappresentare l’automa 5](#_Toc458515518)

[2.3 Chiarimenti riguardanti le scelte implementative 6](#_Toc458515519)

[Capitolo 3 – Metodo risolvente 7](#_Toc458515520)

[3.1 Introduzione 7](#_Toc458515521)

[3.2 Versioni successive del metodo risolvente 7](#_Toc458515522)

[3.2.1 Tempo di verifica delle tre condizioni 8](#_Toc458515523)

[3.3 Implementazione 10](#_Toc458515524)

[3.3.1 Prima e seconda versione del metodo risolvente 10](#_Toc458515525)

[3.3.2 Terza versione del metodo risolvente 11](#_Toc458515526)

[Capitolo 4 – Casi di test 14](#_Toc458515527)

[4.1 Introduzione 14](#_Toc458515528)

[4.2 Generazione automatica dei casi di test 14](#_Toc458515529)

[4.2.1 Premessa 14](#_Toc458515530)

[4.2.2 Parametri in ingresso al generatore automatico di automi 15](#_Toc458515531)

[4.2.3 Logica alla base del generatore di automi 16](#_Toc458515532)

[4.2.4 Il modulo random\_automaton 17](#_Toc458515533)

[Capitolo 5 – Complessità e prestazioni 23](#_Toc458515534)

[5.1 Dimensioni del problema 23](#_Toc458515535)

# 

# Capitolo 1 - Introduzione

## 1.1 Descrizione del problema

L’obiettivo del presente elaborato è quello di realizzare un software che risolva il problema di analizzare il livello di diagnosticabilità di un automa non deterministico finito. In particolare le specifiche del problema richiedono i seguenti input:

* un *automa non deterministico finito*  dove è l’insieme (non vuoto) degli stati (fra cui uno stato iniziale ) e è l’insieme (non vuoto) delle transizioni di stato;
* il sottoinsieme (non vuoto) delle *transizioni osservabili* di , , ciascuna contraddistinta da un *evento osservabile semplice*  (più transizioni uscenti dal medesimo stato possono essere contraddistinte dallo stesso evento osservabile);
* l’alfabeto degli eventi osservabili semplici di , tale che il linguaggio generato da su questo alfabeto sia *vivo*, il che comporta che sia ciclico e che ogni cammino ciclico in contenga almeno una transizione osservabile;
* il sottoinsieme (non vuoto) delle *transizioni di guasto* di , , ciascuna delle quali contraddistinta dal medesimo (tipo di) *evento di guasto f* (più transizioni uscenti dal medesimo stato possono essere contraddistinte dall’evento di guasto);
* il livello (intero) di diagnosticabilità da verificare.

Come output il software deve fornire l’indicazione se sia vero o falso che (come descritto in input attraverso ,, , , ) goda del livello di diagnosticabilità e, nel caso ciò sia falso, il livello di diagnosticabilità massimo di cui gode (dove tale livello sarà ).

## 1.2 Strumenti utilizzati

Per lo sviluppo del software sono stati utilizzati i seguenti strumenti, raggruppati per tipologia:

* **Linguaggi**
  + *Python 2.7*: linguaggio di programmazione dinamico orientato agli oggetti;
  + *XML*: metalinguaggio per la definizione di linguaggi di markup;
  + *DOT*: linguaggio per la descrizione di grafi.
* **Ambiente di sviluppo**
  + *PyCharm*: ambiente di sviluppo integrato, multi-linguaggio e multipiattaforma.
* **Strumento di condivisione**
  + *GitHub*: servizio web di hosting per lo sviluppo di progetti software (e non solo) che usa il sistema di controllo di versione Git.
* **Librerie non standard per Python**
  + *networkx*: libreria per la creazione e manipolazione di grafi e reti complesse;
  + *lxml*: libreria per l’elaborazione di file *XML*;
  + *graphviz*: libreria per il rendering di grafi descritti nel linguaggio *DOT*;
  + *PIL*: libreria per la gestione delle immagini.

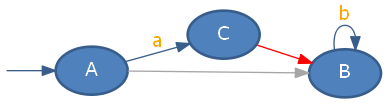
# Capitolo 2 - Definizione delle strutture dati

## 2.1 Rappresentazione dell’input

Per descrivere i dati in ingresso all’applicazione abbiamo utilizzato il linguaggio *XML*.

L’input principale dell’applicazione è un *automa non deterministico finito* , dove è l’insieme (non vuoto) degli stati (fra cui uno stato iniziale ) e è l’insieme (non vuoto) delle transizioni di stato. Mostriamo con l’ausilio di un esempio come un semplice automa possa essere descritto attraverso un file *XML*, secondo la sintassi da noi adottata.

In *Figura 2.1* è rappresentato l’automa preso come esempio.



*Figura 2.1: Un esempio di automa non deterministico finito. In blu sono rappresentate le transizioni osservabili, in grigio quelle non osservabili e in rosso quelle di guasto.*

L’automa rappresentato in *Figura 2.1* è descritto dai seguenti parametri:

Tale automa può essere rappresentato nel linguaggio *XML* secondo la seguente sintassi:

<?xml version=**"1.0"** encoding=**"utf-8"**?>

<automaton>

<states>

<state initial=**"true"**>**A**</state>

<state>**B**</state>

<state>**C**</state>

</states>

<transitions>

<transition>

<source>**A**</source>

<destination>**B**</destination>

</transition>

<transition observable=**"true"**>

<source>**A**</source>

<event>**a**</event>

<destination>**C**</destination>

</transition>

<transition fault=**"true"**>

<source>**C**</source>

<destination>**B**</destination>

</transition>

<transition observable=**"true"**>

<source>**B**</source>

<event>**b**</event>

<destination>**B**</destination>

</transition>

</transitions>

</automaton>

Lo stato iniziale è individuato dall’attributo initial, mentre l’attributo fault indica se una transizione è associata ad un evento di guasto. Le transizioni non osservabili (non di guasto) sono identificabili dall’assenza di attributi, quelle osservabili invece sono caratterizzate dall’attributo observable e l’evento ad esse associato è rappresentato dal tag <event>. Infine i tag <source> e <destination> rappresentano rispettivamente lo stato sorgente e lo stato destinazione di una transizione.

Per trasformare l’automa descritto nel file *XML* in un oggetto Python abbiamo utilizzato la nostra funzione load\_xml che, grazie ad un parser *XML*, esamina la struttura ad albero del file e costruisce progressivamente un oggetto istanza della classe **Automaton**. In questa fase vengono gestiti eventuali errori sintattici, grazie all’utilizzo di un *XML* *Schema*, e semantici, garantendo la correttezza dell’input.

## 2.2 Classi Python utilizzate per rappresentare l’automa

Utilizziamo la classe **Automaton** per rappresentare ogni tipo di automa (bad twin, good twin o sincronizzato) nella sua interezza:

**class** **Automaton:**

**def** \_\_init\_\_**(**self**,** initial\_state**,** states**):**

self**.**initial\_state **=** initial\_state

self**.**states **=** states

Come si può notare dall’estratto di codice la classe **Automaton** è caratterizzata dagli attributi initial\_state e states, che rappresentano rispettivamente il nome dello stato iniziale e gli stati dell’automa. In particolare l’attributo states è un dizionario in cui ogni chiave è un nome ed ogni valore il corrispettivo stato dell’automa, rappresentato dalla classe **State**.

La classe **State** rappresenta appunto uno stato dell’automa:

**class** **State:**

**def** \_\_init\_\_**(**self**,** name**):**

self**.**name **=** name

self**.**neighbours **=** dict**()**

self**.**visited **=** **False**

Come si può notare dall’estratto di codice la classe **State** è caratterizzata dagli attributi name, neighbours e visited, che rappresentano rispettivamente il nome, gli stati adiacenti e un flag che indica se lo stato è già stato visitato (utile in fase di ricerca in profondità). In particolare l’attributo neighbours è un dizionario in cui ogni chiave è uno stato ed ogni valore la lista di transizioni (istanze della classe **Transition**) che legano lo stato corrente a quello rappresentato dalla chiave.

La classe **Transition** rappresenta una transizione dell’automa:

**class** **Transition:**

**def** \_\_init\_\_**(**self**,** event**=None,** fault**=False,** ambiguous**=False):**

self**.**event **=** event

self**.**fault **=** fault

self**.**ambiguous **=** ambiguous

Come si può notare dall’estratto di codice la classe **Transition** è caratterizzata dagli attributi event, fault e ambiguous, che rappresentano rispettivamente l’evento, un flag che indica se la transizione è di guasto e un flag che indica se essa è ambigua. In particolare l’attributo event è **None** se la transizione non è osservabile, altrimenti è un oggetto istanza della classe **Event**.

Infine la classe **Event** rappresenta un evento osservabile, semplice o composto, associato ad una transizione osservabile dell’automa:

**class** **Event:**

**def** \_\_init\_\_**(**self**,** name**=None):**

**if** type**(**name**)** **is** str**:**

name **=** **[**name**]**

self**.**multiset **=** Counter**(**name**)**

Come si può notare dall’estratto di codice la classe **Event** è caratterizzata dall’attributo multiset che rappresenta un multinsieme di eventi osservabili semplici. Per definizione un multinsieme è un insieme (pertanto non conta l’ordine degli elementi in esso contenuti) in cui sono ammessi duplicati. Questa caratteristica è fondamentale per costruire i *twin* di livello *i* > 1, i quali possono contenere eventi osservabili composti di livello *i*. Un evento composto di livelli *i* è costituito da istanze, non tutte identiche, di eventi appartenenti a , ossia l’alfabeto degli eventi osservabili semplici. La cardinalità di un evento composto è pari alla cardinalità del multinsieme dei suoi componenti. La classe Counter, specializzazione del dizionario (dict) presente nel modulo collections, è adatta a rappresentare questo tipo di dato.

## 2.3 Chiarimenti riguardanti le scelte implementative

Risulta evidente dal paragrafo precedente come la struttura dati più utilizzata sia il dizionario (dict). Essa è stata infatti adoperata per rappresentare gli stati nella classe **Automaton**, il vicinato di uno stato e le relative transizioni nella classe **State** e, con una specializzazione, il multinsieme di eventi osservabili semplici nella classe **Event**. Tali scelte sono giustificate dai frequenti accessi, richiesti dal software, a tali strutture dati: ogni accesso ai valori richiede un tempo costante se si conosce la chiave.

# Capitolo 3 – Metodo risolvente

## 3.1 Introduzione

La prima versione del metodo risolvente (si vedano le specifiche dell’elaborato) consiste nel calcolare, ad ogni iterazione del ciclo while, il bad twin di livello *i*, il good twin di livello *i* e l’automa sincronizzato di livello *i*.

**PROPOSIZIONE (C\*)**: Se vale la condizione di diagnosticabilità di livello *i* – 1 e, nell’automa risultante dalla sincronizzazione dei twin di livello *i*, non esiste alcun cammino contenente sia (almeno) una transizione il cui evento osservabile sia (esattamente) di livello *i*, sia una transizione ambigua seguita (anche non immediatamente) da un ciclo[[1]](#footnote-1), allora vale la condizione di diagnosticabilità di livello *i*.

Se tale condizione è verificata, l’algoritmo passa all’iterazione successiva. Ogni iterazione del ciclo while comporta quindi la verifica di tale condizione, o meglio la verifica della negazione di essa: se nell’automa risultante dalla sincronizzazione di livello *i* esiste almeno un cammino dove la prima transizione sincronizzata ambigua, derivante da una transizione di guasto e l’altra no, è seguita (anche non immediatamente) da un ciclo (infinito), allora non sussiste la diagnosticabilità al livello *i* e il livello di diagnosticabilità massimo è *i* – 1. Si noti che dal punto di vista della complessità temporale, l’algoritmo che verifica questa condizione è di classe NP.

## 3.2 Versioni successive del metodo risolvente

La principale differenza tra la prima versione del metodo risolvente e le successive due è costituita dall’utilizzo di tre condizioni sufficienti di diagnosticabilità:

**CONDIZIONE 1 (C1)**: Se nell’automa risultante dalla sincronizzazione dei twin del livello considerato non esiste alcuna transizione ambigua (cioè ), allora a tale livello sussiste la diagnosticabilità.

**CONDIZIONE 2 (C2)**: Se il bad twin è deterministico relativamente all’insieme degli eventi osservabili fino al livello considerato, allora a tale livello sussiste la diagnosticabilità.

**CONDIZIONE 3 (C3)**: Se nel bad twin gli eventi osservabili fino al livello considerato associati alle transizioni di guasto non sono associati ad alcuna transizione che non sia di guasto, allora a tale livello sussiste la diagnosticabilità.

Mentre la prima versione del metodo risolvente non fa utilizzo di alcuna di queste condizioni, la seconda e la terza versione le utilizzano tutte e tre. In particolare le condizioni **C2** e **C3** riguardano solamente il bad twin: se verificate non sarà necessario costruire né il good twin né l’automa sincronizzato di livello *i*. Se le condizioni **C2** e **C3** non sono soddisfatte si passa alla costruzione del good twin e dell’automa sincronizzato di livello *i*, dopodiché si verifica la condizione **C1**: se nemmeno questa è soddisfatta si passa a verificare **C\***. Nel caso pessimo nessuna delle tre condizioni è verificata, pertanto sia la seconda che la terza versione del metodo risolvente ricadono nel caso della prima versione, con l’aggravante che al tempo di calcolo si aggiunge il tempo di verifica delle tre condizioni, stimato nel paragrafo seguente.

### 3.2.1 Tempo di verifica delle tre condizioni

Analizziamo ora dal punto di vista teorico la complessità degli algoritmi che verificano le tre condizioni **C1**, **C2** e **C3**.

**CONDIZIONE 1 (C1)**

Riportiamo di seguito l’algoritmo di verifica.

**def** condition\_C1**(**ambiguous\_transitions**):**

**return** len**(**ambiguous\_transitions**)** **==** 0

Come si può notare l’algoritmo verifica semplicemente che l’insieme ambiguous\_transitions delle transizioni ambigue abbia lunghezza nulla. Il calcolo della lunghezza di una qualsiasi struttura dati che rappresenti una sequenza di elementi richiede un tempo costante, pertanto tale algoritmo ha complessità .

**CONDIZIONE 2 (C2)**

Riportiamo di seguito gli algoritmi utilizzati per la verifica.

**def** condition\_C2**(**bad\_twin**):**

**return** **not** bad\_twin**.**is\_non\_deterministic**()**

**def** is\_non\_deterministic**(**self**):**

**for** state **in** self**.**states**.**values**():**

events **=** dict**()**

**for** transitions **in** state**.**get\_neighbours**().**values**():**

**for** transition **in** transitions**:**

**if** transition**.**is\_observable**():**

**try:**

**if** events**[**transition**.**get\_event\_name**()]:**

**return** **True**

**except** KeyError**:**

events**[**transition**.**get\_event\_name**()]** **=** **True**

**return** **False**

Come si può notare l’algoritmo di verifica calcola la negazione del valore di ritorno dell’algoritmo is\_non\_deterministic. Analizziamo dunque la sua complessità. I due cicli **for** più annidati passano in rassegna tutte le transizioni uscenti da uno stato, pertanto hanno complessità ), assumendo il numero medio di transizioni uscenti da uno stato (*branching factor*). Il ciclo **for** esterno passa in rassegna tutti gli stati, pertanto ha complessità . Dunque l’algoritmo presenta una complessità , ovvero lineare nel numero di transizioni. D’altronde per verificare se un automa sia deterministico nel caso peggiore è necessario passare in rassegna tutte le sue transizioni.

**CONDIZIONE 3 (C3)**

Riportiamo di seguito gli algoritmi utilizzati per la verifica.

**def** condition\_C3**(**bad\_twin**):**

**return** **not** bad\_twin**.**has\_ambiguous\_events**()**

**def** has\_ambiguous\_events**(**self**):**

events **=** dict**()**

**for** state **in** self**.**states**.**values**():**

**for** transitions **in** state**.**get\_neighbours**().**values**():**

**for** transition **in** transitions**:**

**try:**

**if** events**[**transition**.**get\_event\_name**()]** **!=** transition**.**is\_fault**():**

**return** **True**

**except** KeyError**:**

events**[**transition**.**get\_event\_name**()]** **=** transition**.**is\_fault**()**

**return** **False**

Come si può notare l’algoritmo di verifica calcola la negazione del valore di ritorno dell’algoritmo has\_ambiguous\_events. La struttura di quest’ultimo è simile a quella del metodo is\_non\_deterministic, la cui complessità è già stata analizzata. Pertanto anche questo algoritmo ha complessità .

## 3.3 Implementazione

### 3.3.1 Prima e seconda versione del metodo risolvente

La prima e la seconda versione del metodo risolvente rispecchiano lo pseudo-codice fornito nelle specifiche dell’elaborato. È di seguito riportato il codice dei metodi Python da noi implementati.

**PRIMA VERSIONE DEL METODO RISOLVENTE**

**def** method1**(**automaton**,** level**):**

old\_bad\_twin **=** automaton

i **=** 1

**while** i **<=** level**:**

new\_bad\_twin **=** generate\_bad\_twin**(**old\_bad\_twin**,** i**)**

good\_twin **=** generate\_good\_twin**(**new\_bad\_twin**)**

synchronized**,** ambiguous\_transitions **=** synchronize\_1**(**new\_bad\_twin**,** good\_twin**)**

**for** src\_name**,** dst\_name **in** ambiguous\_transitions**:**

states **=** synchronized**.**get\_states**()**

**if** find\_loops**(**states**[**dst\_name**],** **{**src\_name**}):**

**return** i **-** 1

old\_bad\_twin **=** new\_bad\_twin

i **+=** 1

**return** **True**

**SECONDA VERSIONE DEL METODO RISOLVENTE**

**def** method2**(**automaton**,** level**):**

old\_bad\_twin **=** automaton

i **=** 1

**while** i **<=** level**:**

new\_bad\_twin **=** generate\_bad\_twin**(**old\_bad\_twin**,** i**)**

c2 **=** condition\_C2**(**new\_bad\_twin**)**

c3 **=** condition\_C3**(**new\_bad\_twin**)**

**if** **not(**c2 **or** c3**):**

good\_twin **=** generate\_good\_twin**(**new\_bad\_twin**)**

synchronized**,** ambiguous\_transitions **=** synchronize\_1**(**new\_bad\_twin**,** good\_twin**)**

c1 **=** condition\_C1**(**ambiguous\_transitions**)**

**if** **not** c1**:**

**for** src\_name**,** dst\_name **in** ambiguous\_transitions**:**

states **=** synchronized**.**get\_states**()**

**if** find\_loops**(**states**[**dst\_name**],** **{**src\_name**}):**

**return** i **-** 1

old\_bad\_twin **=** new\_bad\_twin

i **+=** 1

**return** **True**

Il metodo find\_loops, utilizzato in tutte le versioni del metodo risolvente, restituisce **True** se la condizione di diagnosticabilità **C\*** non è rispettata, cioè se nell’automa risultante dalla sincronizzazione esiste almeno un cammino in cui la prima transizione ambigua è seguita (anche non immediatamente) da un ciclo (infinito).

### 3.3.2 Terza versione del metodo risolvente

Dalle specifiche dell’elaborato notiamo che la terza versione del metodo risolvente utilizza una seconda versione dell’algoritmo di sincronizzazione. A differenza della prima versione, la seconda non riceve in ingresso il good twin e il bad twin di livello *i*, bensì l’automa sincronizzato di livello *i* – 1 e il bad twin di livello *i*.

Sono sorte spontaneamente alcune considerazioni non trascurabili ai fini della programmazione:

1. Ogni iterazione del ciclo **while** prevede, in primis, la generazione del bad twin di livello *i*: se il bad twin di livello *i* soddisfa una condizione sufficiente (**C2** o **C3**) di diagnosticabilità, si passa all’iterazione successiva del ciclo **while** senza generare l’automa sincronizzato.
2. Ad ogni ciclo può essere necessario disporre dell’automa sincronizzato di livello *i* – 1.
3. Può succedere che all’iterazione *i*-esima del ciclo **while** non si disponga dell’automa sincronizzato di livello *i* – 1 poiché non è stato generato durante l’iterazione precedente.

A seguito di queste considerazioni abbiamo implementato la terza versione del metodo risolvente in due modi diversi. Il primo metodo, method3\_1, si basa sulla seguente logica:

* Se le condizioni (**C2** o **C3**) di diagnosticabilità sono soddisfatte, si passa al ciclo successivo senza generare l’automa sincronizzato;
* Se le condizioni (**C2** o **C3**) di diagnosticabilità non sono soddisfatte, allora è necessario disporre dell’automa sincronizzato di livello *i* – 1: se esso non è disponibile viene generato a partire dal bad twin e dal good twin di livello *i* - 1*,* utilizzando il primo algoritmo di sincronizzazione.

Il secondo metodo, method3\_2, genera, indipendentemente dal soddisfacimento delle condizioni (**C2** o **C3**) di diagnosticabilità, l’automa sincronizzato di livello *i*, in modo che sia sempre disponibile per il ciclo successivo.

È di seguito riportato il codice dei due metodi che implementano la terza versione del metodo risolvente.

**TERZA VERSIONE DEL METODO RISOLVENTE (VERSIONE 1)**

**def** method3\_1**(**automaton**,** level**):**

old\_bad\_twin **=** automaton

i **=** 1

first\_sync **=** **True**

last\_sync\_level **=** 1

**while** i **<=** level**:**

new\_bad\_twin **=** generate\_bad\_twin**(**old\_bad\_twin**,** i**)**

c2 **=** condition\_C2**(**new\_bad\_twin**)**

c3 **=** condition\_C3**(**new\_bad\_twin**)**

**if** **not(**c2 **or** c3**):**

**if** first\_sync**:**

good\_twin **=** generate\_good\_twin**(**new\_bad\_twin**)**

synchronized**,** ambiguous\_transitions **=** synchronize\_1**(**new\_bad\_twin**,** good\_twin**)**

first\_sync **=** **False**

**else:**

**if** last\_sync\_level **<** i **-** 1**:**

old\_good\_twin **=** generate\_good\_twin**(**old\_bad\_twin**)**

synchronized**,** ambiguous\_transitions **=** synchronize\_1**(**old\_bad\_twin**,** old\_good\_twin**)**

synchronized **=** synchronize\_2**(**synchronized**,** ambiguous\_transitions**,** new\_bad\_twin**,** i**)**

last\_sync\_level **=** i

c1 **=** condition\_C1**(**ambiguous\_transitions**)**

**if** **not** c1**:**

**for** src\_name**,** dst\_name **in** ambiguous\_transitions**:**

states **=** synchronized**.**get\_states**()**

**if** find\_loops**(**states**[**dst\_name**],** **{**src\_name**}):**

**return** i **-** 1

old\_bad\_twin **=** new\_bad\_twin

i **+=** 1

**return** **True**

**TERZA VERSIONE DEL METODO RISOLVENTE (VERSIONE 2)**

**def** method3\_2**(**automaton**,** level**):**

old\_bad\_twin **=** automaton

i **=** 1

**while** i **<=** level**:**

new\_bad\_twin **=** generate\_bad\_twin**(**old\_bad\_twin**,** i**)**

**if** i **==** 1**:**

good\_twin **=** generate\_good\_twin**(**new\_bad\_twin**)**

synchronized**,** ambiguous\_transitions **=** synchronize\_1**(**new\_bad\_twin**,** good\_twin**)**

**else:**

synchronized **=** synchronize\_2**(**synchronized**,** ambiguous\_transitions**,** new\_bad\_twin**,** i**)**

c2 **=** condition\_C2**(**new\_bad\_twin**)**

c3 **=** condition\_C3**(**new\_bad\_twin**)**

**if** **not(**c2 **or** c3**):**

c1 **=** condition\_C1**(**ambiguous\_transitions**)**

**if** **not** c1**:**

**for** src\_name**,** dst\_name **in** ambiguous\_transitions**:**

states **=** synchronized**.**get\_states**()**

**if** find\_loops**(**states**[**dst\_name**],** **{**src\_name**}):**

**return** i **-** 1

old\_bad\_twin **=** new\_bad\_twin

i **+=** 1

**return** **True**

# Capitolo 4 – Casi di test

## 4.1 Introduzione

Non essendo forniti oracoli, il primo fine della sperimentazione è quello di confrontare i risultati ottenuti dalle tre versioni del metodo risolvente: ogni differenza denuncia la presenza di difetti o nelle specifiche o nell’implementazione. Il secondo fine della sperimentazione è quello di comparare le prestazioni delle tre versioni del metodo risolvente (argomento affrontato nel Capitolo 5). Poiché non sono forniti dati per la sperimentazione ed è necessario, ai fini di cui sopra, disporre di un insieme significativo di casi di test, abbiamo costruito un generatore automatico di automi non deterministici finiti.

## 4.2 Generazione automatica dei casi di test

### 4.2.1 Premessa

È bene ricordare che ogni automa generato dovrà essere dotato, dato l’input del problema descritto a pagina *1*, di alcune caratteristiche. In particolare è richiesto che:

1. gli insiemi , , e siano non vuoti;
2. il linguaggio generato da sull’alfabeto degli eventi osservabili semplici sia vivo, il che comporta che sia ciclico e che ogni cammino ciclico in contenga almeno una transizione osservabile;
3. la richiesta che il linguaggio generato da su sia vivo implica che ogni stato di sia dotato di almeno una transizione uscente e che, seguendo una qualsiasi transizione uscente da un qualsivoglia stato, dopo un numero finito (magari nullo) di transizioni, si incontri una transizione osservabile.

### 4.2.2 Parametri in ingresso al generatore automatico di automi

Prima di procedere alla costruzione del generatore automatico di automi, abbiamo individuato alcuni parametri di ingresso il cui valore può essere modificato al fine di generare un insieme significativo ed eterogeneo di casi di test:

* ns, il numero di stati dell’automa;
* nt, il numero di transizioni dell’automa;
* no, il numero di transizioni osservabili dell’automa;
* ne, la cardinalità dell’insieme degli eventi osservabili semplici ;
* nf, il numero di transizioni di guasto.

Riteniamo importante sottolineare l’introduzione di altri parametri al fine di comparare sperimentalmente le prestazioni delle tre versioni dell’algoritmo risolvente (argomento affrontato nel Capitolo 5):

* bf, il numero medio di transizioni uscenti da ogni stato (*branching factor*);
* po, la percentuale di transizioni osservabili;
* pe, il rapporto tra la cardinalità dell’insieme degli eventi ed il numero di transizioni osservabili;
* pf, la percentuale di transizioni di guasto.

L’aggiunta di questi parametri introduce alcune relazioni fondamentali ai fini della correttezza della sperimentazione. Supponendo di conoscere ns e i quattro parametri appena introdotti, è possibile ricavare gli altri quattro come segue:

* ;
* ;
* ;
* .

### 4.2.3 Logica alla base del generatore di automi

La logica alla base del generatore di automi è la seguente:

1. Inizialmente si verifica se i parametri di ingresso soddisfano i seguenti vincoli:
   1. nt ns, ovvero non possono esserci più stati che transizioni;
   2. no , ovvero deve esserci almeno una transizione osservabile;
   3. ne no, ovvero non possono esserci più eventi che transizioni osservabili;
   4. nf , ovvero deve esserci almeno una transizione di guasto;
   5. nt nonf, ovvero l’unione delle transizioni osservabili e di quelle di guasto deve essere contenuta nell’insieme delle transizioni .
2. Viene creata la lista dei nomi degli stati, lunga ns:
   1. se ns, il nome di uno stato corrisponde ad una lettera maiuscola dell’alfabeto latino;
   2. se ns, il nome di uno stato corrisponde alla lettera *S*, seguita da un numero incrementale (a partire da 1), diverso per ogni stato.
3. Viene creata la lista dei nomi degli eventi osservabili semplici, lunga no:
   1. se no, il nome di un evento corrisponde ad una lettera minuscola dell’alfabeto latino;
   2. se no, il nome di un evento corrisponde alla lettera *e*, seguita da un numero incrementale (a partire da 1), diverso per ogni evento.
4. Viene estratto casualmente un nome dalla lista creata al punto *2*, associandolo allo stato iniziale.
5. Vengono quindi aggiunte casualmente le transizioni non osservabili attraverso un metodo ricorsivo, a partire dallo stato iniziale: questo garantisce che tutti gli stati siano raggiungibili dallo stato iniziale e che ogni stato abbia almeno una transizione uscente. Poiché tale metodo può portare alla formazione di cicli, è necessario verificare che il numero no di transizioni osservabili sia almeno pari al numero di cicli. Se ciò non fosse verificato, l’automa presenterebbe dei cicli formati esclusivamente da transizioni non osservabili, provocando la non terminazione del metodo find. In tal caso si ripete l’aggiunta delle transizioni non osservabili finché la condizione di cui sopra viene soddisfatta oppure si raggiunge un limite massimo di tentativi, stabilito dall’utente.
6. Vengono poi aggiunte le transizioni osservabili necessarie a rendere vivo il linguaggio generato dall’automa sull’alfabeto degli eventi : in ogni ciclo generato al punto *5* viene estratta casualmente una transizione (al momento ancora non osservabile), associandole un evento osservabile estratto casualmente dalla lista creata al punto *3*, rendendola dunque osservabile.
7. Vengono inoltre aggiunte altre transizioni osservabili fino a raggiungere il numero no richiesto dall’utente. In particolare fin tanto che il numero residuo di transizioni da aggiungere è strettamente maggiore del numero nf di transizioni di guasto, vengono create casualmente delle transizioni osservabili, altrimenti vengono aggiunte associando un evento osservabile a delle transizioni non osservabili estratte casualmente.
8. Vengono infine aggiunte le nf transizioni di guasto. In particolare fin tanto che ci sono transizioni da aggiungere (per colmare il numero nt) vengono create casualmente delle transizioni di guasto, altrimenti vengono aggiunte associando impostando a **True** l’attributo fault di transizioni non osservabili estratte casualmente.
9. Se nei punti *7* e *8* si dovesse raggiungere il numero massimo di tentativi imposto dall’utente (per impossibilità di aggiungere transizioni preservando la correttezza dell’automa), l’algoritmo termina con un opportuno messaggio di errore.

### 4.2.4 Il modulo random\_automaton

Per la generazione automatica di automi abbiamo creato il modulo random\_automaton, il quale offre il metodo generate\_random\_automaton, affiancato da alcuni metodi necessari al fine della generazione di un singolo automa.

Di seguito è riportato il codice del metodo generate\_random\_automaton; saranno descritti a seguire gli altri metodi del modulo random\_automaton.

**Metodo generate\_random\_automaton**

**def** generate\_random\_automaton**(**ns**,** nt**,** no**,** ne**,** nf**):**

check **=** validate\_params**(**ns**,** nt**,** no**,** ne**,** nf**)**

**if** type**(**check**)** **is** **not** bool**:**

**return** check

params **=** read\_params**()**

**if** type**(**params**)** **is** str**:**

**return** params

max\_attempts **=** read\_params**()[**'attempts'**]**

states **=** dict**()**

**if** ns **<=** len**(**string**.**ascii\_uppercase**):**

states\_names **=** string**.**ascii\_uppercase**[:**ns**]**

**else:**

states\_names **=** **[**'S' **+** str**(**i**)** **for** i **in** range**(**1**,** ns **+** 1**)]**

**if** ne **<=** len**(**string**.**ascii\_lowercase**):**

events\_names **=** string**.**ascii\_lowercase**[:**ne**]**

**else:**

events\_names **=** **[**'e' **+** str**(**i**)** **for** i **in** range**(**1**,** ne **+** 1**)]**

**for** name **in** states\_names**:**

states**[**name**]** **=** State**(**name**)**

events\_names **=** initialize\_events\_names**(**events\_names**,** no**)**

initial\_state **=** random**.**choice**(**states\_names**)**

added\_nt **=** add\_minimal\_transitions**(**states**[**initial\_state**],** states**,** ns**,** nt **-** ns**)**

automaton **=** Automaton**(**initial\_state**,** states**)**

loops **=** automaton**.**get\_loops**()**

attempts **=** 1

**while** len**(**loops**)** **>** no**:**

**if** attempts **>** max\_attempts**:**

**return** log**.**MAX\_ATTEMPTS\_ERROR

states **=** initialize\_states**(**automaton**)**

added\_nt **=** add\_minimal\_transitions**(**states**[**initial\_state**],** states**,** ns**,** nt **-** ns**)**

automaton**.**set\_states**(**states**)**

loops **=** automaton**.**get\_loops**()**

attempts **+=** 1

nt **-=** added\_nt

added\_no **=** add\_minimal\_observable\_transitions**(**automaton**,** events\_names**,** loops**)**

no **-=** added\_no

**while** no **>** 0**:**

**if** nt **>** nf**:**

src **=** states**[**random**.**choice**(**states\_names**)]**

dst **=** states**[**random**.**choice**(**states\_names**)]**

event\_name **=** events\_names**[-**1**]**

attempts **=** 1

**while** **not** src**.**add\_transition**(**dst**,** Transition**(**Event**(**event\_name**))):**

**if** attempts **==** max\_attempts**:**

**return** log**.**MAX\_ATTEMPTS\_ERROR

src **=** states**[**random**.**choice**(**states\_names**)]**

dst **=** states**[**random**.**choice**(**states\_names**)]**

attempts **+=** 1

events\_names**.**pop**()**

nt **-=** 1

**else:**

unobservable\_transitions **=** automaton**.**get\_unobservable\_transitions**()**

transition **=** random**.**choice**(**unobservable\_transitions**)**

transition**.**set\_event**(**Event**(**events\_names**.**pop**()))**

no **-=** 1

prev\_nt **=** nt

prev\_nf **=** nf

prev\_automaton **=** automaton

attempts1 **=** 0

**while** **True:**

**if** attempts1 **==** max\_attempts**:**

**return** log**.**MAX\_ATTEMPTS\_ERROR

nt **=** prev\_nt

nf **=** prev\_nf

automaton **=** deepcopy**(**prev\_automaton**)**

states **=** automaton**.**get\_states**()**

**while** nf **>** 0**:**

**if** nt **>** 0**:**

src **=** states**[**random**.**choice**(**states\_names**)]**

dst **=** states**[**random**.**choice**(**states\_names**)]**

attempts2 **=** 1

**while** **not** src**.**add\_transition**(**dst**,** Transition**(**fault**=True)):**

**if** attempts2 **==** max\_attempts**:**

**return** log**.**MAX\_ATTEMPTS\_ERROR

src **=** states**[**random**.**choice**(**states\_names**)]**

dst **=** states**[**random**.**choice**(**states\_names**)]**

attempts2 **+=** 1

nt **-=** 1

automaton**.**set\_states**(**states**)**

**else:**

unobservable\_transitions **=** automaton**.**get\_unobservable\_transitions**()**

transition **=** random**.**choice**(**unobservable\_transitions**)**

transition**.**set\_fault**()**

nf **-=** 1

**if** **not** automaton**.**has\_fault\_loops**():**

**break**

attempts1 **+=** 1

**for** state **in** automaton**.**get\_states**().**values**():**

state**.**set\_visited**(False)**

**return** automaton

Questo metodo implementa la logica descritta nel paragrafo 4.2.3. Esso inizialmente utilizza il metodo validate\_params per verificare la correttezza dei parametri di input. Per la generazione casuale della lista dei nomi degli eventi osservabili viene utilizzato il metodo initialize\_events\_names, basato su un campionamento con rimpiazzo. Infine le transizioni non osservabili sono aggiunte sfruttando il metodo ricorsivo add\_minimal\_transitions e quelle osservabili, necessarie a rendere vivo il linguaggio, attraverso il metodo add\_minimal\_observable\_transitions.

**Metodo validate\_params**

**def** validate\_params**(**ns**,** nt**,** no**,** ne**,** nf**):**

**if** nt **<** ns**:**

**return** log**.**STATES\_TRANSITIONS\_NUMBER\_ERROR

**if** no **==** 0**:**

**return** log**.**ZERO\_OBSERVABLE\_TRANSITIONS\_ERROR

**if** ne **>** no**:**

**return** log**.**OBSERVABLE\_EVENTS\_TRANSITIONS\_NUMBER\_ERROR

**if** nf **==** 0**:**

**return** log**.**ZERO\_FAULT\_TRANSITIONS\_ERROR

**if** nt **<** no **+** nf**:**

**return** log**.**TOTAL\_TRANSITIONS\_NUMBER\_ERROR

**return** **True**

Si tratta del metodo per la convalida dei parametri di input. Esso effettua i controlli descritti nel punto *1* del paragrafo 4.2.3.

**Metodo initialize\_events\_names**

**def** initialize\_events\_names**(**events\_names**,** no**):**

events **=** dict**()**

**for** name **in** events\_names**:**

events**[**name**]** **=** 1

**for** i **in** range**(**no **-** len**(**events\_names**)):**

name **=** random**.**choice**(**events\_names**)**

events**[**name**]** **+=** 1

result **=** list**()**

**for** name**,** count **in** events**.**iteritems**():**

**for** i **in** range**(**count**):**

result**.**append**(**name**)**

random**.**shuffle**(**result**)**

**return** result

Partendo dalla lista, di lunghezza ne, di nomi degli eventi osservabili semplici generata al punto *3* del paragrafo 4.2.3, questo metodo crea una lista, di lunghezza no, di nomi di eventi ordinati casualmente. Come già anticipato viene sfruttato il campionamento con rimpiazzo dalla lista di input, garantendo però che i nomi degli eventi che compaiono nella lista di input compaiano almeno una volta nella lista di output.

**Metodo add\_minimal\_transitions**

**def** add\_minimal\_transitions**(**src**,** states**,** ns**,** nl**):**

state\_names **=** states**.**keys**()**

src**.**set\_visited**()**

dst\_name **=** random**.**choice**(**state\_names**)**

dst **=** states**[**dst\_name**]**

**if** ns **==** 1**:**

src**.**add\_transition**(**dst**,** Transition**())**

**return** 1

**if** dst**.**is\_visited**()** **and** nl **>** 0**:**

**while** **not** src**.**add\_transition**(**dst**,** Transition**())** **and** len**(**state\_names**)** **>** 1**:**

state\_names**.**remove**(**dst\_name**)**

dst\_name **=** random**.**choice**(**state\_names**)**

dst **=** states**[**dst\_name**]**

**if** dst**.**is\_visited**():**

state\_names **=** states**.**keys**()**

next\_name **=** random**.**choice**(**state\_names**)**

next\_state **=** states**[**next\_name**]**

**while** **not** next\_state**.**is\_visited**()** **and** len**(**state\_names**)** **>** 1**:**

state\_names**.**remove**(**next\_name**)**

next\_name **=** random**.**choice**(**state\_names**)**

next\_state **=** states**[**next\_name**]**

**return** 1 **+** add\_minimal\_transitions**(**next\_state**,** states**,** ns**,** nl **-** 1**)**

**return** 1 **+** add\_minimal\_transitions**(**dst**,** states**,** ns **-** 1**,** nl**)**

**while** dst**.**is\_visited**()** **and** len**(**state\_names**)** **>** 1**:**

state\_names**.**remove**(**dst\_name**)**

dst\_name **=** random**.**choice**(**state\_names**)**

dst **=** states**[**dst\_name**]**

src**.**add\_transition**(**dst**,** Transition**())**

**return** 1 **+** add\_minimal\_transitions**(**dst**,** states**,** ns **-** 1**,** nl**)**

Questo metodo, di natura ricorsiva, si occupa di aggiungere le transizioni non osservabili necessarie a garantire che tutti gli stati siano raggiungibili a partire da quello iniziale e che ogni stato abbia almeno una transizione uscente: il metodo ritorna il numero di transizioni non osservabili aggiunte. Particolare attenzione va rivolta ai parametri di ingresso:

* src, lo stato sorgente della transizione da aggiungere, inizialmente lo stato iniziale;
* states, il dizionario contenente gli stati dell’automa;
* ns, il numero residuo di stati da aggiungere, inizialmente quello imposto dall’utente;
* nl, il numero residuo di cicli che si possono formare senza compromettere il soddisfacimento del vincolo di cui sopra.

La logica alla base di questo metodo è la seguente:

1. Viene marcato come visitato lo stato sorgente src e viene estratto casualmente uno stato destinazione.
2. Se il numero residuo ns di stati è pari a 1, viene aggiunta una transizione non osservabile che collega src allo stato destinazione estratto, ritornando 1 al contesto chiamante.
3. Se lo stato destinazione estratto è già stato visitato ed il numero residuo nl di cicli è strettamente positivo si tenta di aggiungere una transizione tra src e lo stato destinazione: finché l’aggiunta della transizione non va a buon fine viene estratto casualmente un altro stato destinazione.
4. Dopo aver aggiunto la transizione si invoca ricorsivamente il metodo in due modi diversi, a seconda del fatto che lo stato destinazione sia già stato visitato o meno:
   1. se è già stato visitato, la transizione aggiunta ha portato alla formazione di un ciclo; il nuovo stato sorgente viene estratto casualmente tra gli stati già visitati ed il numero residuo nl di cicli viene decrementato di 1;
   2. se non è già stato visitato non si è formato alcun ciclo, pertanto l’aggiunta delle transizioni può proseguire dallo stato destinazione, che diventa lo stato sorgente della chiamata ricorsiva; viene inoltre decrementato di 1 il numero residuo ns di stati.
5. Se al contrario non sono ammessi ulteriori cicli, deve essere estratto casualmente un nuovo stato destinazione tra gli stati non ancora visitati: viene dunque aggiunta una transizione non osservabile tra src e il nuovo stato destinazione. Pertanto l’aggiunta delle transizioni può proseguire da quest’ultimo, che diventa lo stato sorgente della chiamata ricorsiva; viene inoltre decrementato di 1 il numero residuo ns di stati.

**Metodo add\_minimal\_observable\_transitions**

**def** add\_minimal\_observable\_transitions**(**automaton**,** events\_names**,** loops**):**

count **=** 0

**while** len**(**loops**)** **>** 0**:**

loop **=** loops**.**pop**(**0**)**

**if** len**(**loop**)** **==** 1**:**

src\_name **=** loop**[**0**]**

dst\_name **=** src\_name

**else:**

src\_name **=** random**.**choice**(**loop**)**

dst\_index **=** **(**loop**.**index**(**src\_name**)** **+** 1**)** **%** len**(**loop**)**

dst\_name **=** loop**[**dst\_index**]**

event\_name **=** events\_names**[-**1**]**

**if** automaton**.**set\_transition\_event**(**src\_name**,** dst\_name**,** Event**(**event\_name**)):**

events\_names**.**pop**()**

count **+=** 1

**return** count

Questo metodo si occupa di aggiungere le transizioni osservabili necessarie a rendere vivo il linguaggio generato dall’automa sull’alfabeto degli eventi . Come già anticipato si considerano tutti i cicli generati nella fase di aggiunta delle transizioni non osservabili: per ogni ciclo viene estratta casualmente una transizione componente e le viene associato un evento osservabile semplice, rendendola a tutti gli effetti osservabile. Il metodo ritorna il numero di transizioni osservabili aggiunte.

# Capitolo 5 – Complessità e prestazioni

## 5.1 Dimensioni del problema

1. Per definizione questa è la condizione (necessaria) di diagnosticabilità (**C\***). [↑](#footnote-ref-1)