ANALISI DI DIAGNOSTICABILITÀ

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

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* $A = (S, T)$ dove S è l'insieme (non vuoto) degli stati (fra cui uno stato iniziale s_0) e T è l'insieme (non vuoto) delle transizioni di stato;

- il sottoinsieme (non vuoto) delle *transizioni osservabili* di A, $T_o \subseteq T$, ciascuna contraddistinta da un *evento osservabile semplice* $e \in \Sigma_o$ (più transizioni uscenti dal medesimo stato possono essere contraddistinte dallo stesso evento osservabile);

- l'alfabeto degli eventi osservabili semplici Σ_o di A, tale che il linguaggio generato da A su questo alfabeto sia *vivo*, il che comporta che A sia ciclico e che ogni cammino ciclico in A contenga almeno una transizione osservabile;

- il sottoinsieme (non vuoto) delle *transizioni di guasto* di A, $T_f \subseteq T \setminus T_o$, 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 $\ell \geq 1$ da verificare.

Come output il software deve fornire l'indicazione se sia vero o falso che A (come descritto in input attraverso S, s_0, T, T_o, T_f) goda del livello di diagnosticabilità ℓ e, nel caso ciò sia falso, il livello di diagnosticabilità massimo di cui gode A (dove tale livello sarà $< \ell$).

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* $A = (S, T)$, dove S è l'insieme (non vuoto) degli stati (fra cui uno stato iniziale s_0) e T 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.

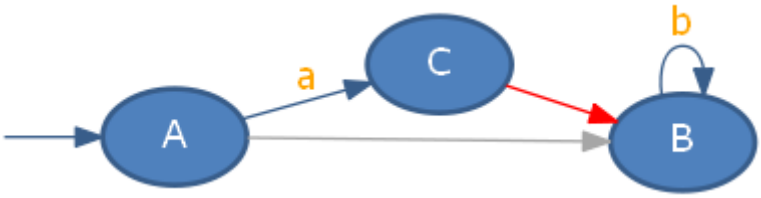

Figura 2.1: Un esempio di automa non deterministico finito. In blu sono rappresentate le transizioni osservabili, in grigio quelle non osservabili e in rosso quelle di guasto.

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

- $S = \{A, B, C\}$
- $s_0 = A$
- $T = \{A \to B, A \to C, C \to B, B \to B\}$
- $\Sigma_o = \{a, b\}$
- $T_o = \{A \xrightarrow{a} C, B \xrightarrow{b} B\}$
- $T_f = \{C \to B\}$

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 livello i è costituito da istanze, non tutte identiche, di eventi appartenenti a Σ_o, 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: nota la chiave, ogni accesso ai valori richiede un tempo costante.

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], 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 NP completo.

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:

[1] Per definizione questa è la condizione (necessaria) di diagnosticabilità (**C***).

CONDIZIONE 1 (C1): Se nell'automa risultante dalla sincronizzazione dei twin del livello considerato non esiste alcuna transizione ambigua (cioè $T_a = \emptyset$), 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 $O(1)$.

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 $O(T_S)$, assumendo $T_S = \frac{|T|}{|S|}$ il numero medio di transizioni uscenti da uno stato (*branching factor*). Il ciclo **`for`** esterno passa in rassegna tutti gli stati, pertanto ha complessità $O(|S|)$. Dunque l'algoritmo presenta una complessità $O(T_S|S|) = O\left(\frac{|T|}{|S|}|S|\right) = O(|T|)$, 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 $O(|T|)$.

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 codifica:

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 attraverso due varianti. La prima variante, rappresentata dal 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.

La seconda variante, rappresentata dal 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 (PRIMA VARIANTE)

```
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 (SECONDA VARIANTE)

```
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

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 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 A generato dovrà essere dotato, dato l'input del problema descritto a pagina *1*, di alcune caratteristiche. In particolare è richiesto che:

1. gli insiemi S, T, T_o e $T_f \subseteq T \setminus T_o$ siano non vuoti;
2. il linguaggio generato da A sull'alfabeto degli eventi osservabili semplici Σ_o sia vivo, il che comporta che A sia ciclico e che ogni cammino ciclico in A contenga almeno una transizione osservabile;
3. la richiesta che il linguaggio generato da A su Σ_o sia vivo implica che ogni stato di A 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 Σ_o;
- `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:

- $nt = \lfloor bf \cdot ns + 0.5 \rfloor$;
- $no = \max(\lfloor po \cdot nt + 0.5 \rfloor, 1)$;
- $ne = \max(\lfloor pe \cdot no + 0.5 \rfloor, 1)$;
- $nf = \max(\lfloor pf \cdot nt + 0.5 \rfloor, 1)$.

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:
    a. `nt` $\geq$ `ns`, ovvero non possono esserci più stati che transizioni;
    b. `no` > 0, ovvero deve esserci almeno una transizione osservabile;
    c. `ne` $\leq$ `no`, ovvero non possono esserci più eventi che transizioni osservabili;
    d. `nf` > 0, ovvero deve esserci almeno una transizione di guasto;
    e. `nt` $\geq$ `no` $+$ `nf`, ovvero l'unione delle transizioni osservabili e di quelle di guasto deve essere contenuta nell'insieme delle transizioni T.
2. Viene creata la lista dei nomi degli stati, lunga `ns`:
    a. se `ns` ≤ 26, il nome di uno stato corrisponde ad una lettera maiuscola dell'alfabeto latino;
    b. se `ns` > 26, 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`:
    a. se `no` ≤ 26, il nome di un evento corrisponde ad una lettera minuscola dell'alfabeto latino;
    b. se `no` > 26, 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 A sull'alfabeto degli eventi Σ_o: 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:
    a. 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;
    b. 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 A sull'alfabeto degli eventi Σ_o. 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

Abbiamo ritenuto significativo considerare, come dimensioni principali del problema, alcuni dei parametri con cui vengono generati casualmente i casi di test. Abbiamo infatti preso in analisi:

- il numero totale di stati dell'automa (parametro `ns`);
- il *branching factor*, ovvero il numero medio di transizioni uscenti da ogni stato (parametro `bf`). Si noti la relazione lineare con il numero di transizioni $nt = \lfloor bf \cdot ns + 0.5 \rfloor$;
- il livello di diagnosticabilità ℓ dato in input al problema.

Altre dimensioni che abbiamo considerato nelle sperimentazioni sono quelle che influenzano la generazione di transizioni ambigue:

- la percentuale di transizioni osservabili (parametro `po`);
- il rapporto tra la cardinalità dell'insieme degli eventi ed il numero di transizioni osservabili (parametro `pe`);
- la percentuale di transizioni di guasto (parametro `pf`).

5.2 Considerazioni preliminari alla sperimentazione

Il problema di determinare se un automa non deterministico finito goda del livello di diagnosticabilità ℓ è notoriamente intrattabile. Senza soffermarci sulla trattazione teorica della complessità, abbiamo posto attenzione alle seguenti considerazioni ed aspettative.

1. È ragionevole pensare che l'introduzione delle tre condizioni di diagnosticabilità (**C1**, **C2** e **C3**), ampiamente descritte nel Capitolo *3*, possa condurre ad una riduzione del tempo di esecuzione dell'algoritmo. Tuttavia nel caso (pessimo) in cui nessuna delle tre condizioni sia verificata, 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 (sezione 3.2.1).
2. È impossibile, in ogni caso, sapere a priori quando e se le condizioni saranno soddisfatte. Per questo motivo abbiamo analizzato l'impatto dell'utilizzo delle tre condizioni di diagnosticabilità soltanto dal punto di vista empirico.
3. In generale ci aspettavamo un tempo di esecuzione che crescesse esponenzialmente al crescere delle dimensioni del problema (in particolare del livello di diagnosticabilità).
4. Il tempo di esecuzione non dipende necessariamente dal livello di diagnosticabilità ℓ dato in input al problema, ma piuttosto dal livello di diagnosticabilità massimo i di cui effettivamente gode l'automa. Nel caso in cui $i < \ell$, il tempo di esecuzione è influenzato solamente dal tempo di generazione degli automi intermedi (bad twin, good twin e sincronizzato) fino al livello i. A parità degli altri parametri (numero di stati e numero di transizioni), due automi che godono di livelli di diagnosticabilità diversi richiedono tempi di esecuzione diversi.

5.2.1 Note sulla complessità spaziale

La complessità spaziale presenta due fronti di analisi:

- lo spazio riservato alle strutture dati;
- lo spazio dedicato al salvataggio dei contesti di eventuali chiamate ricorsive.

Per quanto riguarda le strutture dati, possiamo affermare che lo spazio di memoria occupato dagli automi intermedi (bad twin, good twin e sincronizzato), generati al fine di verificare la diagnosticabilità di livello i, cresca esponenzialmente al crescere del livello di diagnosticabilità considerato. A titolo di esempio consideriamo la cardinalità dell'alfabeto degli eventi osservabili semplici $n = |\Sigma_o|$. Il multinsieme degli eventi osservabili fino al livello i è dato da $\Sigma_o^i = \Sigma_o^{i-1} \cup \Sigma_o^{i*}$, dove Σ_o^{i*} è l'insieme degli eventi osservabili esattamente di livello i. La cardinalità di Σ_o^{i*} è dunque pari al numero di combinazioni con ripetizione di n elementi di classe i, a cui va sottratto n poiché

non possono esistere eventi osservabili di livello i costituiti da istanze tutte identiche dello stesso evento osservabile $\in \Sigma_o$:

$$\left|\Sigma_o^{i*}\right| = \binom{n+i-1}{i} - n$$
$$= \frac{(n+i-1)(n+i-2)\ldots n(n-1)!}{i!\,(n-1)!} - n$$
$$\approx \frac{(n+i-1)(n+i-2)\ldots n}{i!}$$

Potenzialmente il numero di transizioni di un bad twin di livello i è $O\left(\binom{|S|}{2}\left|\Sigma_o^i\right|\right) = O(|S|^2\left|\Sigma_o^i\right|)$.

Per quanto concerne il salvataggio dei contesti è difficile stimare l'altezza dell'albero delle chiamate ricorsive (gli algoritmi ricorsivi coinvolti nel problema sono l'algoritmo `find` e quello che controlla la condizione **C***, implementato dal metodo `find_loops` e introdotto nel Capitolo *3*).

Dopo aver effettuato alcune prove abbiamo trascurato lo studio sperimentale della complessità spaziale: non abbiamo individuato alcuno strumento che fosse in grado di quantificare in modo sufficientemente preciso la memoria occupata dai contesti di esecuzione e dalle strutture dati.

5.3 Uno script Python per la sperimentazione

Per automatizzare il processo di generazione di più casi di test abbiamo eseguito lo script `main` che contiene metodi specifici per ogni parametro che si desidera far variare. In particolare ogni metodo scrive un file *CSV*[2] avente come nome il timestamp della data corrente e salvato nella directory `times/nome_parametro/` dove `nome_parametro` è il nome del parametro che viene fatto variare. Nel file vengono registrati i principali parametri di interesse, accompagnati dai tempi di esecuzione (misurati in secondi) di ognuna delle quattro versioni del metodo risolvente. I tempi di esecuzione salvati nel file sono calcolati mediando i tempi di esecuzione misurati su `n` casi di test, dove `n` è il parametro rappresentante il numero di automi che vengono generati per ogni configurazione dei parametri di input.

[2] *Comma Separated Value*

5.4 Obiettivi della sperimentazione

Nei sotto-paragrafi seguenti vengono riportati i punti salienti delle sperimentazioni condotte grazie al generatore automatico di casi di test ed allo script `main`. Nella fase preliminare alla sperimentazione ci siamo posti due diversi obiettivi:

1. valutare la complessità temporale al variare delle dimensioni principali del problema (livello di diagnosticabilità, numero di stati e *branching factor*), confrontando tra loro i tempi di esecuzione delle quattro versioni del metodo risolvente; a questo scopo sono stati condotti gli esperimenti 1-3;
2. valutare l'impatto della verifica delle tre condizioni di diagnosticabilità, al variare dei parametri responsabili dell'eventuale generazione di transizioni ambigue, quali la cardinalità dell'alfabeto degli eventi, la percentuale di transizioni osservabili e la percentuale di transizioni di guasto; a questo scopo sono stati condotti gli esperimenti 4-6.

Affinché l'analisi fosse sensata è stato necessario rendere comparabili i casi di test: tutti gli automi sui quali invocare ognuna delle quattro versioni del metodo risolvente sono stati generati in modo che godessero di un livello di diagnosticabilità minimo ℓ, impostabile dal file di configurazione. Per questo motivo abbiamo creato il metodo `search_automaton` per cercare un automa diagnosticabile almeno fino al livello ℓ. Di seguito il codice del metodo:

```
def search_automaton(ns, nt, no, ne, nf, level, method):
    automaton = generate_random_automaton(ns, nt, no, ne, nf)
    while type(automaton) is not str and type(method(automaton, level)) is not
bool:
        automaton = generate_random_automaton(ns, nt, no, ne, nf)
    return automaton
```

Come si può notare vengono generati automi casuali attraverso il metodo `generate_random_automaton` fino a trovarne uno per il quale il metodo risolvente ritorna **`True`**.

5.4.1 Esperimento 1

Per confrontare i tempi di esecuzione al crescere del livello di diagnosticabilità abbiamo impostato nel file di configurazione i seguenti parametri: `var=lv`, `n=10`, `lvmin=1`, `lvmax=6`, `lvstep=1`, `ns=6`, `nt=10`, `no=5`, `ne=4` e `nf=1`.

Come si può notare dalla *Figura 5.1*, l'andamento del tempo di esecuzione è chiaramente conforme alle nostre aspettative: esso aumenta esponenzialmente all'aumentare del livello di diagnosticabilità.

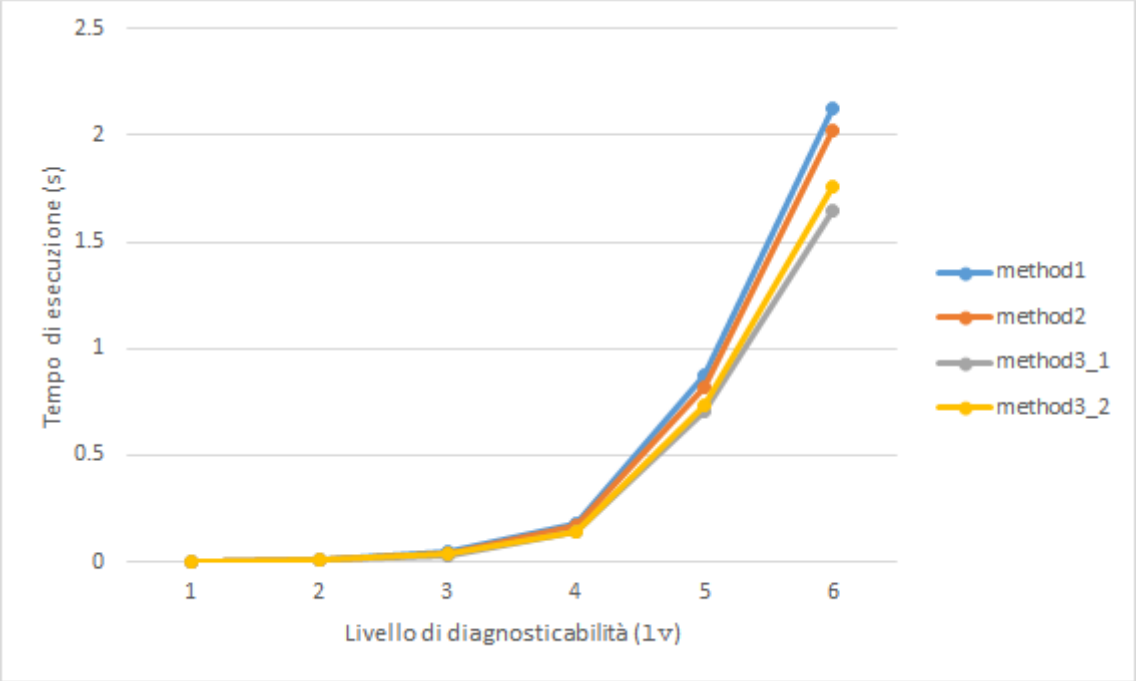

Figura 5.1: Tempi di esecuzione al variare del livello di diagnosticabilità.

5.4.2 Esperimento 2

Per confrontare i tempi di esecuzione al crescere del numero di stati abbiamo impostato nel file di configurazione i seguenti parametri: `var=ns`, `n=10`, `level=3`, `nsmin=5`, `nsmax=50`, `nsstep=5`, `bf=1.75`, `po=0.75`, `pe=0.75` e `pf=0.01`.

Come si può notare dalla *Figura 5.2*, possiamo concludere che, come è giusto aspettarsi, il tempo di esecuzione aumenta all'aumentare del numero di stati, attraverso una relazione difficile da definire in modo preciso.

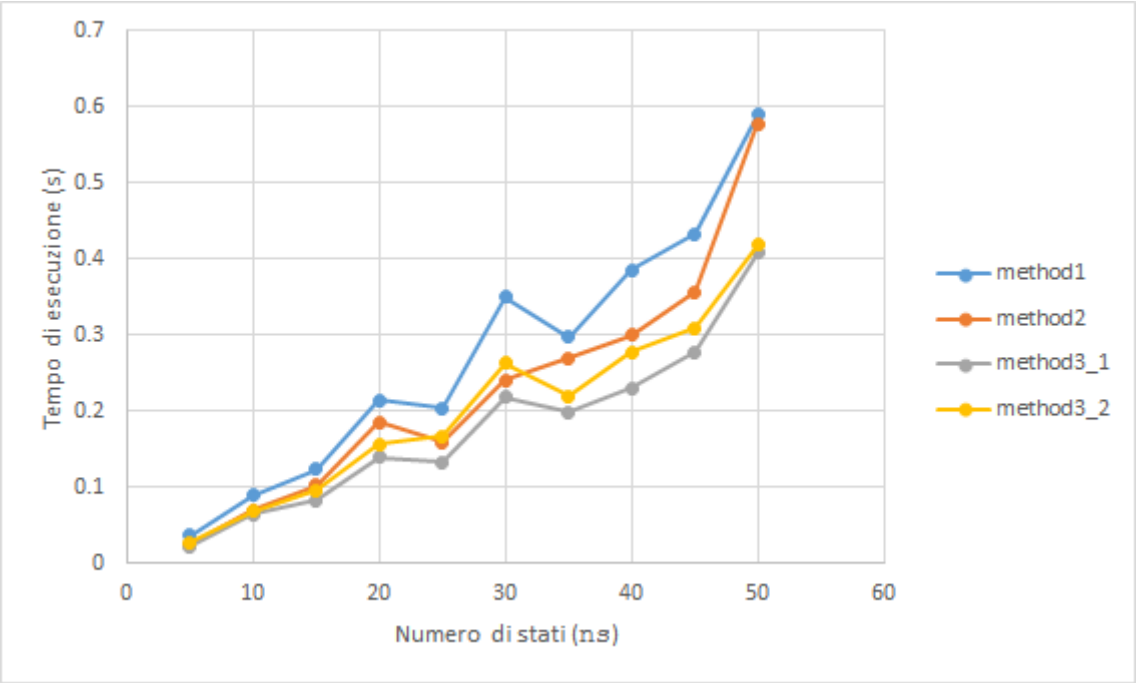

Figura 5.2: Tempi di esecuzione al variare del numero di stati.

5.4.3 Esperimento 3

Per confrontare i tempi di esecuzione al crescere del *branching factor* abbiamo impostato nel file di configurazione i seguenti parametri: `var=bf`, `n=10`, `level=3`, `bfmin=1`, `bfmax=3`, `bfstep=0.5`, `ns=10`, `po=0.75`, `pe=0.75` e `pf=0.01`.

Come si può notare dalla *Figura 5.3*, il tempo di esecuzione aumenta esponenzialmente all'aumentare del *branching factor*.

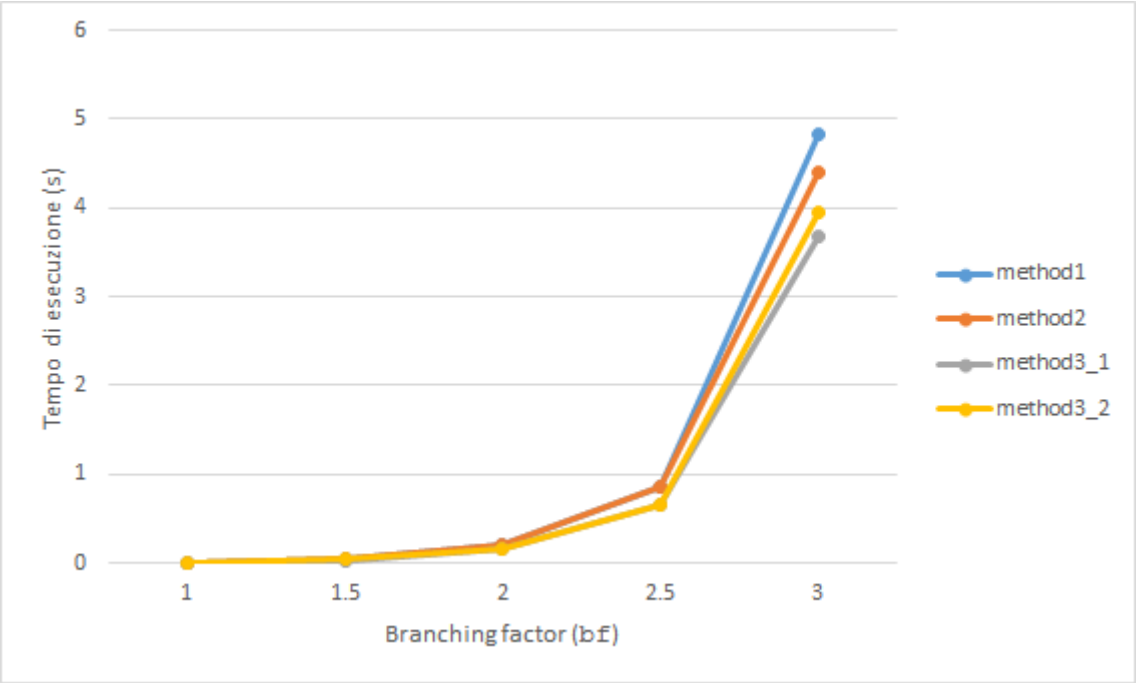

Figura 5.3: Tempi di esecuzione al variare del branching factor.

5.4.4 Esperimento 4

Per valutare l'impatto della verifica delle tre condizioni di diagnosticabilità al variare della cardinalità dell'alfabeto degli eventi abbiamo impostato nel file di configurazione i seguenti parametri: `var=pe`, `n=100`, `level=3`, `pemin=0.1`, `pemax=1.0`, `pestep=0.1`, `ns=10`, `nt=18`, `po=0.9` e `pf=0.1`.

Come si può notare dalla *Figura 5.4*, il tempo di esecuzione non varia in modo significativo al variare del rapporto tra la cardinalità dell'alfabeto degli eventi e il numero di transizioni osservabili. È però importante sottolineare come sia evidente il vantaggio dell'introduzione delle tre condizioni di diagnosticabilità: le due[3] versioni del metodo risolvente che le verificano richiedono tempi di esecuzione decisamente inferiori rispetto alla prima versione.

Una variante di questo esperimento è stata condotta modificando il bilancio tra le transizioni osservabili e quelle di guasto. I parametri utilizzati per questa variante sono gli stessi elencati sopra, ad eccezione di `po=0.5` e `pf=0.5`. Il risultato è mostrato in *Figura 5.5*.

In entrambe le configurazioni gli automi generati sono privi di transizioni non osservabili che non siano di guasto, infatti le percentuali `po` e `pf` sono complementari.

[3] Si intendono le versioni 2 e 3 (entrambe le varianti).

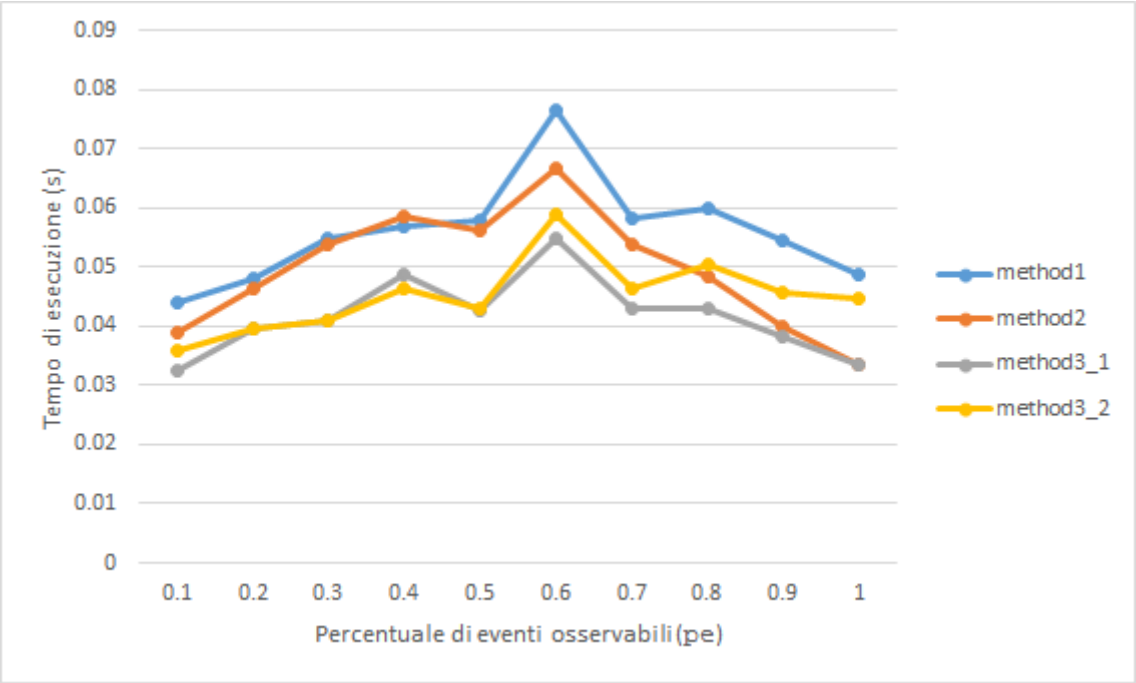

Figura 5.4: Tempi di esecuzione al variare della percentuale di eventi osservabili (prima variante dell'esperimento 4).

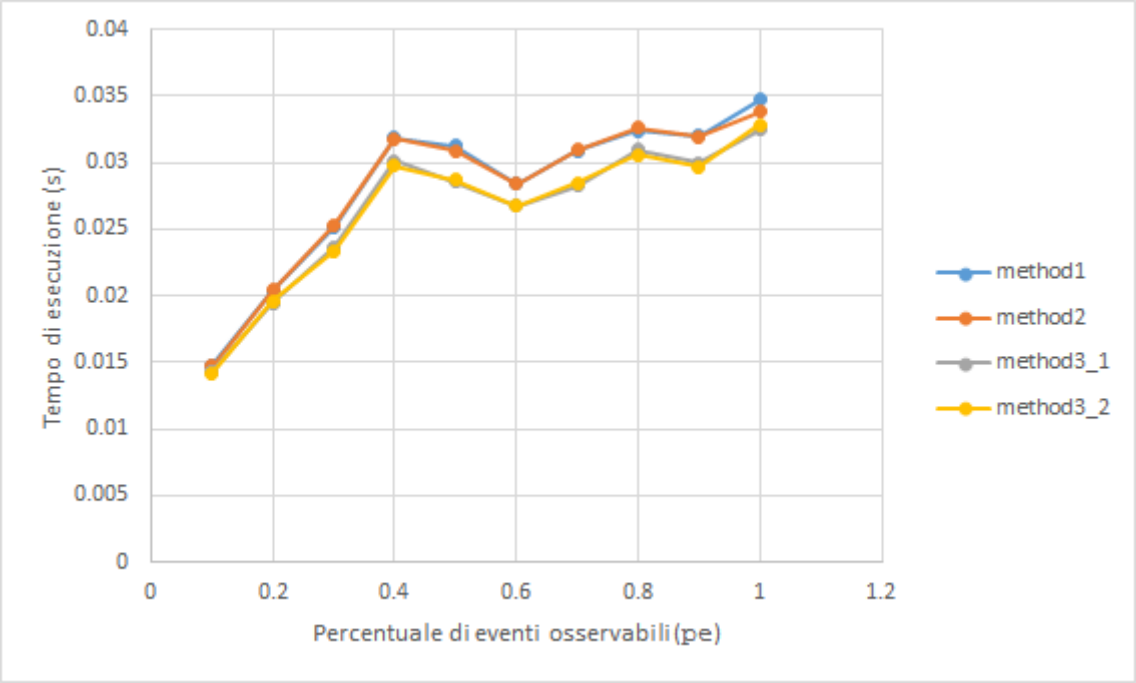

Figura 5.5: Tempi di esecuzione al variare della percentuale di eventi osservabili (seconda variante dell'esperimento 4).

5.4.5 Esperimento 5

Per valutare l'impatto della verifica delle tre condizioni di diagnosticabilità al variare della percentuale di transizioni osservabili abbiamo impostato nel file di configurazione i seguenti parametri: `var=po`, `n=100`, `level=3`, `pomin=0.1`, `pomax=0.9`, `postep=0.1`, `ns=10`, `nt=18`, `pe=1.0` e `pf=0.1`.

È interessante notare in *Figura 5.6* l'andamento del tempo di esecuzione apparentemente simmetrico rispetto all'asse `po=0.5`. In ogni caso le due versioni che utilizzano le condizioni di diagnosticabilità richiedono tempi inferiori rispetto alla prima.

Analogamente all'esperimento 4, è stata condotta una variante di questo esperimento, ottenuta modificando il bilancio tra le transizioni osservabili e quelle di guasto. I parametri utilizzati per questa variante sono gli stessi elencati sopra, ad eccezione di `pomax=0.5` e `pf=0.5`.

Dalla *Figura 5.7* si può notare come il tempo di esecuzione aumenti linearmente all'aumentare della percentuale di transizioni osservabili.

In entrambe le varianti è stato posto `pe=1.0` per forzare la generazione di automi in cui a transizioni osservabili diverse fossero associati eventi diversi.

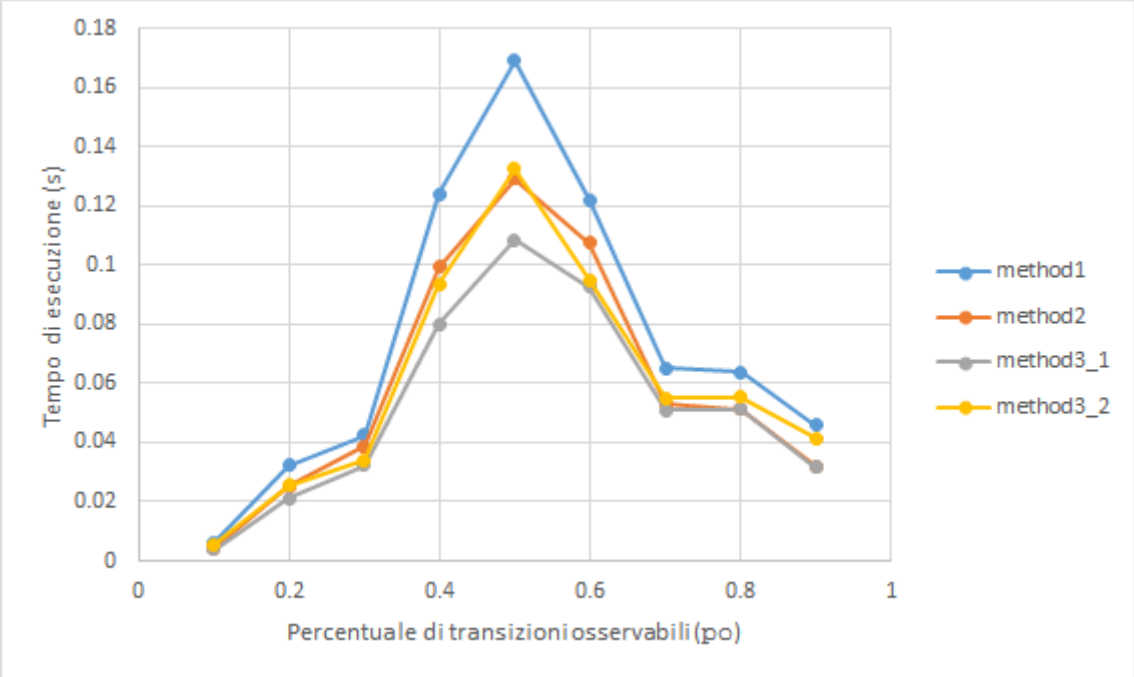

Figura 5.6: Tempi di esecuzione al variare della percentuale di transizioni osservabili (prima variante dell'esperimento 5).

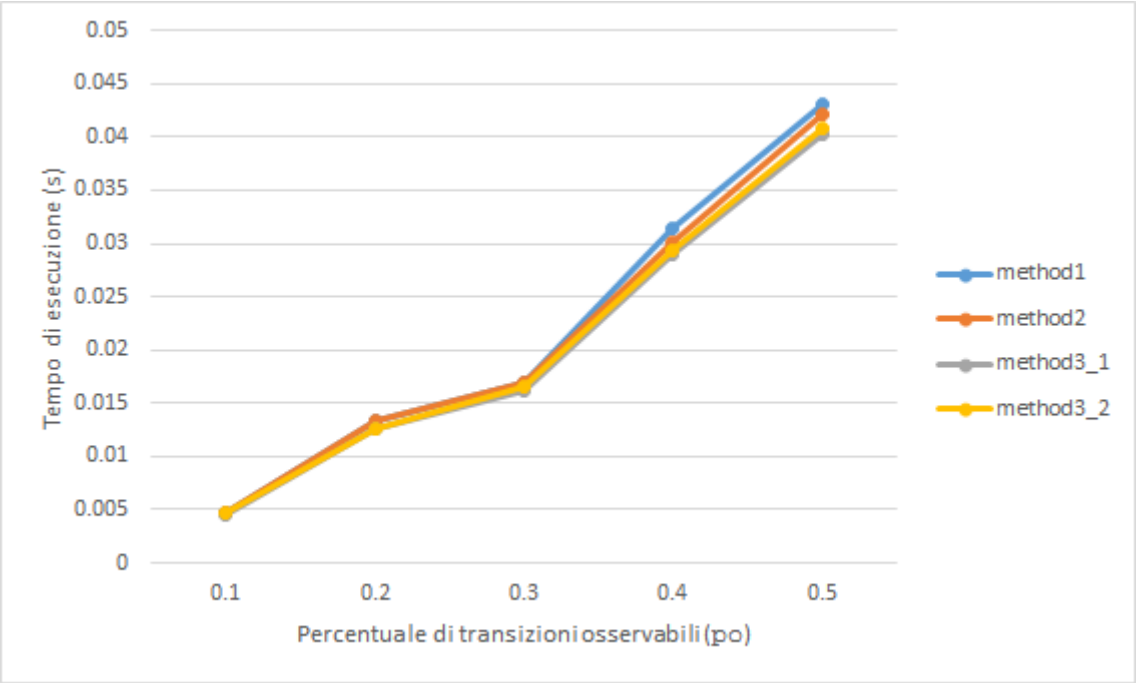

Figura 5.7: Tempi di esecuzione al variare della percentuale di transizioni osservabili (seconda variante dell'esperimento 5).

5.4.6 Esperimento 6

Per valutare l'impatto della verifica delle tre condizioni di diagnosticabilità al variare della percentuale di transizioni di guasto abbiamo impostato nel file di configurazione i seguenti parametri: `var=pf`, `n=100`, `level=3`, `pfmin=0.1`, `pfmax=0.9`, `pfstep=0.1`, `ns=10`, `nt=18`, `pe=1.0` e `po=0.1`.

Come si può notare dalla *Figura 5.8*, il tempo di esecuzione non varia in modo significativo al variare della percentuale di transizioni di guasto.

Anche in questo caso è stata condotta una variante dell'esperimento, ottenuta modificando il bilancio tra le transizioni osservabili e quelle di guasto. I parametri utilizzati per questa variante sono gli stessi elencati sopra, ad eccezione di `pfmax=0.5` e `po=0.5`.

Dalla *Figura 5.9* si può notare come il tempo di esecuzione diminuisca all'aumentare della percentuale di transizioni osservabili.

In entrambe le varianti è stato posto `pe=1.0` per forzare la generazione di automi in cui a transizioni osservabili diverse fossero associati eventi diversi.

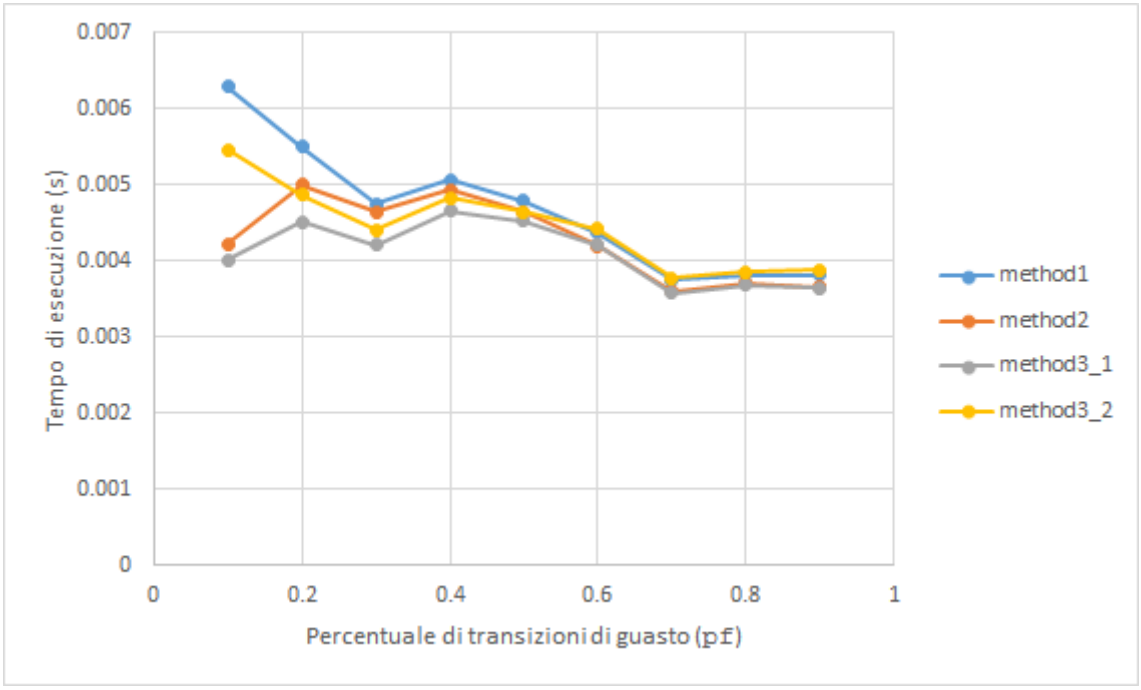

Figura 5.8: Tempi di esecuzione al variare della percentuale di transizioni di guasto (prima variante dell'esperimento 6).

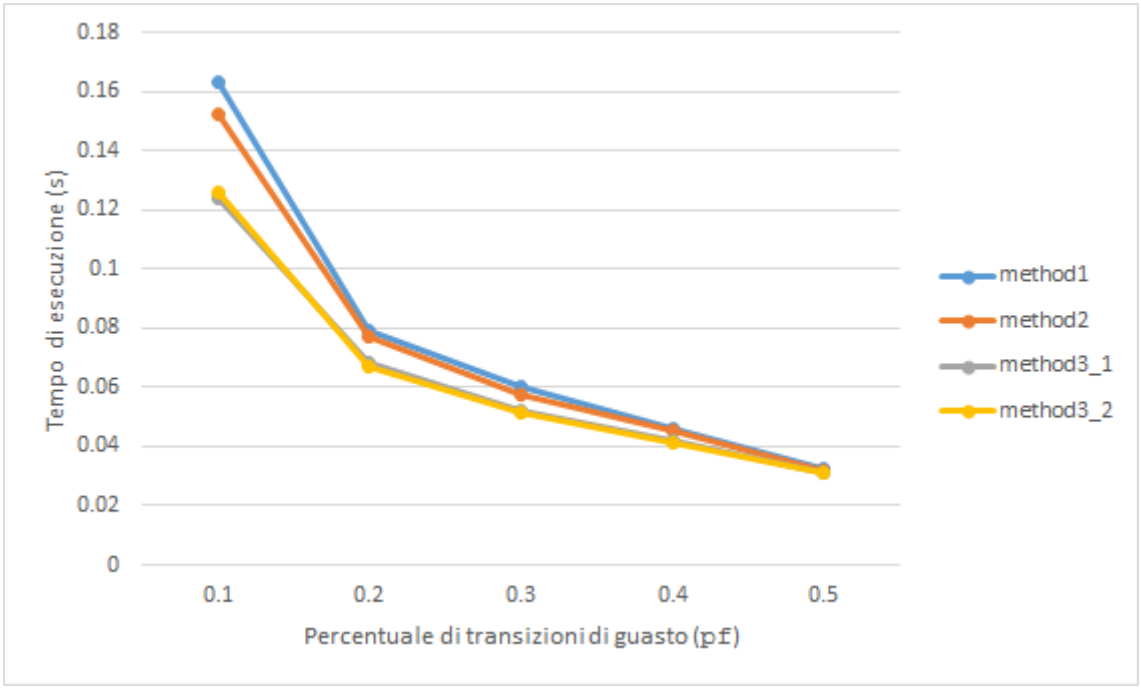

Figura 5.9: Tempi di esecuzione al variare della percentuale di transizioni di guasto (seconda variante dell'esperimento 6).

CAPITOLO 6 – UTILIZZO DEL SOFTWARE

6.1 Il file di configurazione

La parte centrale del software risiede nel file di configurazione, in cui è possibile impostare moltissimi parametri, quali:

- `level`, il livello di diagnosticabilità da verificare;
- `method`, la versione del metodo risolvente da utilizzare per la verifica;
- `save`, il flag che indica se salvare o meno i risultati intermedi (file *XML* e immagini degli automi generati quali bad twin, good twin e sincronizzato);
- `compact`, il flag che indica se rappresentare nelle immagini le transizioni in maniera compatta (la stessa rappresentazione utilizzata nelle specifiche dell'elaborato);
- `ns`, `nt`, `no`, `ne`, `nf`, parametri di input al generatore di automi casuali (sezione *4.2.2*);
- `attempts`, il numero massimo di tentativi per generare un automa in maniera casuale;
- `var`, il parametro da far variare nell'analisi delle prestazioni;
- `n`, il numero di automi da generare per ogni configurazione dei parametri di input;
- `bf`, `po`, `pe`, `pf`, ulteriori parametri utili al generatore di automi casuali (sezione *4.2.2*);
- `lvmin`, `lvmax`, `lvstep`, rispettivamente il valore minimo, massimo e il passo del livello di diagnosticabilità;
- `nsmin`, `nsmax`, `nsstep`, rispettivamente il valore minimo, massimo e il passo del numero di stati;
- `bfmin`, `bfmax`, `bfstep`, rispettivamente il valore minimo, massimo e il passo del *branching factor*;
- `pomin`, `pomax`, `postep`, rispettivamente il valore minimo, massimo e il passo della percentuale di transizioni osservabili;
- `pemin`, `pemax`, `pestep`, rispettivamente il valore minimo, massimo e il passo del rapporto tra la cardinalità dell'alfabeto degli eventi ed il numero di transizioni osservabili;

- `pfmin`, `pfmax`, `pfstep`, rispettivamente il valore minimo, massimo e il passo della percentuale di transizioni di guasto.

6.2 Interfaccia a linea di comando

Come già anticipato, per la sperimentazione è stato utilizzato lo script `main`. Dopo aver impostato gli opportuni parametri nel file di configurazione, è sufficiente eseguire il seguente comando:

```
python main.py
```

Il programma inizierà a generare i casi di test necessari per la sperimentazione. Poiché questo processo può richiedere parecchio tempo, abbiamo ritenuto importante mostrare all'utente le informazioni circa lo stato di avanzamento, come mostrato in *Figura 6.1*.

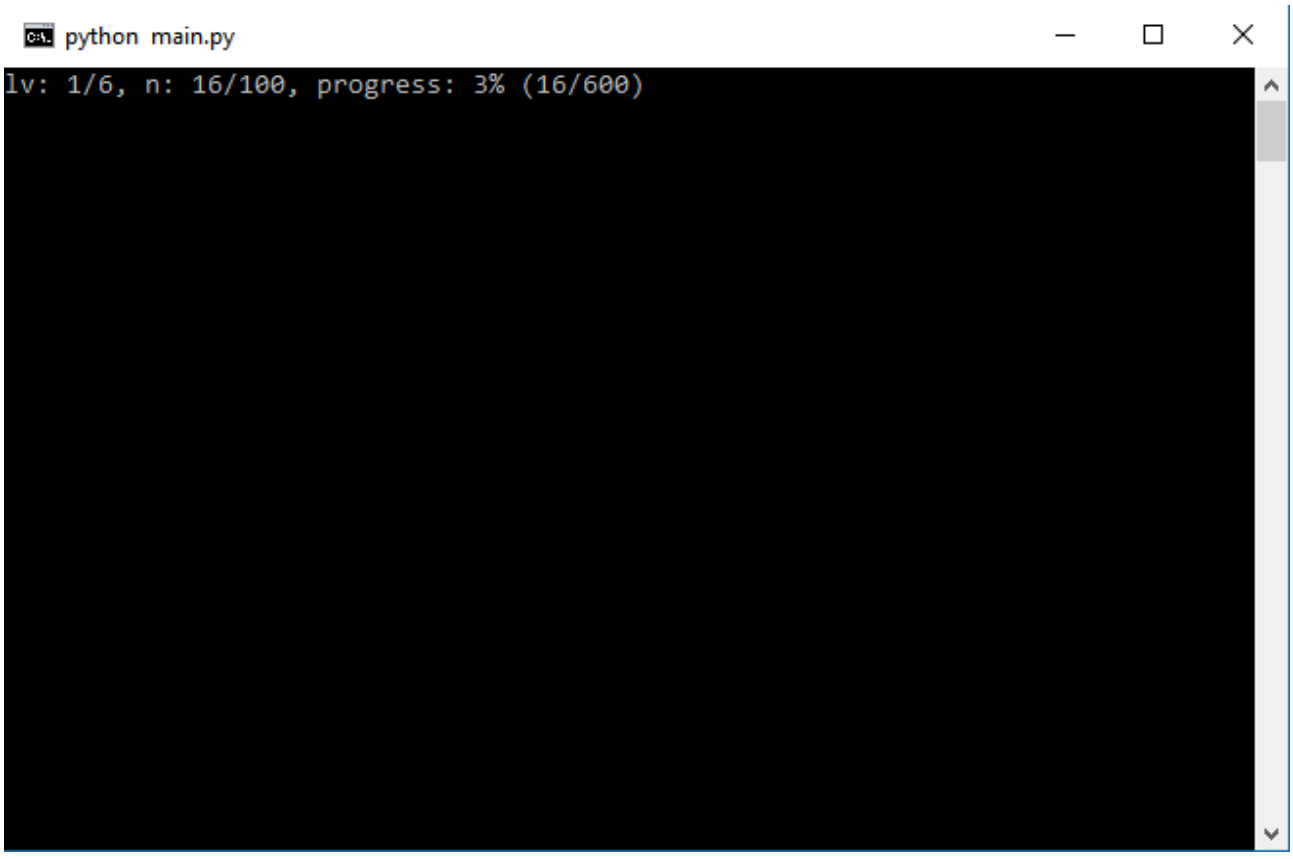

Figura 6.1: Esempio di esecuzione dello script `main`.

Una volta terminato, il programma salva i risultati in un file *CSV* avente la seguente struttura:

```
lv;ns;nt;no;ne;nf;method1;method2;method3_1;method3_2
1;6;10;5;4;1;0.001859903;0.001351968;0.001434749;0.002189921
2;6;10;5;4;1;0.01098624;0.010108693;0.008093215;0.009515529
3;6;10;5;4;1;0.044736649;0.039731035;0.029330577;0.035487384
4;6;10;5;4;1;0.176744121;0.169228753;0.13884125;0.140457864
5;6;10;5;4;1;0.878762283;0.815224246;0.705517457;0.73662488
6;6;10;5;4;1;2.123607096;2.022971653;1.649551594;1.761765217
```

6.3 Interfaccia grafica

Ulteriori funzionalità, che non riguardano la sperimentazione, sono offerte dallo script `main_gui`, quali:

- modifica dei parametri attraverso l'apertura del file di configurazione;
- caricamento di un automa da un file *XML*;
- generazione di un automa casuale, in accordo ai parametri impostati;
- esecuzione del controllo del livello di diagnosticabilità dell'automa, sia esso caricato o generato casualmente;
- salvataggio di un report contenente tutte le informazioni raccolte durante l'analisi di diagnosticabilità (ad esempio il soddisfacimento di eventuali condizioni di diagnosticabilità).

Una volta lanciato il comando

```
python main_gui.py
```

apparirà l'interfaccia mostrata in *Figura 6.2*, in stile "finestra" dell'ambiente *Windows*.

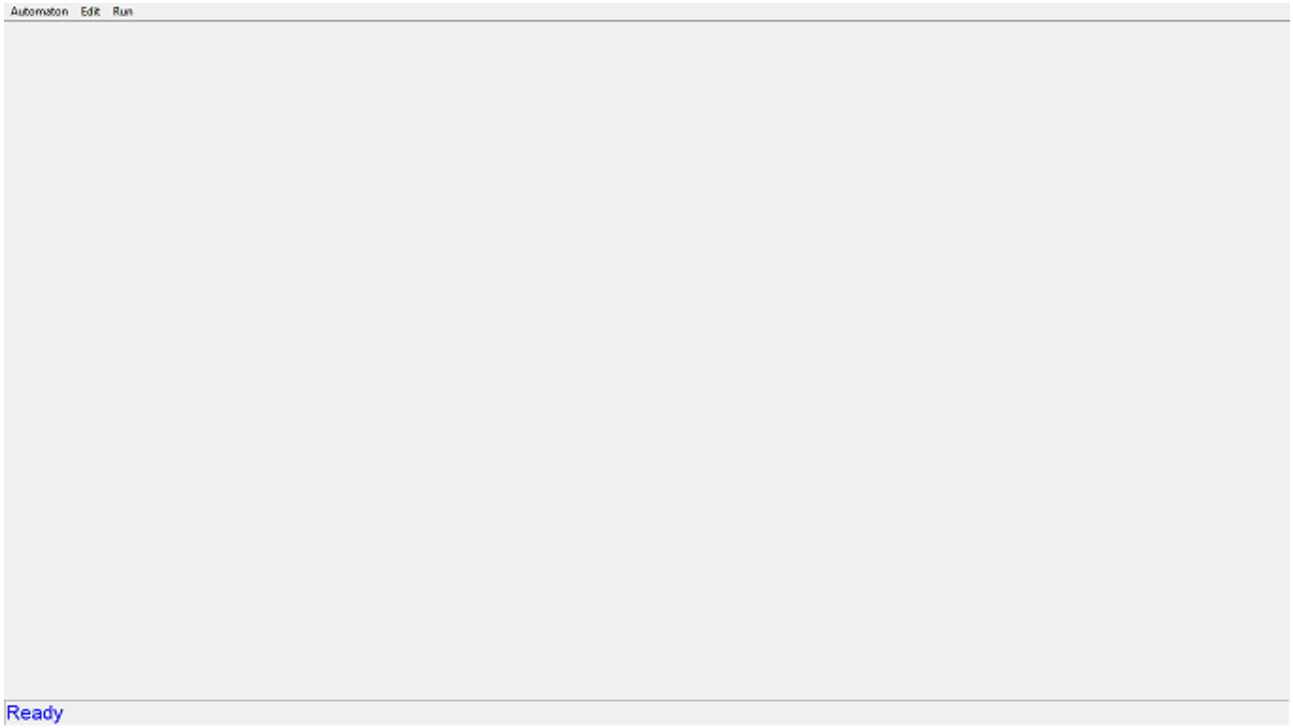

Figura 6.2: Esempio di esecuzione dello script `main_gui`.

6.4 Esempio completo di utilizzo

Per maggior chiarezza, mostriamo ora un esempio completo di utilizzo dell'interfaccia grafica.

6.4.1 Caricamento di un automa

Per caricare un automa è sufficiente cliccare sulla voce **Load** del menu **Automaton**, come mostrato in *Figura 6.3*.

Una volta cliccata la voce, apparirà una finestra di dialogo in cui è possibile selezionare il file *XML* che descrive l'automa da caricare, come mostrato in *Figura 6.4*.

Cliccando sul pulsante "Apri" l'automa viene caricato e ne viene mostrata l'immagine all'interno della finestra, come mostrato in *Figura 6.5*.

Figura 6.3: Selezione della voce per caricare un automa.

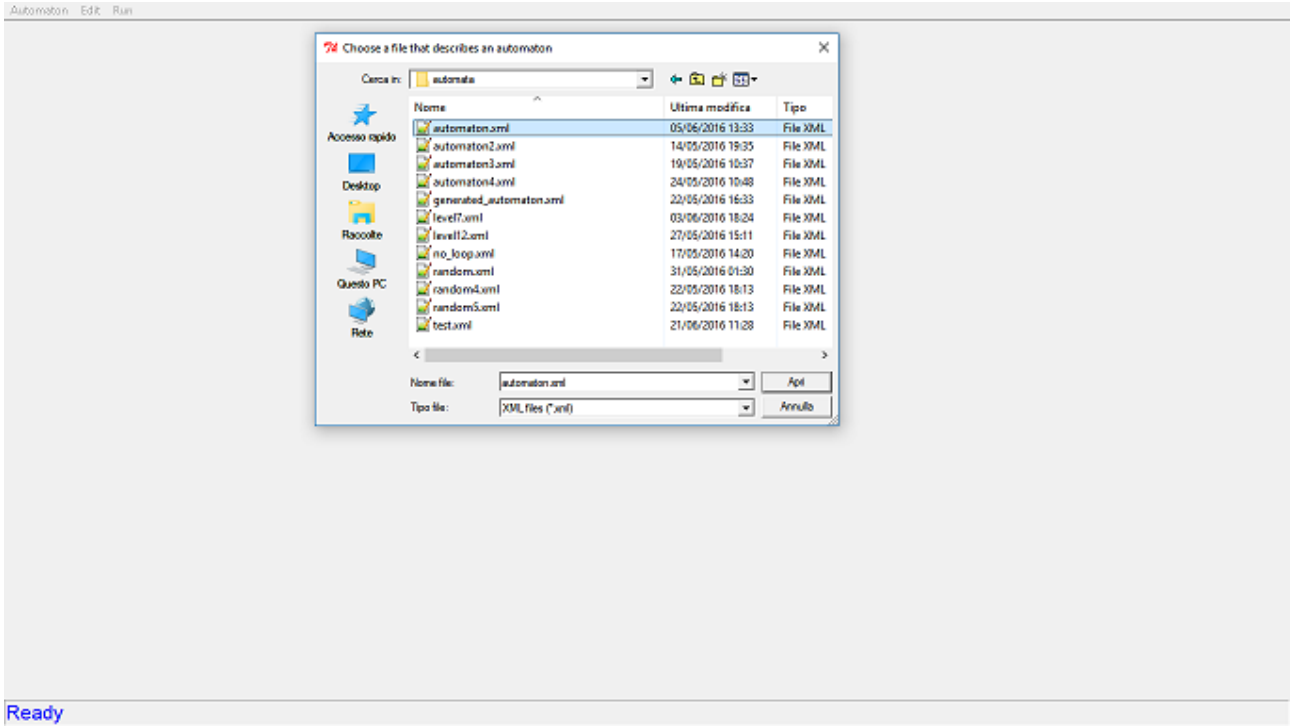

Figura 6.4: Finestra di dialogo per selezionare un file XML.

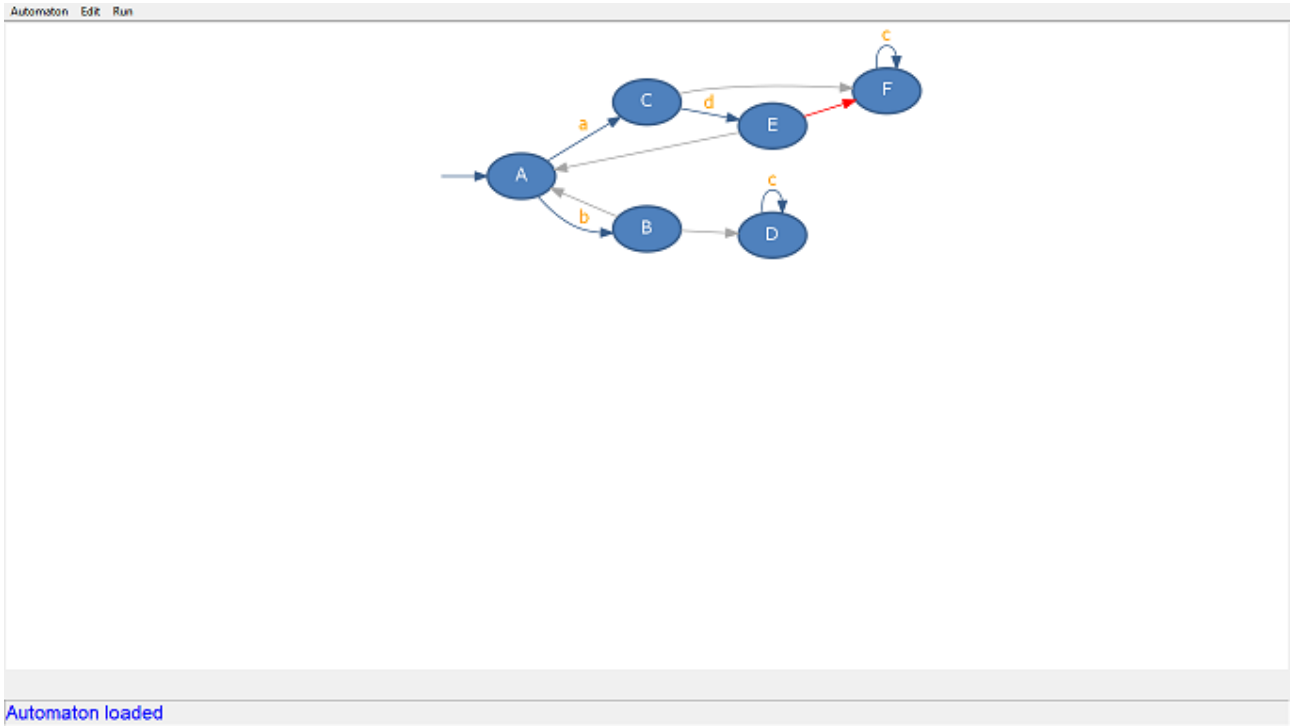

Figura 6.5: Esempio di caricamento di un automa avvenuto con successo.

6.4.2 Generazione di un automa casuale

Supponiamo di voler generare un automa avente 3 stati, 5 transizioni, 2 eventi osservabili differenti, 3 transizioni osservabili ed una di guasto. Prima di procedere con la generazione, assicuriamoci di aver impostato correttamente i parametri, aprendo il file di configurazione (sezione *6.4.3*).

Per generare un automa in maniera casuale è sufficiente cliccare sulla voce **Generate** del menu **Automaton**, come mostrato in *Figura 6.6*.

Una volta cliccata la voce, l'automa viene generato utilizzando i parametri impostati nel file di configurazione e ne viene mostrata l'immagine all'interno della finestra, come mostrato in *Figura 6.7*.

Figura 6.6: Selezione della voce per generare un automa casuale.

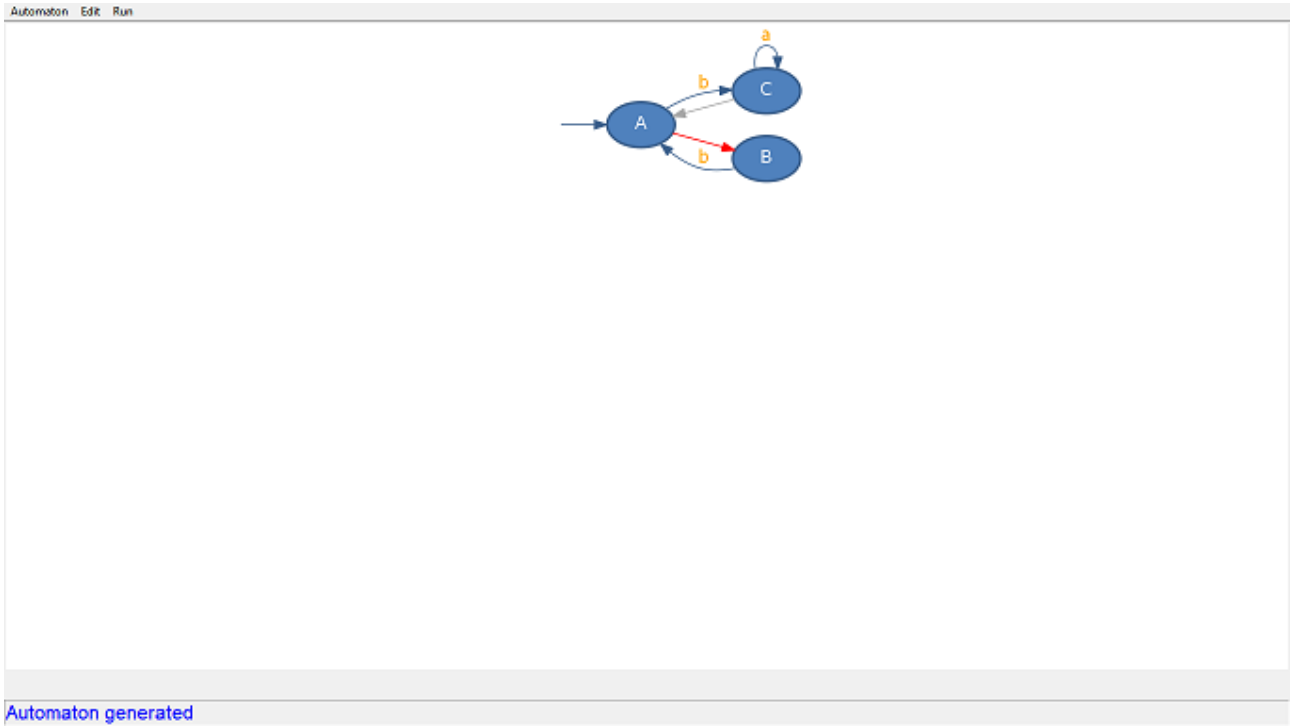

Figura 6.7: Esempio di generazione di un automa casuale avvenuta con successo.

6.4.3 Modifica del file di configurazione

In qualsiasi momento è possibile modificare i parametri all'interno del file di configurazione. Per aprire il file è sufficiente cliccare sulla voce **Parameters** del menu **Edit**, come mostrato in *Figura 6.8*. Una volta cliccata la voce, si aprirà il file di configurazione con un editor di testo, nel quale sarà possibile modificare i parametri, come mostrato in *Figura 6.9*.

Figura 6.8: Selezione della voce per aprire il file di configurazione.

6.4.4 Controllo del livello di diagnosticabilità

Dopo aver caricato o generato un automa, è possibile controllarne il livello di diagnosticabilità. Per effettuare il controllo è sufficiente cliccare sulla voce **Check diagnosability level** del menu **Run**, come mostrato in *Figura 6.10*.

Poiché questa operazione può richiedere parecchio tempo, in qualsiasi momento l'utente può decidere di interrompere il processo, cliccando sulla voce **Stop computation** del menu **Run**, come mostrato in *Figura 6.11*.

Se il controllo non viene interrotto, il programma mostra i risultati dell'elaborazione in basso a destra, come mostrato in *Figura 6.12*.

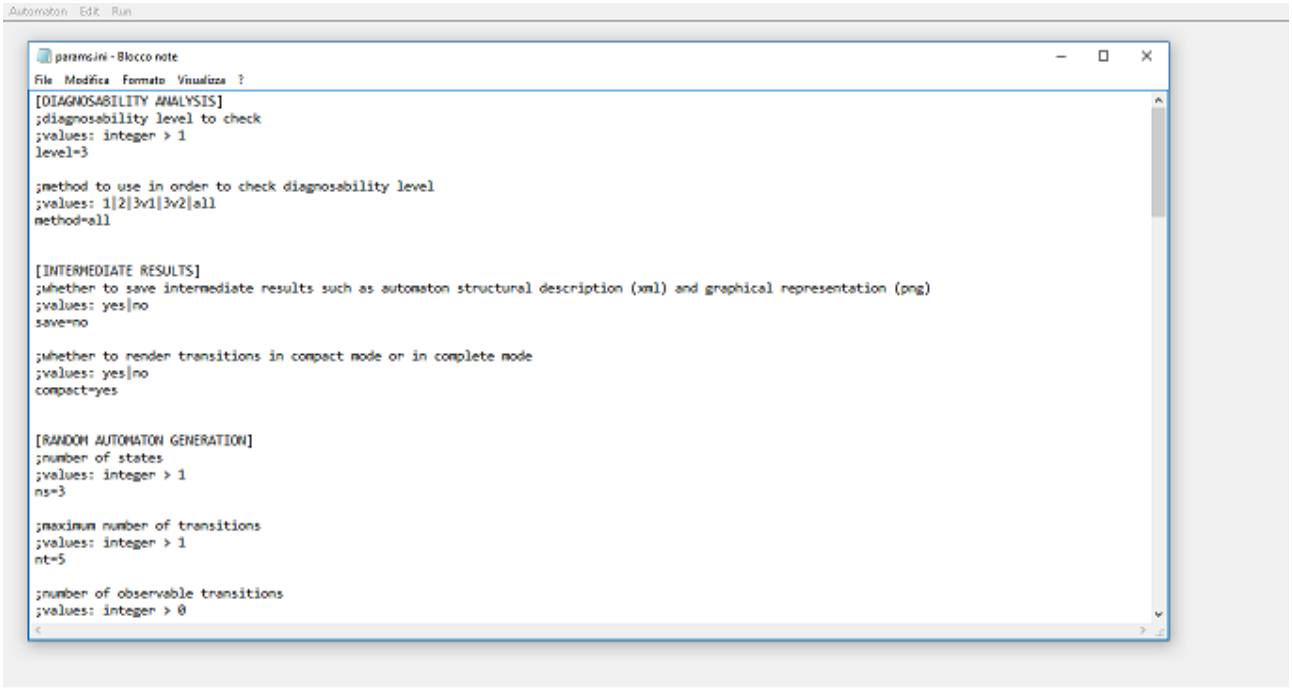

Figura 6.9: Apertura del file di configurazione.

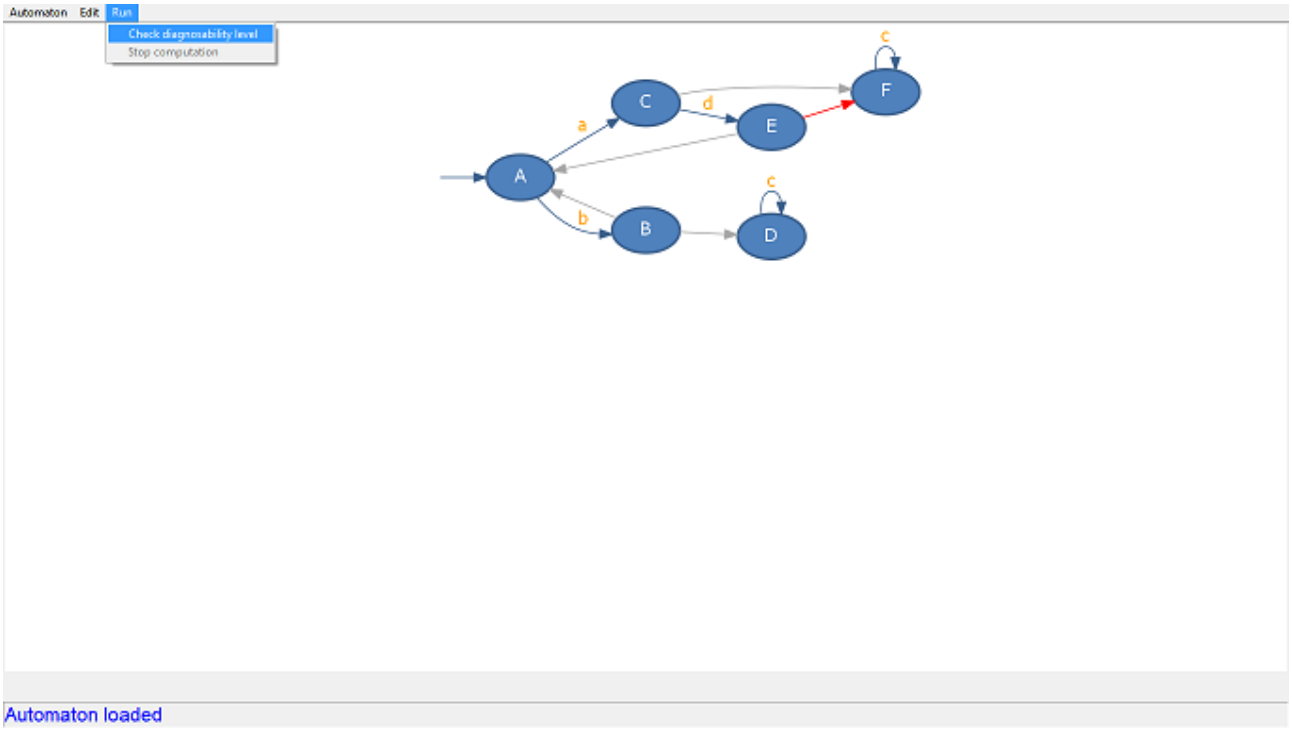

Figura 6.10: Selezione della voce per effettuare il controllo del livello di diagnosticabilità.

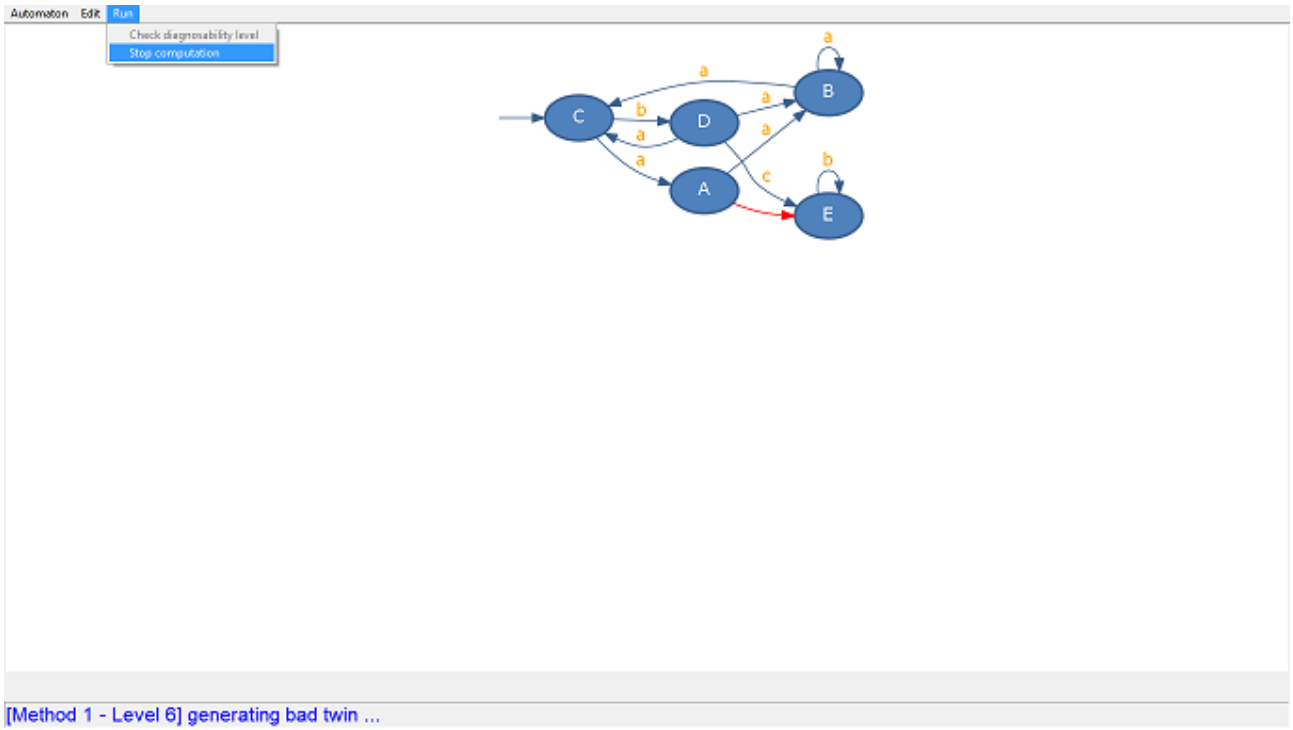

Figura 6.11: Selezione della voce per interrompere il controllo del livello di diagnosticabilità.

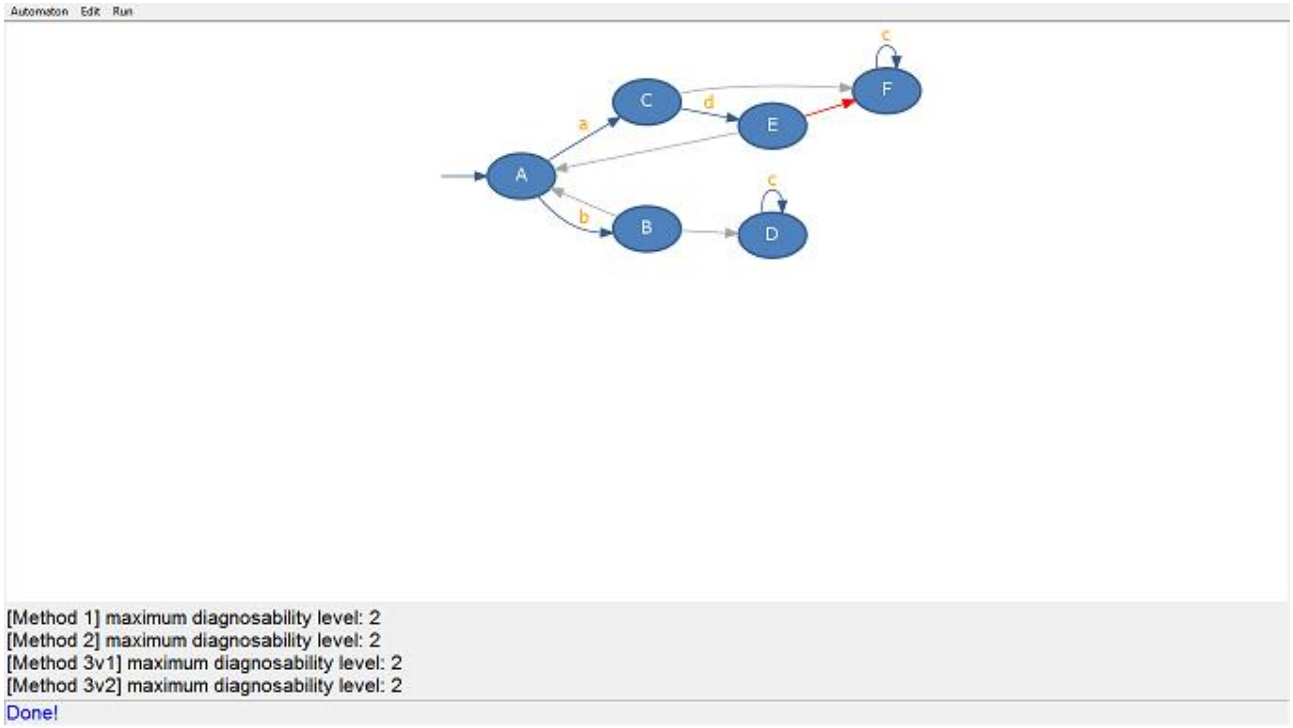

Figura 6.12: Risultati mostrati al termine del controllo del livello di diagnosticabilità.

6.4.5 Salvataggio del report

Una volta prodotti i risultati, è possibile salvare in un report le informazioni raccolte durante l'analisi di diagnosticabilità. Per salvare il report è sufficiente cliccare sulla voce **Save report…** del menu **Automaton**, come mostrato in *Figura 6.13*.

Il report viene salvato all'interno della directory `reports/` come un file con estensione `.log`, avente come nome il timestamp della data corrente.

Riportiamo di seguito per completezza l'estratto di un report:

```
...
######################################################################
[Method 3v1 - Level 1] generating bad twin ...
[Method 3v1 - Level 1] verifying condition C2 ...
[Method 3v1 - Level 1] condition C2 satisfied ...
[Method 3v1 - Level 1] verifying condition C3 ...
[Method 3v1 - Level 1] condition C3 not satisfied ...
[Method 3v1 - Level 1] diagnosability level satisfied ...
----------------------------------------------------------------------
[Method 3v1 - Level 2] generating bad twin ...
[Method 3v1 - Level 2] verifying condition C2 ...
[Method 3v1 - Level 2] condition C2 satisfied ...
[Method 3v1 - Level 2] verifying condition C3 ...
[Method 3v1 - Level 2] condition C3 not satisfied ...
[Method 3v1 - Level 2] diagnosability level satisfied ...
----------------------------------------------------------------------
[Method 3v1 - Level 3] generating bad twin ...
[Method 3v1 - Level 3] verifying condition C2 ...
[Method 3v1 - Level 3] condition C2 not satisfied ...
[Method 3v1 - Level 3] verifying condition C3 ...
[Method 3v1 - Level 3] condition C3 not satisfied ...
[Method 3v1 - Level 3] generating good twin ...
[Method 3v1 - Level 3] synchronizing twins ...
[Method 3v1 - Level 3] verifying condition C1 ...
[Method 3v1 - Level 3] condition C1 not satisfied ...
[Method 3v1 - Level 3] diagnosability level not satisfied ...


[Method 3v1] maximum diagnosability level: 2

...
```

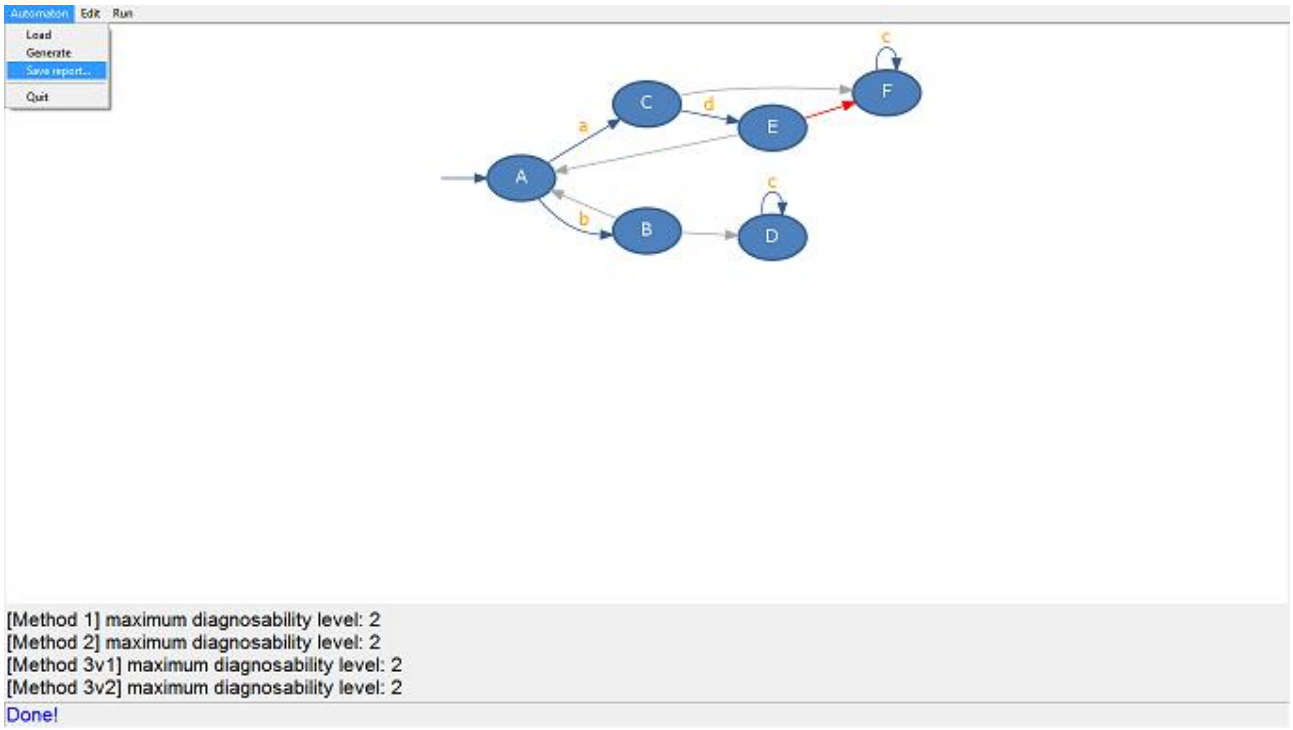

Figura 6.13: Selezione della voce per salvare il report.

CAPITOLO 7 – CURIOSITÀ

7.1 Un dubbio risolto

Durante lo sviluppo del software ci siamo imbattuti in un dubbio circa la possibilità di formazione di transizioni ambigue che non fossero a valle di un ciclo (infinito). Il dubbio ci è sorto leggendo più volte le specifiche, in particolare il vincolo sull'automa di input che impone che il linguaggio generato da A sull'alfabeto degli eventi osservabili semplici Σ_o sia vivo. Ciò che non ci tornava era come fosse possibile la formazione di transizioni ambigue non a valle di un ciclo dato che ogni transizione nell'automa di input fosse a valle di un ciclo, formato ovviamente da almeno una transizione osservabile. Ciò che abbiamo commesso è stato un errore di valutazione: le specifiche affermano infatti che le transizioni ambigue possono formarsi durante la sincronizzazione degli automi, accoppiando due transizioni che sono una di guasto e l'altra no. Se però lo stato destinazione di una transizione sincronizzata deriva dall'accoppiamento di due stati diversi, non è sempre possibile generare una transizione sincronizzata avente come sorgente proprio quello stato. In tal caso esso diventa uno stato "pozzo" e, se una transizione ambigua si forma a valle di tale stato (e di nessun ciclo), si verifica la condizione che smentisce il nostro dubbio.

In *Figura 7.*1 riportiamo un esempio che abbiamo utilizzato per smentire il dubbio.

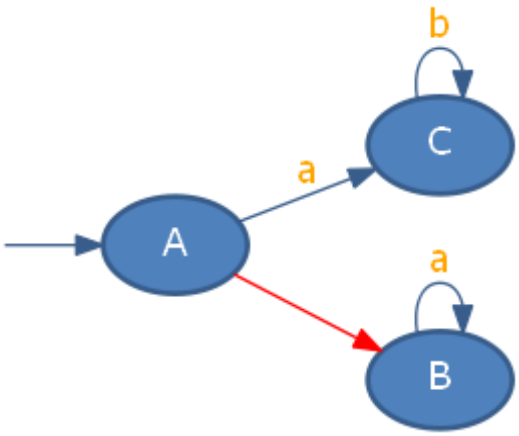

Figura 7.1: Automa utilizzato per smentire il dubbio. In rosso la transizione di guasto.

Tale automa gode di un livello di diagnosticabilità potenzialmente infinito, poiché in nessun livello si forma una transizione ambigua a valle di un ciclo (infinito).

In particolare riportiamo in *Figura 7.2*, *Figura 7.3* rispettivamente il bad twin e il good twin di livello 1. Dalla *Figura 7.4* si può notare come nell'automa sincronizzato si formi lo stato pozzo *B,C* e di conseguenza, poiché l'unica transizione ambigua ha come destinazione proprio lo stato *B,C*, risulta verificata la condizione di diagnosticabilità **C***.

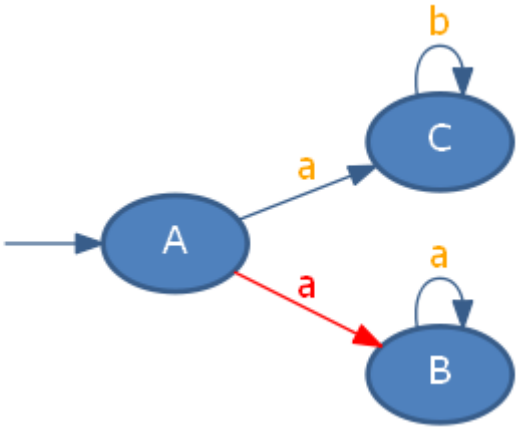

Figura 7.2: Bad twin di livello 1. In rosso la transizione di guasto.

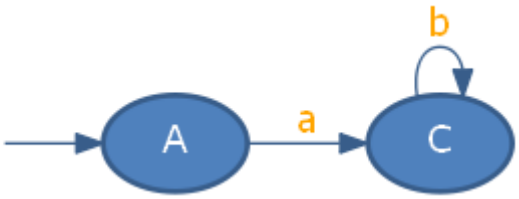

Figura 7.3: Good twin di livello 1.

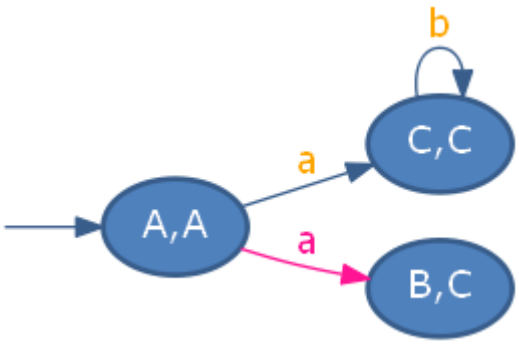

Figura 7.3: Automa sincronizzato di livello 1. In magenta la transizione ambigua.