POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione
(V Facoltà di Ingegneria)

Corso di Laurea Specialistica in Ingegneria Informatica

Reti di Petri Stocastiche Colorate per la modellazione di failure-modes multipli in Sistemi a Componenti

Relatore: Prof.ssa Raffaela Mirandola

Correlatore: Dott. Antonio Filieri

Tesi di Laurea di:

Filippo Spelta Matr. 720439

Anno Accademico 2009-2010

Gli ostacoli non mi fermano; ogni ostacolo conduce a una ferma risoluzione.

(Leonardo da Vinci)

Ai miei genitori e a mio fratello Andrea

Desidero ringraziare il mio Relatore, la Professoressa Raffaela Mirandola, il mio Correlatore e amico dai tempi di Cremona, il Dott. Antonio Filieri e il Dott. Diego Pérez dell'Università di Saragozza per gli spunti, le indicazioni e i suggerimenti che mi hanno fornito durante la stesura della tesi e per la disponibilità dimostratami in ogni circostanza.

INDICE

CAPITOLO 1 Introduzione **1**

CAPITOLO 2 Concetti fondamentali **3**

2.1 Sistemi a componenti 3

2.2 Reti di Petri 6

2.2.1 Reti di Petri stocastiche 7

2.2.2 Reti di Petri stocastiche colorate 11

2.2.2 I) Introduzione e differenze con le reti di petri stocastiche semplici 11

2.2.2 II) Comportamento dinamico delle reti di Petri stocastiche colorate 17

2.2.2 III) Definizione formale 18

CAPITOLO 3 Statement del problema **22**

3.1 Failure modes multipli e propagazione dell'errore 22

3.2 Modellazione dei sistemi a componenti 24

3.2.1 Modello di un singolo componente 24

3.2.2 Modello dell'architettura 26

CAPITOLO 4 Modello risolutivo **28**

4.1 Esempio di un semplice sistema a componenti 28

4.2 Modellazione proposta 31

4.2.1 Modello di un singolo componente 33

4.2.2 Modello dell'architettura 44

4.3 Modellazione di un semplice sistema a componenti 50

4.3.1 Test di ipotesi per valutare l'affidabilità 54

CAPITOLO 5 Misurazioni e analisi di complessità **58**

5.1 Descrizione tool e generazione dei modelli 58

5.1.2 Interfaccia grafica e creazione di modelli elementari 59

5.1.3 XML e creazione di modelli di complessità generica 59
5.2 Simulazioni e setting dei parametri 60
5.3 Misurazioni e analisi di complessità 63

CAPITOLO 6 Conclusioni e sviluppi futuri 74
6.1 Conclusioni 74
6.2 Sviluppi futuri 75

BIBLIOGRAFIA 77

APPENDICE 80

INDICE DELLE FIGURE

Figura 4.1: Architettura del sistema 29
Figura 4.2: Tipo strutturato “Errore” 32
Figura 4.3: Funzione di guardia di una transizione 35
Figura 4.4: Firing weights di una transizione 36
Figura 4.5: Componente con un output e due failure modes 37
Figura 4.6: Posto P1 37
Figura 4.7: Transizione T3 38
Figura 4.8: Transizione T4 38
Figura 4.9: Transizione T5 39
Figura 4.10: Transizione T6 39
Figura 4.11: Componente con un output e tre failure modes 41
Figura 4.12: Componente con due output e tre failure modes 42
Figura 4.13: Primo componente del sistema 44
Figura 4.14: Ultimo componente del sistema 46
Figura 4.15: Transizione T1000000 48
Figura 4.16: Connessioni tra componenti 49
Figura 4.17: Componenti effettivi del sistema 50
Figura 4.18: Primo componente del sistema 52
Figura 4.19: Ultimo componente del sistema 53
Figura 4.20: Modello del sistema completo 53
Figura 4.21: Artificio implementativo 54
Figura 5.1: Parte del file XML di un semplice modello 60
Figura 5.2: Parametri della simulazione 61
Figura 5.3: Struttura dei componenti di TimeNET 62

INDICE DELLE TABELLE

Tabella 3.1: Tipi base 15

Tabella 4.1: Proprietà di affidabilità 30

Tabella 4.2: Dati empirici ottenuti dalla simulazione del sistema 56

Tabella 5.1: Tempi di building e di simulazione dei modelli 64

INDICE DEI GRAFICI

Grafico 5.1: Sistemi con 2 porte di output 66

Grafico 5.2: Sistemi con 3 porte di output 66

Grafico 5.3: Sistemi con 4 porte di output 67

Grafico 5.4: Sistemi con 5 componenti 68

Grafico 5.5: Sistemi con 6 componenti 68

Grafico 5.6: Sistemi con 7 componenti 69

Grafico 5.7: Sistemi con 8 componenti 69

Grafico 5.8: Sistemi con 9 componenti 70

Grafico 5.9: Sistemi con 2 failure modes 72

Grafico 5.10: Sistemi con 3 failure modes 73

Grafico 5.11: Sistemi con 4 failure modes 73

RIASSUNTO

L'affidabilità è un aspetto fondamentale della qualità del software. Considerando il sempre maggior numero di contesti critici (medico, finanziario) in cui sono presenti sistemi software di elevata complessità, sorge la necessità di garantire requisiti di affidabilità per tali sistemi.
Risulta inoltre di fondamentale importanza la possibilità di determinare i requisiti di affidabilità dei sistemi software già dalla loro fase di progettazione in modo che gli sviluppatori siano in grado di compiere fin dall'inizio scelte efficaci.
Considerando l'analisi di affidabilità del software, si può rilevare come le due caratteristiche fondamentali che dovrebbero essere considerate, e sono state finora per lo più trascurate dalle tecniche esistenti, siano la possibilità di avere differenti tipologie di failure e di valutare la propagazione degli errori all'interno dei sistemi software. Infatti, appare troppo semplificato un approccio che preveda di considerare un'unica tipologia di failure e, inoltre, l'attivazione di un fault dà luogo ad un errore, ma si manifesta come failure solamente se si propaga fino all'interfaccia esterna del sistema.
Questo lavoro propone una modellazione in grado di descrivere l'affidabilità di sistemi basati su componenti software considerando la possibilità di avere differenti tipologie di failure e tenendo conto della propagazione degli errori.
La modellazione è realizzata utilizzando le Reti di Petri stocastiche colorate e permette di descrivere accuratamente le modalità secondo cui gli errori si propagano, vengono mascherati o trasformati in errori di una tipologia differente.
La simulazione del sistema permette un'accurata stima delle proprietà di affidabilità.
Il lavoro è completato dall'analisi di complessità dei modelli realizzati e dalla valutazione dell'efficienza del tool nell'eseguire le simulazioni.

CAPITOLO 1

Introduzione

Lo sviluppo software basato su componenti è un'estensione dello sviluppo software convenzionale che prevede di offrire un nuovo approccio al progetto, alla realizzazione e all'implementazione delle applicazioni software. È possibile far coesistere componenti scritti in differenti linguaggi di programmazione ed eseguibili su più piattaforme.
In particolare, considerando componenti software che operano in ambienti critici per la sicurezza, appare di fondamentale importanza il requisito non funzionale dell'**affidabilità**.
Nei sistemi a componenti l'affidabilità deriva dal fatto che il componente fornisca o meno una corretta e completa implementazione della sua specifica. Tale definizione, valida per un componente, non è tuttavia sufficientemente generalizzabile per poter descrivere l'affidabilità dell'intero sistema. Passando dal componente al sistema, infatti, entrano in gioco i legami tra i componenti e le modalità di propagazione di eventuali deviazioni dal servizio corretto denominate failure. L'affidabilità diventa quindi la misura probabilistica della capacità del sistema di eseguire con successo il suo compito.
Questo lavoro nasce dalla necessità di ottenere una tecnica per poter valutare l'affidabilità di sistemi a componenti software che operano in ambienti critici per la sicurezza. In particolare, viene fornita una tecnica per poter valutare l'affidabilità dell'intero sistema a componenti conoscendo le affidabilità dei singoli componenti e il modo in cui sono tra loro connessi.
Per descrivere l'affidabilità di tali sistemi appare troppo riduttiva e irrealistica l'assunzione di avere un comportamento di tipo booleano delle failure. Non è sufficiente limitare la descrizione a situazioni corrette o errate. Si presenta la necessità di avere **tipi differenti di failure**, ad esempio sono possibili content failure causate dalla consegna di un contenuto errato o timing failure causate dalla consegna del contenuto in un tempo non valido. La seconda caratteristica fondamentale per poter considerare nella sua interezza il problema

dell’affidabilità nei sistemi a componenti, è rappresentata dalla possibilità di valutare la **propagazione degli errori** tra componenti. Un errore dà luogo ad una failure solamente se si manifesta all’interfaccia del sistema. Tuttavia è possibile che un errore in un componente venga mascherato nel percorso compiuto all’interno del sistema e non dia luogo a failure. È inoltre possibile che un errore cambi di tipologia nel suo percorso tra i componenti del sistema. Il contributo originale di questo lavoro può essere inquadrato nell’ottica di voler fornire una descrizione di tali aspetti presentando, attraverso il formalismo delle **Reti di Petri stocastiche colorate**, una modellazione in grado di garantire un’accurata analisi di affidabilità e, conseguentemente, di permettere al progettista di software di compiere scelte efficaci fin dall’inizio della progettazione.

Il documento è organizzato secondo la seguente struttura:

- Capitolo 2: viene fornita una descrizione del background necessario allo sviluppo successivo del lavoro. Nella prima sezione si introducono i sistemi a componenti presentandone le peculiarità e i vantaggi, nella seconda si descrivono le Reti di Petri stocastiche colorate che verranno in seguito utilizzate.
- Capitolo 3: vengono precisati i termini del problema presentando la descrizione accurata di un sistema a componenti software.
- Capitolo 4: viene inizialmente presentato un esempio di un semplice sistema a componenti. Nella seconda sezione si descrive accuratamente la modellazione realizzata e, nella terza, si applica tale modellazione all’esempio illustrato nella prima sezione, mostrando il soddisfacimento delle proprietà di affidabilità.
- Capitolo 5: viene descritto il tool utilizzato per la modellazione e le simulazioni, TimeNET, successivamente, per valutarne l’efficienza, si mostrano i risultati delle simulazioni, l’ultima sezione comprende un’analisi di complessità del modello proposto.
- Capitolo 6: vengono riassunti gli obiettivi raggiunti mostrando possibili sviluppi futuri.

CAPITOLO 2

Concetti fondamentali

Il presente capitolo introduce gli ambiti entro cui si sviluppa il lavoro realizzato. La prima sezione è dedicata alla presentazione dei principi fondamentali dell'ingegneria del software basata su componenti e, più in dettaglio, dei sistemi a componenti.

2.1 Sistemi a componenti

L'ingegneria del software basata su **componenti** è un ramo dell'ingegneria del software che ha l'obiettivo [4] di offrire un **nuovo approccio** al progetto, alla realizzazione, all'implementazione ed all'evoluzione delle applicazioni software. Secondo questa prospettiva, le applicazioni software sono assemblate a partire da molteplici fonti, i componenti stessi possono essere scritti in differenti linguaggi di programmazione ed essere eseguibili su più piattaforme. Si può inquadrare lo sviluppo software basato su componenti come **un'estensione dello sviluppo software convenzionale**, sfruttando le possibilità offerte dai componenti per soddisfare alcuni requisiti e mantenendo l'utilizzo di tecniche convenzionali per soddisfare altri requisiti. In questa prospettiva si può considerare lo sviluppo software convenzionale come caso particolare dello sviluppo basato su componenti, in quanto il primo manca di alcune tecniche e opportunità (e conseguentemente vantaggi) che caratterizzano il secondo. Si può quindi spaziare in un ampio range di possibilità, dallo sviluppo puramente convenzionale ad un'estrema frammentazione in componenti, una delle questioni chiave per il progettista è proprio quella di riuscire a determinare a che profondità è ragionevole suddividere in componenti a seconda della particolare situazione considerata. Volendo dare una definizione di componente software si può fare riferimento ad esso come ad un qualcosa che è possibile

rappresentare a **scatola nera**, con un'**interfaccia esterna** che è indipendente dai suoi **meccanismi interni**. In generale gli aspetti rilevanti comuni a tutte le possibili accezioni del concetto di componente software tendono ad enfatizzare la presenza di un "interno", di un "esterno" e di una "relazione" tra essi. Si ha inoltre la presenza di un "contesto" implicito per la relazione. L'interno è uno strato software che soddisfa proprietà, è un dispositivo che può essere gestito per ottenere riusabilità. L'esterno è un'interfaccia che soddisfa determinate proprietà, ha il ruolo di fornire un servizio ad utilizzatori umani o ad altri strati software. La relazione tra interno ed esterno è descritta facendo riferimento ai concetti di specifica, implementazione ed incapsulamento e non viene qui ulteriormente approfondita. Il contesto per la relazione stabilisce come il software deve essere gestito e utilizzato, all'interno di un ben definito processo per lo sviluppo e il mantenimento del software.

Sono possibili anche altre definizioni di componente software a seconda dell'aspetto che si vuole enfatizzare. Ad esempio una definizione che procede dall'esterno verso l'interno si concentra sull'interfaccia di servizio esterna e classifica il componente come un'interfaccia con una (o più) implementazioni software. Tuttavia non sempre si è in grado di inferire dalla specifica dell'interfaccia esterna se l'implementazione interna coinvolge realmente uno strato software, si può pensare al semplice esempio di un servizio di controllo del credito che potrebbe essere effettuato da un operatore umano o da un software in automatico senza che il cliente esterno si renda conto della differenza. Un'altra definizione che compie un percorso opposto e procede dall'interno all'esterno, delinea il componente come uno strato software che fornisce servizi attraverso una (o più) interfacce. Anche questa prospettiva tuttavia ha la lacuna di non garantire univocità nel legame tra software interno e servizio offerto, e in particolare uno stesso strato software può essere in grado di fornire la stessa funzionalità a due o più differenti livelli di servizio (tipicamente a prezzi differenti). Si può pensare al cliente che paga pochi centesimi per avere una previsione meteorologica non dettagliata, e alla fornitura di una previsione accurata in cambio di una somma molto maggiore, ma ottenibile utilizzando lo stesso software.

Alcuni esperti evitano queste ambiguità producendo metamodelli complicati nei quali la nozione originale di componente software è decomposta in concetti separati, quali ad esempio: interfaccia di servizio, specifica dell'interfaccia, progetto di unità software, eseguibile software. Questi metamodelli sono fondamentali per gli sviluppatori di tool dedicati ma possono creare confusione in quanto il concetto di componente software scompare originando frammenti concettuali separati.

Volendo confrontare i componenti con gli **oggetti** e fare un paragone tra essi, si nota che, di fatto, i componenti ereditano molte delle caratteristiche degli oggetti nel paradigma orientato agli oggetti. La nozione di **componente**, però, va ad un **maggiore livello di profondità** della nozione di oggetto. Infatti il riuso di oggetti, solitamente, prevede il riuso di librerie in un particolare linguaggio di programmazione orientato agli oggetti. Si deve quindi essere sufficientemente pratici, ad esempio, di Java per poter essere in grado di riusare una classe Java. Nel paradigma a componenti invece è possibile riusare un componente senza sapere quale linguaggio di programmazione o piattaforma utilizza internamente. La stessa specifica può essere implementata in modi differenti. Inoltre, così come la specifica è una descrizione del comportamento di un componente, e il comportamento può essere descritto in modi differenti, si ha che uno stesso componente può soddisfare specifiche differenti.
Si conclude la sezione mostrando come i sistemi a componenti possono migliorare le applicazioni software. La normativa ISO 9126 fornisce una definizione generica di qualità del software sintetizzandola in sei caratteristiche fondamentali. Si mostra come i sistemi a componenti contribuiscono ad ognuna di esse:

- **Funzionalità**: l'utilizzo di componenti preesistenti permette una disponibilità più rapida di una migliore funzionalità.
- **Manutenibilità**: la struttura modulare di una soluzione basata su componenti permette di rimpiazzare facilmente i singoli componenti.
- **Usabilità**: l'utilizzo di componenti standard supporta la comunanza delle interfacce grafiche. Lo sviluppo basato su componenti inoltre supporta l'integrazione desktop che è in grado di dare all'utente una visione unitaria di dati eterogenei.
- **Efficienza**: i colli di bottiglia nelle prestazioni possono essere identificati e quindi, solitamente, la necessità di ottimizzare le prestazioni può essere localizzata in un piccolo numero di componenti critici. I componenti possono essere ottimizzati internamente per migliorare le prestazioni, senza influenzare la loro specifica, o analogamente, i componenti possono essere spostati tra piattaforme per migliorare le prestazioni, senza influenzare la funzionalità o l'usabilità dell'applicazione.
- **Affidabilità**: data una specifica formale e completa di un componente, l'affidabilità del componente deriva dal fatto che esso fornisca o meno una corretta e completa implementazione della sua specifica. L'affidabilità dell'intera applicazione è una questione

separata, ma è chiaramente rafforzata se l'applicazione è costituita da componenti affidabili.

- **Portabilità**: la specifica di un componente è indipendente dalla piattaforma. Un componente, quindi, può essere rapidamente rigenerato per una nuova piattaforma, senza influenzare nessun altro componente.

La sezione successiva descrive il formalismo matematico adottato per modellare il problema in esame: le reti di Petri stocastiche colorate. Vengono inizialmente accennate le reti di Petri tradizionali, se ne mostra poi la loro evoluzione introducendo le reti di Petri stocastiche, mentre l'ultima parte della sezione è dedicata alla definizione e all'approfondimento delle reti di Petri stocastiche colorate.

2.2 Reti di Petri

Le reti di Petri sono state introdotte per la prima volta da Carl Adam Petri nel 1962 e da allora ne sono state definite molteplici varianti in grado di adattarsi a specifici obiettivi modellistici.
Diffusamente utilizzate per rappresentare matematicamente sistemi a eventi discreti [1], le reti di Petri tradizionali sono grafi bipartiti diretti composti da posti, transizioni e archi che connettono i primi alle seconde.
L'obiettivo di questa sezione non è quello di fornire un'analisi esaustiva e formale delle diverse versioni di reti di Petri, piuttosto si vuole dare un'efficace descrizione delle basi di questo potente strumento modellistico, lasciando alle successive sezioni il compito di definire la particolare tipologia di rete utilizzata delineandone progressivamente gli opportuni dettagli.
Le reti di Petri sono particolarmente indicate per sistemi che prevedono attività concorrenti, sincronizzate, conflittuali o nondeterministiche, inoltre permettono la descrizione modulare di sistemi complessi considerando solamente stati locali e cambiamenti tra gli stati. Va inoltre ricordato come le basi matematiche su cui poggia il formalismo, permettano sia un'analisi qualitativa basata sulle equazioni di stato o sul grafo di raggiungibilità, sia una valutazione quantitativa basata sul grafo di raggiungibilità o realizzata tramite simulazione.
Elementi caratterizzanti le reti di Petri sono i **posti** (raffigurati da cerchi), le **transizioni** (raffigurate da sbarre) e gli **archi** diretti per realizzare le connessioni, è possibile

connettere posti a transizioni ma non sono permesse connessioni tra posti e posti o tra transizioni e transizioni. I posti possono contenere **token** e assegnare un certo numero di token ad un determinato posto assume, a seconda del contesto, un preciso significato definendo di fatto lo stato del modello (denominato **marcatura**). Invece le transizioni, solitamente, modellano attività quali ad esempio cambiamenti di stato o eventi.
Considerando un sistema ad eventi discreti appare naturale come certi eventi possano accadere in certi stati del sistema, quando ciò si verifica nel nostro formalismo si dirà che la transizione è **abilitata** nella marcatura considerata, in particolare quindi avremo un numero di token sufficiente nei posti di input alla transizione. In questa situazione può accadere che gli eventi si svolgano in modo atomico e quindi la transizione viene eseguita e si ottiene un cambiamento dello stato del sistema.
L'abilitazione delle transizioni, il firing e il relativo cambio di marcatura sono definiti da opportune **regole di abilitazione** e **regole di firing** proprie della classe di rete di Petri effettivamente utilizzata. Se il sistema prevede attività che impieghino una certa quantità di tempo, è possibile tenere conto di tale aspetto utilizzando il **firing delay** associato alle transizioni. Tale ritardo può essere stocastico (una variabile casuale) e quindi descritto da una distribuzione di probabilità, e può essere visto come il tempo che deve trascorrere tra l'abilitazione della transizione e il conseguente firing.
Le reti di Petri standard ora considerate non prevedono l'impiego di tipologie differenti di token, si ha un unico tipo di token ed essi sono raffigurati mediante pallini neri o tramite un numero che indichi la loro numerosità nel posto associato. Nella presente esposizione si utilizza la scala dei tempi continua.

2.2.1 Reti di Petri stocastiche

Si fornisce ora una definizione formale delle reti di Petri stocastiche per poi andare a descrivere dettagliatamente i vari elementi che caratterizzano tale variante.
Una rete di Petri stocastica è formalmente definita [1] da una tupla:

$SPN = (P, T, \Pi, Pre, Post, Inh, \Lambda, W, Deg, m_0, RV)$

- P è l'insieme dei posti, i quali possono contenere token.

- La marcatura m della rete di Petri associa un numero (intero non negativo) di token al posto corrispondente:

$m: P \rightarrow N$

La marcatura m può anche essere vista come un vettore di numeri naturali con la dimensione del numero di posti:

$m \in N^{|P|} = (m(p_1), m(p_2),\ldots, m(p_{|P|}))$

M indica l'insieme di tutte le marcature teoricamente possibili di una rete di Petri:

$M = \{m/ \forall p \in P: m(p) \in N\}$

- T indica l'insieme delle transizioni che contiene l'insieme delle transizioni temporizzate T^{tim} e delle transizioni immediate T^{im}.

Naturalmente un nodo di una rete di Petri può essere o un posto o una transizione e la rete non dovrebbe mai essere vuota e quindi devono valere le relazioni:

$T \cap P = \varnothing$

$T \cup P \neq \varnothing$

- Π rappresenta la funzione di priorità e associa ad ogni transizione un numero naturale:

$\Pi: T \rightarrow N$

Numeri più alti indicano una priorità maggiore, e solamente le transizioni immediate possono avere una priorità più grande di zero. La priorità, quindi, definisce implicitamente il tipo di transizione:

$T^{tim} = \{t \in T \setminus \Pi(t) = 0\}$

$T^{im} = \{t \in T \setminus \Pi(t) > 0\}$

Nella rappresentazione grafica le transizioni possono essere etichettate con la loro priorità, ma solitamente lo si evita per le transizioni che hanno una priorità di default come tutte le transizioni temporizzate e le transizioni immediate con priorità pari a 1.

- Pre descrive la molteplicità degli archi di input che connettono posti a transizioni. Il caso più generale è quello di una molteplicità dell'arco di input dipendente dalla marcatura, quindi Pre viene definito come una funzione che mappa ogni coppia posto-transizione unita ad un vettore di marcatura su un numero naturale rappresentante la cardinalità dell'arco (il numero di token).

$\text{Pre}: P \times T \times N^{|P|} \rightarrow N$

Nel semplice caso di cardinalità indipendente dalla marcatura si scrive:

$\text{Pre}(p_i, t_j, .) \in N$

e se non ci sono archi di input che connettono il posto p_i alla transizione t_j allora si scrive:

$\text{Pre}(p_i, t_j, .) = 0$

- Post indica la molteplicità degli archi di output che connettono transizioni a posti. La definizione è analoga agli archi di input:

$\text{Post}: P \times T \times N^{|P|} \rightarrow N$

- Inh descrive la molteplicità degli **archi inibitori**. Questo tipo di archi ha la particolarità di connettere solamente posti a transizioni e mai transizioni a posti ed è utilizzato per inibire l'abilitazione di una transizione. In particolare la transizione non può essere abilitata se nel posto che le è connesso tramite un arco inibitore è presente un numero di token almeno pari alla molteplicità dell'arco inibitore (1 nel caso di default). Gli archi inibitori sono rappresentati graficamente come gli archi tradizionali e per distinguerli da

quest'ultimi si aggiunge un pallino vicino alla sbarra raffigurante la transizione cui sono connessi.
La definizione di Inh è analoga alla definizione di Pre, la presenza del valore 0 indica che non è presente un arco inibitore per la corrispondente coppia posto-transizione:

$Inh: P \times T \times N^{|P|} \rightarrow N$

- Il ritardo Λ di una transizione specifica il tempo per il quale la transizione stessa deve restare abilitata prima di poter essere eseguita. Il ritardo è definito da una funzione di distribuzione di probabilità che permette di descrivere ritardi casuali.

$\Lambda: T \rightarrow F^{+}$

- W associa ad ogni transizione immediata un numero reale. Questo valore rappresenta il **"firing weight"** delle transizioni immediate.

$W: T^{im} \rightarrow R$

Tali pesi saranno diffusamente utilizzati nella successiva modellazione per rappresentare le probabilità relative di firing di transizioni concorrenti.
Nella rappresentazione grafica i firing weights per le transizioni immediate possono essere scritti vicino alla transizione. Il valore di default è 1.

- Deg descrive il grado di concorrenza per ogni transizione:

$Deg: T \rightarrow \{SS, IS\}$

SS indica "single server" e IS "infinite server".

- m_0 indica la marcatura iniziale del modello. Siccome m_0 è una marcatura ha la forma:

$m_0: P \rightarrow N$

- RV specifica l'insieme delle variabili di rinforzo del modello di rete di Petri stocastica.

2.2.2 Reti di Petri stocastiche colorate

Le reti di Petri stocastiche colorate [1] sono particolarmente utili per descrivere sistemi a eventi discreti stocastici complessi e possono essere viste come un'estensione delle reti di petri stocastiche appena introdotte. Solitamente, nelle reti di petri tradizionali, i posti e le transizioni corrispondono rispettivamente a buffer e ad attività o a entità simili, gli oggetti che sono creati, modificati e movimentati attraverso il sistema sono descritti dai token nei posti. Le applicazioni in cui questi oggetti presentano attributi significativi hanno portato all'introduzione di reti con **token distinguibili** (**colorati**). Gli attributi dei token devono essere strutturati e specificati mediante l'appartenenza a un colore (o tipo).
L'effettiva esecuzione delle transizioni, in questa tipologia di rete, può dipendere dai valori degli attributi dei token e a sua volta può modificare tali valori.
Una transizione può avere differenti modi di abilitazione e di firing dipendenti dai suoi token di input. La tipologia di rete di Petri ora considerata usa le variabili di arco per descrivere queste possibilità.

2.2.2 I) Introduzione e differenze con le reti di petri stocastiche semplici

La principale differenza tra le reti di Petri tradizionali e la versione colorata è che in quest'ultima i token possono avere attributi definiti arbitrariamente. Diventa quindi possibile identificare token diversi contrariamente a quanto accadeva con le reti di Petri semplici nelle quali i token erano tra loro identici. Tale semplice estensione comporterà un incremento della complessità di posti, transizioni e archi.

Tipi o colori

I token appartengono ad uno specifico **tipo** o **colore**, che definisce l'intervallo di valori che gli attributi possono assumere e le operazioni applicabili allo stesso modo di un tipo di variabile in un qualsiasi linguaggio di programmazione. I termini colore e tipo sono sinonimi.
I tipi possono essere tipi base o tipi strutturati, gli ultimi sono definiti dall'utente. Nel tool TimeNET [2] (utilizzato per la modellazione e la simulazione) i tipi base disponibili sono i seguenti: Tipo vuoto, Integer (nome: int), Real (real), Boolean (bool), String (string), Data

e ora (DateTime). Il Tipo vuoto è simile al "tipo" dei token nelle reti di Petri semplici. I token di questo tipo non possono essere distinti, sono raffigurati come punti neri e non possiedono nessun attributo. Integer e Real sono tipi numerici i cui valori possono essere comparati e usati in espressioni aritmetiche e sono simili ai tipi int e double dei linguaggi di programmazione. I valori booleani possono essere comparati, negati e usati in espressioni contenenti AND e OR logici. Le stringhe rappresentano array di caratteri, le costanti di tipo String sono racchiuse tra virgolette. I tempi possono essere memorizzati in attributi di tipo DateTime. Il formato in cui viene rappresentato il tempo è ore:minuti:secondi e la data è rappresentata dal formato mese/giorno/anno, separata dal simbolo @. I valori temporali possono essere sottratti e il risultato è espresso in secondi, l'addizione e la sottrazione tra interi funzionano in maniera analoga. Il tempo corrente è indicato da NOW.
I tipi strutturati sono definiti dall'utente e possono contenere un qualsiasi numero di tipi base e di altri tipi strutturati esattamente come una struct nel linguaggio C, naturalmente non sono permesse definizioni circolari. La notazione di accesso ai tipi strutturati così come la specifica di oggetti strutturati prevede l'utilizzo di parentesi graffe. Le parentesi racchiudono una lista di valori di attributi uniti ai nomi degli attributi per definire il valore di un token complesso. Ad esempio un token di tipo Prodotto può essere specificato da {nome = "leva", passo = 2} se prima era stata data la coerente definizione di tipo che mostrasse il tipo strutturato Product come composto dagli elementi nome e passo di tipo rispettivamente string e int. I valori di default nella creazione dei token e gli attributi che non cambiano nelle operazioni tra token possono essere omessi. Il tipo vuoto non può essere utilizzato come parte di tipi strutturati.
La sola operazione permessa su un tipo strutturato è il confronto. Due oggetti di un tipo strutturato sono uguali se tutti i loro attributi sono uguali. Benché un attributo di un tipo strutturato possa essere internamente implementato come un puntatore, non ci sono riferimenti accessibili al livello del modello. I token possono solo essere copiati e non è possibile avere differenti riferimenti allo stesso token o attributo.

Posti

I posti sono simili a quelli nelle reti di Petri semplici e sono rappresentati mediante cerchi e servono come contenitori di token. Essi rappresentano elementi passivi del modello e il loro contenuto corrisponde allo stato locale del modello stesso. Siccome i token hanno un

tipo, in una rete di Petri colorata, è utile restringere il tipo di token che possono esistere in un posto ad un unico tipo, che quindi diviene anche **il tipo o il colore del posto**. Questo tipo può essere sia un tipo base predefinito sia un tipo strutturato definito nel modello. In entrambi i casi, nelle figure, è mostrato in corsivo vicino al posto, il tipo vuoto è omesso.
Il nome unico di un posto è scritto vicino al posto stesso.
La marcatura iniziale di un posto è una serie di token del tipo corretto e descrive il contenuto del posto all'inizio di una valutazione.
Possono esserci token differenti con attributi uguali che li rendono simili, ma non identici.
La marcatura è un multinsieme di token, per convenzione si mostra solamente il numero dei token iniziali mentre il numero corrente è mostrato altrove, se richiesto. Un'utile estensione importante per molte applicazioni reali è la specifica della **capacità** di un posto.
La capacità indica il numero massimo di token che possono esistere in un posto ed è rappresentata mediante l'utilizzo di parentesi quadre vicino al posto, viene omessa se la capacità è illimitata (il default).

Archi e iscrizioni sugli archi

I posti e le transizioni sono connessi da archi diretti come in qualsiasi altro tipo di rete di Petri. Un arco che va da un posto ad una transizione è detto arco di input della transizione e il posto connesso è chiamato posto di input (e analogamente per gli archi e i posti di output). Contrariamente alle reti di Petri semplici, nelle quali un numero è l'unico attributo di un arco, in questa tipologia di reti di Petri è possibile specificare quale tipo di token è influenzato e quali operazioni sugli attributi dei token sono effettuate quando una transizione viene eseguita. Tutto questo è realizzato mediante le **iscrizioni sugli archi**.
Nella rappresentazione grafica le iscrizioni sugli archi sono racchiuse in parentesi angolari (<,>).
Gli archi di input delle transizioni e le relative iscrizioni descrivono quanti token sono rimossi durante il firing di una transizione e assegnano un nome (mediante le variabili) ai token stessi mediante il quale essi saranno identificati nelle espressioni di guardia e negli archi di output. Il nome delle variabili è racchiuso tra parentesi e può essere presente anche un numero intero ad indicare il numero di token da rimuovere, il valore di default 1 è omesso. Durante il fiirng la variabile assume il valore corrente del token, mentre quest'ultimo viene rimosso dal posto corrispondente. Per evitare ambiguità ogni identificatore di variabile di input può essere usato solamente in un arco di input di ogni

transizione, non è quindi possibile avere due differenti archi di input di una stessa transizione che presentino la stessa variabile nelle corrispondenti iscrizioni.

Gli archi di output delle transizioni definiscono quali token vengono aggiunti ai relativi posti connessi al momento del firing della transizione. Ci sono due possibilità per descrivere questa operazione: i token possono essere "trasferiti" oppure "creati". Nel caso di trasferimento di token, il nome della variabile associata al token di input scelto è utilizzato nell'arco di output. Il token che era legato a questa variabile viene quindi spostato nel corrispondente posto di output. La molteplicità dei token deve essere la stessa per evitare ambiguità e cioè se, ad esempio, vengono rimossi tre token da un posto di input non è poi possibile spostarne solamente due in un posto di output ed eliminare il terzo ma dovranno essere spostati tutti e tre nel posto di output. Analogamente, considerando un'unica transizione, non è possibile utilizzare la stessa variabile di input in differenti posti di output. I token associati a variabili di input che non vengono utilizzate negli archi di output sono distrutti alla fine del firing della transizione.

Se invece si vuole avere la creazione di nuovi token in un posto, è sufficiente non specificare nessuna variabile di input nel relativo arco di output. In questo modo verranno creati token del tipo associato al posto di output. Se si vuole creare più di un token nell'iscrizione sull'arco di output bisogna indicare il numero di token seguito dal simbolo #. Gli attributi dei token così creati saranno settati ai loro valori di default secondo quanto specificato dalla Tabella 3.1.

Sia nel caso di token trasferiti sia nel caso di token creati, è possibile settare i valori degli attributi di un token (o il suo valore nel caso il token sia di un tipo predefinito) ad un valore costante o ad un valore che dipende da altri token di input. Nel caso di token trasferiti la sintassi prevede di indicare, nell'iscrizione dell'arco di output, la variabile seguita dall'attributo che si vuole modificare e dal valore tra virgolette separati dal simbolo = e racchiusi tra parentesi tonde, nella successiva descrizione della fase di modellazione sarà mostrato un esempio concreto dell'utilizzo di tale procedura. Il tipo delle variabili contenute nelle iscrizioni di input e di output è dato dal tipo del corrispondente posto connesso e quindi non deve essere definito esplicitamente. È possibile introdurre restrizioni sui token di input mediante l'utilizzo di guardie come sarà mostrato nella sezione successiva. Per convenzione tutte le variabili presenti nelle iscrizioni sugli archi di una transizione t, sono indicate come "variabili di transizione" di t.

Tipo	Nome	Valore di default
Tipo vuoto	-	-
Integer	int	0
Real	real	0.0
Boolean	bool	false
String	string	“”
Data e ora	DateTime	0:0:0@1/1/0

Tabella 3.1: Tipi base

Transizioni

Le transizioni (raffigurate come rettangoli di differenti forme) modellano le attività del sistema. Possono essere attivate (abilitate) quando tutti i token di input necessari sono disponibili e una funzione di guardia (opzionale) è vera. Il loro firing modella l'occorrenza di un'attività e cambia la marcatura dei posti (lo stato del sistema). Ci sono diversi tipi di transizioni accompagnate dalle corrispondenti rappresentazioni: le **transizioni immediate**, che vengono eseguite senza ritardo, sono raffigurate come sottili rettangoli, le **transizioni temporizzate** sono più grandi e vuote, mentre le transizioni di sostituzione (utilizzate per definire modelli gerarchici) hanno rettangoli neri nella parte inferiore e superiore. Le transizioni hanno molteplici attributi, il nome è una stringa che identifica univocamente la transizione nel modello, il firing delay, proprio delle transizioni temporizzate, descrive la distribuzione di probabilità dell'intervallo di tempo che deve trascorrere tra l'abilitazione e il firing della transizione, il **firing weight**, proprio delle transizioni immediate, è un valore reale che permette di definire la probabilità relativa di firing e, la priorità, anch'essa propria delle transizioni immediate è un intero maggiore di 0 che permette di assegnare precedenze nella scelta tra transizioni concorrenti, sia il firing weight sia la priorità hanno come valore di default 1. Le transizioni in una rete di Petri colorata hanno differenti firing modes a seconda degli attributi dei token che rimuovono dai loro posti di input. In ogni stato di una rete di Petri stocastica colorata tutti i possibili assegnamenti dei token di input alle rispettivi variabili di arco (binding) possono essere firing modes validi. Una **funzione di guardia** può ridurre i token per i quali una transizione può essere abilitata. La guardia è una funzione booleana che può dipendere dallo stato del modello e dalle variabili sugli archi di input. La transizione è abilitata solamente, con un certo binding tra token e

variabili in uno specifico stato del modello, se la funzione di guardia è valutata Vera per quella configurazione.
Le espressioni nelle funzioni di guardia spesso contengono il confronto di attributi dei token di input con costanti o con altri attributi dei token di input. Gli operatori e la sintassi saranno presentati nella sessione successiva. La funzione di guardia di default è vuota ed è valutata Vera per definizione. La **semantica del server** specifica se una transizione può essere concorrentemente abilitata con se stessa oppure no, può avere il valore (di default) "single server" o "infinite server". La semantica di tipo "single server" modella il naturale comportamento di una risorsa ristretta che può eseguire una sola azione alla volta (su una parte/token), quindi la transizione può essere abilitata con un solo binding alla volta. La semantica "infinite server" invece modella il caso di un numero arbitrario di risorse e tutti i binding che possono essere eseguiti insieme sono abilitati concorrentemente, quindi un token in arrivo in un posto di input di una transizione single server è ignorato mentre nel caso di transizione infinite server viene generata una nuova abilitazione della transizione.

Sintassi delle espressioni

Le espressioni con operatori sono utilizzabili nelle iscrizioni ovunque è permesso utilizzare variabili e costanti se non diversamente specificato, ad esempio le iscrizioni degli archi di input non possono avere associate espressioni con operatori. Il risultato di un'espressione deve sempre corrispondere al tipo richiesto da un posto. Il numero corrente di token nel posto p è indicato da #p, è possibile identificare i posti nelle espressioni facendo riferimento al nome unico che viene assegnato ad ogni elemento del modello. I valori degli attributi dei token di input possono essere utilizzati nelle espressioni sugli archi di output così come nelle funzioni di guardia. Il nome di una variabile su un arco di input identifica il token che è associato alla variabile in un particolare stato del modello. Gli attributi dei token strutturati sono accessibili con l'abituale notazione puntata.

Misure di prestazioni

Per la valutazione quantitativa dei modelli basati su reti di Petri stocastiche colorate sono sostanzialmente disponibili due classi di misure di prestazioni: la prima basata sul conteggio del numero dei token in un posto, la seconda basata sul conteggio del numero di volte che una transizione viene eseguita. Il numero di token nel posto P è misurato

specificando l'espressione #P e analogamente il numero di volte che la transizione T viene eseguita è misurato attraverso l'espressione #T. Entrambe le misure possono essere filtrate specificando condizioni sugli attributi analogamente alle funzioni di guardia. Nel tool TimeNET ciascuna misura ha un nome, un'espressione, un tipo e un valore calcolato. Il tipo può essere scelto tra istantaneo, cumulativo e medio a seconda delle esigenze modellistiche e le espressioni possono contenere costanti numeriche e operazioni.

2.2.2 II) Comportamento dinamico delle reti di Petri stocastiche colorate

Ogni posto contiene un insieme di token del corrispondente tipo: la marcatura del posto. Le marcature di ogni posto insieme definiscono lo stato del sistema modellato. Le transizioni possono essere abilitate a seconda dello stato corrente del modello ed essere eseguite cambiando di conseguenza lo stato. Lo stato iniziale del modello è indicato dalla marcatura iniziale m_0. Le transizioni nelle reti di Petri colorate hanno differenti modi di attivazione e firing, che corrispondono ai possibili binding tra i token nei posti di input e le variabili sugli archi di input. Quindi le transizioni possono essere abilitate ed eseguite solo a condizione che si realizzi uno specifico binding. Dire che una transizione è abilitata è solamente un'abbreviazione per indicare che almeno uno dei suoi binding è abilitato. Per ogni transizione devono essere valutati tutti i possibili binding per verificare quali sono abilitati. Un binding è abilitato se la funzione di guardia della transizione è valutata vera e se c'è abbastanza capacità libera nei posti di output della transizione per i token che verranno creati in essi. Se la semantica della transizione è single server, come nella modellazione proposta in seguito, allora non è permessa concorrenza e solamente uno tra i binding abilitati viene effettivamente scelto in modo probabilistico e la transizione può quindi essere eseguita.

2.2.2 III) Definizione formale

Una rete di Petri stocastica colorata è formalmente definita [1] dalla tupla:

$SCPN = (P, T, \tau, C, Cap, Pre, Post, G, \Pi, W, \Lambda, Deg, m_0, RV)$

La definizione delle reti di Petri stocastiche colorate si appoggia su quella delle reti di Petri stocastiche essendo le prime un'estensione delle seconde e quindi, in questa sezione, non si andranno a ridefinire gli elementi già illustrati per le reti di Petri stocastiche (per i quali si può fare riferimento alla Sezione 2.2.1), invece vengono ora descritti i nuovi elementi tipici delle reti colorate.

- τ indica l'insieme di tutti i tipi (o colori) che è permesso utilizzare in una rete di Petri stocastica colorata. Var indica l'insieme di variabili sull'insieme di tipi τ, Var_τ indica l'insieme di variabili su un singolo tipo τ e $Expr_{Var}$ indica l'insieme di tutte le espressioni costruite da variabili presenti in Var. L'insieme delle variabili contenute in un'espressione e è indicato da Var(e), mentre il tipo di una variabile o del risultato di un'espressione è indicato da C, e si ha che:

$\forall v \in Var: C(v) \in \tau$

- C è la funzione di dominio del colore e associa ogni colore (o tipo) ad ogni posto.

$C: P \rightarrow \tau$

- Cap è la funzione che indica quanti token è in grado di contenere un posto. Il valore ∞ viene utilizzato per indicare che il posto ha capacità illimitata.

$Cap: P \rightarrow \{N^+ \cup \infty\}$

- L'informazione delle iscrizioni sugli archi di input e di output è formalmente descritta dalle matrici di incidenza Pre e Post. Gli archi che vanno da un posto p a una transizione t corrispondono alla matrice Pre e hanno un multinsieme (M) di variabili come iscrizione. Il tipo delle variabili deve corrispondere al dominio di colore del posto p, C(p).

$\forall\, p \in P, t \in T: Pre(p,t) \in M_{VarC(p)}$

Il firing mode di una transizione in una rete di Petri stocastica colorata è rappresentato dal mapping di valori sulle variabili di transizione. Queste variabili sono quelle che appaiono nelle iscrizioni sugli archi di input Pre.

Si definisce il corrispondente insieme di variabili di transizione Var(t) di una transizione t basandosi sulle variabili che sono contenute nelle espressioni Pre come:

$\forall\, t \in T: Var(T) = \cup_{p\in P} Var(Pre(p,t))$

Una possibile combinazione di valori per tutte le variabili di transizione è chiamata binding (o a volte colore della transizione) di t ed è indicata da $\beta(t)$:

$\forall\, t \in T, v \in Var(t): \beta(t,v) \in C(v)$

L'insieme di tutti i binding di una transizione t teoricamente possibili è indicato da $\beta^*(t)$, ed è dato dal prodotto di tutti gli insiemi di valori di tutte le variabili di transizione di t:

$\beta^*(t) = C(v_1) \times C(v_2) \times \ldots \times C(v_{|Var(t)|})$

per $Var(t) = \{v_1, v_2, \ldots, v_{|Var(t)|}\}$

Il binding β di una transizione t mappa le variabili di transizione Var (t) su valori ed è così utilizzato per ricavare il valore reale delle espressioni nel modello. Il valore di un'espressione $Expr_{Var(t)}$, nell'ipotesi che ci sia il binding $\beta \in \beta^*(t)$, è calcolato valutando l'espressione dopo il mapping di tutte le variabili sui valori dati dal binding. Il risultato è indicato da $Expr^{\beta}_{Var(t)}$.

Gli archi di output (vanno da una transizione t ad un posto P) presentano come iscrizioni le espressioni sulle variabili di transizione:

$\forall\ p \in P, t \in T$: Post (p,t) $\in Expr_{Var(t)}$

Sulla base dei binding è ora possibile definire il tipo del risultato delle espressioni sugli archi di input e di output. Per gli archi di input, l'impostazione di una variabile porta ad un multinsieme di token del corrispondente tipo di posto di input. Ogni espressione su un arco di output, analogamente, restituisce un multinsieme sul dominio di colore del posto connesso, quando è valutata per un mapping di valori sulle variabili contenute. Si ha quindi:

$$\forall\ p \in P, t \in T, \beta \in \beta^*(t): \qquad Pre(p,t)^\beta \in M_{C(p)}$$
$$Post(p,t)^\beta \in M_{C(p)}$$

- In ogni stato solo alcuni binding possono essere abilitati. La guardia G di una transizione è una funzione booleana che ritorna Vero per un binding β se, nello stato m del modello, è verificata. La funzione di guardia di una transizione è definita da un'espressione che comprende le variabili di transizione e lo stato del modello:

$\forall\ t \in T: G(t): \beta^*(t) \times M \rightarrow B$

- Uno stato di uno rete di Petri stocastica colorata corrisponde ad una specifica associazione dei multinsiemi di token ai posti, ed è chiamato marcatura. Ogni marcatura m è quindi un vettore indicizzato dai posti, le cui entry sono multinsiemi di colori. Un token è un oggetto di un colore (tipo).

$\forall\ p \in P: m(p) \in M_{C(p)}$

La marcatura iniziale m_0 descrive lo stato del modello di Reti di Petri dal quale prende origine il comportamento dinamico. M_0 è una marcatura ed è quindi della forma m_0: P →

M_τ con la restrizione che i token in ogni posto devono essere del corrispondente tipo. La marcatura iniziale non deve violare la restrizione sulle capacità dei posti:

$\forall\ p \in P: |\ m_{0(p)}| \leq Cap(p)$

L'insieme di tutte le possibili marcature è indicato da M (da non confondere col simbolo di multinsieme).

- RV indica le misure di performance della rete di Petri stocastica e colorata. Come precedentemente mostrato ci sono due tipi base di misure: il numero di token in un posto e il numero di volte che una transizione viene eseguita. È inoltre possibile definire misure più complesse utilizzando queste misure di base come termini in espressioni numeriche con operatori.

CAPITOLO 3

Statement del problema

Nel presente capitolo si introdurrà il problema in esame mostrando come sia di fondamentale importanza la possibilità di considerare failure modes multipli per poter avere risultati significativi in merito allo studio di affidabilità dei sistemi a componenti.

3.1 Failure modes multipli e propagazione dell'errore

Il problema in esame [3] considera sistemi a componenti software che operano in ambienti critici per la sicurezza, in essi è quindi fondamentale l'affidabilità del sistema e cioè la misura probabilistica della capacità del sistema di eseguire con successo il suo compito.
Il termine **failure** indica una deviazione dal servizio corretto offerto dal sistema. Un **errore** è la parte dello stato del sistema che può condurre alla generazione di una failure ed è a sua volta causato dall'attivazione di un **fault**. La deviazione dal funzionamento corretto del sistema può manifestarsi in diversi modi, corrispondenti ai diversi **failure modes** del sistema. In tale ambito sorge in maniera del tutto naturale l'esigenza di utilizzare **failure modes multipli**, infatti una scelta che preveda di avere un comportamento di tipo booleano in grado di distinguere solamente tra presenza e assenza di failure non garantisce l'espressività necessaria a rappresentare comportamenti significativamente patologici. Inoltre risulta essere rilevante anche la possibilità di descrivere il processo di propagazione dell'errore tra i componenti del sistema. Gli aspetti significativi da considerare nella modellazione sono quindi la possibilità di avere failure modes multipli e la capacità di quest'ultimi di propagarsi in differenti modi attraverso il flusso di esecuzione arrivando anche ad affiorare a livello dell'interfaccia applicativa. In particolare si vuole porre l'accento sulla possibilità di avere failure modes in grado di trasformarsi attraverso i

componenti e cioè, nella modellazione realizzata, i failure modes non sono immutabili ma il loro tipo può variare passando da un componente ad un altro, questo aspetto permette di considerare il problema ad un maggiore livello di generalità, infatti si riesce a descrivere il caso di un componente che, invocato con un particolare failure mode, restituisce in output un differente failure mode.

Focalizzandosi sulla definizione dei failure modes, appare naturale come la loro caratterizzazione sia un'attività strettamente legata al particolare sistema considerato. Ad esempio due tipici failure modes che ricorrono in più sistemi sono i **content failure** e i **timing failure**. I primi descrivono il caso in cui il contenuto dell'output del sistema si discosta da quello corretto, i secondi modellano la situazione in cui il tempo di consegna del contenuto stesso non è quello atteso. Si possono definire altri failure modes, ad esempio introducendo una gradualità nei content e nei timing failure, o ancora combinando questi ultimi arrivando a definire dei particolari failure modes nei quali sia il contenuto sia il relativo tempo di consegna sono scorretti, tali failure modes sono detti **halt failure**, il loro verificarsi rende l'attività del sistema, se presente, impercettibile a livello dell'interfaccia del sistema stesso. Gli errori possono sorgere sia a causa di fault interni silenti sia a causa della ricezione di un input errato. Errori nei componenti non si manifestano necessariamente come failure nei componenti stessi e a loro volta failure nei singoli componenti non danno necessariamente luogo a failure del sistema. Una failure in un componente si verifica solo quando un errore si propaga all'interno del componente fino alla sua interfaccia, e una failure nel sistema si verifica solo quando un errore si propaga attraverso i componenti fino all'interfaccia del sistema. In questo percorso di propagazione, un errore può essere mascherato quando, ad esempio, un valore errato viene sovrascritto prima di essere fornito all'interfaccia. Un errore può anche essere trasformato, ad esempio nel caso in cui una content failure, ricevuta da un altro componente, causa computazioni addizionali portando al verificarsi di una timing failure. Se si vuole analizzare l'affidabilità di un sistema a componenti si deve considerare l'intero insieme di fattori appena delineati, i quali possono portare al manifestarsi di una failure. A livello dei componenti, questo aspetto richiede di stimare la verosimiglianza che un dato failure mode si manifesti all'interfaccia del componente a causa di un fault interno, o a causa della propagazione dello stesso (o di un differente) failure mode ricevuto all'interfaccia di input del componente. A livello del sistema, si dovrebbero considerare i possibili percorsi di propagazione attraverso i componenti, e le rispettive verosimiglianze.

L'approccio proposto può essere applicato ai primi passi della progettazione del software, e può fornire un accurato modello predittivo in grado di guidare le decisioni a riguardo degli aspetti architetturali e comportamentali.

3.2 Modellazione dei sistemi a componenti

Per supportare lo sviluppo basato su componenti, ogni componente deve possedere tutte le informazioni a riguardo delle sue proprietà funzionali per poter così realizzare l'interazione con gli altri componenti. Queste informazioni includono, ad esempio, la specifica dei servizi richiesti o forniti dal componente, che viene spesso indicata con il nome di **interfaccia costruttiva**.

Per poter analizzare le proprietà non-funzionali, quali ad esempio l'affidabilità, è necessario considerare informazioni aggiuntive espresse attraverso un'opportuna **interfaccia analitica**. Ed è su questo secondo aspetto che si focalizza la modellazione successiva.

Il modello presentato in questa sezione definisce un'**interfaccia analitica orientata all'affidabilità**. In questo modello si assume che ogni componente (e quindi l'intero sistema costituito dai singoli componenti) sia caratterizzato da **N** differenti **failure modes**. Ogni modo r, ($1 \leq r \leq N$) può essere uno dei failure mode elementari definiti nella sezione precedente, o una combinazione di alcuni di essi, o un qualunque altro failure mode dedicato. Per ragioni di uniformità, viene introdotto un failure mode addizionale (il failure mode 0), che corrisponde alla propagazione del servizio corretto.

3.2.1 Modello di un singolo componente

Un singolo componente C_i comprende:

- una porta di input ip_i;
- un insieme di porte di output $O_i = \{op_{ik}\}$;

- un modello operativo definito dalle probabilità: $p_i(k)$ ($\forall op_{ik} \in O_i$), dove ogni $p_i(k)$ è così definito: $p_i(k) = Pr\{C_i$ produce un output sulla porta $op_{ik} \in O_i \setminus C_i$ riceve un input sulla sua porta di input$\}$

Vale la seguente relazione: $\sum_{opik \in Oi} p_i(k) = 1$;

- un modello di failure definito dalle probabilità: $f_i(r,s)(0 \leq r \leq N, 0 \leq s \leq N\}$ dove ogni $f_i(r,s)$ è così definito: $f_i(r,s) = Pr\{C_i$ produce un output con il failure mode s $\setminus C_i$ riceve un input con il failure mode r$\}$

Vale la seguente relazione: $\sum^{N}_{s=0} f_i(r,s) = 1$.

In questo modello, si fa l'assunzione che il trasferimento, sia dei dati sia del controllo, si realizzi attraverso le porte di input e di output: C_i riceve dati e controllo dalla sua porta di input, e produce dati e trasferisce il controllo, verso altri componenti, attraverso le sue porte di output. Il **modello operativo** dà una caratterizzazione stocastica del profilo di utilizzo degli altri componenti quando C_i è attivo. Ogni probabilità $p_i(k)$ può essere vista come la frazione di trasferimento di dati e controllo che avviene attraverso la porta di output op_{ik} di C_i rispetto a al trasferimento totale di dati e controllo generato da C_i. Analogamente, il **modello di failure** dà una caratterizzazione stocastica dello stato delle failure nel componente C_i. La probabilità $f_i(r,s)$ può essere utilizzata come base per definire interessanti proprietà di affidabilità di un componente software. Alcuni esempi di queste proprietà sono citati di seguito:

- la **probabilità di internal failure** relativa al failure mode s, $s > 0$, è la probabilità $f_i(0,s)$.
- La **robustezza** rispetto all'error mode r, $r > 0$, è la probabilità $f_i(r,0)$.
- La **suscettibilità** rispetto all'error mode r, $r > 0$, è la probabilità $1 - f_i(r,0)$.
- La **tendenza** relativa al failure mode s è la probabilità $\sum_r \beta_r \cdot f_i(r,s)$, dove β_r è la probabilità di ricevere in input un failure mode r.

Queste proprietà formali sono facilmente comprensibili dall'utilizzatore e permettono una semplice formalizzazione dei requisiti su un singolo componente, unita alla possibilità di fornire senza difficoltà un feedback allo sviluppatore.

Come ultimo aspetto è utile mostrare il comportamento dell'interfaccia costruttiva precedentemente definita nel caso in cui ci si trovi a dover modellare componenti che prevedano di avere più porte di input. Sono possibili due interpretazioni: la prima prevede di astrarre dalla presenza di porte di input multiple nel componente reale condensandole in un'unica porta di input. In questo caso, sia il modello operativo, sia quello di failure, di questo componente astratto, rappresentano una sorta di profilo medio del comportamento del componente reale. La seconda interpretazione prevede di considerare il componente astratto come una proiezione del comportamento reale rispetto a una delle sue porte di input. Da questo punto di vista, un componente reale risulta essere modellato da un insieme di componenti astratti analoghi a quello precedentemente definito, dove ogni elemento dell'insieme modella il comportamento del componente reale condizionato dall'aver ricevuto dati e controllo attraverso una specifica porta di input.

3.2.2 Modello dell'architettura

Un'architettura A comprende:

- un insieme di componenti: $C = \{C_0, C_1, \ldots, C_M\}$con le rispettive interfacce analitiche;
- un mapping: : $\cup^M_{i=0} O_i \rightarrow \cup^M_{i=0} \{ip_i\}$

Data una porta di output op_{ik} di un componente C_i, $map_A(op_{ik})$ definisce a quale porta di input è collegata la porta di output op_{ik}.

In questa definizione dell'architettura, C_1, C_2, …, C_{M-1}, corrispondono ai componenti utilizzati per costruire l'applicazione modellata da A. C_0 e C_M hanno invece un ruolo particolare. Essi sono componenti fittizi utilizzati per modellare rispettivamente l'inizio dell'applicazione modellata da A e il risultato finale prodotto dall'applicazione stessa. C_0 ha tante porte di output quanti sono i possibili punti di ingresso dell'applicazione modellata da A. Inoltre, la porta di input di C_0 non è connessa a nessuna delle altre porte di output dei componenti di A. C_M ha solo una porta di input, e non ha porte di output. Data una porta di output $op_{0k} \in O_0$, $map_A(op_{0k}) = ip_i$, significa che C_i è un componente dal quale l'applicazione può iniziare la sua esecuzione. Analogamente, data una porta di output op_{ik}

$\in O_i$ ($1 \leq i \leq M - 1$), $map_A(op_{0k}) = ip_M$, significa che C_i, è un componente che può produrre il risultato finale dell'applicazione.

Il modello operativo associato a C_0, (dato dalle probabilità $p_0(k)$) può così essere utilizzato per modellare l'incertezza stocastica relativa al punto di ingresso dell'applicazione e al profilo di failure dell'utente. La terminazione dell'applicazione è invece modellata dal trasferimento del controllo a C_M. Data la natura speciale di C_0 e C_M, i loro failure modes sono così definiti:

$f_0(r,r) = f_M(r,r) = 1$ ($0 \leq r \leq N$),

$f_0(r,s) = f_M(r,s) = 0$ ($0 \leq r \leq N$, $0 \leq s \leq N$, $r \neq s$),

che significa che C_0 e C_M, non modificano i failure modes che ricevono.

Vengono ora definite le seguenti probabilità a livello dell'architettura:

$F_A(r,s)$($0 \leq r \leq N$, $0 \leq s \leq N$), dove ogni $F_A(r,s)$ è così definito: $F_A(r,s) = Pr\{A$ termina con il failure mode $s \setminus A$ è stata attivata con il failure mode $r\}$

Analogamente alle proprietà a livello dei componenti definite precedentemente, ora è possibile utilizzare le probabilità $F_A(r,s)$ per definire proprietà di affidabilità a livello dell'applicazione:

- l'**affidabilità** è la probabilità $F_A(0,0)$.
- La **robustezza** rispetto all'error mode r ($r > 0$) è la probabilità $F_A(r,0)$.
- La **suscettibilità** rispetto all'error mode r ($r > 0$) è la probabilità $1 - F_A(r,0)$.
- La **tendenza** relativa al failure mode s è la probabilità $\sum_r \beta_r \cdot F_A(r,s)$, dove β_r è la probabilità di ricevere in input un failure mode r.

I modelli dei componenti e dell'architettura qui presentati, permettono la definizione di una vista astratta, orientata all'affidabilità, del sistema a componenti.

CAPITOLO 4

Modello risolutivo

Il presente capitolo descrive la modellazione realizzata per poter efficacemente trattare il problema delineato nel capitolo precedente. Il formalismo matematico utilizzato, le Reti di Petri stocastiche colorate, è stato dettagliatamente precisato nel Capitolo 2.
La prima sezione descrive un esempio applicativo, lo scopo è quello di introdurre un semplice sistema a componenti [3], secondo la formalizzazione fornita nel capitolo precedente, che verrà poi utilizzato nell'ultima sezione con la funzione di esemplificare la modellazione proposta nella seconda sezione.

4.1 Esempio di un semplice sistema a componenti

Il sistema presentato è costituito da tre componenti. Il primo componente (C_1) ha il ruolo di dispatcher per tutte le richieste che giungono al sistema e ne rappresenta l'unico punto di ingresso. Il dispatcher analizza le richieste e le invia al server che è modellato con un secondo componente (C_2). A seconda della specifica operazione richiesta, il server può svolgere il lavoro autonomamente o può generare ulteriori richieste. Nel primo caso l'output del server deve passare attraverso un server di guardia, modellato dal terzo componente (C_3), prima di raggiungere l'utente. Nel secondo caso, la richiesta del server è rimandata al dispatcher per poter essere nuovamente eseguita. Il server di guardia ha il compito di analizzare l'output del server principale ed assicurarsi che non contenga nessun informazione confidenziale, se il server di guardia rileva informazioni inattese allora rimanda la richiesta al dispatcher per renderla nuovamente elaborabile, se invece non vengono rilevate informazioni confidenziali allora i risultati vengono forniti all'utente.

L’architettura del sistema è rappresentata in Figura 4.1. Si ricorda che, nella modellazione realizzata, una transizione prevede il trasferimento sia del controllo, sia dei dati, da un componente all’altro. Le probabilità che esprimono il modo operativo di ogni componente sono indicate in grassetto in corrispondenza della freccia direzionale di ogni connessione.

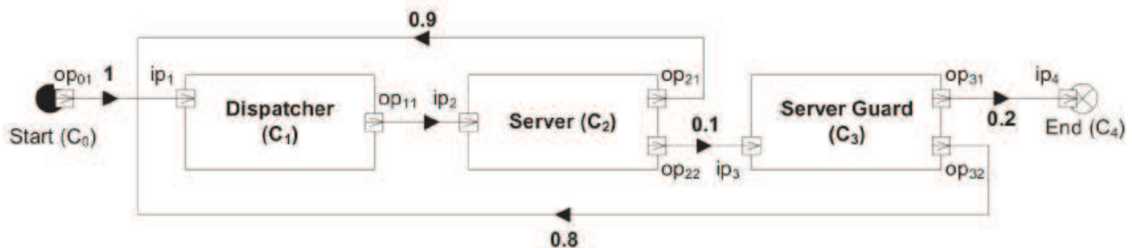

Figura 4.1: Architettura del sistema

In questo esempio si considerano due failure modes (indicati da 1 e 2 rispettivamente) oltre al modo di esecuzione corretto (failure mode 0).
Il profilo atteso di avvio dei failure modes all’attivazione del sistema è descritto dalle seguenti probabilità:

failure mode 0 = 0,8
failure mode 1 = 0,1
failure mode 2 = 0,1

Per poter analizzare l’impatto che le proprietà di affidabilità dei singoli componenti hanno sull’affidabilità del sistema, si introducono le seguenti proprietà a livello dei componenti:

- **Affidabilità** (R): è la probabilità che il componente C_i, a condizione di essere stato invocato senza errori, non produca un output errato ($f_i(0,0)$).
- **Robustezza** (rispetto all’error mode r, B_r): è la probabilità che il componente C_i, invocato con il failure mode r, produca un output corretto ($f_i(r,0)$).
- **Failure interna** (con il modo s, F_s): è la probabilità che il componente C_i, invocato senza errori, produca un output errato con il failure mode s ($f_i(0,s)$).
- **Switching** (dal modo r al modo s con r,s > 0, S_{rs}): è la probabilità che il componente C_i, produca in output il failure mode s, dato in input l’error mode r ($f_i(r,s)$).

La Tabella 4.1 mostra i valori assegnati alle proprietà di affidabilità dei componenti del sistema.

Componente	**Proprietà**	**Valore**
C_1	R	0.94
C_1	F_1	0,03
C_1	F_2	0,03
C_1	B_1	0,05
C_1	S_{11}	0,92
C_1	S_{12}	0,03
C_1	B_2	0,70
C_1	S_{21}	0,00
C_1	S_{22}	0,30
C_2	R	0,64
C_2	F_1	0,13
C_2	F_2	0,23
C_2	B_1	0,03
C_2	S_{11}	0,84
C_2	S_{12}	0,13
C_2	B_2	0,03
C_2	S_{21}	0,13
C_2	S_{22}	0,84
C_3	R	0,96
C_3	F_1	0,02
C_3	F_2	0,02
C_3	B_1	0,65
C_3	S_{11}	0,169
C_3	S_{12}	0,181
C_3	B_2	0,05
C_3	S_{21}	0,336
C_3	S_{22}	0,614

Tabella 4.1: Proprietà di affidabilità

4.2 Modellazione proposta

In questa sezione si mostra come si è giunti ad ottenere la modellazione del problema considerato. Si presenta inizialmente la descrizione di un singolo componente, successivamente verrà illustrata la modellazione dell'intera architettura. L'obiettivo di questa sezione è quello di mostrare le scelte modellistiche compiute per poter descrivere il problema delineato nel Capitolo 3 con le Reti di Petri Stocastiche Colorate introdotte nel Capitolo 2. I formalismi necessari sono stati ampiamente descritti nei capitoli precedenti e, in questa sezione, si farà ricorso ad essi solamente in casi di potenziale ambiguità, preferendo invece dare maggior enfasi alla corrispondenza tra modello e problema mostrando, di volta in volta, le soluzioni individuate e la loro bontà nell'ottica di ottenere una modellazione corretta e completa del problema.
Nel seguito, talvolta, verranno utilizzati indistintamente i termini "error mode" e "failure mode" benché essi abbiano una sfumatura differente come delineato nella Sezione 3.1. Si fa questa scelta per non appesantire troppo l'esposizione e soprattutto per non spostare l'attenzione del problema dall'aspetto centrale che non è la distinzione tra error e failure ma la modellazione del meccanismo che ne permette la propagazione all'interno del sistema. Il contesto chiarirà l'accezione nella quale si sta considerando il termine utilizzato.
Nella sezione verranno mostrate immagini tratte dal tool utilizzato per la modellazione e la simulazione (TimeNET) che permetteranno di seguire lo sviluppo complessivo della creazione del modello.

Failure modes multipli

L'aspetto da cui parte la descrizione della modellazione proposta, e quello che meglio mostra l'adeguatezza del formalismo matematico adottato, giustificandone la bontà della scelta, è la presenza di failure modes multipli.
I token, nel modello realizzato, possono essere pensati come una rappresentazione concreta del trasferimento del flusso di dati e controllo da un componente all'altro. Ma è la possibilità di dotarli di un tipo che permette di considerare i differenti failure modes.
La presenza dei colori, o tipi, è l'elemento che permette di avere, a differenza delle Reti di Petri tradizionali, **token distinguibili** e quindi offre la possibilità di modellare la gestione del comportamento voluto.

Si è scelto di utilizzare un unico tipo per tutti i token per evitare descrizioni ridondanti e in particolare si è fatto ricorso ad un **tipo strutturato** definito appositamente: il tipo "Errore". Esso contiene a sua volta un attributo di tipo "string" denominato "tipo" e sarà proprio il valore di quest'ultimo che determinerà il failure mode corrispondente. Il failure mode n-esimo è indicato dal valore "erroren". Poteva anche essere evitato il ricorso ad un tipo strutturato utilizzando direttamente token di tipo "string", ma si è fatta tale scelta in quanto è sembrata migliore dal punto di vista della linearità nella creazione del modello e soprattutto in ottica di migliorarne l'usabilità e l'utilizzo da parte di utenti meno esperti. Infatti la maggior caratterizzazione dell'attributo "tipo" ne permette una più efficace collocazione all'interno della modellazione, ed è del tutto coerente ed allineata alla descrizione del corrispondente elemento nella presentazione del problema iniziale.
In Figura 4.2 viene mostrata la rappresentazione grafica del tipo strutturato "Errore" nell'interfaccia di TimeNET, e la corrispondente finestra nella quale si possono specificare gli attributi.

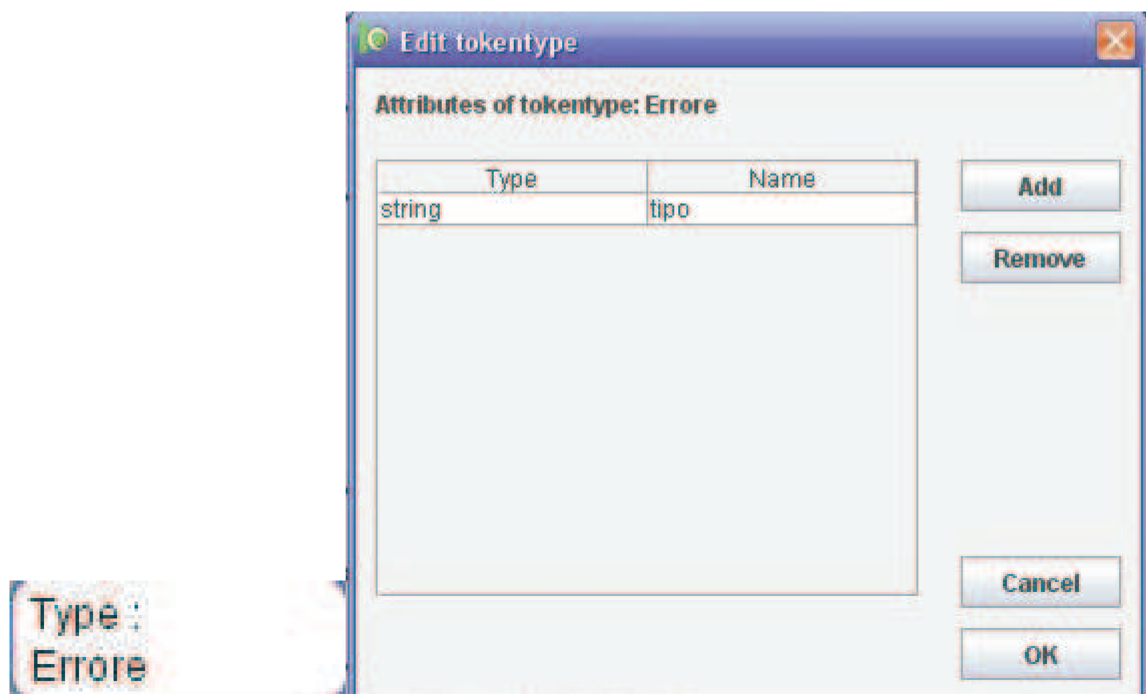

Figura 4.2: Tipo strutturato "Errore"

Si procede ora alla descrizione della modellazione di un singolo componente e successivamente dell'architettura del sistema complessivo.

4.2.1 Modello di un singolo componente

Porte del componente

Considerando il problema in esame e lo strumento matematico con il quale si è deciso di affrontarlo, la scelta più naturale, nella modellazione di un singolo componente, è stata quella di rappresentare le porte di input e di output con i **posti** delle Reti di Petri. In accordo con il problema precedentemente presentato [3], ogni singolo componente è quindi dotato di un posto che rappresenta l'unica porta di input del componente stesso. Sono poi presenti tanti posti quante sono le porte di output del componente.

Modello operativo e modello di failure

La modellazione del modello operativo e del modello di failure è stata effettuata ricorrendo all'utilizzo delle **transizioni**. La transizione, nella modellazione proposta, può essere vista come l'azione compiuta dal componente al fine di scegliere a quale componente trasferire dati e controllo da una parte (modello operativo), e scegliere se propagare (o modificare) un particolare failure mode dall'altra (modello di failure).
Si è scelto di utilizzare transizioni immediate in quanto il problema in esame non richiedeva l'introduzione di particolari ritardi. La creazione dei due modelli (operativo e di failure) è stata realizzata in maniera coordinata al fine di rendere più compatta la rappresentazione senza per questo perdere in generalità o correttezza. In particolare gli aspetti fondamentali cui si è fatto ricorso per ottenere una modellazione corretta e completa sono stati le **funzioni di guardia** e i **firing weights**. Sostanzialmente considerando l'esempio mostrato nella sezione precedente, e le relative proprietà di affidabilità a livello dei componenti, ci si accorge che esse sono tutte rielaborazioni della probabilità $f_i(r,s)$ definita nella Sezione 3.2.1 e qui richiamata per comodità:
$f_i(r,s)$ = Pr{C_i produce un output con il failure mode s \ C_i riceve un input con il failure mode r}.
Senza riportare la caratterizzazione delle quattro proprietà presentata nella Sezione 4.1 ma semplicemente tenendo presente la definizione appena ricordata e valutando i failure modes presenti nell'esempio (tre, due effettivi ed uno per indicare il modo di funzionamento corretto) si nota, osservando la Tabella 4.1, che ogni componente viene descritto da nove proprietà esattamente corrispondenti a tutte le possibili situazioni di

propagazione dei failure modes. Generalizzando ad un numero qualsiasi di failure modes è immediato ricavare che per avere una completa caratterizzazione, in grado di fornire una rappresentazione modellistica coerente, è necessario che le transizioni associate ad ogni porta di output del componente siano pari in numero al quadrato dei failure modes. Utilizzando tali transizioni è possibile descrivere tutti i casi di propagazione dei failure modes sia che essi rimangano inalterati, sia che subiscano una modifica del loro stato.

Una volta determinato il numero di transizioni necessario per poter descrivere il comportamento voluto, la modellazione prevede di inserire opportune funzioni di guardia. In particolare le funzioni di guardia hanno il compito di "filtrare" il token di ingresso valutandone il valore del campo "tipo" e quindi, conseguentemente, il failure mode associato. Le funzioni di guardia saranno del tipo "x.tipo=="erroren"", dove n indica il failure mode che si vuole considerare. Ad esempio se la transizione ha associata la funzione di guardia "x.tipo=="errore1"" essa risulta abilitata solamente se il token nel posto di ingresso associato ha l'attributo "tipo" del tipo "Errore" settato ad "errore1". In questo caso, il significato modellistico è quello di voler prendere in considerazione il caso di componente attivo con failure mode di tipo 1, e quindi, ritornando alla descrizione mediante la probabilità $f_i(r,s)$, si sta considerando il caso $f_i(1,s)$ nel componente C_i. La parte che rimane da specificare è come possa avvenire il cambio di failure mode (o la conferma) all'interno del componente. Per modellare questo comportamento si è fatto ricorso alle iscrizioni sugli archi di output delle transizioni. Esse permettono di modificare i valori degli attributi dei token e quindi, ad esempio, un'iscrizione del tipo "x(tipo="errore2")" su un arco di output ha il significato di settare ad "errore2" l'attributo "tipo" del token e quindi modella il comportamento di un cambiamento (o conferma) verso il failure mode 2 in uscita. Riproponendo la descrizione parallela con la probabilità $f_i(r,s)$, una tale iscrizione modella il caso $f_i(r,2)$ nel componente C_i. Combinando adeguatamente le guardie delle transizioni e le iscrizioni degli archi di output si riesce ad esplorare tutto lo spazio dei casi possibili ottenendo una completa caratterizzazione della probabilità $f_i(r,s)$. In particolare il **numero di guardie differenti** sarà sempre uguale al **numero di iscrizioni differenti** ed entrambi coincideranno con il **numero di failure modes del sistema**. Per comprendere meglio il meccanismo si può poi pensare ad una sorta di "prodotto" tra guardie e iscrizioni che dà luogo ad **n^2 abbinamenti** (avendo **n failure modes**), i quali vengono mappati in una corrispondenza biunivoca con le n^2 transizioni per porta di output. In Figura 4.3 viene mostrata, a titolo di esempio, la finestra che permette di specificare le proprietà di una transizione facendo particolare riferimento alla funzione di guardia. Nel

caso considerato, la guardia specificata, prevede di abilitare la transizione solamente se il token nel posto di input associato ha l'attributo "tipo" settato ad "errore1".

Qualified Name	Value
text	T21
priority	1
weight	2.0E0
globalGuard	
localGuard	x.tipo=="errore1"
takeFirst	false
serverType	ExclusiveServer
timeGuard	
specType	Automatic
manualCode	
watch	false
logfileName	
logfileDescription	
logfileExpression	
displayExpression	

Figura 4.3: Funzione di guardia di una transizione

La modellazione prosegue poi descrivendo quantitativamente le probabilità $f_i(r,s)$. Per modellare tale aspetto, si è fatto ricorso ai firing weights delle transizioni. I weights, infatti, permettono di specificare le probabilità relative di firing di transizioni concorrenti. Ad esempio se si ha un sistema con due failure modes e si vogliono specificare le due probabilità $f_i(0,0) = 0,8$ e $f_i(0,1) = 0,2$ sarà sufficiente settare i weights delle due transizioni corrispondenti, rispettivamente a 8 e 2. L'utilizzo dei firing weights permette quindi di descrivere completamente il profilo dei failure modes del componente considerato. Va inoltre considerato che i firing weights, nella modellazione finale, dovranno essere modificati per tenere conto del modello operativo del sistema, questo aspetto verrà specificato nell'ultimo esempio di questa sezione e nelle sezioni successive dove sarà mostrata la modellazione dell'architettura dell'intero sistema.
In Figura 4.4 viene mostrata la finestra di una transizione evidenziando la specifica del corrispondente firing weight, che nel caso specifico è settato al valore 6.

Qualified Name	Value
text	T23
priority	1
weight	6.0E0
globalGuard	
localGuard	x.tipo=="errore2"
takeFirst	false
serverType	ExclusiveServer
timeGuard	
specType	Automatic
manualCode	
watch	false
logfileName	
logfileDescription	
logfileExpression	
displayExpression	

Figura 4.4: Firing weight di una transizione

Ora, dopo aver spiegato le scelte modellistiche effettuate per descrivere il meccanismo di propagazione dei failure modes all'interno di un singolo componente, si mostrano alcuni esempi di complessità graduale in cui sono state applicate le tecniche precedentemente descritte.

Il primo esempio considerato, e mostrato in Figura 4.5, è un semplicissimo componente avente una sola porta di output e due failure modes: il failure mode 0 (rappresentante il modo di esecuzione corretto) e il failure mode 1. Si è partiti dal caso più semplice possibile (escluso il caso poco significativo di un output e un failure mode) per poter illustrare meglio gli aspetti fondamentali della modellazione realizzata. Nell'esempio in Figura 4.5 il posto P1 rappresenta la porta di input del componente, il posto P2 rappresenta l'unica porta di output e il token presente il P1 indica che il controllo è assegnato al componente considerato. Entrambi i posti sono del tipo "Errore", in Figura 4.6 viene mostrata, a titolo di esempio, la finestra che descrive le proprietà del posto P1, evidenziandone il tipo (si ricorda che ad ogni posto può corrispondere un solo tipo). Sono poi presenti quattro transizioni coerentemente con le considerazioni precedenti che prevedono un numero di transizioni pari al quadrato dei failure modes. Dalla figura si nota inoltre la presenza di due differenti iscrizioni sugli archi di output: l'iscrizione "x(tipo="errore0")" e l'iscrizione "x(tipo="errore0")", ciascuna riportata due volte.

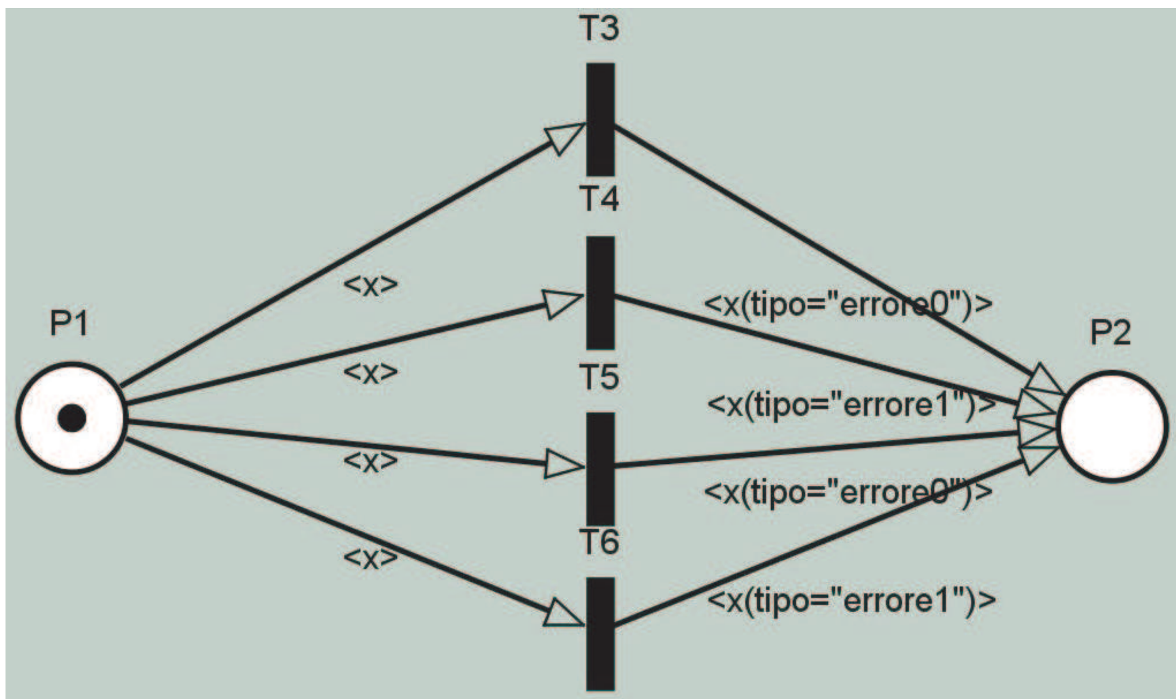

Figura 4.5: Componente con un output e due failure modes

Le iscrizioni sugli archi di input, non esplicitamente menzionate nella descrizione iniziale, non presentano elementi degni di particolare rilievo, semplicemente esse sono costituite dalla variabile "x", che è poi la stessa utilizzata dalle funzioni di guardia nelle transizioni e dalle iscrizioni sugli archi di output, ed hanno il compito di identificare il token del posto di input permettendone quindi il controllo da parte delle funzioni di guardia e la modifica da parte delle iscrizioni sugli archi di output.

Qualified Name	Value
text	P1
queue	Random
capacity	0
tokentype	Errore
watch	false

Figura 4.6: Posto P1

Per mostrare nel dettaglio la specifica delle funzioni di guardia, e dei firing weights, nelle quattro figure successive vengono riportate le finestre di descrizione delle quattro transizioni del modello.

Qualified Name	Value
text	T3
priority	1
weight	1.0E0
globalGuard	
localGuard	x.tipo=="errore0"
takeFirst	false
serverType	ExclusiveServer
timeGuard	
specType	Automatic
manualCode	
watch	false
logfileName	
logfileDescription	
logfileExpression	
displayExpression	

Figura 4.7: Transizione T3

Qualified Name	Value
text	T4
priority	1
weight	3.0E0
globalGuard	
localGuard	x.tipo=="errore0"
takeFirst	false
serverType	ExclusiveServer
timeGuard	
specType	Automatic
manualCode	
watch	false
logfileName	
logfileDescription	
logfileExpression	
displayExpression	

Figura 4.8: Transizione T4

Qualified Name	Value
text	T5
priority	1
weight	5.0E0
globalGuard	
localGuard	x.tipo=="errore1"
takeFirst	false
serverType	ExclusiveServer
timeGuard	
specType	Automatic
manualCode	
watch	false
logfileName	
logfileDescription	
logfileExpression	
displayExpression	

Figura 4.9: Transizione T5

Qualified Name	Value
text	T6
priority	1
weight	1.0E0
globalGuard	
localGuard	x.tipo=="errore1"
takeFirst	false
serverType	ExclusiveServer
timeGuard	
specType	Automatic
manualCode	
watch	false
logfileName	
logfileDescription	
logfileExpression	
displayExpression	

Figura 4.10: Transizione T6

Interpretando le informazioni sulle funzioni di guardia e sui firing weights secondo quanto descritto in precedenza, e osservando gli archi di output delle transizioni considerate, si

evince che le transizioni T3, T4, T5, T6 (e i rispettivi archi di output) modellano, rispettivamente, le probabilità:

$f_i(0,0) = 0,1$

$f_i(0,1) = 0,3$

$f_i(1,0) = 0,5$

$f_i(1,1) = 0,1$

Si mostra un secondo esempio in cui viene considerato un componente leggermente più complicato del precedente, esso ha sempre un solo output ma tre diversi failure modes, con la solita convenzione che uno di essi rappresenti il modo di funzionamento corretto e gli altri due siano effettivi failure modes.
L'esempio è presentato in Figura 4.11. In questo caso non si ripeteranno nel dettaglio tutte le considerazioni fatte per il sistema precedente, ma si mostreranno le differenze da esso con lo scopo di cogliere gli aspetti generali che permettono di modellare un componente qualsiasi.
Posti di input e output hanno un ruolo del tutto analogo a quello del caso precedente. La differenza evidente tra il sistema ora considerato e quello con due failure modes risiede nel numero di transizioni e di archi. In questo caso si nota la presenza di nove transizioni in accordo con la considerazione che esse devono essere in numero pari al quadrato dei failure modes. Analogamente, per quanto riguarda gli archi di output delle transizioni, si nota che essi presentano tre iscrizioni differenti ciascuna ripetuta tre volte. Tale aspetto è del tutto coerente con la descrizione iniziale e permette di mostrare un esempio pratico della generalizzazione ad un componente con tre failure modes. Non si ripete ora l'analisi delle singole finestre delle transizioni, ma in ogni caso se ne fornisce sinteticamente il contenuto per mostrare nel dettaglio la realizzazione del meccanismo di propagazione dell'errore in un componente con tre failure modes. In particolare le transizioni dalla T3 alla T5 hanno la seguente funzione di guardia: "x.tipo=="errore0"", le transizioni dalla T6 alla T8 hanno la seguente funzione di guardia: "x.tipo=="errore1"" e quelle dalla T9 alla T11 hanno la funzione di guardia: "x.tipo=="errore0"". Il loro ruolo è del tutto analogo a quello delle funzioni di guardia nel modello precedente e non viene qui ulteriormente dettagliato.

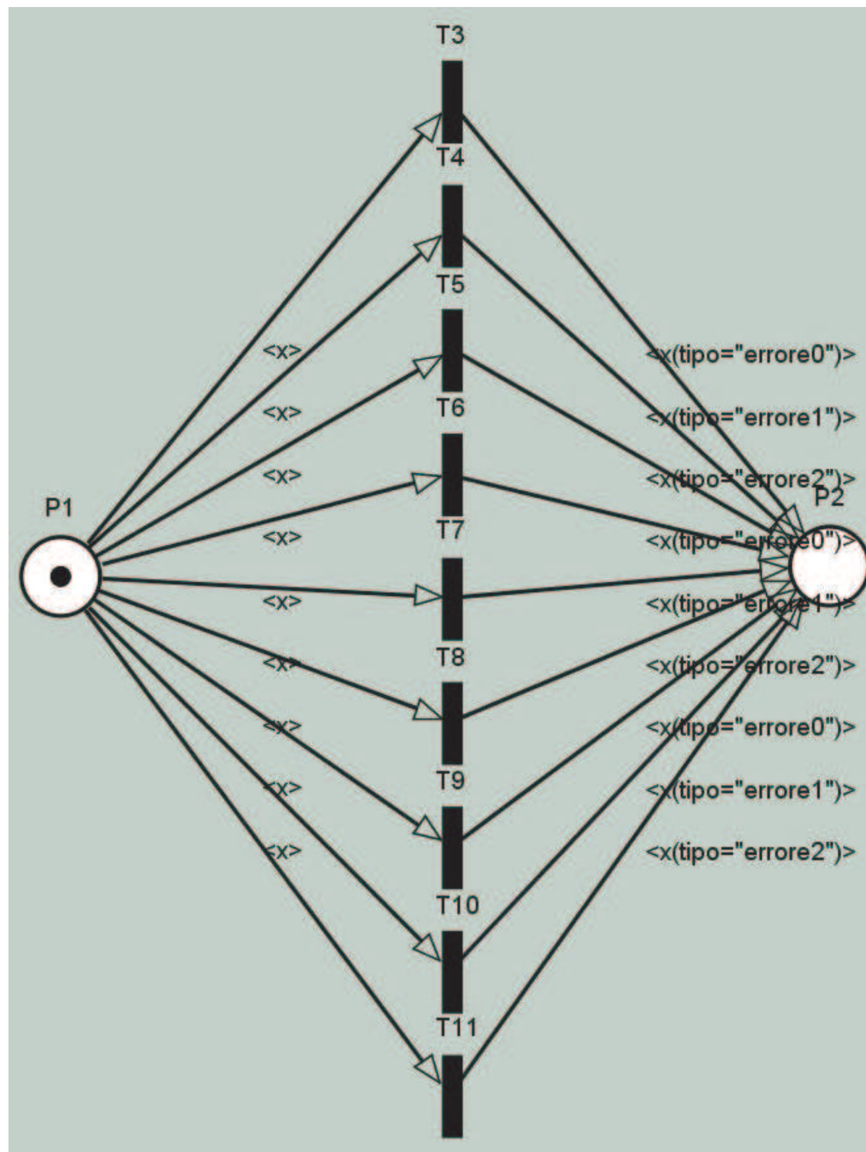

Figura 4.11: Componente con un output e tre failure modes

Dopo aver presentato la generalizzazione di un componente aumentando il numero di failure modes, si conclude questa sezione illustrando la seconda generalizzazione possibile, essa riguarda l’introduzione di un’ulteriore (e in generale n) porta di output.

In Figura 4.12 viene mostrato un componente con due porte di output e tre failure modes. Tale componente è dotato di tre posti, P1 rappresenta la porta di input e P2 e P3 le due porte di output. Oltre ad avere un posto in più, rispetto all’esempio precedente, il modello è caratterizzato dall’avere il doppio delle transizioni, precisamente ora sono presenti diciotto transizioni. Questo aspetto è coerente con le considerazioni precedenti ed è del tutto in linea con l’interpretazione modellistica seguita.

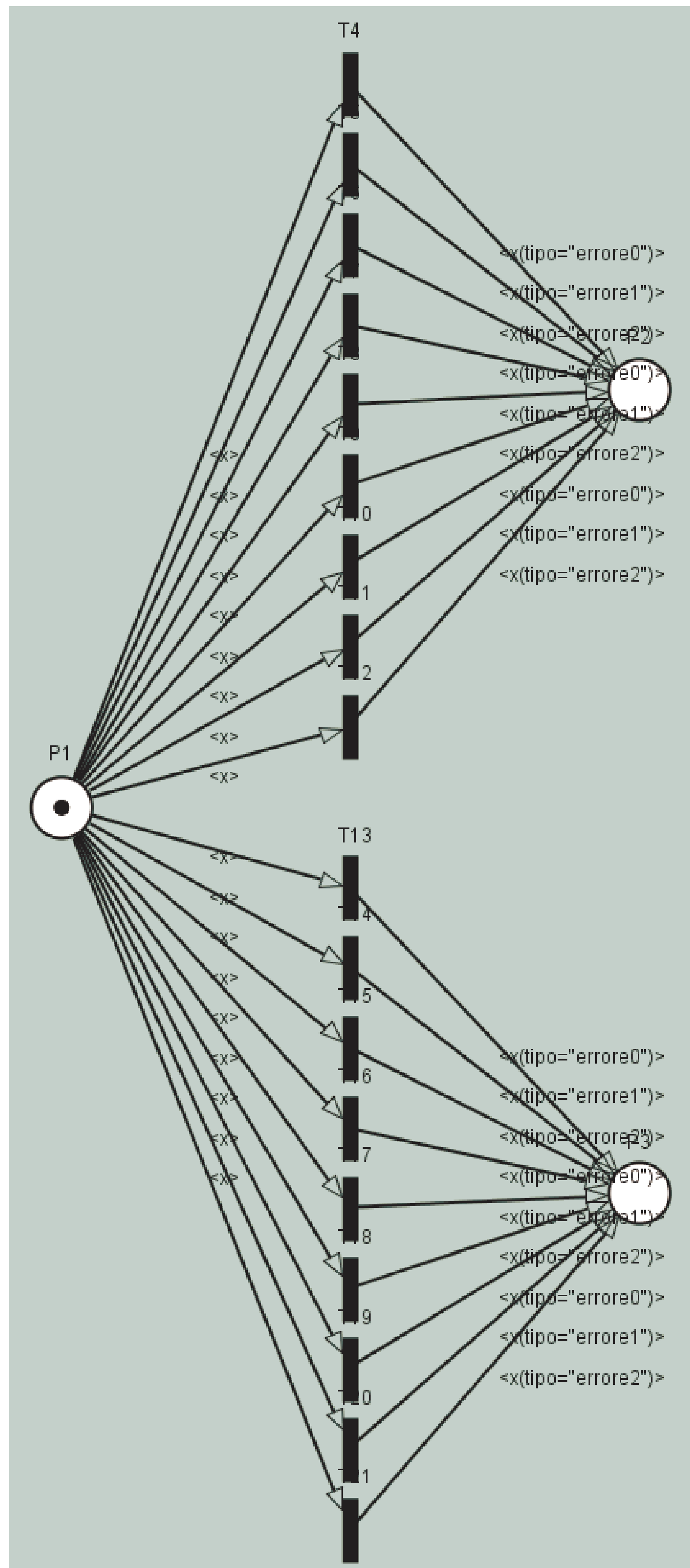

Figura 4.12: Componente con due output e tre failure modes

Infatti, nella modellazione proposta, si è rilevato che, per ogni componente, deve essere presente un numero di transizioni pari al quadrato del numero di failure modes per ciascuna porta di output. Nell'esempio considerato si hanno tre failure modes e due porte di output ed è dunque ragionevole avere diciotto transizioni. Una volta chiarito questo aspetto il resto del modello è del tutto simile a quelli già visti in precedenza, le guardie e le iscrizioni sugli archi hanno un comportamento analogo a quello già descritto. Sostanzialmente, le transizioni da T4 a T12 sono molto simili a quelle da T13 a T21, infatti le loro funzioni di guardia sono rispettivamente le stesse. La **differenza importante** tra questi due gruppi di transizioni risiede nei **firing weights**. Si è già citato in precedenza questo aspetto rimandandone la trattazione, ora lo si descrive nei dettagli. Si è detto che è stata fatta la scelta di fornire una descrizione coordinata del modello operativo e del modello di failure. Finora non si è mai considerato esplicitamente il **modello operativo** in quanto sono stati considerati solo componenti con una porta di output e quindi, implicitamente, un comportamento puramente sequenziale nel quale fosse univocamente determinato il componente verso il quale il componente corrente avrebbe trasferito i dati e il controllo. Ora, avendo introdotto un componente con due porte di output, si pone il problema di descrivere la modellazione del modello operativo. Essa è stata realizzata mediante l'utilizzo dei firing weights. In particolare, i firing weights vengono modificati per tenere conto del modello operativo in modo che non cambino le probabilità relative all'interno della stessa porta di output variandole invece tra porte di output differenti. Precisando maggiormente, si è fatta la scelta di moltiplicare per un fattore dipendente dalle probabilità del modello operativo tutti i firing weights relativi alla stessa porta di output. Nell'esempio considerato, se la prima porta di output è connessa ad un componente che avrà il 20% di probabilità di assumere il controllo e i dati e la seconda ad uno che avrà l'80% di probabilità, allora i firing weights delle transizioni connesse al posto di output corrispondente alla prima porta di output andranno moltiplicati per 2 e quelli relativi alla seconda porta di output per 8. Tale accorgimento permette di descrivere in maniera compatta il modello operativo. Al momento può non risultare del tutto chiaro l'aver voluto condensare la rappresentazione del modello operativo e del modello di failure, ma proseguendo nell'esposizione si mostrerà la bontà di tale scelta. In particolare nelle prossime due sezioni, nelle quali saranno introdotte, rispettivamente, la modellazione dell'architettura dell'intero sistema e la modellazione dell'esempio proposto nella sezione 4.1, si comprenderà meglio il perché di questa scelta in quanto sarà possibile descriverla

nell'ambito del sistema completo arrivando ad illustrare come essa garantisca una corretta interpretazione del modello operativo.

4.2.2 Modello dell'architettura

Dopo aver descritto la modellazione di un singolo componente, ora si mostra come è possibile realizzare il **collegamento tra componenti** arrivando quindi ad ottenere la modellazione dell'intera architettura del sistema.
Sostanzialmente, in tale fase, l'aspetto rilevante a cui prestare attenzione è la corretta connessione dei componenti, in modo da poter gestire, sfruttando gli accorgimenti descritti alla fine della sezione precedente, il **modello operativo** in maniera corretta.
Introducendo la visione globale del sistema è opportuno, prima di descrivere come si sono realizzati i legami tra componenti, introdurre due componenti particolari: il primo e l'ultimo che possono essere concettualmente ricondotti ai componenti C_0 e C_m, introdotti nella Sezione 3.2.2.

Primo componente

Il primo componente è, come illustrato nella Sezione 3.2.2, un componente fittizio che costituisce il punto di inizio del sistema ed ha l'importante ruolo di caratterizzare il profilo di avvio dei failure modes. In Figura 4.13 si illustra il modello del primo componente.

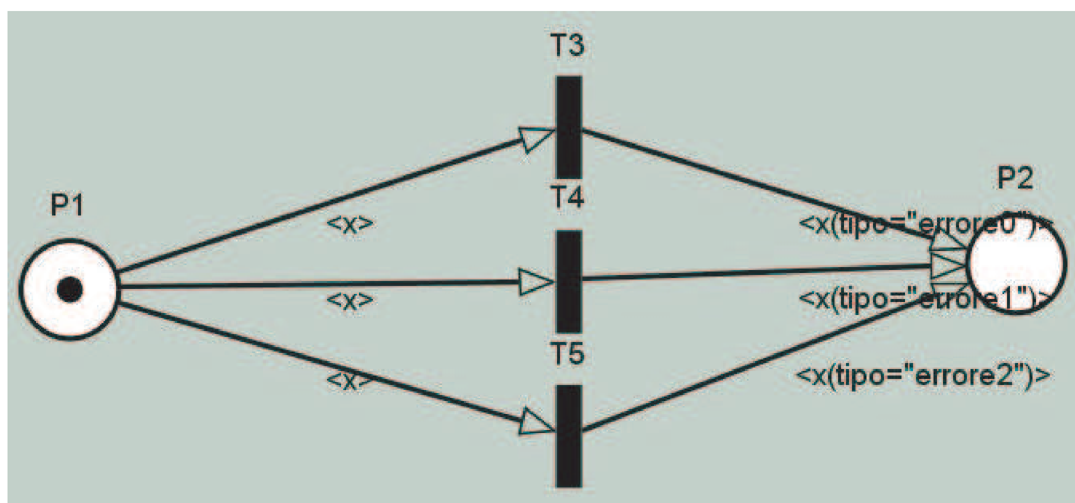

Figura 4.13: Primo componente del sistema

Nella modellazione effettuata, esso è sempre composto da una sola porta di input e da una sola porta di output. Le transizioni sono in numero pari ai failure modes. Sono poi presenti tanti archi di output anch'essi pari in numero ai failure modes, ciascuno recante un'iscrizione relativa al corrispondente failure mode. Analogamente al caso precedentemente descritto di un componente generico, sono ancora le iscrizioni sugli archi di output a permettere, in questo caso, il setting iniziale dei failure modes. Le transizioni non hanno nessuna funzione di guardia, sono invece importanti i firing weights che permettono di specificare le probabilità di avvio dei failure modes del sistema. Considerando un sistema che abbia come primo componente il componente mostrato in Figura 4.13 e presenti un profilo di avvio dei failure modes descritto dalle seguenti probabilità:

failure mode 0 = 0,2
failure mode 1 = 0,1
failure mode 2 = 0,7

allora, per ottenere una modellazione corretta, sarà necessario settare i firing weights delle transizioni T3, T4, T5 rispettivamente a 2,1,7.

Ultimo componente

L'ultimo componente, analogamente al primo, è anch'esso un componente fittizio. Tale componente viene utilizzato per modellare la terminazione dell'applicazione, in particolare l'applicazione termina quando viene trasferito il controllo all'ultimo componente C_M. Esso ha inoltre l'importante ruolo di permettere l'effettiva misurazione delle probabilità $F_A(r,s)$ a livello dell'architettura. In Figura 4.14 è rappresentato il modello dell'ultimo componente del sistema nel caso di un sistema avente tre failure modes e un componente C_{M-1} con due porte di output. I posti P1 e P2 sono le porte di output dell'ultimo componente effettivo mentre l'ultimo componente può concettualmente essere immaginato come composto dai sei archi di input e dalle corrispondenti transizioni temporizzate. Precisando meglio sarebbe meglio sottolineare come, in effetti, anche le porte P1 e P2 devono essere considerate parte dell'ultimo componente. O meglio lo sono solamente nel caso in cui l'ultimo componente effettivo C_{M-1} ha due porte di output entrambe connesse a C_M. Nel caso in cui, ad esempio, il componente C_{M-1} ha due porte di output, una connessa a C_M e

una connessa ad un componente precedente, allora solamente la prima deve essere considerata concettualmente parte di C_M mentre la seconda ha solo il ruolo di porta di output di C_{M-1} e non sarà collegata a transizioni temporizzate.

Le transizioni inserite hanno il compito di permettere, in modo indiretto, la misurazione delle probabilità $F_A(r,s)$. In particolare il meccanismo utilizzato prevede di sollecitare il sistema, a partire dal primo componente, con un numero noto di token e di contare alla fine il numero di token presenti nei posti di output per ciascun differente failure mode. Per realizzare questa operazione si è pensato di inserire un numero opportuno di transizioni temporizzate e di abilitarne il "watch", cioè la funzione che permette di ottenere, al termine della simulazione, un grafico rappresentante il numero di volte che la transizione è stata eseguita.

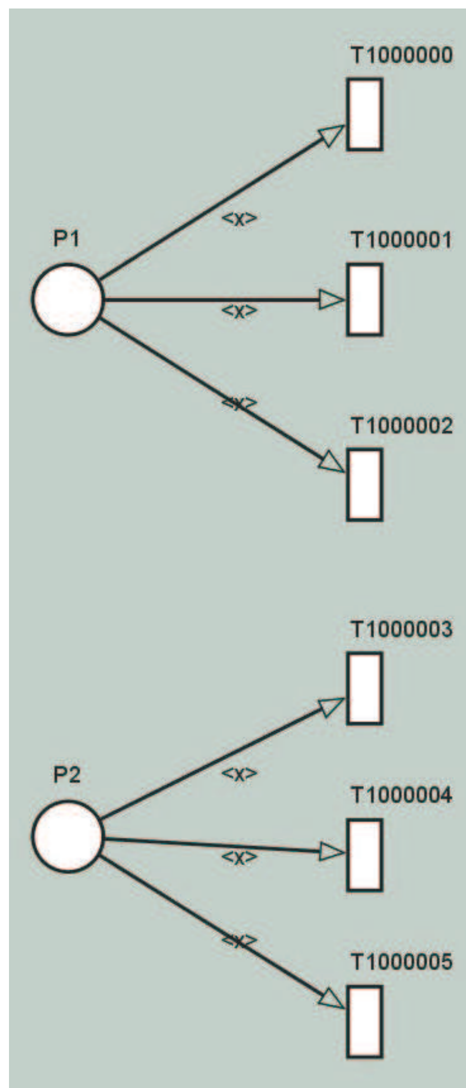

Figura 4.14: Ultimo componente del sistema

In particolare è necessario inserire tante transizioni quanti sono i failure modes per ciascuna porta di output. Bisogna poi dotare ogni transizione di un'adeguata funzione di guardia a seconda del failure mode che si vuole rilevare. Considerando l'esempio in Figura 4.14. la transizione T1000000 ha la seguente funzione di guardia: "x.tipo=="errore0"", la transizione T1000001 ha la funzione di guardia "x.tipo=="errore1" mentre la transizione T1000002 ha la guardia: "x.tipo=="errore2"". Le transizioni T1000003, T1000004, T1000005 hanno rispettivamente le stesse funzioni di guardia. Tutte e sei le transizioni hanno, come descritto in precedenza, il campo "watch" settato a true. Per maggior chiarezza si riporta in Figura 4.15 la finestra che descrive la transizione T1000000. Il ruolo delle transizioni è quello di rilevare i token che presentano il campo "tipo" settato ad un particolare valore di failure mode e quindi è naturale che ne siano presenti una per ciascun tipo di failure mode e siano poi applicate a tutte le porte di output dell'ultimo componente. Bisogna poi contare il numero di volte che ciascuna transizione temporizzata viene eseguita, facendo attenzione a sommare i dati relativi a transizioni corrispondenti, ad esempio, nel caso considerato, per avere il numero di token in output caratterizzati dal failure mode 0 sarà necessario sommare il numero di volte che è stata eseguita la transizione T1000000 con il numero di volte che è stata eseguita la transizione T1000003 e analogamente per le coppie T1000001-T1000004 e T1000002-T1000005 per ottenere, rispettivamente, il numero di token in output caratterizzati dal failure mode 1 e dal failure mode 2. Conoscendo il numero di token iniziali e la loro natura è poi possibile determinare facilmente le probabilità $F_A(r,s)$. Si sono utilizzate transizioni temporizzate solamente perché, a livello implementativo, il tool dava problemi a rilevare il "watch" di transizioni immediate.

Connessione tra componenti

Dopo aver dettagliatamente descritto la modellazione dei componenti generici, del primo componente e dell'ultimo, l'ultimo passo che rimane è mostrare come avvenga la connessione tra essi per poter così ottenere la modellazione definitiva dell'architettura del sistema completo.
Ricordando che la gestione del modello operativo è già stata effettuata nelle fasi precedenti della modellazione ora si è in grado di capirne la ragione. La natura delle Reti di Petri prevede che siano possibili solamente collegamenti tra posti e transizioni e viceversa, ma non tra posti e posti o tra transizioni e transizioni.

Qualified Name	Value
text	T1000000
timeFunction	EXP(1.0)
globalGuard	
localGuard	x.tipo=="errore0"
takeFirst	false
serverType	ExclusiveServer
timeGuard	
specType	Automatic
manualCode	
watch	true
logfileName	
logfileDescription	
logfileExpression	
displayExpression	

Figura 4.15: Transizione T1000000

Per poter quindi connettere la porta di output di un componente con l'opportuna porta di input, e quindi un posto ad un altro posto, si deve passare attraverso una transizione, che non ha un significato modellistico particolare (non presenta né guardie né firing weights diversi da 1) ma ha semplicemente lo scopo di permettere il concreto realizzarsi della connessione. Naturalmente sarà necessario un numero di transizioni pari al numero di porte di output del componente da cui parte la connessione, ciascuna connessa al rispettivo posto di output. Le transizioni avranno poi un'unica connessione in uscita verso il posto di input desiderato modellando quindi il trasferimento del controllo e dei dati (oltre alle informazioni sui failure modes) alla porta di input del rispettivo componente. In Figura 4.16 viene mostrato un esempio che permette di cogliere meglio questi aspetti, in particolare si rappresenta il caso di tre componenti generici con tre failure modes e caratterizzati ciascuno da due porte di output. Si può notare che il primo componente è connesso al secondo mediante la prima porta di output, mentre la seconda lo riconnette a se stesso creando un autoanello. Il secondo componente è invece connesso al terzo con la prima porta di output e al primo con la seconda porta di output. L'immediatezza della modellazione in questa fase non richiede ulteriori dettagli. È invece opportuno soffermarsi un'ultima volta sulla gestione del modello operativo per mostrare l'adeguatezza della scelta precedentemente fatta.

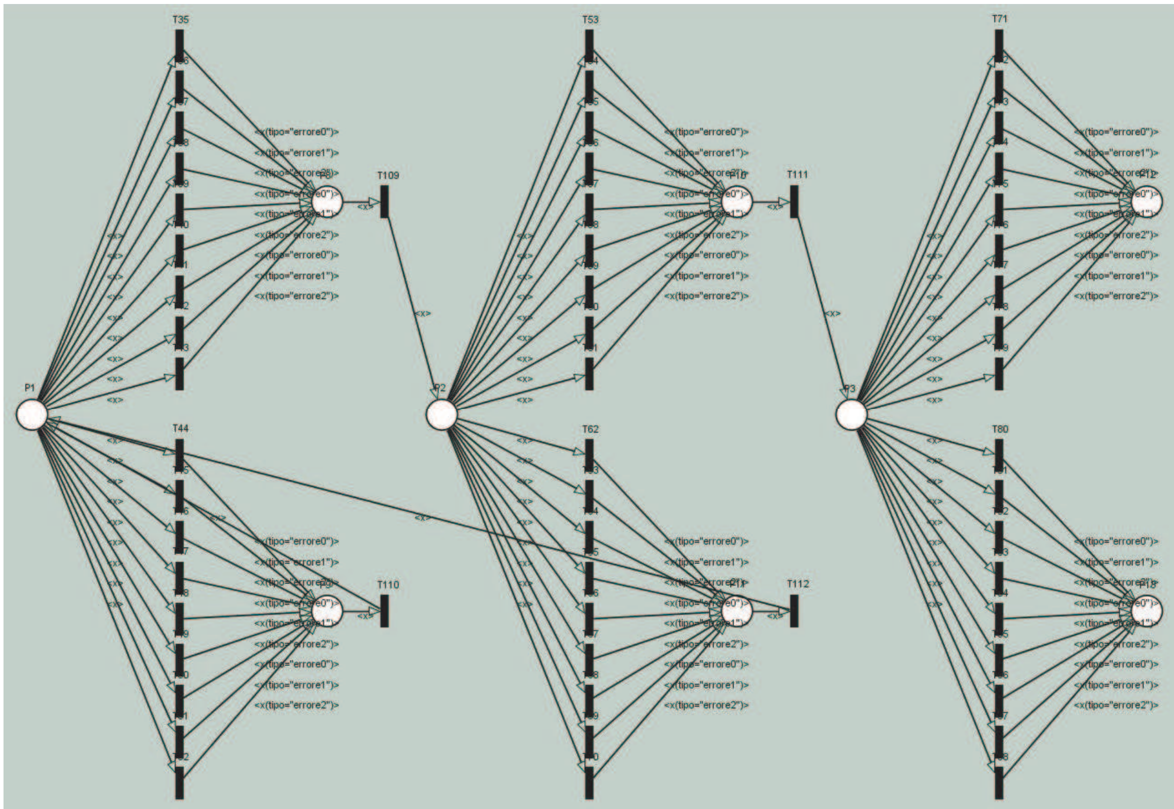

Figura 4.16: Connessioni tra componenti

Considerando l'esempio in Figura 4.16 risulta chiaro che le transizioni fittizie aggiunte per connettere posti di output a posti di input non sono in realtà mai in concorrenza tra di loro. Infatti, tali transizioni, sono abilitate solamente se è presente almeno un token nel corrispondente posto di input. Tuttavia, per come è stato realizzato il modello, la presenza di un token in un posto di input di transizioni di questo tipo significa che la scelta su quale tipo di failure mode propagare è già stata effettuata e ci si rende conto che anche quella legata al modello operativo deve essere già stata fatta. Infatti queste transizioni hanno un'unica connessione che non permette di discriminare la porta di input di destinazione. Quindi per non tralasciare le informazioni relative al modello operativo è necessario considerarle nel punto in cui sia garantita un'effettiva concorrenza tra transizioni in grado di guidare il flusso di informazioni e controllo verso strade differenti, tale possibilità è proprio assicurata dalla scelta modellistica descritta in precedenza, che permette di sfruttare opportunamente i firing weights per condensare la gestione di modello operativo e modello di failure.

4.3 Modellazione di un semplice sistema a componenti

In questa sezione si applica la modellazione mostrata nella sezione precedente al semplice sistema a componenti introdotto nella sezione 4.1. Dopo aver descritto la realizzazione del modello, si mostrerà mediante l'utilizzo di un test di verifica d'ipotesi, la bontà dello stesso nel garantire risultati verosimili relativamente alle proprietà di affidabilità. Lo scopo di questa sezione non è quello di descrivere in modo formale la corrispondenza tra modello ed esempio (aspetto già ampiamente trattato nella sezione precedente), piuttosto si vuole mostrare il processo che ha portato alla creazione del modello mettendo in rilievo gli aspetti critici a cui prestare maggiore attenzione.
Inizialmente si considera il sistema presentato nella sezione 4.1 e si nota che esso è composto da tre componenti effettivi. In Figura 4.17 vengono riportati i tre componenti rilevanti del modello. Il primo modella il dispatcher (C_1), il secondo il server principale (C_2) e il terzo il server di guardia (C_3).

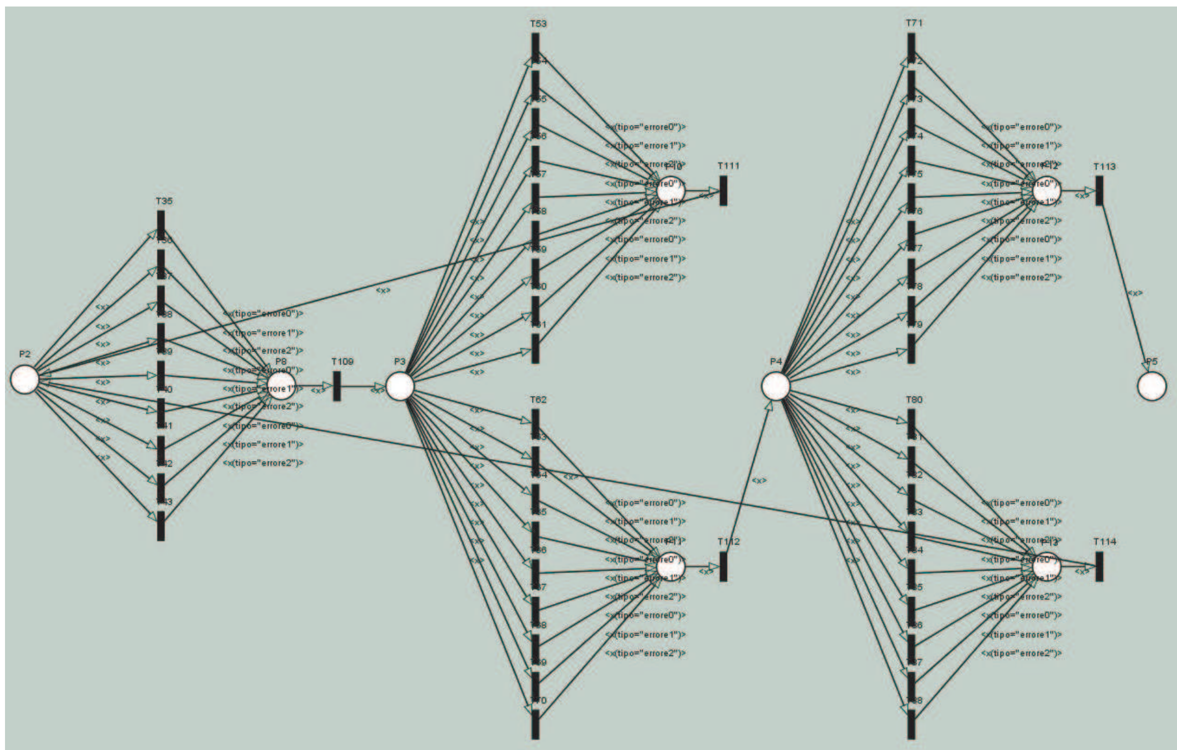

Figura 4.17: Componenti effettivi del sistema

Dalla Figura 4.17 si nota subito che il modello presenta tre failure modes coerentemente al sistema che si vuole descrivere. In questa fase bisogna fare attenzione a dotare ciascun

componente del numero corretto di porte di output. Successivamente si procede a settare funzioni di guardia e firing weights delle transizioni non fittizie, cioè di tutte le transizioni tranne quelle utilizzate per garantire le connessioni tra componenti. Settando i firing weights non bisogna dimenticare di fare attenzione a gestire, oltre al modello di failure, il modello operativo secondo le procedure precedentemente descritte. A titolo di esempio si riportano di seguito i firing weights delle transizioni relative al secondo componente effettivo così da poter concretamente mostrare la gestione coordinata di modello operativo e modello di failure. Le transizioni della prima porta di output del componente C_2 sono quelle comprese tra T53 e T61 ed hanno i seguenti firing weights:

T53 = 576

T54 = 117

T55 = 207

T56 = 27

T57 = 756

T58 = 117

T59 = 27

T60 = 117

T61 = 756

Le transizioni relative alla seconda porta di output del componente C_2 sono quelle comprese tra T62 e T70 ed hanno i seguenti firing weights:

T62 = 64

T63 = 13

T64 = 23

T65 = 3

T66 = 84

T67 = 13

T68 = 3

T69 = 13

T70 = 84

Andando ora a riconsiderare la Figura 4.1 che descrive l’architettura del sistema e la Tabella 4.1 che comprende i valori delle proprietà di affidabilità dei componenti, è quasi immediato rendersi conto del significato dei firing weights. Infatti il modello operativo prevede di trasferire controllo e dati, attraverso la prima porta di output, al componente C_1 con una probabilità pari a 0,9 e prevede di trasferire controllo e dati, attraverso la seconda porta di output, al componente C_3 con una probabilità 0,1. Conseguentemente i firing weights sono stati settati in modo da rispettare tali probabilità e in particolare si nota che i firing weights delle transizioni T53-T61 sono ottenuti esattamente moltiplicando per 9 i firing weights delle transizioni T62-T70 che a loro volta sono stati ricavati in accordo con le proprietà di affidabilità del componente C_2 specificate in Tabella 4.1. Analoghe considerazioni valgono per i firing weights delle altre transizioni del modello. Dopo aver correttamente definito funzioni di guardia e firing weights, bisogna fare attenzione a connettere i componenti con gli archi secondo lo schema fornito in Figura 4.1.
Una volta realizzato il modello dei tre componenti effettivi, si devono aggiungere i due componenti fittizi, quello iniziale e quello finale, essi non presentano particolarità che non siano state messe in rilievo nella sezione precedente e, pertanto, non ci si dilunga nella loro descrizione limitandosi a riportarli, rispettivamente, in Figura 4.18 e in Figura 4.19. Infine, in Figura 4.20, si riporta il modello del sistema completo.

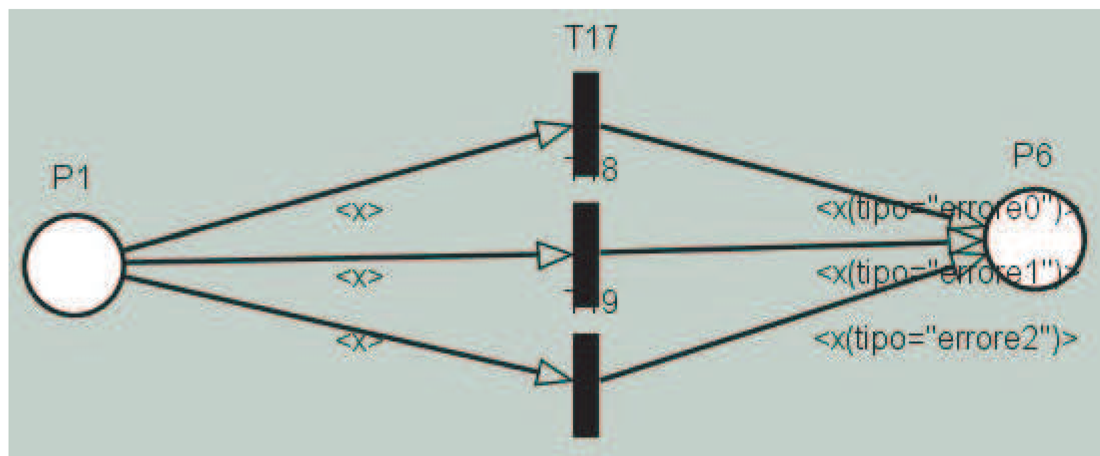

Figura 4.18: Primo componente del sistema

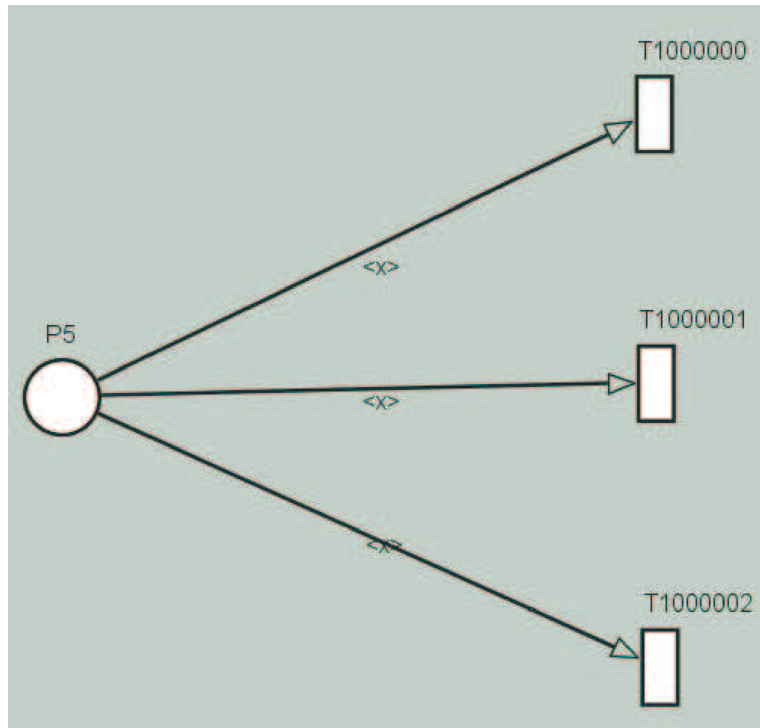

Figura 4.19: Ultimo componente del sistema

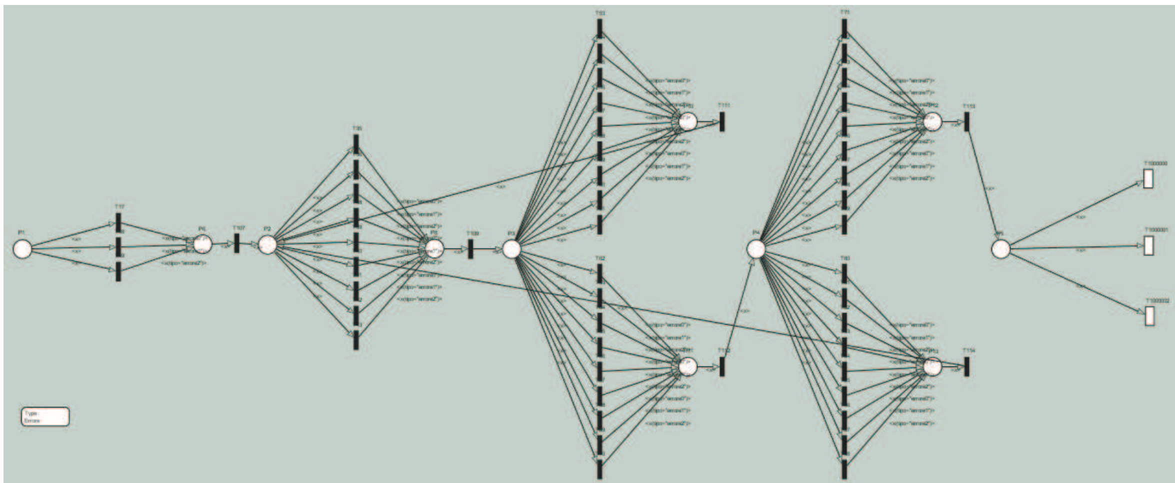

Figura 4.20: Modello del sistema completo

4.3.1 Test di ipotesi per valutare l'affidabilità

In questa sezione, l'ultima del capitolo, si mostra il calcolo dell'affidabilità del sistema e si valuta la bontà del risultato ottenuto, effettuando un test di ipotesi che preveda di confrontare il dato empirico con il valore di affidabilità fornito in [3] e calcolato per mezzo di formule analitiche (che per comodità verrà in seguito chiamato "valore teorico").
Prima di impostare il test occorre brevemente mostrare come si è calcolato il valore empirico di affidabilità. Precedentemente si è mostrato il ruolo dell'ultimo componente nell'effettuare le "misure" di probabilità. Si ricorda la definizione fornita nella Sezione 3.2.2 di probabilità $F_A(r,s)$ a livello dell'architettura:

$F_A(r,s)(0 \leq r \leq N, 0 \leq s \leq N)$, dove ogni $F_A(r,s)$ è così definito: $F_A(r,s) = Pr\{A$ termina con il failure mode s \ A è stata attivata con il failure mode r$\}$

In questo caso si vuole valutare l'**affidabilità** del sistema e cioè la probabilità $F_A(0,0)$.

Si è quindi sollecitato il sistema inserendo 100 token nel primo componente tutti con l'attributo "tipo" settato al valore "errore0", si è poi eseguita la simulazione e si è rilevato il numero di token che alla fine avevano l'attributo "tipo" settato ancora al valore "errore0" (per farlo si è sfruttato il "watch" delle transizioni come descritto nella Sezione 4.2.2). Tale numero fornisce la probabilità $F_A(0,0)$ e cioè la proprietà di affidabilità del sistema. Per completezza, va rilevato che, per ragioni implementative, si sono in realtà introdotti un posto ed una transizione fittizi iniziali per sollecitare il sistema perché il tool non permetteva di settare gli attributi iniziali dei token, per farlo si è quindi fatto ricorso ad un'opportuna iscrizione sull'arco di uscita della transizione fittizia introdotta. In Figura 4.21 è riportato quanto descritto.

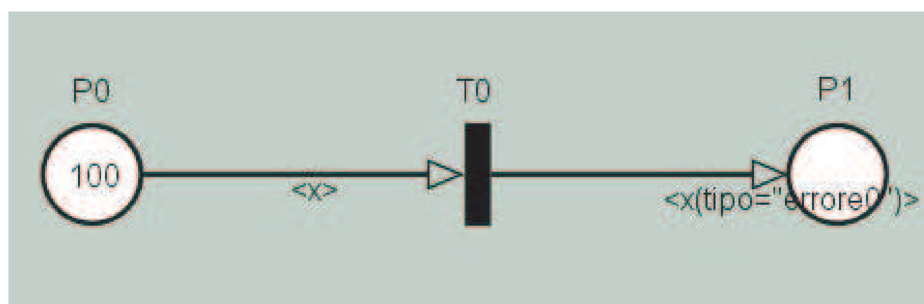

Figura 4.21: Artificio implementativo

Questo artificio non ha alcun valore concettuale, ma lo si è riportato per spiegare tutti i passi compiuti per poter ottenere la misura di probabilità voluta. Il posto P0 è il posto fittizio e contiene 100 token di tipo "Errore" aventi l'attributo "tipo" non settato. La transizione T0 è la transizione fittizia e non contiene né guardie né firing weights particolari. P1 rappresenta il posto di input del primo componente del sistema e l'iscrizione sull'arco di output della transizione T1 ha il ruolo di settare l'attributo "tipo" di tutti i 100 token al valore "errore0". Naturalmente se si fosse voluta una differente configurazione iniziale sarebbe bastato modificare l'iscrizione sull'arco di output. È da rilevare che sono possibili anche configurazioni iniziali "miste", cioè che prevedano di sollecitare il sistema con token aventi failure modes differenti. Per ottenerle basta introdurre tante coppie di posti-transizioni fittizie quanti sono i differenti failure modes dei token iniziali, si devono poi connettere tali coppie al posto di input del primo componente facendo attenzione a settare le iscrizioni sugli archi di output delle transizioni delle varie coppie in modo da rappresentare tutti i failure modes voluti.

Precisato questo aspetto si ritorna ora alla descrizione dell'esperimento. In particolare sono stati raccolti 20 dati, ricavati da altrettante simulazioni e riportati in Tabella 4.2. Successivamente è stata fatta la media aritmetica di tali dati ottenendo il valore 0,6115. Si è poi proceduto ad impostare il test di ipotesi. Non si richiama ora la teoria dei test di ipotesi limitandosi a ricordare che essi vengono ampiamente utilizzati in ambito scientifico/ingegneristico per poter verificare l'attendibilità di un'ipotesi attraverso l'applicazione del metodo sperimentale, volta a valutare se la realtà effettivamente osservata si accorda con la deduzione.
In particolare si è impostato un test di ipotesi parametrico per la media di popolazioni gaussiane con varianza incognita, quindi un t-test. Il test prevede di confrontare il valore sperimentale ora ricavato con il valore teorico ottenuto da [3].

Le due ipotesi nel dettaglio sono:

$H_0: \mu = \mu_0$ versus $H_1: \mu \neq \mu_0$

dove μ_0 rappresenta il valore teorico fornito in [3], in particolare si ha:

$\mu_0 = 0{,}6163$

Simulazione	**$F_A(0,0)$**
1	0,62
2	0,58
3	0,64
4	0,66
5	0,62
6	0,62
7	0,58
8	0,66
9	0,56
10	0,68
11	0,62
12	0,67
13	0,55
14	0,58
15	0,60
16	0,58
17	0,61
18	0,59
19	0,60
20	0,61

Tabella 4.2: Dati empirici ottenuti dalla simulazione del sistema

Il livello di significatività del test α è stato fissato al 95% e quindi si ha:

$\alpha = 0,95$

Il test considerato prevede di rifiutare l'ipotesi nulla H_0 se:

$|X - \mu_0| / (s / \sqrt{n}) \geq t_{n-1}(1 - \alpha/2)$

Dove i simboli utilizzati hanno il seguente significato:

- μ_0 e α sono già stati definiti.
- X è la media campionaria delle osservazioni sperimentali: $X = \sum^{n}_{i=1} X_i / n$. Sostanzialmente è già stata ricavata in precedenza quando si è calcolata la media aritmetica dei dati sperimentali ed è X = 0,6115.
- s, la radice quadrata della varianza campionaria, è ricavato a partire dalla varianza campionaria che è a sua volta uno stimatore della varianza non nota. La varianza campionaria è definita a partire dalle osservazioni: $S^2 = \sum^{n}_{i=1}(X_i - X)^2 / (n - 1)$. Nel caso in esame è s = 0,0366.
- n rappresenta la numerosità del campione, in questo caso il numero di volte che si è realizzata la simulazione ed è n = 20.
- $t_{n-1}(1 - \alpha/2)$ rappresenta il quantile della funzione di ripartizione t di Student con n – 1 gradi di libertà.

Eseguendo il test, consultando le tavole della t di Student, e inserendo i valori numerici nella regola di rifiuto prima citata, si ottiene che la disuguaglianza che rappresenta la regola di rifiuto dell'ipotesi nulla H_0 non è verificata e quindi si può concludere che il valore empirico trovato è una buona rappresentazione di quello teorico al livello di significatività fissato.

Questo risultato mostra l'efficacia del modello in quanto il calcolo dell'affidabilità del sistema dà un risultato coerente con il valore teorico atteso.
Più in generale, il test di ipotesi effettuato, mostra la correttezza dell'approccio seguito nella modellazione, e la bontà delle scelte modellistiche effettuate nella creazione dei modelli per descrivere failure modes multipli nei sistemi a componenti.

CAPITOLO 5

Misurazioni e analisi di complessità

Dopo aver descritto la modellazione del problema e averne mostrato l'efficacia mediante un esempio applicativo nel capitolo precedente, in questo, si vuole valutarne l'efficienza. La prima parte del capitolo contiene una breve descrizione del tool utilizzato, TimeNET, e delle tecniche per realizzare modelli generici, successivamente si illustra la scelta dei parametri di simulazione. La sezione conclusiva è dedicata alla valutazione delle prestazioni del tool, per realizzarla si è fatto ricorso alla simulazione di numerosi sistemi ottenuti variando numero di componenti, di porte di output e di failure modes, l'analisi dei tempi di costruzione del modello e di esecuzione permette di mostrare l'andamento delle prestazioni del tool. Infine viene presentata un'analisi di complessità raffrontandola con i dati empirici ricavati.

5.1 Descrizione tool e generazione dei modelli

Il tool impiegato per la modellazione e la simulazione è TimeNET versione 4.02. In questa sezione non si vuole dare una descrizione dettagliata e rigorosa del tool, per la quale si rimanda al manuale di utilizzo dello stesso [2], in quanto gli aspetti rilevanti per la creazione dei modelli sono stati già mostrati implicitamente nei capitoli precedenti (le numerose figure sono tutte tratte dall'interfaccia di TimeNET). Lo scopo di questa sezione è piuttosto quello di mostrare una panoramica del tool evidenziando le differenze tra la creazione di modelli elementari e la creazione di modelli di complessità generica.

5.1.2 Interfaccia grafica e creazione di modelli elementari

Il tool presenta un'intuitiva interfaccia grafica per la creazione dei modelli. Sostanzialmente la creazione di modelli di complessità ridotta è semplice e richiede solamente la conoscenza delle Reti di Petri stocastiche colorate. I vari elementi posti, transizioni, archi sono infatti rappresentati con l'usuale convenzione grafica ed è sufficiente selezionarli dalla parte inferiore dell'interfaccia, trascinarli nella parte superiore e comporre il modello. Per la specifica delle proprietà delle transizioni, dei tipi di token e di altri dettagli basta modificare le opzioni desiderate nelle apposite finestre (ne sono state mostrate alcune nel Capitolo 4).

5.1.3 XML e creazione di modelli di complessità generica

L'utilizzo dell'interfaccia grafica, semplice ed intuitivo, accennato nella sezione precedente, può essere sufficiente per creare modelli elementari, nel nostro caso specifico modelli dotati di un numero ridotto di componenti, di porte di output e di failure modes. Tuttavia all'aumentare di questi parametri non è conveniente realizzare manualmente i modelli. Per superare tale difficoltà si è fatto ricorso alla creazione di un apposito software in grado di generare modelli di complessità generica. I file che descrivono i modelli in TimeNET sono file XML con una particolare struttura specificata dal relativo schema. A titolo di esempio in Figura 5.1 è riportata una parte del file XML corrispondente al modello di esempio presentato nella Sezione 4.3. È stato quindi necessario studiare la struttura di tali file per poter essere in grado di capire la corrispondenza tra informazioni nel file XML e relativi elementi del modello. Una volta terminata tale fase, si è proceduto a creare un software, utilizzando il linguaggio di programmazione Java, che fosse in grado di generare modelli in maniera automatica permettendo all'utente di specificare il numero di componenti del sistema, il numero di porte di output di ogni componente e il numero di failure modes. In questa sezione non si vuole entrare nella descrizione dettagliata della realizzazione del software generatore di modelli preferendo riportare in Appendice l'intero codice sorgente adeguatamente commentato.

5.2 Simulazioni e setting dei parametri

In questa sezione si vuole mostrare come sono state eseguite le simulazioni dei modelli con il tool TimeNET soffermandosi in particolare sul setting dei parametri significativi, tralasciando invece i parametri privi di particolare interesse, per i quali si rimanda al manuale utente del tool [2].Verrà inoltre accennata l'architettura del sistema su cui poggiano le simulazioni eseguite.


```
- <place capacity="0" id="0.1" initialMarking="" queue="Random" tokentype="Errore" type="node" watch="false">
    <graphics orientation="0" x="190" y="510" />
  - <label id="0.1.0" text="P1" type="text">
      <graphics x="-10" y="-40" />
    </label>
  </place>
- <place capacity="0" id="0.2" initialMarking="" queue="Random" tokentype="Errore" type="node" watch="false">
    <graphics orientation="0" x="691" y="501" />
  - <label id="0.2.0" text="P2" type="text">
      <graphics x="-10" y="-40" />
    </label>
  </place>
- <place capacity="0" id="0.3" initialMarking="" queue="Random" tokentype="Errore" type="node" watch="false">
    <graphics orientation="0" x="1190" y="510" />
  - <label id="0.3.0" text="P3" type="text">
      <graphics x="-10" y="-40" />
    </label>
  </place>
```


Figura 5.1: Parte del file XML di un semplice modello

Dopo aver completato la creazione del modello e verificato l'assenza di errori, per eseguire la simulazione si deve selezionare il menu "Simulation" nella parte superiore dell'interfaccia grafica e successivamente la voce "Stationary Simulation". Una volta compiute queste operazioni si aprirà la finestra mostrata in Figura 5.2. In questa finestra è possibile settare i vari parametri della simulazione, quelli maggiormente significativi sono stati evidenziati. Si è scelto di settare l'intervallo di confidenza al 95% e l'errore relativo massimo al 5%. I concetti di intervallo di confidenza e di errore relativo, propri della teoria statistico-probabilistica, possono qui essere interpretati pensando al fatto che, con i parametri così settati, se vengono eseguite 100 simulazioni in media almeno 95 di esse avranno un risultato che si discosta al massimo del 5% da quello corretto. La scelta di settare i parametri a questi valori è sembrata ragionevole in quanto è un buon compromesso tra accuratezza dei risultati e complessità delle simulazioni.

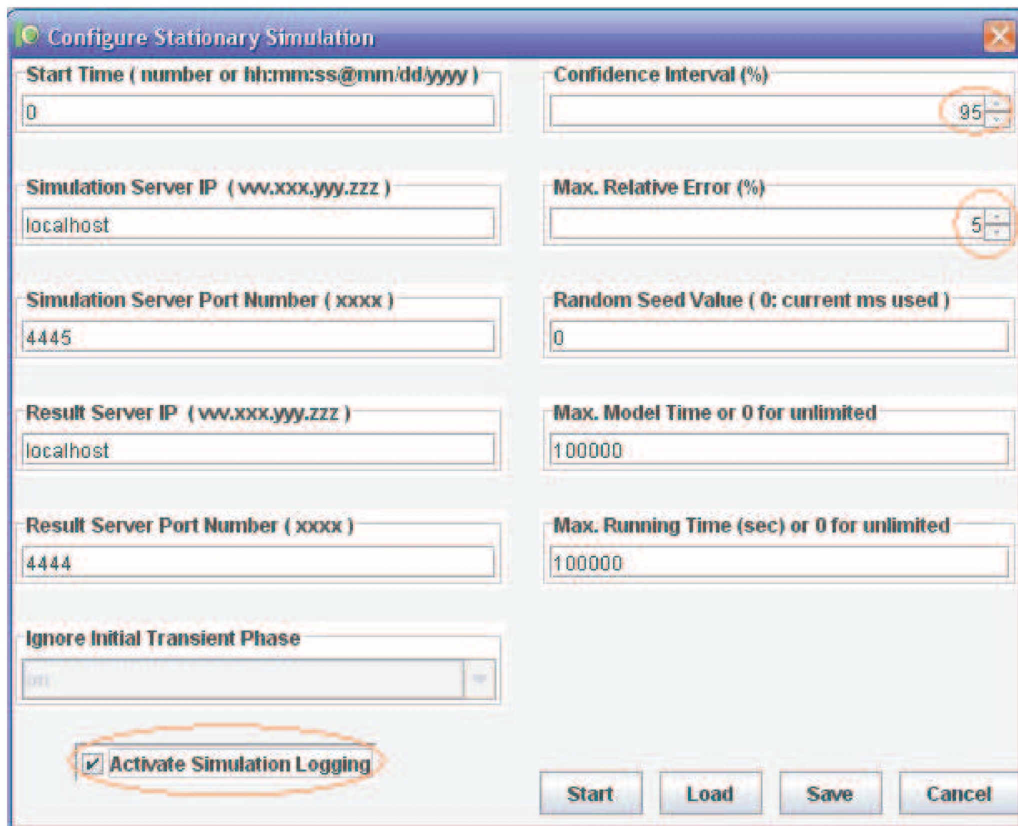

Figura 5.2: Parametri della simulazione

Va tuttavia sottolineato come le misurazioni eseguite, e mostrate nella sezione successiva, siano state ripetute anche settando l'intervallo di confidenza al 99% e l'errore relativo massimo al 1% e in tale situazione i tempi di simulazione siano rimasti pressoché inalterati (e comunque sempre dello stesso ordine di grandezza) di quelli precedenti. Questo risultato da un lato mostra che il simulatore scala bene rispetto a tali parametri, dall'altro dà un'ulteriore conferma della bontà delle scelte modellistiche effettuate, infatti il test di ipotesi realizzato nella Sezione 4.3.1, che dava risultati relativi all'affidabilità coerenti con quelli attesi, è stato realizzato impostando il simulatore con un intervallo di confidenza al 95% e un errore relativo massimo al 5% e cioè le rispettive simulazioni non sono state eseguite estremizzando questi due valori, quindi i risultati ottenuti non derivano da simulazioni eccessivamente intolleranti e la loro adeguatezza è, di conseguenza, testimone di una corretta modellazione del problema in esame.

Nella finestra mostrata in Figura 5.2 è stata evidenziata anche una terza opzione, essa abilita la generazione di un file di log ogni volta che viene eseguita la simulazione, tale file, in particolare, contiene le scelte che vengono compiute dal simulatore in presenza di

transizioni concorrenti e, esaminandolo, ci si può rendere conto della correttezza della propagazione delle informazioni relative a controllo e dati, e ai failure modes.

Per farsi un'idea dell'architettura del sistema sottostante alle simulazioni, è possibile fare riferimento all'illustrazione in Figura 5.3, tratta dal manuale utente di TimeNET. Il tool, per simulare modelli realizzati con Reti di Petri stocastiche colorate, utilizza una struttura basata su componenti remoti [2]. Questa struttura permette, potenzialmente, un'esecuzione distribuita dei differenti componenti della simulazione: il server di simulazione, il monitor dei risultati, e il database (che non viene tuttavia mai utilizzato a meno di avere modelli con marcature iniziali molto complesse). Ognuno di questi componenti può essere eseguito sul computer locale o su un altro computer della rete. Il secondo aspetto rilevante riguarda l'utilizzo combinato di ambienti Java/XML e C/C++. Come è possibile evincere anche dall'illustrazione stessa, il primo, proprio dell'interfaccia grafica, viene utilizzato per garantire interoperabilità, il secondo invece, proprio dei componenti preposti alla simulazione, viene utilizzato per garantire velocità di esecuzione.

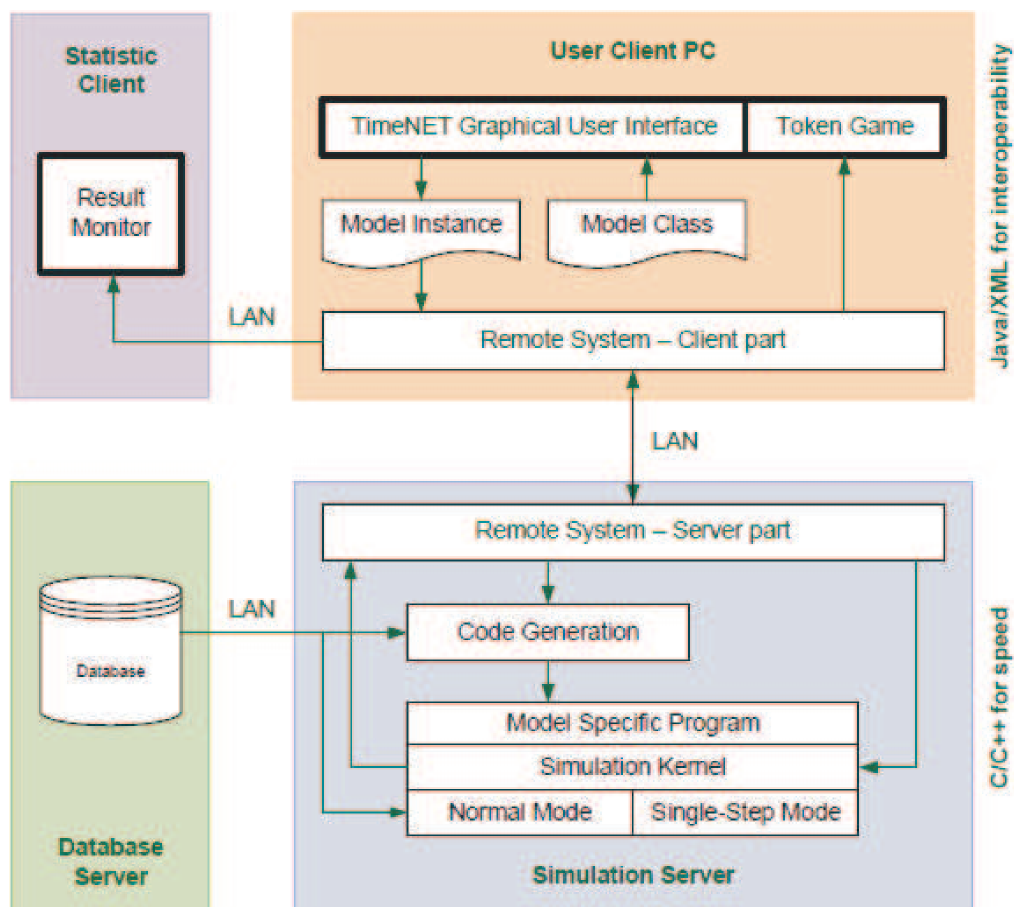

Figura 5.3: Struttura dei componenti di TimeNET

5.3 Misurazioni e analisi di complessità

In questa sezione si riportano le misurazioni effettuate sui modelli generati. In particolare, per ogni modello simulato, sono stati rilevati due dati: il tempo di building totale del modello e il tempo di simulazione. La macchina utilizzata per eseguire le simulazioni è una macchina con prestazioni assicurate in quanto non erano presenti processi che potessero rallentare l'esecuzione delle simulazioni. Essa comprende un processore Intel Xeon E5530 2.40GHz, è dotata di 4,00GB di memoria RAM e il sistema operativo utilizzato è Windows Server 2008 R2 Standard a 64 bit. I modelli sono stati generati con il software generatore di modelli precedentemente descritto e comprendono tutte le possibili combinazioni di sistemi che abbiano dai 5 ai 9 componenti, dalle 2 alle 4 porte di output, dai 2 ai 4 failure modes, assumendo che nei modelli considerati ogni componente sia caratterizzato dallo stesso numero di porte di output. Il software è stato realizzato in modo da generare modelli che presentino, con frequenza casuale, cicli di lunghezza variabile e ogni porta di output può essere connessa ad input diversi rispetto alle altre porte di output dello stesso componente, sono state fatte tali scelte in modo da avere una situazione potenzialmente ad elevata complessità, e quindi il più generale possibile. I dati raccolti sono riportati in Tabella 5.1.

Componenti	Output	Failure mode	Building (s)	Simulazione (s)
5	2	2	81	0
5	2	3	165	0
5	2	4	336	1
5	3	2	124	1
5	3	3	290	0
5	3	4	714	0
5	4	2	165	1
5	4	3	456	0
5	4	4	1386	0
6	2	2	95	0
6	2	3	201	0
6	2	4	442	1
6	3	2	161	1
6	3	3	377	0

6	3	4	1070	1
6	4	2	208	0
6	4	3	629	0
6	4	4	2213	0
7	2	2	123	1
7	2	3	259	1
7	2	4	614	1
7	3	2	179	0
7	3	3	486	0
7	3	4	1544	0
7	4	2	259	1
7	4	3	889	0
7	4	4	3485	0
8	2	2	130	1
8	2	3	304	0
8	2	4	794	0
8	3	2	201	1
8	3	3	623	0
8	3	4	2216	0
8	4	2	314	1
8	4	3	1239	0
8	4	4	5269	2
9	2	2	141	0
9	2	3	364	1
9	2	4	1029	1
9	3	2	242	0
9	3	3	836	1
9	3	4	3112	1
9	4	2	382	0
9	4	3	1654	1
9	4	4	7647	3

Tabella 5.1: Tempi di building e di simulazione dei modelli

Osservando la tabella, il primo aspetto che è subito evidente, è rappresentato dai ridottissimi tempi di simulazione anche nei casi più complessi. Il tempo infatti oscilla sempre tra i valori di 0 e 1 secondo (dove il valore 0 non sta ad indicare un tempo nullo ma un tempo inferiore ad 1 secondo) fatta eccezione per le due simulazioni corrispondenti ai due modelli più complessi, quelli caratterizzati da 8 e 9 componenti, ciascuno con 4 porte di output e 4 failure modes. In essi il tempo di simulazione è rispettivamente di 2 e 3 secondi ed è comunque ridotto considerata la complessità dei modelli. I dati relativi al tempo di building dei modelli sono invece maggiormente articolati e necessitano di un'analisi più approfondita che viene realizzata con l'aiuto di opportuni grafici. I grafici dal 5.1 al 5.3 trattano il caso di sistemi con un numero di porte di output fissato. In ascissa presentano il numero di failure modes del sistema e in ordinata il tempo di building. Per ognuno di questi grafici linee di colore differente indicano sistemi con un differente numero di componenti secondo la legenda presente accanto al grafico. Sostanzialmente tali grafici permettono di vedere come varia il tempo di building in relazione al numero di failure modes e all'aumentare del numero di componenti del sistema. Si può rilevare un andamento del tempo di building di tipo sempre più prossimo all'esponenziale all'aumentare dei failure modes (maggiormente pronunciato al crescere del numero dei componenti) e un incremento più che lineare all'aumentare del numero di componenti. Confrontando i tre grafici, all'aumentare del numero di porte di output, si rileva un incremento più che lineare (maggiore di quello che si ha all'aumentare del numero dei componenti ma non esponenziale) del tempo di building. I grafici presentano lo stesso andamento qualitativo ma è possibile rendersi conto dell'incremento guardando i valori temporali sulla scala delle ordinate.

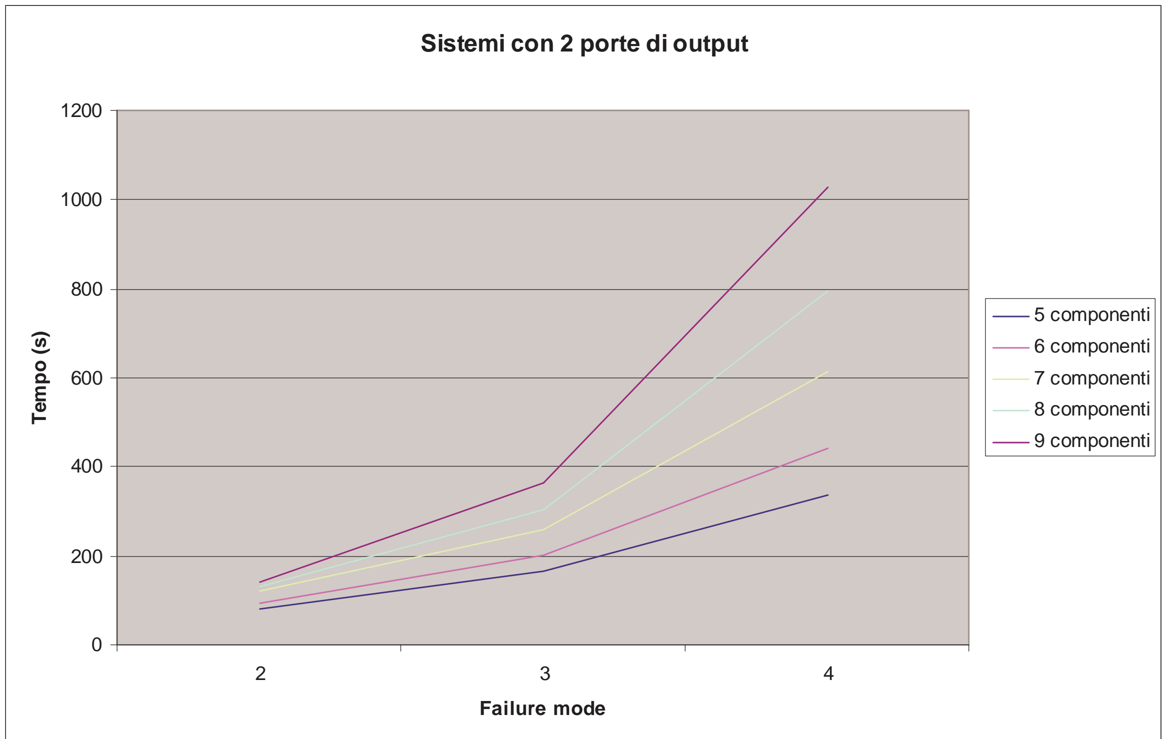

Grafico 5.1: Sistemi con 2 porte di output

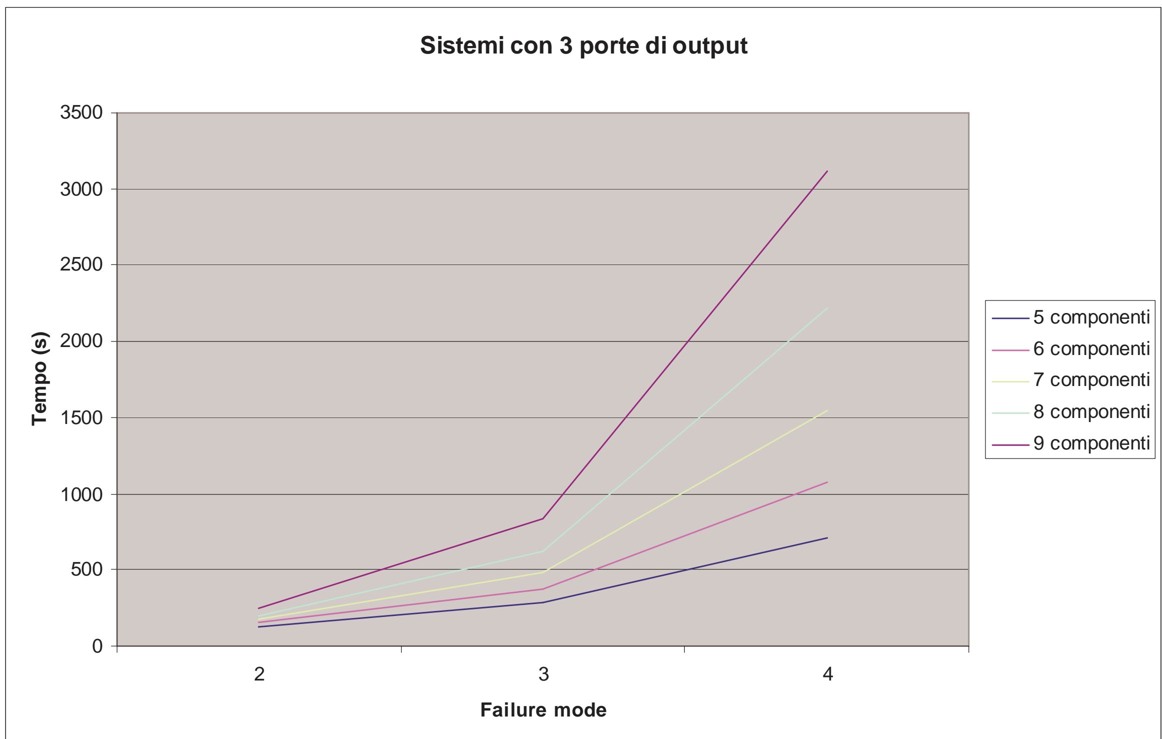

Grafico 5.2: Sistemi con 3 porte di output

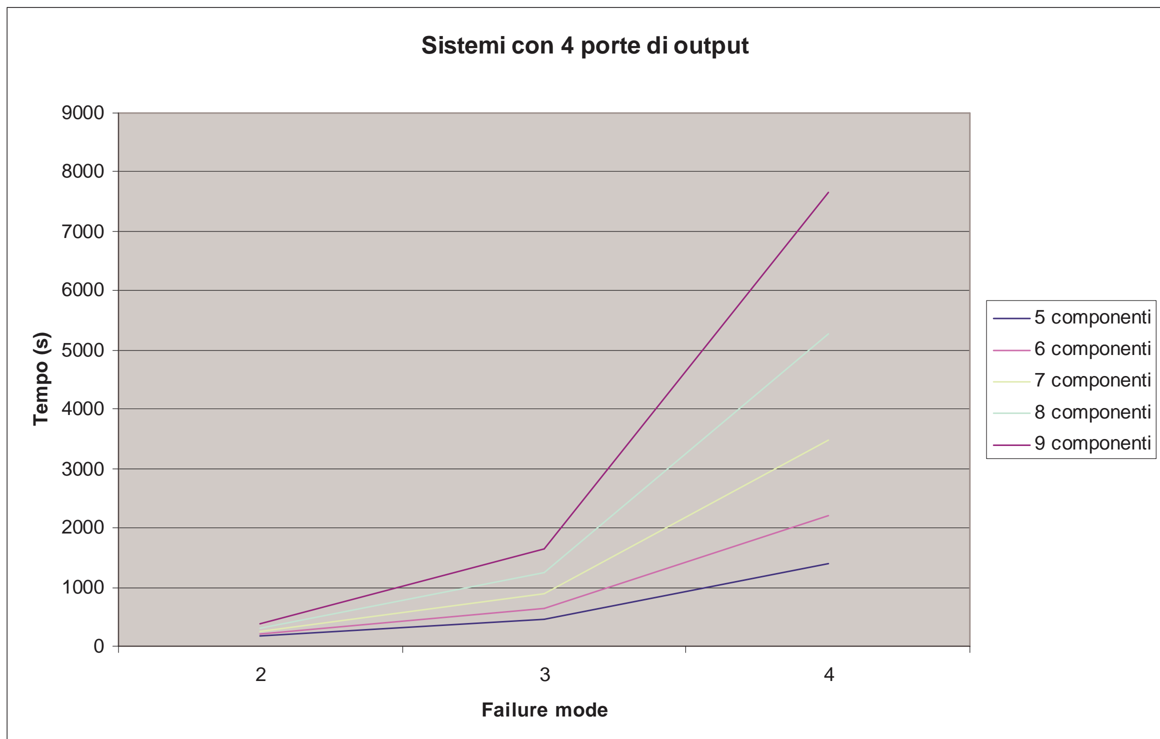

Grafico 5.3: Sistemi con 4 porte di output

La prossima serie di grafici, dal 5.4 al 5.8, mostra il caso di sistemi con un numero di componenti fissato evidenziando l'andamento del tempo di building all'aumentare dei failure modes, in essi linee di colore differente indicano sistemi con un differente numero di porte di output secondo quanto precisato dalla leggenda accanto al grafico. Questi grafici sono sostanzialmente una rielaborazione dei precedenti e non presentano un maggior contenuto informativo, ma sono stati realizzati per mostrare ancora una volta, che il fattore che determina l'incremento più significativo del tempo di building è l'aumentare del numero di failure modes del sistema. I grafici hanno in ascissa e ordinata le stesse grandezze dei precedenti, e mostrano l'andamento esponenziale del tempo di building all'aumentare dei failure modes (maggiormente pronunciato al crescere del numero di porte di output) e un incremento più che lineare (ma non esponenziale) all'aumentare del numero di porte di output. Confrontando tra loro i grafici, è possibile rilevare un incremento più che lineare (ma minore di quello che si ha all'aumentare del numero di porte di output) del tempo di building all'aumentare del numero di componenti del sistema, completando così la dualità dell'analisi rispetto al precedente insieme di grafici.

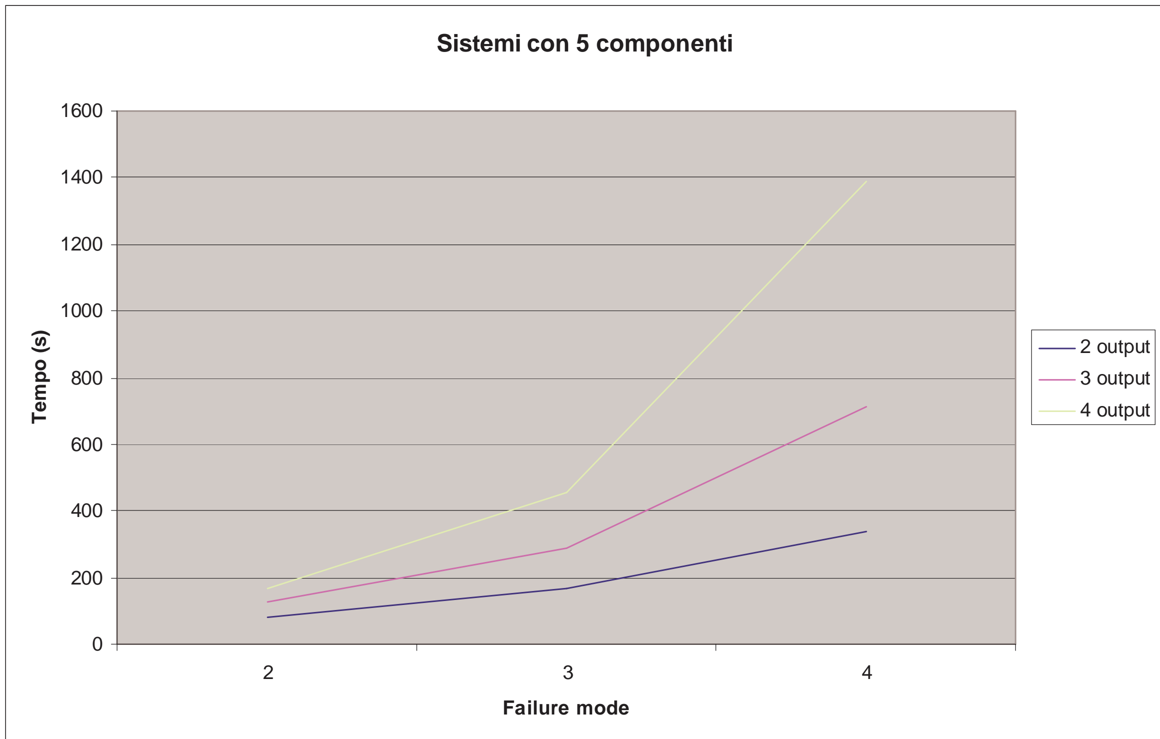

Grafico 5.4: Sistemi con 5 componenti

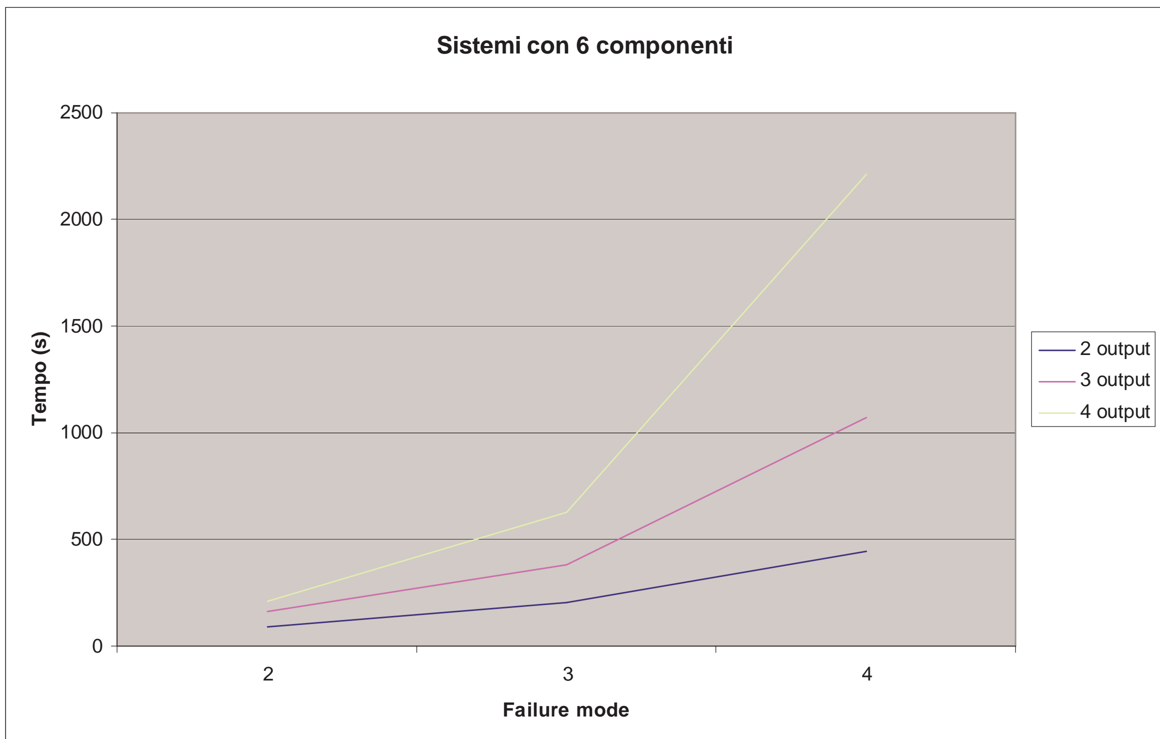

Grafico 5.5: Sistemi con 6 componenti

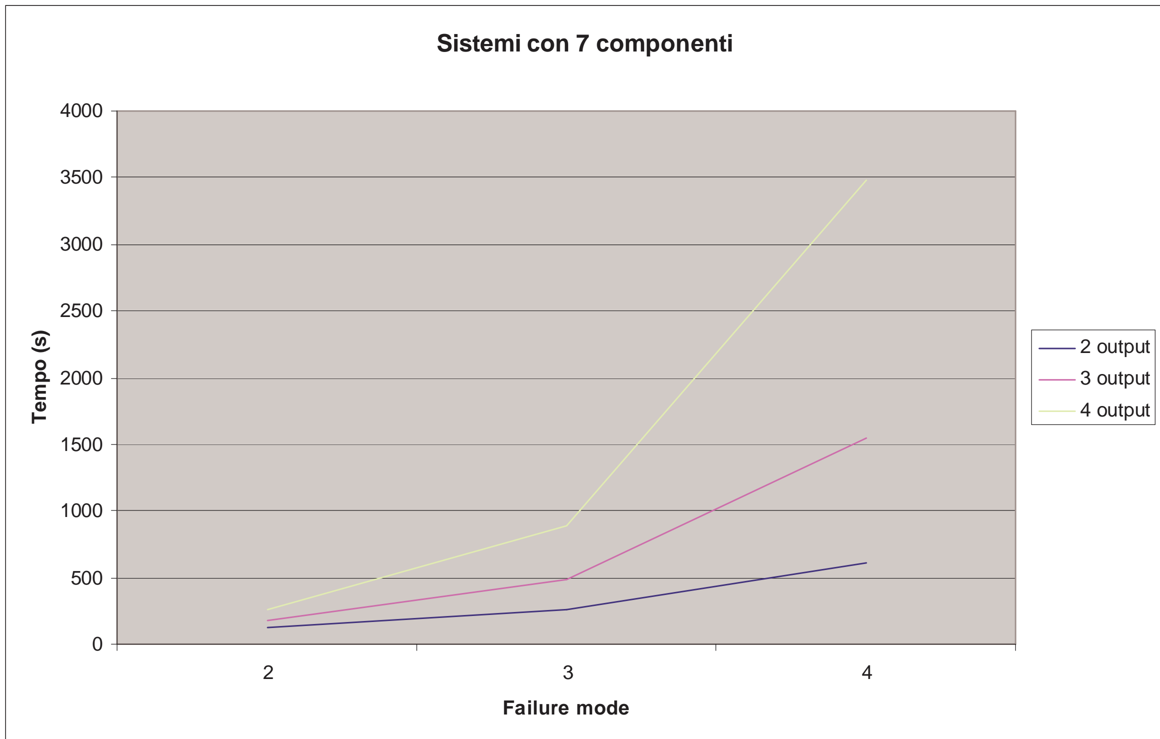

Grafico 5.6: Sistemi con 7 componenti

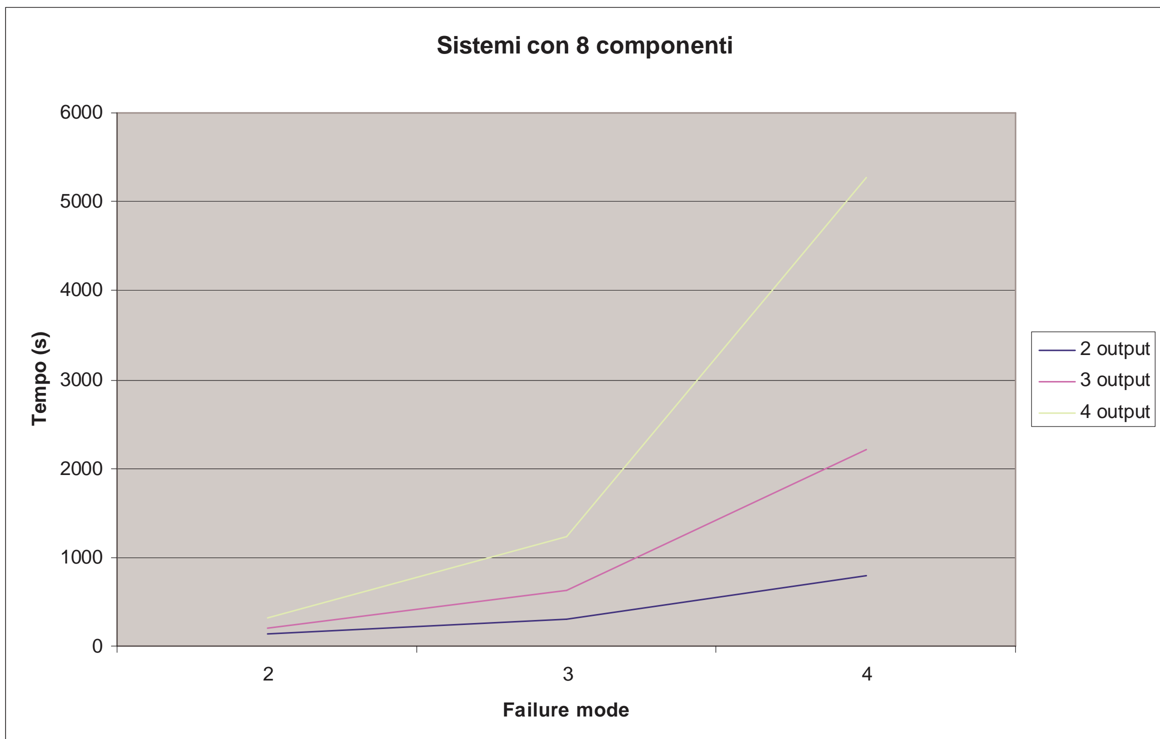

Grafico 5.7: Sistemi con 8 componenti

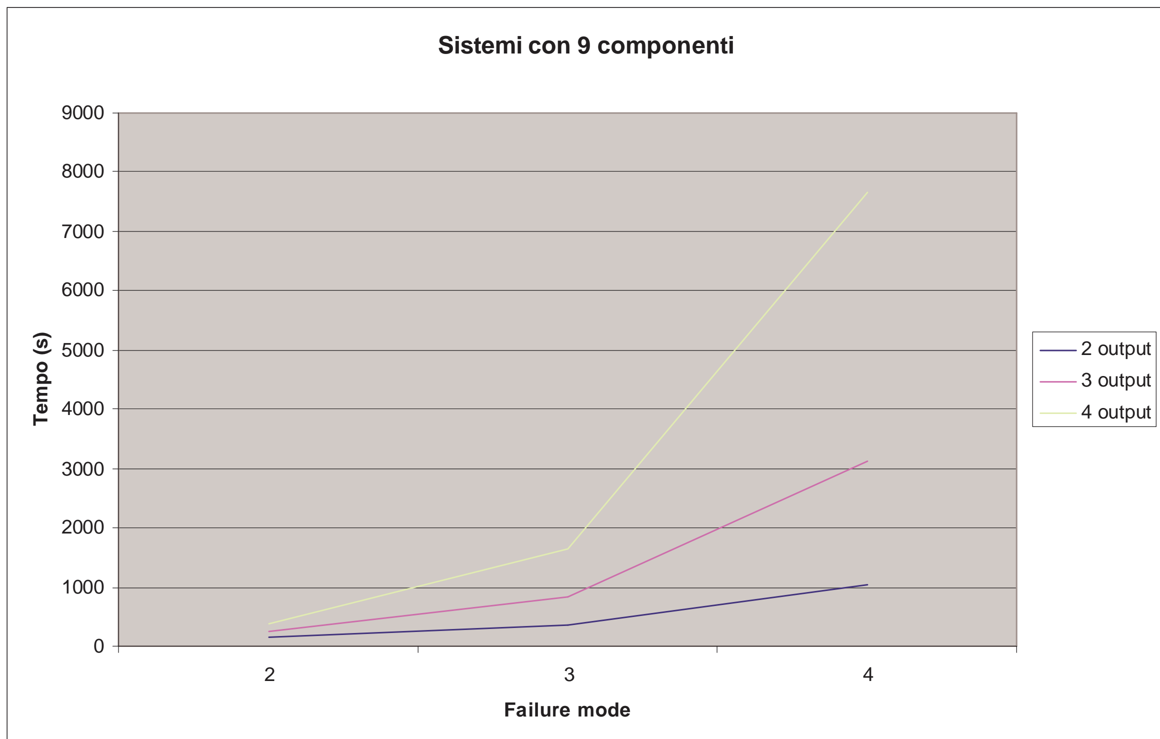

Grafico 5.8: Sistemi con 9 componenti

L'ultima serie di grafici, dal 5.9 al 5.11, riporta il caso di sistemi con un numero di failure mode fissato. Non si ripetono tutte le considerazioni precedenti limitandosi a sottolineare che l'andamento di questi grafici può apparire simile a quelli già mostrati, ma in realtà vi è una differenza. Infatti in ascissa ora è riportato il numero di porte di output (il cui crescere non comporta un incremento esponenziale del tempo di building). Si comprende meglio tale aspetto osservando il Grafico 5.9, esso mostra l'andamento più che lineare (ma non esponenziale) del tempo di building all'aumentare del numero di porte di output, gli altri due grafici risentono dell'aumentato numero di failure modes e sembrano avere un andamento esponenziale mentre la crescita è solamente più che lineare. In questa serie di grafici, l'andamento di crescita esponenziale andrà invece ricercato nel confronto tra i grafici valutando i valori dei tempi in ordinata, in accordo con le osservazioni sugli insiemi di grafici precedenti.

Per maggior chiarezza si riassumono, sinteticamente, i risultati ricavati dall'analisi dei grafici precedenti:

- fissando il numero di porte di output, si ha un incremento esponenziale del tempo di building all'aumentare del numero di failure modes, che risulta tanto più marcato quanto maggiore è il numero di componenti del sistema,
- fissando il numero di porte di output e di failure modes si ha un incremento più che lineare (ma non esponenziale) del tempo di building all'aumentare del numero di componenti del sistema,
- fissando il numero di componenti del sistema, si ha un incremento esponenziale del tempo di building all'aumentare del numero di failure modes, che risulta tanto più marcato quanto maggiore è il numero di porte di output,
- fissando il numero di componenti del sistema e di failure modes si ha un incremento più che lineare (ma non esponenziale) del tempo di building all'aumentare del numero di porte di output,
- a parità di altre condizioni, l'incremento non lineare del tempo di building, dovuto ad un aumento di porte di output, è più marcato rispetto a quello dovuto ad un aumento di componenti del sistema,
- fissando il numero di failure modes del sistema si ha un incremento più che lineare (ma non esponenziale) del tempo di building all'aumentare del numero di porte di output, che risulta tanto più marcato quanto maggiore è il numero di componenti del sistema.

Concludendo, l'analisi dei grafici ha rilevato come l'elemento critico che influenza maggiormente il tempo di building del modello sia il numero di failure modes. Tale aspetto trova riscontro con quanto precedentemente delineato, in particolare, nella Sezione 4.2, infatti, è stato più volte sottolineato che, nella modellazione proposta, deve essere presente un numero di transizioni per porta di output di ogni componente pari al quadrato del numero di failure modes. Questo permette di valutare la complessità di un singolo componente del modello come $O(mn^2)$ dove n è il numero di failure modes e m è il numero di porte di output, la complessità del modello completo sarà, quindi, $O(kmn^2)$ dove k è il numero di componenti. La complessità reale sarà poi maggiore in quanto sono presenti anche altre transizioni nel modello (ad esempio quelle per connettere componenti differenti). Si comprende quindi il maggiore impatto dell'aumento del numero di failure modes sulla complessità del sistema rispetto all'aumento del numero di porte di output o di

componenti. Rigorosamente il contributo teorico dei failure modes alla complessità è quadratico mentre quello di porte di output e componenti è lineare, invece i dati mostrano un andamento esponenziale per i failure modes, e più che lineare per porte di output e componenti. Tale discrepanza è dovuta al fatto che, nella realizzazione pratica del modello, il tool deve considerare le connessioni tra tutti le possibili parti dello stesso ed il loro numero esplode all'aumentare dei parametri introducendo overhead computazionali. Tuttavia, lo scopo di questo paragone, non è quello di stabilire una rigorosa corrispondenza tra analisi di complessità teorica e dati empirici, ma piuttosto è quello di mostrare che l'aver riscontrato un andamento esponenziale del tempo di building del modello all'aumentare del numero di failure modes sia del tutto in linea con i risultati attesi e coerente con la modellazione proposta.

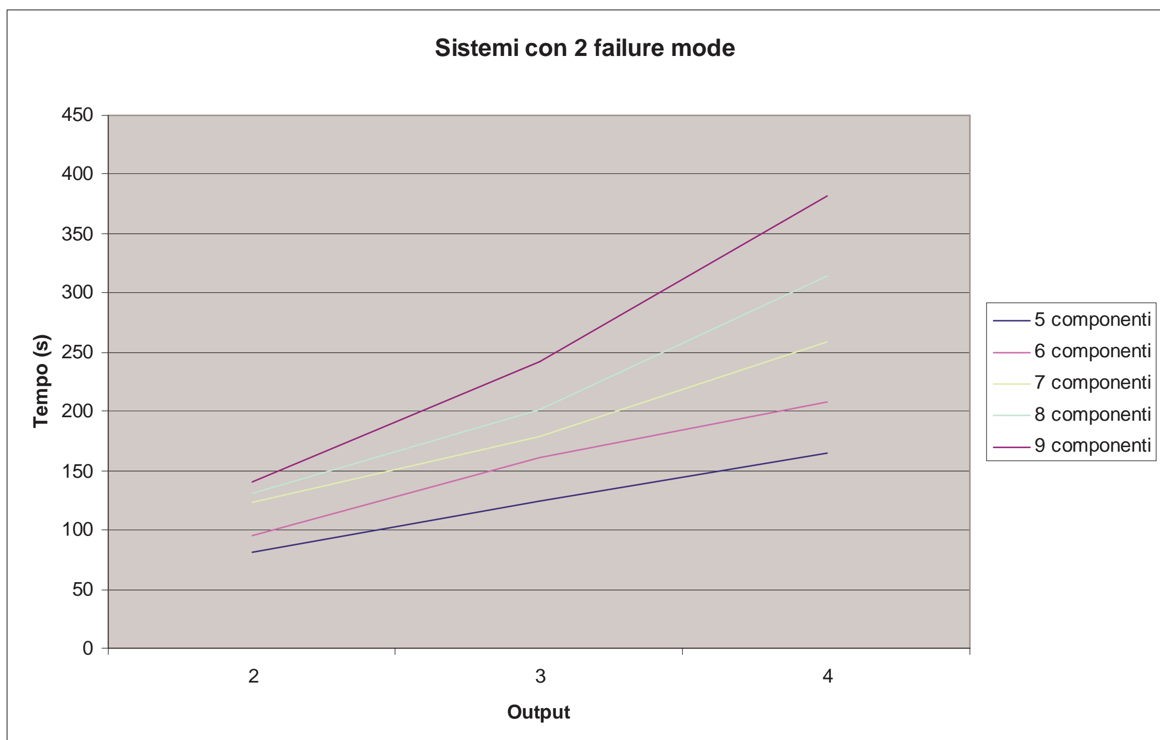

Grafico 5.9: Sistemi con 2 failure modes

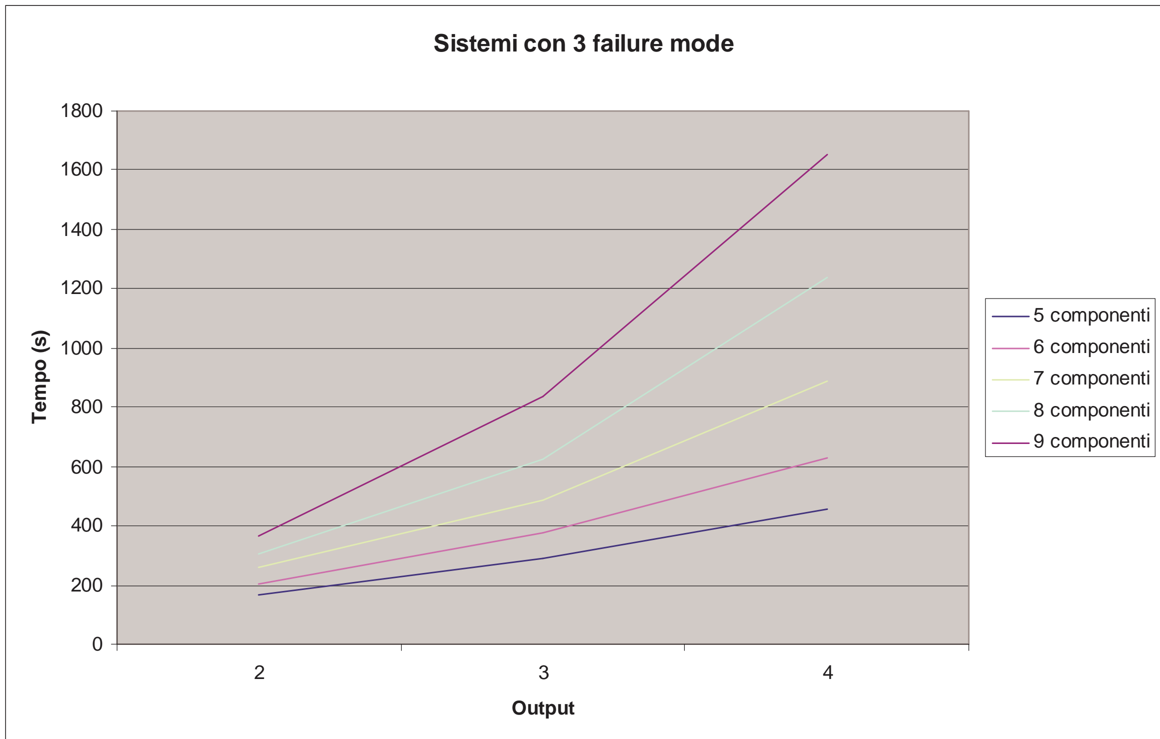

Grafico 5.10: Sistemi con 3 failure modes

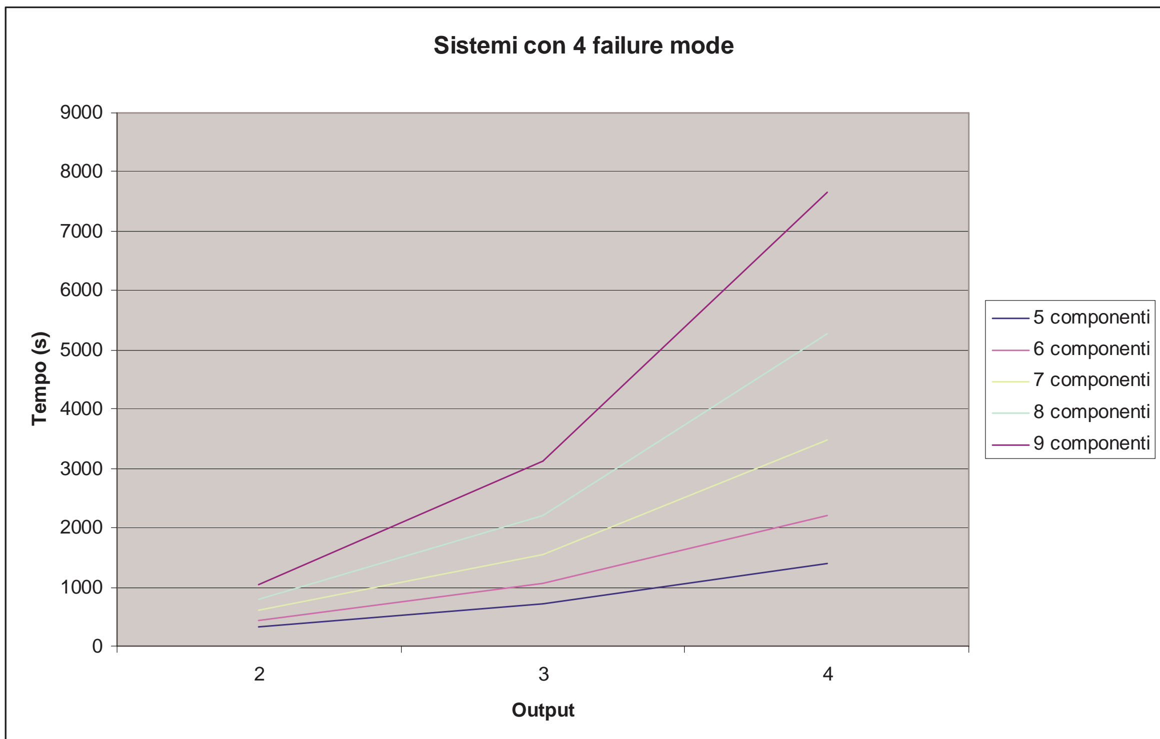

Grafico 5.11: Sistemi con 4 failure modes

CAPITOLO 6

Conclusioni e sviluppi futuri

6.1 Conclusioni

Questo lavoro ha permesso la modellazione di failure modes multipli in sistemi a componenti software utilizzati in ambienti critici per la sicurezza. In tali sistemi l'affidabilità rappresenta un requisito fondamentale.
Si è proposta una modellazione attraverso le Reti di Petri stocastiche colorate che fosse in grado, da un lato, di descrivere adeguatamente il problema, dall'altro, di mettere in condizione l'utilizzatore esperto del sistema, ma non necessariamente dei formalismi matematici utilizzati per modellarlo, di compiere scelte efficaci fin dai primi passi della progettazione del software.
Si è detto dell'importanza dell'affidabilità nei sistemi considerati. In particolare si è ritenuto fondamentale il poter considerare failure modes multipli. È sembrato riduttivo limitare le possibilità di errore ad una presenza o assenza dell'errore stesso.
La modellazione effettuata ha permesso di tenere conto della presenza di failure modes multipli descrivendone la propagazione e i meccanismi che ne permettono la modifica nel passaggio da un componente ad un altro.
Il progettista software può quindi servirsi di tali modelli in fase di progettazione del software e può, ad esempio, valutare l'affidabilità dell'intero sistema in relazione alle affidabilità dei singoli componenti. Da un altro punto di vista è in grado di discriminare nella scelta tra componenti equivalenti dal punto di vista funzionale ma con diversi requisiti di affidabilità. Infatti se il progettista vuole garantire un certo livello di affidabilità del sistema complessivo, può fare un'analisi inversa volta a trovare il livello minimo di affidabilità dei componenti del sistema e quindi la modellazione proposta è in grado di guidarlo anche nella scelta dei componenti software appropriati.

Il Capitolo 4 oltre ad aver descritto la modellazione realizzata, ha avuto l'importante ruolo di mostrare la bontà delle scelte modellistiche effettuate tramite un'analisi delle proprietà di affidabilità.
Il Capitolo 5 ha permesso di valutare l'efficienza del tool utilizzato e ha dato indicazioni in merito alla complessità dei modelli creati. In particolare si è rilevata la ridotta durata delle simulazioni e l'andamento esponenziale del tempo di building del modello all'aumentare del numero di failure modes del sistema.

6.2 Sviluppi futuri

Vista la precedentemente sottolineata efficacia del modello, mostrata dalla corrispondenza dei dati ricavati dall'analisi di affidabilità con i valori attesi, una possibile direzione di espansione del lavoro è costituita dal tentativo di migliorare l'efficienza del tool nell'eseguire le simulazioni e quindi diminuire la complessità dei modelli realizzati.
Si è mostrato come la complessità dei modelli abbia un andamento di tipo quadratico rispetto al numero dei failure modes. Tale relazione deriva direttamente dalla scelta modellistica di creare un numero di transizioni, per porta di output di ogni componente, pari al quadrato dei failure modes. Questa interpretazione, dato il formalismo delle Reti di Petri stocastiche colorate, è sembrata essere la più ragionevole per descrivere il comportamento voluto. Non è tuttavia escluso che possano essere trovate differenti strategie modellistiche in grado di garantire una minore complessità. Tuttavia, appare alquanto poco probabile la possibilità di individuare alternative equivalenti più efficienti che non prevedano di stravolgere completamente le scelte modellistiche effettuate, con il rischio, quindi, di perdere di vista gli scopi della modellazione diminuendone l'efficacia.
Un'altra possibile direzione di espansione del lavoro è rappresentata dal voler dotare il modello della capacità di gestire sia componenti black-box che white-box, in tal modo si potrebbe di volta in volta scendere al livello di generalità ritenuto adatto al particolare caso considerato.
Un ulteriore sviluppo del lavoro proposto è costituito dal poter considerare sistemi in cui la storia passata del sistema è rilevante per l'evoluzione del sistema stesso. Infatti, nella modellazione effettuata, si è implicitamente fatta l'assunzione di avere, in ogni stato, indipendenza dalla storia passata del sistema. Ad esempio, nel modello realizzato, un token in un posto di input, per "evolvere", tiene conto solamente del suo tipo e delle probabilità

delle transizioni cui è connesso, cioè considera solo stato presente e ingressi, potrebbe costituire un’estensione interessante la possibilità di discriminare la scelta dello stato futuro anche sulla base di informazioni legate alla storia del sistema.

BIBLIOGRAFIA

[1] A. Zimmermann. Stochastic Discrete Event Systems: Modeling, Evaluation, Applications. Springer Berlin. 2008.

[2] A.Zimmermann, M. Knoke. TimeNET 4.0 User Manual. 2007.

[3] A. Filieri, C. Ghezzi, V. Grassi, R. Mirandola. Reliability Analysis of Component-Based Systems with Multiple Failure Modes. 2010.

[4] URL http://www.users.globalnet.co.uk/~rxv/CBDmain/cbdfaq.htm.

[5] M. Ajmone Marsan. Stochastic Petri Nets: An Elementary Introduction.

[6] G. Florin, C. Fraize, S. Natkin. Stochastic Petri Nets: Properties, Applications and Tools. Microelectron. Reliab., Vol. 31, No. 4, pp. 669-697, 1991.

[7] G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad. Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications. IEEE Transactions on Computers, Vol. 42, No. 11, November 1993.

[8] S. Bernardi, J. Merseguer. Performance evaluation of UML design with Stochastic Well-formed Nets.The Journal of Systems and Software 80 (2007) 1843-1865.

[9] G. Ciardo, J. Muppala, K. Trivedi. SPNP: Stochastic Petri Net Package.

[10] A. Filieri. Failure propagation analysis with multiple failure modes in component-based systems reliability. Master of Science in Computer Systems Engineering, November 2009.

[11] A. Avizienis et al. Basic concepts and taxonomy of dependable and secure computing. IEEE JDSC 1(1) (2004) 11-33.

[12] V. Cortellessa, V. Grassi. A modelling approach to analyze the impact of error propagation on reliability of component-based systems. LNCS 4608 (2007) 140.

[13] R. Reussner, H. V. Schmidt, I. Poernomo. Reliability prediction for component-based software architectures. Journal of Systems and Software 66 (3) (2003) 241-252.

[14] K. K. Lau, Z. Wang. Software component models. IEEE Transactions on Software Engineering 33(10) (2007) 709-724.

[15] A. Immonen, E. Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. Software and Systems Modeling, 7(1): 49-65, 2008.

[16] S. Krishnamurthy, A. P. Mathur. On the estimation of reliability of a software system using reliabilities of its components 146-155.

[17] G. Rodrigues, D. Rosenblum, S. Uchitel. Using scenarios to predict the reliability of concurrent component-based software systems 111 – 126.

[18] D. Woit. Specifying component interactions for modular reliability estimation. Proc. 1st International Software Quality Week Europe (QWE ’97).

[19] R. C. Cheung. A user-oriented software reliability model. IEEE Transactions on Software Engineering. 6(2): 118-125, 1980.

[20] B. Littlewood. Software reliability model for modular program structure. IEEE Transactions on Reliability, 28(3): 241-246, 1979.

[21] K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.

[22] Y. Atamna. Definition of the model "stochastic timed well formed coloured nets". Proc.5th Int. Workshop on Petri Nets and Performance Models, Toulouse, 1993, pp. 24-33.

[23] M. Ajmone Marsan , G. Balbo, G. Conte, S. Donatelli, G. Franceschinis. Modelling with Generalized Stochastic Petri Nets, Series in parallel computing. John Wiley and Sons, 1995.

[24] C. Lin, D. C. Marinescu. On stochastic high-level Petri nets. Proc. 2nd Int. Workshop on Petri Nets and Performance Models, Madison, Wisconsin, 1987, pp. 34-43.

[25] C. Lin, D. C. Marinescu. Stochastic high-level Petri nets and applications. IEEE Transactions on Computers, vol. 37, pp. 815-825, 1988.

[26] C. Lindemann. Performance Modelling with Deterministic and Stochastic Petri Nets. Wiley, 1998.

[27] M. Knoke, A. Zimmermann. Distributed simulation of colored stochastic Petri nets with TimeNET 4.0. Proc. 3rd Int. Conf. Quantitative Evaluation of Systems (QEST '06), Riverside, CA, USA, Sept. 2006, pp. 117-118.

APPENDICE

In questa sezione si riporta il codice sorgente del generatore di modelli. Esso permette di generare file XML che forniti al tool TimeNET danno luogo ai modelli desiderati. In particolare è possibile specificare il numero di componenti del sistema, il numero di porte di output dei componenti e il numero di failure modes del sistema. I punti maggiormente siginificativi del codice sono adeguatamente commentati per chiarirne il ruolo nell'ambito dell'intero software.

```
package XML;
import org.jdom.*;
import org.jdom.output.*;
import java.io.*;
import java.io.InputStreamReader;
import java.io.BufferedReader ;
import java.io.IOException;

public class Generatore2 {

    public static void main(String[] args) {
        try {

            //acquisizione parametri

            System.out.println("Inserire numero porte di output per singolo
            componente:");
            InputStreamReader reader = new InputStreamReader (System.in);
            BufferedReader myInput = new BufferedReader (reader);
            String outs = new String();
            outs = myInput.readLine();
```

```
System.out.println("Inserire numero di failure modes:");
InputStreamReader reader2 = new InputStreamReader (System.in);
BufferedReader myInput2 = new BufferedReader (reader2);
String fails = new String();
fails = myInput2.readLine();

System.out.println("Inserire numero di componenti:");
InputStreamReader reader3 = new InputStreamReader (System.in);
BufferedReader myInput3 = new BufferedReader (reader3);
String comps = new String();
comps = myInput3.readLine();

int out = Integer.parseInt(outs); //numero di porte di output per
componente
int fail = Integer.parseInt(fails); //numero di failure modes
int comp = Integer.parseInt(comps); //numero componenti


//Elemento radice

Element root = new Element("net");


//Documento

Document document = new Document(root);


//Inizializzazione radice

root.setAttribute("id","0");
root.setAttribute("netclass", "SCPN");
```

```
int output = out; //numero di output effettivi per componente tenendo
conto dei failure modes
int posti = ((1 + out) * comp) + 1; //numero totale di posti, l'ultimo
posto aggiunto è per l'inizializzazione
int postout = out * comp; //numero totale di posti di output
int postin = posti - postout; //numero totale posti di input
int transin = (fail * fail) * out * comp; //numero totale di transizioni
interne
int id = 0; //identificatore oggetto
String ids; //identificatore oggetto string
int offsetasc = 500; //offset tra ascisse di oggetti corrispondenti di
componenti diversi
int asc = 190; //ascissa posto
int ord; //ordinata posto
String ascs; //ascissa posto string
String ords; //ordinata posto string
int offset; //offset per sincronizzare ordinate di transizioni interne e di
posti output e transizioni esterne
offset = (((fail * fail) / 2) * 50);


//Posto di inizializzazione (si usa per aggirare il fatto che il tool non
permette di settare inizialmente il tipo di failure mode dei token)

asc = 25;
ord = 510;
Element posto = new Element ("place");
ids = Integer.toString(id);
id = id + 1;
posto.setAttribute("capacity", "0");
posto.setAttribute("id", "0." + ids);
posto.setAttribute("initialMarking", "");
posto.setAttribute("initialMarking", "{}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {},
{}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {},
```

```
{}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}");

posto.setAttribute("queue", "Random");
posto.setAttribute("tokentype", "Errore");
posto.setAttribute("type", "node");
posto.setAttribute("watch", "false");
Element graph = new Element ("graphics");
ascs = Integer.toString(asc);
ords = Integer.toString(ord);
graph.setAttribute("orientation", "0");
graph.setAttribute("x", ascs);
graph.setAttribute("y", ords);
posto.addContent(graph);
Element label = new Element ("label");
label.setAttribute("id", "0." + ids + ".0");
label.setAttribute("text", "P"+ ids);
label.setAttribute("type", "text");
Element graphics = new Element ("graphics");
graphics.setAttribute("x", "-10");
graphics.setAttribute("y", "-40");
label.addContent(graphics);
posto.addContent(label);
root.addContent(posto);


//Posti input

for (int l = 0; l < comp; l++){
        asc = 190;
    ord = 510;
    Element posto1 = new Element ("place");
    ids = Integer.toString(id);
    id = id + 1;
    posto1.setAttribute("capacity", "0");
```

```
            posto1.setAttribute("id", "0." + ids);
            posto1.setAttribute("initialMarking", "");
            posto1.setAttribute("queue", "Random");
            posto1.setAttribute("tokentype", "Errore");
            posto1.setAttribute("type", "node");
            posto1.setAttribute("watch", "false");
            Element graph1 = new Element ("graphics");
            ascs = Integer.toString(asc + (offsetasc * l));
            ords = Integer.toString(ord);
            graph1.setAttribute("orientation", "0");
            graph1.setAttribute("x", ascs);
            graph1.setAttribute("y", ords);
            posto1.addContent(graph1);
            Element label1 = new Element ("label");
            label1.setAttribute("id", "0." + ids + ".0");
            label1.setAttribute("text", "P"+ ids);
            label1.setAttribute("type", "text");
            Element graphics1 = new Element ("graphics");
            graphics1.setAttribute("x", "-10");
            graphics1.setAttribute("y", "-40");
            label1.addContent(graphics1);
            posto1.addContent(label1);
            root.addContent(posto1);
        }


//Posti output


ord = 0;
        for (int l = 0; l < comp; l++){
            asc = 550;
            ord = 0;
              for(int i = 0; i < output; i++){
                     Element posto1 = new Element ("place");
```

```
ids = Integer.toString(id);
id = id + 1;
posto1.setAttribute("capacity", "0");
posto1.setAttribute("id", "0." + ids);
posto1.setAttribute("initialMarking", "");

posto1.setAttribute("queue", "Random");
posto1.setAttribute("tokentype", "Errore");
posto1.setAttribute("type", "node");
posto1.setAttribute("watch", "false");
Element graph1 = new Element ("graphics");
ascs = Integer.toString(asc + (offsetasc * l));
if (i == 0){
        ord = ord + offset + 50;
}
if (i > 0){
        ord = ord + ((offset + 50) * 2);
}
ords = Integer.toString(ord);
graph1.setAttribute("orientation", "0");
graph1.setAttribute("x", ascs);
graph1.setAttribute("y", ords);
posto1.addContent(graph1);
Element label1 = new Element ("label");
label1.setAttribute("id", "0." + ids + ".0");
label1.setAttribute("text", "P"+ ids);

label1.setAttribute("type", "text");
Element graphics1 = new Element ("graphics");
graphics1.setAttribute("x", "-10");
graphics1.setAttribute("y", "-40");
label1.addContent(graphics1);
posto1.addContent(label1);
root.addContent(posto1);
```

```
                    }
            }


//Transizioni temporizzate per le misure

            int id1 = 1000000; //si utilizza un nuovo id perchè queste transizioni
            sono state aggiunte alla fine ma devono essere
            //messe in questa posizione per la struttura del file xml e se si utilizza il
            solito id ci si trova
            //a dover riaggiornare input e output di tutti gli archi, così invece non
            bisogna modificare
            //nessun altro id, si è scelto 1000000 perchè ragionevolmente non ci
             saranno così tanti oggetti nel
            //modello ma se ci fossero basterebbe aumentare il valore di id1 (e
             analogamente il valore di
            //tratemp negli archi corrispondenti)
            String js = ""; //indice per il numero di failure modes nelle guardie
            string
            ord = 0;
            for (int i = 0; i < output; i++){
                    ord = ord + 50;
                    for (int j = 0; j < fail; j++){
                            Element transizione1 = new Element ("timedTransition");
                              ids = Integer.toString(id1);
                              id1 = id1 + 1;
                              js = Integer.toString(j);
                              transizione1.setAttribute("id", "0." + ids);
                              transizione1.setAttribute("localGuard", "x.tipo==" + "'"
                              + "errore" + js + "'");
                              transizione1.setAttribute("serverType",
                              "ExclusiveServer");
                              transizione1.setAttribute("specType", "Automatic");
                              transizione1.setAttribute("takeFirst", "false");
```

```
			transizione1.setAttribute("timeFunction", "EXP(1.0)");
			transizione1.setAttribute("type", "node");
			transizione1.setAttribute("watch", "true");

			Element graph1 = new Element ("graphics");
			ascs = Integer.toString((offsetasc * comp) + 200);
			ord = ord + offset;

			ords = Integer.toString(ord);
			graph1.setAttribute("orientation", "0");
			graph1.setAttribute("x", ascs);
			graph1.setAttribute("y", ords);
			transizione1.addContent(graph1);
			Element label1 = new Element ("label");

			label1.setAttribute("id", "0." + ids + ".0");
			label1.setAttribute("text", "T"+ ids);

			label1.setAttribute("type", "text");
			Element graphics1 = new Element ("graphics");
			graphics1.setAttribute("x", "-10");
			graphics1.setAttribute("y", "-40");
			label1.addContent(graphics1);
			transizione1.addContent(label1);
			root.addContent(transizione1);
	}
}


//Transizione di inizializzazione

asc = 108;
ord = 510;
Element transizione2 = new Element ("immediateTransition");
```

```
ids = Integer.toString(id);
id = id + 1;
transizione2.setAttribute("id", "0." + ids);
transizione2.setAttribute("priority", "1");
transizione2.setAttribute("serverType", "ExclusiveServer");
transizione2.setAttribute("specType", "Automatic");
transizione2.setAttribute("takeFirst", "false");
transizione2.setAttribute("type", "node");
transizione2.setAttribute("watch", "false");
transizione2.setAttribute("weight", "1.0E0");
Element graph2 = new Element ("graphics");
ascs = Integer.toString(asc);
ords = Integer.toString(ord);
graph2.setAttribute("orientation", "0");
graph2.setAttribute("x", ascs);
graph2.setAttribute("y", ords);
transizione2.addContent(graph2);
Element label2 = new Element ("label");
label2.setAttribute("id", "0." + ids + ".0");
label2.setAttribute("text", "T"+ ids);
label2.setAttribute("type", "text");
Element graphics2 = new Element ("graphics");
graphics2.setAttribute("x", "-10");
graphics2.setAttribute("y", "-40");
label2.addContent(graphics2);
transizione2.addContent(label2);
root.addContent(transizione2);


//Transizioni interne


for (int l = 0; l < comp; l++){
    int sup = 10; //estremo superiore dell'intervallo dei pesi delle
    transizioni, così i pesi saranno tutti compresi tra 1 e 10
```

```
int random; //pesi random
String randoms; //pesi random stringa
int k = 0; //indice per il numero di failure modse nelle guardie
String ks = ""; //indice per il numero di failure modes nelle
guardie string
int j = 0; //usato per gestire il numero di failure modes  nelle
guardie
asc = 370;
ord = 10;
for (int m = 0; m < output; m++){
    for (int i = 0; i < ((transin / comp) / output); i++){
            Element transizione1 = new Element
            ("immediateTransition");
              ids = Integer.toString(id);
              id = id + 1;
              transizione1.setAttribute("id", "0." + ids);
              if (j == fail){
                    j = 0;
                    k = k + 1;
              }
              ks = Integer.toString(k);
              j = j + 1;
              transizione1.setAttribute("localGuard",
              "x.tipo==" + '"' + "errore" + ks + '"');
              transizione1.setAttribute("priority", "1");

              transizione1.setAttribute("serverType",
              "ExclusiveServer");
              transizione1.setAttribute("specType",
              "Automatic");
              transizione1.setAttribute("takeFirst", "false");
              transizione1.setAttribute("type", "node");
              transizione1.setAttribute("watch", "false");
              random = (int)(sup * Math.random()) + 1;
```

```
                    randoms = Integer.toString(random);
                    transizione1.setAttribute("weight", randoms +
                    ".0E0");
                    Element graph1 = new Element ("graphics");
                    ascs = Integer.toString(asc + (offsetasc * l));
                    ord= ord + 50;
                    ords = Integer.toString(ord);
                    graph1.setAttribute("orientation", "0");
                    graph1.setAttribute("x", ascs);
                    graph1.setAttribute("y", ords);
                    transizione1.addContent(graph1);
                    Element label1 = new Element ("label");

                    label1.setAttribute("id", "0." + ids + ".0");
                    label1.setAttribute("text", "T"+ ids);

                    label1.setAttribute("type", "text");
                    Element graphics1 = new Element ("graphics");
                    graphics1.setAttribute("x", "-10");
                    graphics1.setAttribute("y", "-40");
                    label1.addContent(graphics1);
                    transizione1.addContent(label1);
                    root.addContent(transizione1);
                }
            ord = ord + 50;
            j = 0;
            k = 0;
        }

    }


//Transizioni esterne
```

```
for (int l = 0; l < comp - 1; l++){
        asc = 620;
        ord = 0;
          for (int i = 0; i < output; i++){
                  Element transizione1 = new Element
                  ("immediateTransition");
                    ids = Integer.toString(id);
                    id = id + 1;
                    transizione1.setAttribute("id", "0." + ids);
                    transizione1.setAttribute("priority", "1");

                    transizione1.setAttribute("serverType",
                    "ExclusiveServer");
                    transizione1.setAttribute("specType", "Automatic");
                    transizione1.setAttribute("takeFirst", "false");
                    transizione1.setAttribute("type", "node");
                    transizione1.setAttribute("watch", "false");
                    transizione1.setAttribute("weight", "1.0E0");

                    Element graph1 = new Element ("graphics");
                    ascs = Integer.toString(asc + (offsetasc * l));

                    if (i == 0){
                          ord = ord + offset + 50;
                    }
                    if (i > 0){
                          ord = ord + ((offset + 50) * 2);
                    }
                    ords = Integer.toString(ord);
                    graph1.setAttribute("orientation", "0");
                    graph1.setAttribute("x", ascs);
                    graph1.setAttribute("y", ords);
                    transizione1.addContent(graph1);
```

```
			Element label1 = new Element ("label");

			label1.setAttribute("id", "0." + ids + ".0");
			label1.setAttribute("text", "T"+ ids);

			label1.setAttribute("type", "text");
			Element graphics1 = new Element ("graphics");
			graphics1.setAttribute("x", "-10");
			graphics1.setAttribute("y", "-40");
			label1.addContent(graphics1);
			transizione1.addContent(label1);
			root.addContent(transizione1);
		}

}



//Arco di inizializzazione input

int traistart1 = posti ; //id transizione inizializzazione
String traistart1s; //id transizione inizializzazione string
String etichettai1 = "x"; //etichetta arco di input
Element arco2 = new Element ("arc");
ids = Integer.toString(id);
id = id + 1;
traistart1s = Integer.toString(traistart1);
arco2.setAttribute("fromNode", "0.0");
arco2.setAttribute("id", ids);
arco2.setAttribute("toNode", "0." + traistart1s);
arco2.setAttribute("type", "connector");
Element inscr2 = new Element ("inscription");
inscr2.setAttribute("id", "0." + ids + ".0");
inscr2.setAttribute("text", etichettai1);
inscr2.setAttribute("type", "inscriptionText");
```

```
Element graphics3 = new Element ("graphics");
graphics3.setAttribute("x", "0");
graphics3.setAttribute("y", "0");
inscr2.addContent(graphics3);
arco2.addContent(inscr2);
root.addContent(arco2);


//Arco di inizializzazione output

int num; //variabile usata per rendere casuale il tipo di errore iniziale
dei token
String nums; //variabile usata per rendere casuale il tipo di errore
 iniziale dei token string
num = (int)(fail * Math.random());
nums = Integer.toString(num);
String err = "'" + "errore" + nums + "'";
String etichettai2 = "x(tipo=" + err + ")"; //etichetta arco di input

Element arco3 = new Element ("arc");
ids = Integer.toString(id);
id = id + 1;
traistart1s = Integer.toString(traistart1);
arco3.setAttribute("fromNode", "0." + posti);
arco3.setAttribute("id", ids);
arco3.setAttribute("toNode", "0.1");
arco3.setAttribute("type", "connector");
Element inscr3 = new Element ("inscription");
inscr3.setAttribute("id", "0." + ids + ".0");
inscr3.setAttribute("text", etichettai2);
inscr3.setAttribute("type", "inscriptionText");
Element graphics4 = new Element ("graphics");
graphics4.setAttribute("x", "0");
graphics4.setAttribute("y", "0");
```

```
inscr3.addContent(graphics4);
arco3.addContent(inscr3);
root.addContent(arco3);



//Archi di input

int idposto = 0; // id dei posti di partenza degli archi di input
String idpostos; // id dei posti di partenza degli archi di input string
int traistart; //id della prima transizione interna negli archi di input

String traistarts; //id della prima transizione interna negli archi di input
 string
String etichettai; //etichetta arco di input
for (int n = 0; n < comp; n++){ //ciclo per scandire i componenti
      idposto = idposto + 1;
         traistart = posti + 1 + (((fail * fail) * output) * n);
         etichettai = "x";
         for (int i = 0; i < ((fail * fail) * output); i++){ //ciclo per
         scandire le transizioni interne
               Element arco1 = new Element ("arc");
                  ids = Integer.toString(id);
                  id = id + 1;
                  idpostos = Integer.toString(idposto);

                  traistarts = Integer.toString(traistart);
                  traistart = traistart + 1;
                  arco1.setAttribute("fromNode", "0." + idpostos);
                  arco1.setAttribute("id", ids);
                  arco1.setAttribute("toNode", "0." + traistarts);
                  arco1.setAttribute("type", "connector");
```

```
            Element inscr1 = new Element ("inscription");

            inscr1.setAttribute("id", "0." + ids + ".0");
            inscr1.setAttribute("text", etichettai);
            inscr1.setAttribute("type", "inscriptionText");
            Element graphics1 = new Element ("graphics");
            graphics1.setAttribute("x", "0");
            graphics1.setAttribute("y", "0");
            inscr1.addContent(graphics1);
            arco1.addContent(inscr1);
            root.addContent(arco1);
        }
}



//Archi di output

int idpostout; //id dei posti di arrivo degli archi di input
String idpostouts; //id dei posti di arrivo degli archi di input string
int traostart; //id transizioni interne negli archi di output
String traostarts; // id transizioni interne negli archi di output string

for (int o = 0; o < comp; o++){
    int m = 0; //indice per il numero di failure modes nelle etichette
      String ms = ""; //indice per il numero di failure modes nelle
       etichette string
    idpostout = postin + (output * o);
    traostart = posti + 1 + (((fail * fail) * output) * o);
      for (int k = 0; k < output; k++){ //ciclo per scandire i posti di
      output
          for (int j = 0; j < (fail * fail); j++){ //ciclo per scandire le
          transizioni interne
                Element arco1 = new Element ("arc");
```

```
            ids = Integer.toString(id);
            id = id + 1;
            idpostouts = Integer.toString(idpostout);

            traostarts = Integer.toString(traostart);
            traostart = traostart + 1;
            if (m == fail){
                m = 0;
            }
            ms = Integer.toString(m);

            m = m + 1;
            arco1.setAttribute("fromNode", "0." + traostarts);
            arco1.setAttribute("id", ids);
            arco1.setAttribute("toNode", "0." + idpostouts);
            arco1.setAttribute("type", "connector");

            Element inscr1 = new Element ("inscription");

            inscr1.setAttribute("id", "0." + ids + ".0");
            inscr1.setAttribute("text", "x(tipo=" + '"' +
            "errore" + ms + '"' + ')' );
            inscr1.setAttribute("type", "inscriptionText");
            Element graphics1 = new Element ("graphics");
            graphics1.setAttribute("x", "0");
            graphics1.setAttribute("y", "0");
            inscr1.addContent(graphics1);
            arco1.addContent(inscr1);
            root.addContent(arco1);

        }
      idpostout = idpostout + 1;

  }
```

```
}

//Archi di connessione input

 int idpostart; //id dei posti di partenza degli archi di connessione input
String idpostarts; //id dei posti di partenza degli archi di connessione
input string
 int tranout; //id transizioni esterne
 String tranouts; //id transizioni esterne string
for (int p = 0; p < comp - 1; p++){
        idpostart = postin + (output * p);
        tranout = posti + 1 + transin + (output * p);
        for (int k = 0; k < output; k++){
                Element arco1 = new Element ("arc");
                   ids = Integer.toString(id);
                   id = id + 1;
                   idpostarts = Integer.toString(idpostart);
                   tranouts = Integer.toString(tranout);
                   idpostart = idpostart + 1;

                   tranout = tranout + 1;
                   arco1.setAttribute("fromNode", "0." + idpostarts);
                   arco1.setAttribute("id", ids);
                   arco1.setAttribute("toNode", "0." + tranouts);
                   arco1.setAttribute("type", "connector");

                   Element inscr1 = new Element ("inscription");

                   inscr1.setAttribute("id", "0." + ids + ".0");
                   inscr1.setAttribute("text", "x");
                   inscr1.setAttribute("type", "inscriptionText");
                   Element graphics1 = new Element ("graphics");
                   graphics1.setAttribute("x", "0");
```

```
                graphics1.setAttribute("y", "0");
                inscr1.addContent(graphics1);
                arco1.addContent(inscr1);
                root.addContent(arco1);
        }
}


//Archi di connessione output

int idpostarr; //id dei posti di arrivo degli archi di connessione output
String idpostarrs; //id dei posti di arrivo degli archi di connessione
output string
int tranout2; //id transizioni esterne
String tranout2s; //id transizioni esterne string
idpostarr = 1;
for (int q = 0; q < comp - 1; q++){
      idpostarr = 1 + idpostarr;
      tranout2 = posti + 1 + transin + (output * q);

      Element arco1 = new Element ("arc");
         ids = Integer.toString(id);
         id = id + 1;
         idpostarrs = Integer.toString(idpostarr);
         tranout2s = Integer.toString(tranout2);

         tranout2 = tranout2 + 1;
         arco1.setAttribute("fromNode", "0." + tranout2s);
         arco1.setAttribute("id", ids);
         arco1.setAttribute("toNode", "0." + idpostarrs);
         arco1.setAttribute("type", "connector");

         Element inscr1 = new Element ("inscription");
         inscr1.setAttribute("id", "0." + ids + ".0");
```

```
            inscr1.setAttribute("text", "x");
            inscr1.setAttribute("type", "inscriptionText");
            Element graphics1 = new Element ("graphics");
            graphics1.setAttribute("x", "0");
            graphics1.setAttribute("y", "0");
            inscr1.addContent(graphics1);
            arco1.addContent(inscr1);
            root.addContent(arco1);
        }


//Archi di connessione output per i loop

        int idpostarr2; //id dei posti di arrivo degli archi di connessione output
        String idpostarr2s; //id dei posti di arrivo degli archi di connessione
        output string
        int tranout3; //id transizioni esterne
        String tranout3s; //id transizioni esterne string
        idpostarr2 = 1;
        int a = 0; //variabile usata per introdurre loop casualmente
        int b = 0; //variabile usata per riportare in posizione corretta l'indice
        dei posti
        int c = 0; //variabile usata per memorizzare l'indice dei posti se viene
         introdotto un loop
        int offset2; //variabile usata per generare casualmente la lunghezza del
        loop
        for (int q = 0; q < comp - 1; q++){
            tranout3 = posti + 1 + transin + (output * q) + 1;
            idpostarr2 = idpostarr2 + 1;
            for (int k = 0; k < output - 1; k++){
                    a = (int)(2 * Math.random()) + 1; //a vale 1 con
                    probabilità 50%
                    offset2 = (int)((q - 1) * Math.random()) + 1;
```

```
if (a == 1){
        c = idpostarr2;
            idpostarr2 = idpostarr2 - offset2;
            a = 0;
            b = 1;


        }
Element arco1 = new Element ("arc");
    ids = Integer.toString(id);
    id = id + 1;
    idpostarr2s = Integer.toString(idpostarr2);
    if (b == 1){
            b = 0;
            idpostarr2 = c;
    }
    tranout3s = Integer.toString(tranout3);

    tranout3 = tranout3 + 1;
    arco1.setAttribute("fromNode", "0." + tranout3s);
    arco1.setAttribute("id", ids);
    arco1.setAttribute("toNode", "0." + idpostarr2s);
    arco1.setAttribute("type", "connector");

    Element inscr1 = new Element ("inscription");

    inscr1.setAttribute("id", "0." + ids + ".0");
    inscr1.setAttribute("text", "x");
    inscr1.setAttribute("type", "inscriptionText");
    Element graphics1 = new Element ("graphics");
    graphics1.setAttribute("x", "0");
    graphics1.setAttribute("y", "0");
    inscr1.addContent(graphics1);
    arco1.addContent(inscr1);
    root.addContent(arco1);
```

```
        }
}


//Archi per le transizioni temporizzate per le misure

int idpostarr3; //id dei posti di partenza
idpostarr3 = postin + output * (comp - 1) - 1;
String idpostarr3s; //id dei posti di partenza string
int tratemp; //id delle transizioni di arrivo
String tratemps; //id delle transizioni di arrivo string
tratemp = 1000000; //analogo a id1
for (int i = 0; i < output; i++){
        idpostarr3 = idpostarr3 + 1;
        for (int j = 0; j < fail; j++){


                Element arco1 = new Element ("arc");
                  ids = Integer.toString(id);
                  id = id + 1;
                  idpostarr3s = Integer.toString(idpostarr3);
                  tratemps = Integer.toString(tratemp);

                  tratemp = tratemp + 1;
                  arco1.setAttribute("fromNode", "0." + idpostarr3s);
                  arco1.setAttribute("id", ids);
                  arco1.setAttribute("toNode", "0." + tratemps);
                  arco1.setAttribute("type", "connector");

                  Element inscr1 = new Element ("inscription");

                  inscr1.setAttribute("id", "0." + ids + ".0");
                  inscr1.setAttribute("text", "x");
                  inscr1.setAttribute("type", "inscriptionText");
```

```
            Element graphics1 = new Element ("graphics");
            graphics1.setAttribute("x", "0");
            graphics1.setAttribute("y", "0");
            inscr1.addContent(graphics1);
            arco1.addContent(inscr1);
            root.addContent(arco1);
    }
}


//Tipo "Errore"

id = id + 1;
ids = Integer.toString(id);
Element tipo = new Element ("recordTokentype");
tipo.setAttribute("id", ids );
tipo.setAttribute("name", "Errore");
tipo.setAttribute("type", "text");
Element graphics1 = new Element ("graphics");
graphics1.setAttribute("x", "190");
graphics1.setAttribute("y", "900");
tipo.addContent(graphics1);
Element attributo = new Element ("attribute");
attributo.setAttribute("dataType", "string");
attributo.setAttribute("name", "tipo");
tipo.addContent(attributo);
root.addContent(tipo);


//Creazione dell'oggetto XMLOutputter

XMLOutputter outputter = new XMLOutputter();
```

```
      //Imposto il formato dell'outputter come "bel formato"

      outputter.setFormat(Format.getPrettyFormat());


      //Produco l'output sul file modello.xml

      outputter.output(document, new FileOutputStream("modello.xml"));

    }
    catch (IOException e) {
      System.err.println("Errore durante il parsing del documento");
      e.printStackTrace();
    }

  }

}
```