UNIVERSITÀ DEGLI STUDI DI PALERMO

SCUOLA DELLE SCIENZE DI BASE E APPLICATE
Corso di Laurea in Informatica
Dipartimento di Matematica e Informatica

Tecniche di Monitoring di Servizi per la Soddisfazione Parziale dei Requisiti

TESI DI LAUREA DI:
ZICHICHI
MIRKO

RELATORE:
Ing. COSSENTINO
MASSIMO

CORRELATORI:
Ing. SABATUCCI
LUCA
Dott. DE SIMONE
GIADA

---

a.a. 2016/2017

Indice

Introduzione

Il Monitoring di un Servizio consiste nell'operazione di Supervisione del comportamento di quest'ultimo, finalizzata al raggiungimento del risultato sperato. In quest'ambito, la Supervisione può avvenire specificando tramite la Logica Temporale Lineare i Requisiti Parziali da soddisfare. Il Monitoring di questa logica si basa sul metodo del Model Checking che mette a confronto formule logiche e modelli di sistemi per verificarne la compatibilità.
In questo lavoro è stato sviluppato un Supervisore per Formule LTL implementato in MUSA (Middleware for User-driven Service Adaptation), un middleware basato su agenti per lo sviluppo di sistemi adattativi guidato dagli utenti. MUSA è basato sul modello del Cloud Computing e su un'architettura Service Oriented. Il primo è un modello in cui le risorse informatiche, come elaborazione, archiviazione o trasmissione dati, sono fornite on demand da un Provider senza un particolare sforzo di gestione. La seconda è un'architettura dove l'esecuzione di processi business viene effettuata tramite micro-servizi, ovvero funzionalità indipendenti accessibili da remoto. MUSA è stato sviluppato per la ricerca di Workflow dinamici che portano all'ottenimento di questi processi business gestendo le configurazioni a run-time tramite la Self-Adaptability.
La Visione di MUSA si basa su un Reasoner che associa alla giusta maniera il $cosa\ fare$, rappresentato dei Requisiti Funzionali chiamati Goals, a $come\ farlo$, rappresentato da micro-servizi chiamati Capabilities. Tramite questo Reasoner viene costruito un Grafo Computazionale che contiene tutti i possibili Stati del Mondo, ovvero rappresentazioni degli attributi del sistema in un certo istante, dove i percorsi dal primo nodo ad un altro rappresentano soluzioni.
In quest'ambito il Supervisore di Formule LTL fa sì che siano rispettati i Requisiti Funzionali, rappresentati dai Goals, durante la ricerca delle soluzioni e l'esecuzione della soluzione scelta. Lo sviluppo del nuovo Supervisore è basato sui concetti del Model Checking ma utilizza la Rete di Petri come struttura principale. La soluzione ha un fondamento teorico, basato sulla verifica di formule temporali, messo in atto durante la fase di progettazione e di realizzazione del Supervisore.

Capitolo 1

Contesto

1.1 Logica Temporale Lineare

La Logica Temporale Lineare (LTL) è un'estensione della Logica Proposizionale che descrive l'evoluzione delle proprietà di un sistema nel tempo. Infatti laddove la logica proposizionale pone dei limiti nel caratterizzare gli aspetti dinamici di un sistema, è possibile utilizzare la Logica Temporale per specificare e verificare questi aspetti.
LTL è una Logica Temporale orientata al futuro che modella il tempo attraverso una sequenza infinita di stati, ovvero istanti discreti di tempo. Essa è definita Lineare poiché prevede che ogni istante abbia solamente un successore, al contrario della CTL (Computation Tree Logic) che si basa su un modello del tempo ramificato in diversi successori. Anche se più utilizzata in informatica, CTL non verrà trattata in quanto LTL è sufficiente per modellare diversi aspetti del software. Le formule LTL sono formate da proposizioni atomiche combinate usando Connettivi Logici e Operatori Temporali. Mentre i connettivi logici possono descrivere solamente un comportamento statico, e vengono quindi valutati su un singolo stato, gli operatori temporali indicano in quali stati devono essere veri i loro argomenti.

Definizione 1.1 (**Sintassi di LTL).** Il linguaggio LTL è definito come segue:

- Se φ è una proposizione atomica, allora φ appartiene al linguaggio LTL (è una formula LTL).
- Se φ e ψ sono formule LTL allora lo sono anche:
  - $\neg\varphi$
  - $\varphi \vee \psi$
  - $\varphi \wedge \psi$
  - $\varphi \Rightarrow \psi$
  - $\varphi \Leftrightarrow \psi$
  - X φ
  - F φ
  - G φ
  - φ U ψ
  - φ R ψ

Gli Operatori Temporali sono:
L'operatore X è chiamato Next ed impone che φ sia vera nello stato successivo.

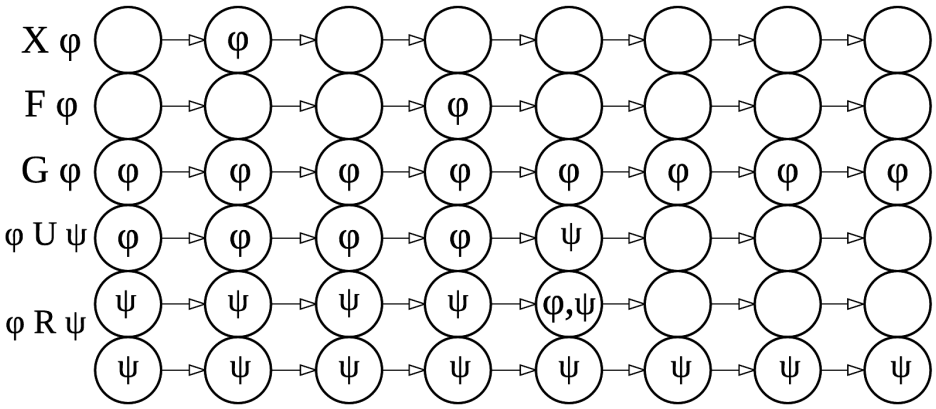


Figura 1.1: Rappresentazione grafica delle formule temporali considerando ogni stato

L'operatore F è chiamato Finally (o Eventually) e stabilisce che prima o poi almeno una volta in un uno stato futuro sia vera φ. Inoltre è un operatore derivato, perché corrisponde alla formula **true** U φ.
L'operatore G è chiamato Globally e viene usato per avere φ vera in tutti gli istanti di tempo. Così come Finally è un operatore derivato, infatti corrisponde alla formula **false** R φ.
L'operatore U è chiamato Until e afferma che φ sia vera in ogni istante finché ψ non diventi vera e impone che prima o poi ψ dovrà diventarla.
L'operatore R è chiamato Release ed è il duale di U, in quanto ψ sarà vera fin quando φ diventerà vera (φ rilascia ψ) oppure ψ resterà vera all'infinito perché φ non diventerà mai vera.

1.2 Model Checking

Il Model Checking è il problema di determinare se una formula è verificata da un sistema dinamico dato il modello di quest'ultimo. Definiti un modello M che rappresenta le possibili esecuzioni del sistema e una formula α che rappresenta il comportamento voluto, il Model Checking verifica le correttezza del sistema rispetto alla specifica data, ovvero $M \models \alpha$.
In generale gli algoritmi di Model Checking si basano su una ricerca esaustiva degli stati in un modello, andando a controllare il corretto comportamento in ognuno di essi. In particolare, per la verifica di proprietà temporali, il Model Checking viene sviluppato tramite la costruzione di un Automa su parole infinite a partire da formule Logico Temporali.

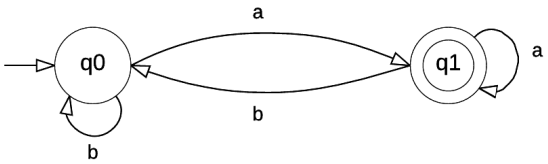


Figura 1.2: Automa di Büchi

1.2.1 Automa di Büchi

Essendo interessati alla verifica di proprietà che in un sistema possono presentarsi all'infinito, nel Model Checking l'automa da costruire deve accettare sequenze infinite. L'Automa di Büchi è un Automa a Stati Finiti che accetta una sequenza infinita in input se esiste un cammino che visita uno stato finale infinite volte.

Definizione 1.2 (Automa di Büchi). L'automa di Büchi $A = \{\Sigma, S, \delta, S_0, F\}$ è una tupla dove:

- Σ è l'alfabeto dei simboli
- S è l'insieme degli stati
- $\delta : S \times \Sigma \to S$ è la funzione di transizione nel caso deterministico ($\delta : S \times \Sigma \to 2^S$ nel caso non deterministico)
- S_0 è lo stato iniziale (l'insieme di stati iniziali nel caso non deterministico)
- F è l'insieme degli stati finali

Data una funzione di etichettatura $l : \mathbf{N} \to S$, una sequenza w è accettata dall'automa A se esiste un cammino dallo stato iniziale ad uno stato finale che effettua un ciclo infinite volte su uno stato finale.

L'Automa di Büchi Generalizzato differisce da un Automa di Büchi per la condizione di accettazione. Infatti quest'ultimo è formato da un insieme di insiemi di stati finali e la condizione di accettazione richiede che almeno uno stato di ogni insieme appaia infinite volte.
Dato un Automa di Büchi Generalizzato è possibile dimostrare che si può costruire un Automa di Büchi equivalente considerando k copie degli stati dell'Automa Generalizzato, dove k è il numero degli insiemi di stati finali.

1.3 Rete di Petri

Una Rete di Petri è un linguaggio grafico e matematico che permette la modellazione di un sistema distribuito, in particolare si presta bene alla modellazione di un

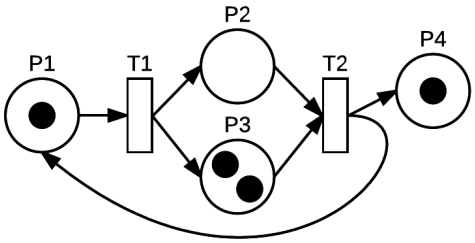


Figura 1.3: Rete di Petri

sistema non deterministico che lavora in parallelo, in modo asincrono. Una Rete di Petri è un grafo diretto che contiene due tipi di nodi connessi tramite Archi diretti: Posti e Transizioni.

Definizione 1.3 (**Rete di Petri).** Una Rete di Petri è una tupla $PN = \{P, T, A\}$ dove:

- P è l'insieme dei Posti
- T è l'insieme delle Transizioni
- $F \subseteq (P \times T) \cup (T \times P)$ è l'insieme degli Archi

I Posti sono raffigurati da cerchi, le Transizioni da rettangoli, mentre gli Archi da frecce e possono collegare solamente un Posto ad una Transizione o viceversa. Un Posto può essere Marcato, ovvero può contenere uno o più Token. Questi sono alla base del funzionamento della Rete in quanto rappresentano l'aspetto dinamico del passaggio da uno stato ad un altro. Infatti, il principio fondamentale della Rete di Petri sta nello Scatto (Firing) della Transizione che avviene in presenza dei Token nel Posto che la precede.
Ogni Transizione ha almeno un Posto che la precede e almeno un Posto che la segue e quindi possono essere identificati in Posti di Input e Posti di Output in riferimento a quella specifica Transizione. Se ogni Posto di Input contiene almeno un Token, la Transizione si dice Abilitata e quindi può Scattare andando a consumare un Token per ogni Place di Input e a produrne uno per ogni Place di Output.

Capitolo 2

Introduzione a MUSA e Supervisori di Formule LTL

Un Sistema Adattativo è un insieme di più parti, indipendenti e non, che formano un'unica entità capace di adattarsi a cambiamenti nell'ambiente esterno o a cambiamenti nelle interazioni interne. Un Sistema Self-Adaptive risponde in maniera autonoma modificando il suo comportamento o la sua struttura per raggiungere l'obiettivo fissato.
Nello studio dei Sistemi Adattativi in Biologia, Fisica, Economia e Sistemi Sociali si può vedere come sia centrale il ruolo del Feedback Loop, un processo circolare in cui l'output del sistema ritorna come input, rendendo il sistema effettivamente Adattativo. Senza un'operazione di Feedback Loop, infatti, il sistema non avrebbe modo di adattarsi alle nuove condizioni ambientali e non sopravviverebbe.
La continua evoluzione della complessità sistemi basati su software rende quest'ultimo uno strumento per portare avanti lo studio sulla Self-Adaptability. In particolare, un Sistema Software che riesce ad organizzarsi da sé, è in grado di cambiare comportamento a run-time in modo da mantenere o migliorare le proprie funzionalità.

2.1 MUSA - Middleware for User-driven Service Adaptation

MUSA è un middleware basato su agenti per lo sviluppo di Sistemi Self-Adaptive guidato dagli utenti. In particolare è sviluppato per gestire l'evoluzione e la Self-Adaptability a run-time di business process in ambito di Workflow adattivi.
I concetti principali su cui si basa la Visione di MUSA sono:

i. Rappresentare *cosa fare* (Goals) e *come fare* (Capabilities) sotto forma di artefatti a run-time sui quali il sistema può lavorare

ii. Un Reasoning che associ alla giusta maniera il *come fare* fare al *cosa fare*

iii. Una semantica comune per tutto il sistema

Il funzionamento di MUSA si basa sulla specifica, da parte dell'utente, di business process descritti come un insieme di Goals e inseriti tramite un linguaggio human-oriented. Con una traduzione di Requisiti Funzionali in entità a run-time è possibile mantenere una forma di Requirement Awareness, in questa maniera è quindi possibile inserire o modificare Goals durante l'esecuzione. Di conseguenza, il core di MUSA consiste in un framework che permette al sistema di cambiare dinamicamente comportamento per adattarsi. L'adattamento viene effettuato considerando lo Stato del Mondo, che rappresenta un'istantanea delle proprietà e degli attributi del sistema, e le Capabilities che sono operazioni che può utilizzare, rendendo il sistema capace di decidere autonomamente su come operare per ottenere i Goals.

Specifica dei Goals e delle Capability

Il primo compito di MUSA consiste nell'accettare e inserire nel sistema a run-time una serie di requisiti, rappresentati dai Goals. Nell'ottica di requisiti Goal-Oriented un Goal è un obiettivo che un attore vuole raggiungere e in questo caso si traduce come un cambiamento nello Stato del Mondo che un attore vuole raggiungere. Uno Stato del Mondo W_{t} al tempo t, considerando che il sistema ha una parziale conoscenza dell'ambiente in cui gira, è l'insieme delle proposizioni del primo ordine di un dominio prefissato (chiamate *fatti*) che sono vere in quel momento, in pratica ciò che conosce il sistema dell'ambiente durante l'esecuzione.
Per la specifica dei Goals viene utilizzato il GoalSPEC, un linguaggio basato sul linguaggio naturale adatto per l'uso in ambito di business. GoalSPEC è stato definito per descrivere le principali operazioni disponibili in BPMN, con l'obiettivo di specificare cosa si vuole ottenere senza indicare come ottenerlo e inoltre è un linguaggio pensato per l'uso in diversi domini di applicazione potendo definire un'ontologia per ogni dominio specifico.

Definito cosa il sistema deve ottenere attraverso la specifica dei Goals, il come ottenerla è compito delle Capabilities. Dall'architettura service-oriented il concetto di Capability può essere definito come un micro-servizio da integrare nel sistema, utilizzato per espletare una determinata azione. Nell'ambito di MUSA la Capability ha una natura duale:

- la $capability\ astratta$ è la descrizione formale dell'effetto che essa produce
- la $capability\ concreta$ è una piccola unità computazionale che esegue effettivamente il servizio concreto

Ogni Capability può essere inserita in ogni momento, non essendo una componente che forma la struttura del sistema e inoltre, essendo micro-servizio, può essere sviluppata da Developer esterni per ottenere diverse soluzioni.

Self-Configuring per arrivare ad una soluzione

Trovare una soluzione coincide con il Proactive Means-End Reasoning, ovvero il problema di trovare l'insieme minimo e completo di operazioni per ottenere i Goals inseriti, avendo a disposizione un insieme di Capabilities e partendo da uno Stato del Mondo W_0.
Nel momento in cui si effettua una ricerca delle soluzioni ottimali, considerando i Goals come punti da raggiungere e le Capabilities come evoluzioni da un punto all'altro, la self-configuration diventa un problema di ricerca nello spazio. L'algoritmo principale, infatti, si basa su un Grafo Computazionale che viene costruito in maniera incrementale per l'esplorazione delle varie combinazioni delle Capabilities, dove ogni nodo ingloba uno Stato del Mondo differente. Vengono estrapolati i cammini dal Grafo che, dal nodo iniziale, portano a nodi che contengono Stati del Mondo in cui sono soddisfatti tutti i Goals, per rappresentare le diverse soluzioni.

Feedback Loop

Riprendendo quanto scritto sopra sull'importanza del Feedback Loop nei Sistemi Adattativi, in MUSA sono presenti due loop principali, uno interno ed uno esterno.

- Durante i periodi di stabilità del sistema viene eseguito il loop interno, portando alla creazione di una configurazione (il processo di self-configuring sopra descritto) e all'esecuzione di un sotto ciclo implementato seguendo il modello MAPE :

  - Monitor - Componente che si occupa di aggiornare il sistema sullo stato dell'ambiente
  - Analyze - Componente che controlla la stabilità dell'ambiente
  - Plan - Componente che ordina le Capabilities in modo da preferire quelle che portano al soddisfacimento dei Goals più velocemente
  - Execute - Componente che si occupa di invocare il servizio appropriato

- Non appena vengono rilevati malfunzionamenti o una violazione dei requisiti, il sistema diviene instabile e mostra la sua natura adattativa cercando di ritornare ad uno stato di stabilità. Sfruttando la proprietà del Self-Healing viene bloccata l'esecuzione del sistema per eseguire il loop esterno composto da: un monitor per i Goals, che permette la loro manipolazione, un Monitor per l'Ambiente e per i Requisiti, che verifica il corretto processamento delle variabili d'ambiente e dei requisiti e dal Proactive Means-End Reasoning che riporta alla creazione di una configurazione.

Orchestrazione a run-time e Oloni

MUSA è un Sistema Multi-Agente (M.A.S) composto da più Agenti che interagiscono in un ambiente comune, quindi il comportamento del sistema è affidato a

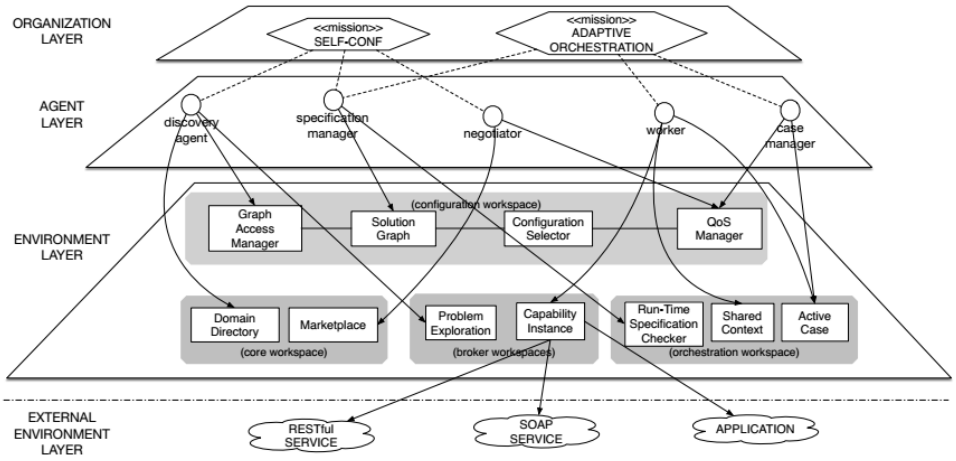


Figura 2.1: Organizzazione degli Agenti e degli Artefatti in MUSA 2.0

loro. Un Agente è un'entità autonoma che può vivere in un ambiente complesso e che osserva tramite dei "sensi" che gli permettono di percepire e di agire nell'ambiente, per portare a termine dei compiti.
L'organizzazione degli Agenti in MUSA, per generare la composizione di servizi, è gestita tramite sistemi multi-agente Olonici. Un Olone è un sistema con una struttura auto-organizzata in evoluzione, che può essere identificata nello stesso tempo sia come un individuo, che come una parte di un'entità più grande. Questo metodo garantisce una maggiore robustezza oltre ad una condivisione del sapere e ad una coordinazione distribuita.
L'architettura di MUSA è basata sull'approccio di JaCaMo, un framework utilizzato per la progettazione e la programmazione di Agenti che cooperano nello stesso ambiente in un MAS. Un sistema programmato in JaCaMo è caratterizzato da un'organizzazione degli Agenti tramite **Mo**ise, da un linguaggio per implementare ed eseguire gli Agenti programmato in **Ja**son e da un linguaggio per lavorare in un ambiente basato su Artefatti distribuiti programmato in **Ca**rtago.
In Figura 2.1 è possibile vedere come l'ambiente sia modellato tramite un layer (il terzo scendendo dall'alto), che contiene risorse e oggetti (programmati in Java) che l'Agente può usare e manipolare a run-time, chiamati Artefatti. In quest'architettura le azioni di un Agente sono rappresentate da operazioni descritte negli Artefatti e ciò che percepiscono dall'ambiente sono delle proprietà presenti o eventi generati negli Artefatti. Sopra il layer degli Agenti è presente l'Organization Layer che li organizza secondo un certo modello di società. Inoltre, il layer più in basso rappresenta gli oggetti esterni al sistema con i quali gli Agenti interagiscono.

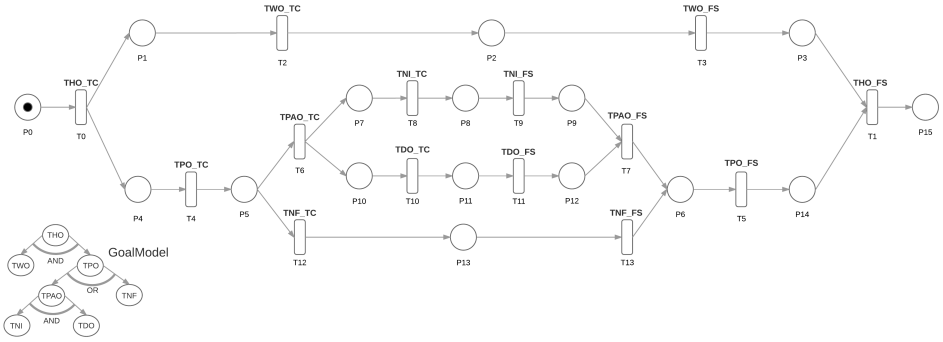


Figura 2.2: Rappresentazione GoalModel e Rete di Petri associata

2.2 Supervisori in un Sistema Goal-Oriented

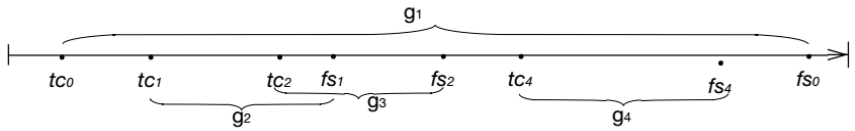
Nell'attuale configurazione di MUSA il soddisfacimento dei Goals assume un aspetto statico, in quanto è possibile ottenerlo solamente supervisionando le proprietà statiche del sistema e dell'ambiente. Un Goal viene raggiunto se esistono due Stati del Mondo W_{i} e W_{f} (sempre considerando che si opera sul Grafo Computazionale per la Self-Configuration e che quindi i due Stati devono trovarsi in un cammino del Grafo) tali che W_{i} soddisfi la Trigger Condition del Goal e W_{f} soddisfi il Final State del Goal (con i $\leq$ f). In un Goal la Trigger Condition tc descrive le proprietà che devono essere presenti in uno Stato del Mondo affinché possa iniziare il soddisfacimento del Goal, mentre il Final State fs rappresenta lo Stato che si vuole ottenere per soddisfare il Goal.
La composizione di più Goal è gestita tramite il Goal Model, un grafo diretto dove i Goal sono Nodi e le Relazioni tra i Goals sono Archi. Possono esserci due tipi di relazioni:

- In una Relazione AND il Goal Root è soddisfatto se tutti i sub-Goals sono soddisfatti $g_{\mathrm{r}} \Rightarrow (g_1 \wedge g_2 \wedge ... \wedge g_{\mathrm{n}})$
- In una Relazione OR il Goal Root è soddisfatto se almeno uno dei sub-Goals è soddisfatto $g_{\mathrm{r}} \Rightarrow (g_1 \vee g_2 \vee ... \vee g_{\mathrm{n}})$

In questa configurazione il Supervisore viene rappresentato tramite una Rete di Petri, costruita partendo dal Goal Model, in cui sono presenti solamente tre casi da da analizzare: caso Base, caso Parallelo (AND) e caso Condizionale (OR). Come è possibile vedere dalla Figura 2.2 ogni caso è rappresentato nella Rete con un Template diverso. Inoltre, in questa Rete i Posti sono di attesa e alle Transizioni sono associati tc o fs e ogni Goal viene rappresentato tramite i suoi tc e fs e le sue relazioni. L'avanzamento nella Rete viene gestito facendo Scattare ogni Transizione, a cui è associata una tc o un fs soddisfacibile nello Stato del Mondo attuale.

2.2.1 Supervisore di Formule LTL

Alla base del Monitoring in MUSA, per come visto finora, vi è una struttura limitata ad un Goal Model con le sole relazioni logiche AND e OR. In questo contesto, l'operazione monitoraggio di un Goal consiste nel controllare che ci siano due Stati del Mondo che soddisfino tc e fs o, nel caso di Goal composti tramite le relazioni, controllare che i sub-Goals siano soddisfatti. Come mostra la seguente immagine, l'unica relazione temporale che è possibile controllare in questo modello è solamente quella che vi è tra tc e fs, ovvero che la prima deve verificarsi in uno Stato del Mondo precedente allo Stato in cui si verifica la seconda (e implicitamente la relazione tra root Goal e sub-Goals).



La Logica Temporale Lineare si presta bene alla descrizione dei Goals introducendo delle proprietà temporali che aumentano la potenza del modello. Con le formule LTL è possibile descrivere proprietà che deve possedere il sistema in un determinato istante nel tempo, aumentando le possibili specifiche dei Goals da inserire nel sistema.
Come già visto il Monitoring per LTL avviene tramite il Model Checking, attraverso la costruzione dell'Automa di Büchi. Data una formula LTL che rappresenti il comportamento voluto dal sistema, la costruzione dell'Automa di Büchi avviene in quattro passi principali:

i. Forma Normale
   Data una formula φ, questa viene trasformata in forma normale negativa in cui le negazioni sono riportate alle proposizioni atomiche (es. $\neg$(X φ) $\equiv$ X ($\neg\varphi$)). Inoltre vengono considerati solamente gli operatori fondamentali true, false, $\vee$ e $\wedge$ per gli operatori logici e X, U e R per quelli temporali. Il resto degli operatori viene sostituito con la loro derivazione (es. F $\varphi \equiv$ **true** U φ ; G $\varphi \equiv$ **false** R φ).

ii. Grafo
   Per ottenere l'Automa viene costruito un grafo partendo dalle espansioni della formula normale φ. Ogni nodo nel grafo contiene le formule che deve espandere (il primo nodo inizia espandendo la formula iniziale) e i letterali, rappresentanti le proposizioni atomiche, che ha processato.
   Se un nodo espande una formula del tipo X ψ, verrà creato un nuovo nodo con un arco entrante dal primo e con ψ da espandere.
   Se un nodo espande una formula del tipo $\vee$, U o R, viene creato un nodo copia per considerare entrambe le formule della disgiunzione: se $\phi_1 \vee \phi_2$, ϕ_1 sarà espansa in un nodo e ϕ_2 nell'altro. In questo caso sono fondamentali

le uguaglianze:
ϕ_1 U $\phi_2 \equiv \phi_2 \vee (\ \phi_1 \wedge$ X(ϕ_1 U ϕ_2))
ϕ_1 R $\phi_2 \equiv (\phi_2 \wedge \phi_1) \vee (\phi_2 \wedge$ X(ϕ_1 R ϕ_2))

iii. Automa Generalizzato
Costruito il Grafo, l'Automa viene ottenuto come segue:

- Σ è la congiunzione dei Letterali usati
- S è dato dall'insieme dei nodi del Grafo
- La funzione δ è rappresentata dagli archi
- S_0 è il nodo iniziale (oppure l'insieme dei nodi iniziali)
- Vi è una funzione di etichettatura che ad ogni stato associa i Letterali usati nel nodo corrispondente
- L'insieme degli stati finali F è dato da tutti gli stati associati a nodi che non hanno espanso formule del tipo ϕ_1 U ϕ_2 oppure che hanno espanso formule ϕ_2 (provenienti da una formula U già espansa in un nodo precedente), in sintesi i nodi in questione non devono trovarsi in una forma di Eventualità (caso dell'Until)

iv. Automa Finale
L'ultimo passo consiste nel trasformare un Automa di Büchi Generalizzato con k insiemi di accettazione in un Automa di Büchi. Il procedimento consiste nel ricopiare k volte gli stati dell'Automa Generalizzato e considerare le varie transizioni.

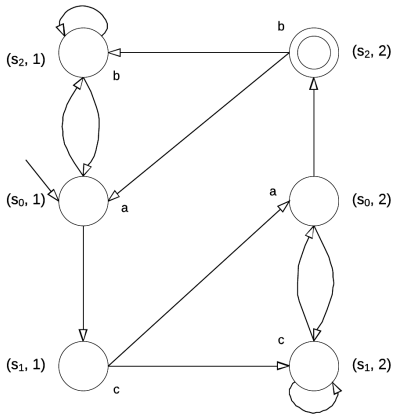


Figura 2.3: Esempio di Automa Finale ottenuto dopo aver effettuato i quattro passi

La complessità computazionale per l'algoritmo di costruzione è $O(2^{|\varphi|})$, ovvero esponenziale alla dimensione della formula in quanto l'automa contiene un numero di stati proporzionale al numero di sottoformule di φ.

Dopo aver ottenuto l'Automa di Büchi finale è possibile effettuare l'operazione di Model Checking modellando il sistema con un altro Automa di Büchi A_{sys}. La verifica del sistema avviene tramite i linguaggi descritti dai due automi, ma l'Automa finale viene costruito partendo dalla formula negata $\neg\varphi$.
Infatti la verifica consiste nel controllare che l'intrsezione tra il linguaggio accettato da A_{sys} e il linguaggio accettato dall'Automa di Büchi costruito partendo dalla negazione della formula che rappresenta il comportamento voluto $A_{\neg\varphi}$ sia vuota

$$L(A_{\text{sys}}) \cap L(A_{\neg\varphi}) = \emptyset$$

Per verificare ciò, viene costruito l'Automa che rappresenta l'intersezione tra i due $A_{\text{p}} = A_{\text{sys}} \cap A_{\neg\varphi}$ e si controlla se il linguaggio di A_{p} è vuoto.

2.2.2 Il Problema

Considerando l'approccio classico appena descritto per il Model Checking LTL, il Problema consiste nel Monitoring del sistema tramite le formule LTL sia durante la costruzione del Grafo Computazionale che porta alla soluzione, sia durante l'Esecuzione effettiva delle operazioni che producono un risultato. Il Supervisore non assume gli aspetti tipici di un Model Checker, ma si basa sugli stessi principi per la verifica di formule LTL. La costruzione di questo Supervisore, infatti, si basa sull'utilizzo delle Reti di Petri e dall'integrazione con le restanti componenti di MUSA. In questa maniera la costruzione del modello risulta più veloce rispetto a quella classica dell'Automa di Büchi, in quanto si tratta di un'operazione di tempo lineare rispetto alla dimensione della formula.

Capitolo 3

Soluzione - Teoria

In questo capitolo procedo a mostrare come avvenga la costruzione di un Supervisore per formule LTL grazie all'utilizzo della Rete di Petri ed ad argomentare la parte Teorica su cui si fonda implementazione di questo Supervisore in MUSA, mostrando un nuovo modello in grado di supportare la verifica del linguaggio LTL. Il problema alla base è sempre quello, descritto nel capitolo precedente, del Proactive Means-End Reasoning, quindi la soluzione andrà a ricadere nell'ambito dell'algoritmo che lo risolve: la $PMR\ ability$. Questa è una soluzione incentrata sull'utilizzo degli Stati del Mondo per la costruzione dinamica del Grafo Computazionale. La strategia attuata si basa sulla composizione di più Capabilities che portano a diversi Stati Computazionali, che rappresentano i diversi comportamenti attuabili dal sistema. La costruzione è guidata dalla specifica dei Goals, utilizzando adesso le Formule LTL, in modo tale che il comportamento del sistema porti al soddisfacimento di questi. Il Supervisore, ovvero colui che permette il soddisfacimento dei Goals, è modellato tramite l'utilizzo della Rete di Petri, così come in precedenza. La differenza del nuovo approccio sta nel grado di soddisfacimento del Goal che, dall'essere considerato in maniera statica solamente tramite Trigger Condition e Final State, passa ad essere considerato dinamicamente durante l'avanzamento da uno stato all'altro.

3.1 Costruzione del Supervisore

Per la costruzione di un modello del Supervisore basato sulla Rete di Petri, viene considerato il processo che porta alla costruzione di un Automa di Büchi a partire da una formula LTL. È possibile comparare una Rete di Petri etichettata con un Automa a Stati Finiti in quanto è simile la logica di transizione da uno stato all'altro e in casi specifici esiste anche un'equivalenza nei linguaggi generati. Lo studio degli Automi di Büchi per le formule LTL porta, quindi, ad una serie di modelli realizzabili di Rete di Petri in grado di rappresentare le formule base del linguaggio.

3.1.1 Forma Normale Negativa

Per la costruzione di ogni Rete viene utilizzata la Forma Normale Negativa, ovvero la forma in cui tutte le negazioni appaiono solamente davanti le proposizioni atomiche. Ogni formula LTL è logicamente equivalente ad una formula in forma normale negativa, basta applicare le seguenti trasformazioni:

$$\neg(\mathrm{X}\ \varphi) \Rightarrow \mathrm{X}(\neg\varphi)$$
$$\neg(\mathrm{F}\ \varphi) \Rightarrow \mathrm{G}(\neg\varphi)$$
$$\neg(\mathrm{G}\ \varphi) \Rightarrow \mathrm{F}(\neg\varphi)$$
$$\neg(\varphi\ \mathrm{U}\ \psi) \Rightarrow (\neg\varphi)\ \mathrm{R}\ (\neg\psi)$$
$$\neg(\varphi\ \mathrm{R}\ \psi) \Rightarrow (\neg\varphi)\ \mathrm{U}\ (\neg\psi)$$

Rispetto alla costruzione dell'Automa di Büchi verranno considerati anche gli operatori derivati come F e G per una migliore ottimizzazione. Infatti successivamente sarà mostrato come gli operatori derivati abbiano un modello di Rete che deriva dalle Reti degli operatori fondamentali.

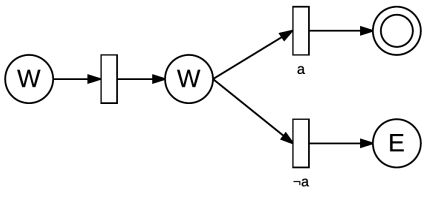
3.1.2 Notazione Rete di Petri

Per la rappresentazione di una formula LTL tramite una Rete di Petri verrà usata una funzione di etichettatura per le Transizioni e verranno considerati diversi stati associati ai Posti.
La funzione di etichettatura associa una Condizione ad una Transizione, in maniera tale che la Transizione può Scattare solamente se la Condizione è verificata. Una Condizione può essere rappresentata da uno dei seguenti tre tipi di informazione: una proposizione atomica, una formula del linguaggio, una composizione di più formule del linguaggio.
Ogni Posto nella rete ha uno Stato associato che rappresenta lo stato della rete quando un Token si trova nel Posto. Gli Stati associati ad un Posto sono di tre tipi: Stato di Accettazione A, rappresenta uno stato in cui la formula rappresentata dalla rete è accettata; Stato di Attesa W, rappresenta uno stato di attesa in cui la formula rappresentata dalla rete non risulta né accettata né rifiutata; Stato di Errore E, rappresenta uno stato in cui la formula rappresentata dalla rete è rifiutata.

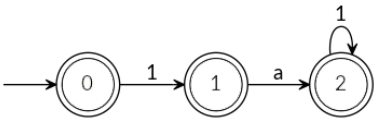
3.2 Modello di Rete per ogni Operatore

Con lo scopo di ottenere un Supervisore LTL, ogni formula base composta da un solo operatore è stata modellata tramite una Rete di Petri, in maniera tale da arrivare ad un'operazione di Monitoring efficiente. Ogni rete è costruita in maniera tale che il Posto iniziale, ovvero quello dove sarà presente il Token per iniziare le operazioni di supervisione, sia sempre quello più a sinistra. Ogni Scatto da un Posto ad un altro equivale ad un passaggio da uno stato temporale ad un altro.

3.2.1 Next



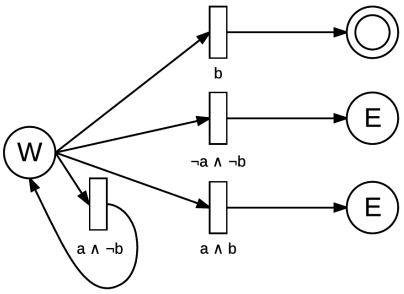
(a) Rete di Petri per Xa



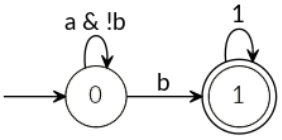
(b) Automa di Büchi per Xa

La costruzione della Rete per la formula Xa si basa sul fatto che nello stato iniziale può essere vera qualsiasi proposizione, perché la verifica avviene nello stato successivo. Infatti la prima Transizione a sinistra scatta per qualunque Condizione, mentre le altre due rimanenti decidono il raggiungimento o meno della formula. Il Token, trovandosi nel Posto di attesa W al centro della Rete, permette ad entrambe le Transizioni con Condizioni a e $\neg a$ di scattare. La differenza con l'Automa è evidente in questo passaggio: nel modello della Rete viene controllata la Condizione a che può portare ad uno stato di accettazione o di errore (Posto di Accettazione in alto e Posto di Errore in basso), mentre nell'Automa viene accettato una stringa di linguaggio che contenga a in quella posizione per poter arrivare ad uno stato finale (comportando un'accettazione della stringa). Quindi è già possibile, nella costruzione delle Reti, notare una differenza con il Model Checking standard.

3.2.2 Until



(c) Rete di Petri per aUb

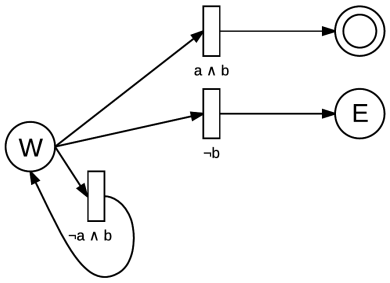


(d) Automa di Büchi per aUb

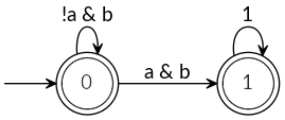
Il Modello della Rete utilizzato per l'operatore Until nella formula aUb è una conseguenza dell'equivalenza $aUb \equiv b \vee (a \wedge X(aUb))$.
La formula quindi, data la disgiunzione, è verificata in due casi:
- Il caso in cui è vera b, rappresentato dalla Transizione con Condizione b che fa scattare il Token ad uno stato di Accettazione.
- Il caso in cui è vera a ed al prossimo stato sarà verificata la formula aUb, rappresentato dalla Transizione con Condizione a che riporta il Token con uno scatto al

Posto iniziale effettuando un ciclo, in maniera tale che la formula aUb possa essere verificata nuovamente allo stato successivo.
Confrontando la Rete con l'Automa è possibile trovare un'altra differenza strutturale. Un Automa descrive un linguaggio e nel riconoscere se una stringa appartiene a questo linguaggio è solamente necessario un percorso che porti da uno stato iniziale ad uno stato finale. Nell'accettazione di una stringa non è quindi contemplato l'uso di stati particolari che indichino l'errore, perché, essendo la stringa da riconoscere finita, prima o poi il percorso nell'Automa arriverà ad uno stato finale o non. Nella Rete invece è necessario riconoscere quando una Condizione porta ad un Errore, in modo da rigettare la formula, per considerare un comportamento infinito e non una stringa finita. Da questa motivazione derivano le due Transizioni che portano ad un Posto di Errore nella Rete.
Infine, oltre il Modello della Rete, bisogna sottolineare come la formula aUb (quindi anche una derivata come $Fa \equiv trueUa$) sia l'unica a poter restare in uno stato di Attesa per un tempo infinito, restando in bilico tra Accettazione ed Errore.

3.2.3 Relase

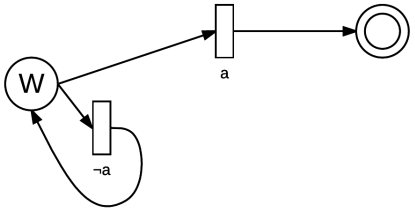


(e) Rete di Petri per aRb

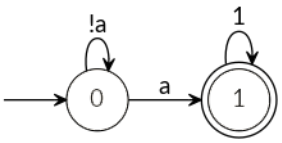


(f) Automa di Büchi per aRb

Analogamente alle considerazioni fatte per l'operatore Until, anche il Modello di Rete per Relase è basato sull'equivalenza $aRb \equiv (a \wedge b) \vee (b \wedge X(aRb)$). Quindi:
- a e b vere contemporaneamente è un comportamento riportato nella Rete dalla Transizione più in alto che porta ad un Posto di Accettazione.
- La Condizione in cui b è vera e deve essere verificata la formula aRb nello stato temporale successivo è associata nuovamente ad una Transizione che implementa un ciclo.
La struttura della Rete è simile a quella dell'operatore Until, ma la differenza sostanziale risiede nel Posto iniziale. Infatti, diversamente dall'Until, una formula del tipo aRb è verificata anche se il secondo operando b è vero in tutti gli stati temporali futuri all'infinito, rendendo il primo Posto di Attesa un Posto Accettato all'infinito (quindi alla fine dell'esecuzione).



(g) Rete di Petri per Fa

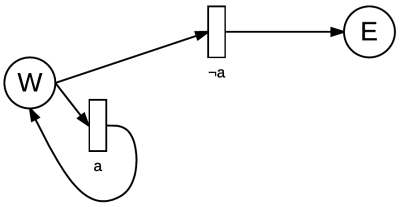


(h) Automa di Büchi per Fa

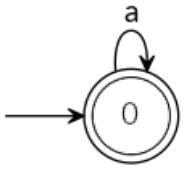
3.2.4 Finally

Essendo Finally un operatore derivato dall'operatore fondamentale Until ($Fa \equiv trueUa$), anche il modello della Rete viene derivato dal modello Until. Considerando $true$ come una proposizione sempre vera, $trueUa$ può essere definita come tutto quello che non è a Until a : $(\neg a)Ua$. In questa maniera, nella Rete che modella $(\neg a)Ua$, la Transizione che effettua il ciclo sullo stato iniziale avrà come Condizione associata $\neg a$ e le Transizioni che portano ad un Errore non potranno mai scattare perché le Condizioni associate saranno sempre false : $(\neg(\neg a) \wedge \neg a) \rightarrow false$, $((\neg a) \wedge a) \rightarrow false$. Effettuando quindi delle semplificazioni nella struttura si ottiene il Modello di Rete per la formula Fa.
È importante sottolineare l'assenza di Posti di Errore poiché l'operatore Finally aspetterà all'infinito solamente che si verifichi a e di conseguenza non potrebbe mai verificarsi un Errore.

3.2.5 Globally



(i) Rete di Petri per Ga



(j) Automa di Büchi per Ga

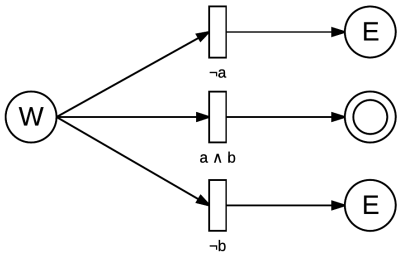
Anche il Modello Globally è ottenuto considerando la derivazione di questo operatore da un operatore fondamentale. Si tiene conto dell'equivalenza $Ga \equiv falseRa$, dove $false$ è una proposizione che non si avvererà mai. Effettuando delle semplificazioni dalla Rete che modella $falseRa$, la Transizione che porta al Posto di Accettazione in alto sarà rimossa perché non sarà mai verificata $(false \wedge a) \rightarrow false$, mentre il ciclo sul primo Posto di Attesa e la Transizione che porta ad un Errore saranno condizionati rispettivamente da a e $\neg a$.

Una formula del tipo Ga, risulta "accettata" durante il ciclo nel Posto di Attesa, ma prima o poi potrebbe essere Rifiutata per il verificarsi di uno stato in cui a non sia vera. In questo aspetto è possibile scorgere la dualità tra Finally e Globally: mentre il primo aspetta all'infinito che arrivi uno stato di Accettazione, il secondo aspetta all'infinito che arrivi uno stato di Errore.

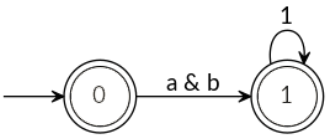
3.2.6 Operatori Logici

Gli operatori logici sono tutti caratterizzati da una Rete simile in cui avviene solamente uno scatto da uno Posto di Attesa ad un Posto di Accettazione o Errore. Questo è dovuto al fatto che basta verificare la Condizione una volta per verificare la formula, in quanto il comportamento di questi operatori è statico.

And



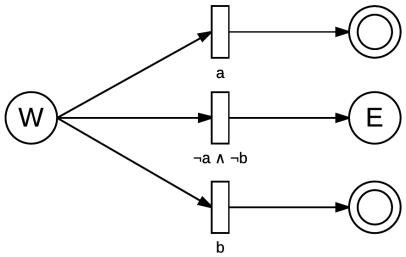
(k) Rete di Petri per $a \wedge b$



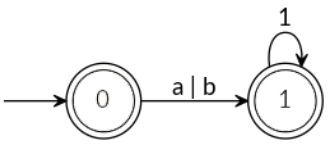
(l) Automa di Büchi per $a \wedge b$

Nel Modello And, una formula viene accettata solamente se si verificano sia a che b, in tutti gli altri casi si passa ad un Posto di Errore.

Or



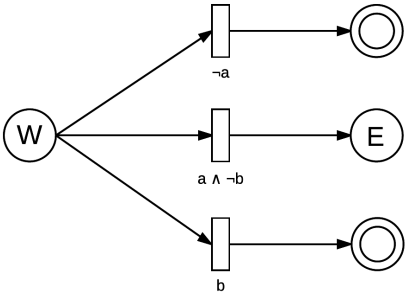
(m) Rete di Petri per $a \vee b$



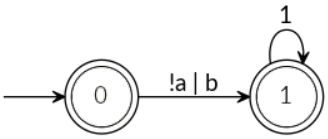
(n) Automa di Büchi per $a \vee b$

Nel Modello Or scatta la Transizione con associata la Condizione che viene verificata per prima tra a e b, se entrambe sono rigettate allora si passa al Posto di Errore.

Implicazione



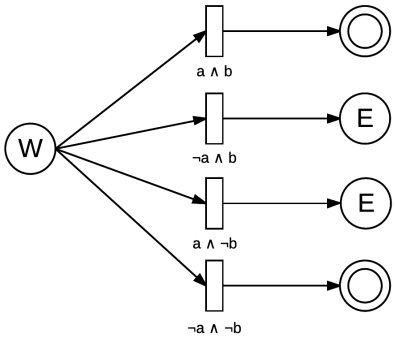
(o) Rete di Petri per $a \Rightarrow b$



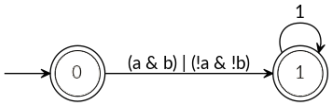
(p) Automa di Büchi per $a \Rightarrow b$

Per il Modello Implicazione basta considerare l'equivalenza $a \Rightarrow b \equiv \neg a \vee b$ ed utilizzare il Modello di Rete Or.

Doppia Implicazione



(q) Rete di Petri per $a \Leftrightarrow b$



(r) Automa di Büchi per $a \Leftrightarrow b$

La Doppia Implicazione $a \Leftrightarrow b$ equivale a $(a \wedge b) \vee (\neg a \wedge \neg b)$, difatti per ottenere il Modello in questo caso si utilizza il Modello di Rete Or con i due operandi $(a \wedge b)$ e $(\neg a \wedge \neg b)$. La Condizione che porta all'Errore sarà $\neg(a \wedge b) \wedge \neg(\neg a \wedge \neg b)$, che è possibile trasformare in $\neg a \wedge b$ per una Transizione e $a \wedge \neg b$ per l'altra.

3.3 Composizione di Reti

Dati i Modelli delle Reti per formule base composte da un solo operatore, il passo successivo consiste nel costruire un Modello che possa supportare tutte le infinite formule esprimibili in LTL. Essendo quest'ultimo un linguaggio formato da formule ottenibili dalla composizione degli operatori, la composizione delle Reti dei Modelli sopra descritti si presta come soluzione.
Per esprimere formule del tipo θ = $F(Ga)$ basta utilizzare il modello di Rete per

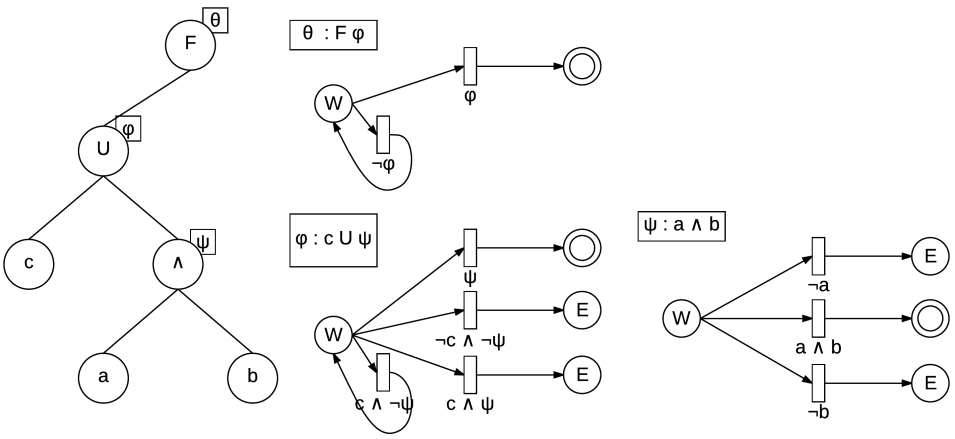


Figura 3.1: Esempio di composizione di più Reti di Petri partendo dalla formula $F(aU(a \wedge b))$

Fa e sostituire le Condizioni che interessano a con $\varphi = Ga$. In questo modo si ottengono due Reti dove la prima, che rappresenta $F\varphi$, dipende dalla seconda, che rappresenta Ga.
La Composizione di più Reti per modellare una qualsiasi Formula LTL, allora, si ottiene tramite un Albero Binario dove:

- La radice e i nodi intermedi contengono operatori LTL
- I figli di un nodo contengono gli operandi dell'operatore contenuto nel nodo
- Le foglie contengono solamente proposizioni atomiche

Questa struttura permette di tenere una gerarchia tra formule e sotto-formule per poter gestire le dipendenze tra di loro. Una volta ottenuto l'albero dalla formula iniziale, visitando ogni nodo, è possibile costruire una Rete per ogni sotto-formula seguendo i Modelli per ogni singolo operatore.

3.4 Operazione di Monitoring

Ottenuto un insieme di Reti di Petri interconnesse tramite le dipendenze create durante la fase di Composizione, è possibile iniziare l'operazione di Monitoring del sistema. La dipendenza di una formula dall'altra, nella Rete, si traduce in una Condizione associata ad una Transizione. Quindi, una formula dipende da una sotto-formula se, nella Rete che modella la formula, è presente come condizione la sotto-formula tramite un riferimento.
Prendendo come esempio la configurazione in Figura 3.1, è possibile vedere come la formula θ dipenda da φ e sua volta φ dipenda da ψ. Per ogni stato temporale si inizia il monitoring dalla prima formula, ovvero quella contenuta nella radice dell'Albero Binario, e si controllano le Transizioni che possono scattare nella Rete.

Nell'esempio, ipotizzando che il Monitoring sia appena iniziato e che il Token si trovi nel Posto iniziale di Attesa, le Transizioni che possono scattare nella prima Rete sono φ e $\neg\varphi$ (per brevità le Transizioni verranno indicate con le formule delle Condizioni associate). La Transizione φ scatterà solamente se la Rete φ è in uno stato di Accettazione, mentre la Transizione $\neg\varphi$ scatterà solamente se la Rete φ è in uno stato di Errore. Quindi il controllo è passato alla Rete che rappresenta φ e conseguentemente la Rete φ per far scattare una Transizione dovrà passare il controllo alla Rete ψ. Nel caso in cui una Rete di una sotto-formula, come nell'esempio ϕ, rimanga in uno stato di Attesa, non può essere né in uno stato di Accettazione né in uno di Errore. Allora nessuna Transizione della sopra-formula può scattare e la configurazione di Token in tutte le Reti viene conservata per essere continuata nel prossimo stato temporale. Ciò permetterà di continuare la verifica di formule che richiedono più stati per essere valutate.
L'operazione di Monitoring quindi consiste, andando di stato in stato, nell'avanzamento dei Token nelle Reti e nel controllo delle Condizioni associate alle Transizioni. Le Condizioni, a sua volta, possono essere verificate tramite altre Reti nel caso di formule o tramite gli attributi dell'ambiente del sistema nel caso di proposizioni atomiche. L'operazione di Monitoring si concluderà quando la Rete che modella la prima formula (quella contenuta nella radice) avrà raggiunto un Posto di Accettazione o di Errore dal quale non può più scattare.

Capitolo 4

Soluzione - Architettura

Per l'implementazione del Supervisore in MUSA, in questo capitolo, considero il suo ruolo nel sistema, la sua costruzione partendo da una formula LTL e l'effettivo comportamento realizzato.
Anche se MUSA è basato su una programmazione orientata agli Agenti, implementata tramite Jason, nella versione 2.0 assumono una grande importanza gli artefatti, programmati in Java, che rappresentano le operazioni effettuabili da questi Agenti. Il Supervisore quindi, rappresentando un'operazione effettuata da un Agente, è implementato in Java tramite un insieme di classi che portano alla sua costruzione e che sono integrate con le altre parti del sistema per l'esecuzione del Monitoring.
Per una più chiara comprensione delle procedure, utilizzo come esempio un caso basato sulla formula $\theta = (Fa) \wedge (Gb)$, mostrando tutti i passaggi dall'inserimento nel sistema della formula sotto forma di Goal sino all'esecuzione del Supervisore costruito su di questa.

4.1 Supervisione nell'architettura di MUSA

Come discusso nel Capitolo 2 e in particolare tramite la Figura 2.1, l'architettura di MUSA 2.0 è organizzata su più layer, tra i quali l'Agent Layer e l'Environment Layer. Nell'Agent Layer sono presenti tutti gli Agenti che tramite le loro azioni costituiscono il comportamento del sistema tra cui, in particolare, la categoria di Agenti che si occupa di trovare diverse soluzioni per il soddisfacimento dei Goals, rappresentata dal Discovery Agent. Nell'Environment Layer, tra tutti i workspaces che rappresentano gli ambienti di lavoro dove possono interagire gli Agenti, ogni singolo Discovery Agent opera nel suo Discovery Workspace, in particolare tramite l'artefatto ProblemExploration. Ogni Discovery Agent è costantemente a lavoro per generare nuovi Stati del Mondo tramite le Capabilities che possiede. In questo ambito, l'agente comunica con il GraphAccessManager sia per ricevere, dal SolutionGraph, lo Stato del Mondo sul quale deve lavorare, sia per mandare, eventualmente, dei nodi da aggiungere al Grafo. La comunicazione e le operazioni vengono eseguite in loop seguendo quest'ordine:

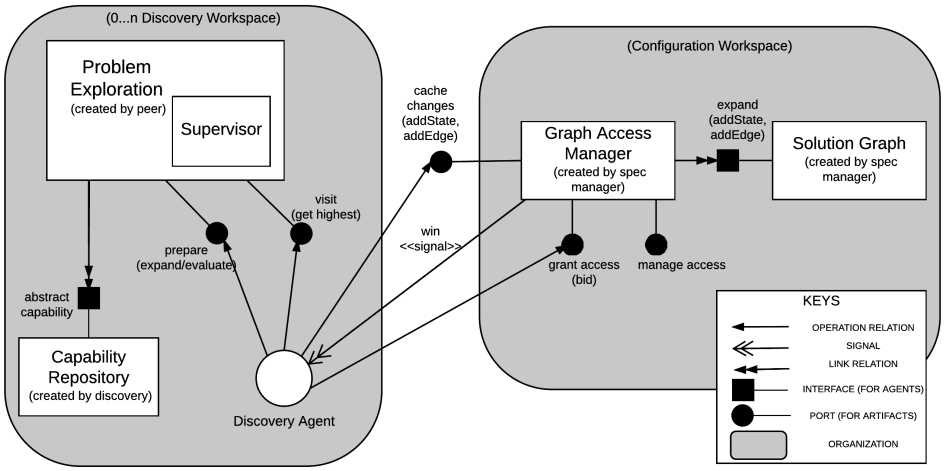


Figura 4.1: Porzione di sistema interessata per il Supervisore in MUSA 2.0

- L'agente riceve un nuovo nodo che incapsula uno Stato del Mondo appena inserito nel SolutionGraph e lo inserisce nella lista di nodi che deve "espandere"
- Ogni agente è costantemente a lavoro sui nodi che riceve, infatti per ogni Capability che possiede, cerca di applicarla allo Stato del Mondo incapsulato in un nodo, in maniera tale da espanderlo e per ottenerne uno nuovo
- Ogni nuovo Stato del Mondo espanso da un agente viene re-incapsulato in un nodo con associato uno punteggio (Score), che indica il grado di vicinanza del nuovo Stato del Mondo al raggiungimento dei Goals
- Ad intervalli di tempo prestabiliti, viene indetta un'asta nella quale ogni agente effettua una puntata di un nodo, scegliendo quello con punteggio più alto. Vince l'asta l'agente con il nodo con punteggio migliore e questo nodo viene dapprima aggiunto al SolutionGraph e dopo distribuito agli altri agenti che ci lavoreranno su

In questo ciclo continuo che caratterizza la vita del Discovery Agent, la Supervisione prende atto durante il lavoro di Espansione che l'agente esegue costantemente su tutti i nodi che incapsulano gli Stati del Mondo che riceve.

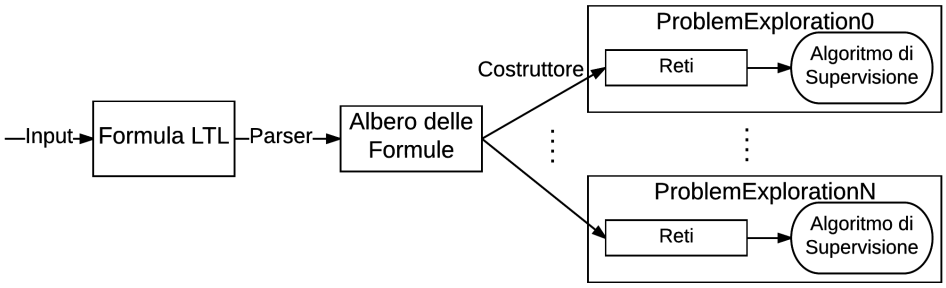
4.1.1 Espansione di un Nodo

L'operazione di Espansione di un nodo sta al centro del funzionamento dell'artefatto ProblemExploration e quindi è fondamentale comportamento dell'Agente. L'espansione viene effettuata estrapolando lo Stato del Mondo da un nodo e applicando le "Evoluzioni" che può apportare una Capability compatibile. Ogni Capability

possiede delle Pre-Condizioni e delle Post-Condizioni: le pre-condizioni permettono di controllare se una Capability è compatibile lo Stato del Mondo, mentre le post-condizioni permettono di generare un nuovo Stato a partire da quello precedente aggiungendo e/o rimuovendo predicati.
Nella moltitudine di Stati che vengono elaborati il Supervisore ha il compito di monitorare l'avanzamento nella soddisfazione dei Goals. Infatti, durante questo processo, dato che ogni Capability associata all'artefatto viene testata, è possibile che ne venga applicata una che porta ad uno stato di errore o che sia meno efficace rispetto ad un'altra, perciò è necessario l'utilizzo di un Supervisore. In questo contesto, subito dopo aver elaborato un nuovo Stato del Mondo tramite le "Evoluzioni" applicate da una Capability, inizia l'operazione di Supervisione.

4.2 Costruzione

Nel sistema MUSA 2.0 la struttura del Supervisore viene mantenuta a prescindere dall'artefatto ProblemExploration. Infatti la costruzione di una parte del Supervisore avviene in seguito all'inserimento della formula LTL e successivamente ogni artefatto costruisce la sua copia per poter eseguire le operazioni.



4.2.1 Albero delle Formule

Partendo dalla formula LTL che l'utente inserisce in input nel sistema la prima operazione che viene effettuata è quella di creazione dell'Albero Binario delle Formule. Per ottenere questa struttura è necessaria l'azione di un Parser che sia in grado di riconoscere il linguaggio LTL e restituire un oggetto che descriva interamente la formula. In particolare il Parser utilizzato è stato implementato utilizzando la libreria Java ANTLR e permette di riconoscere ogni formula LTL e la logica del prim'ordine. Durante la scansione della formula viene riempita una Pila con le proposizioni atomiche e gli operatori che il Parser incontra, che nel caso della formula θ sarebbe: $\wedge, G, F, b, a$.
Non appena verificata la stringa in input come facente parte del linguaggio LTL, inizia la costruzione, tramite gli elementi della pila, dell'Albero Binario, rappresentato dalla classe FormulaBT. Inoltre avviene la riduzione in Forma Normale

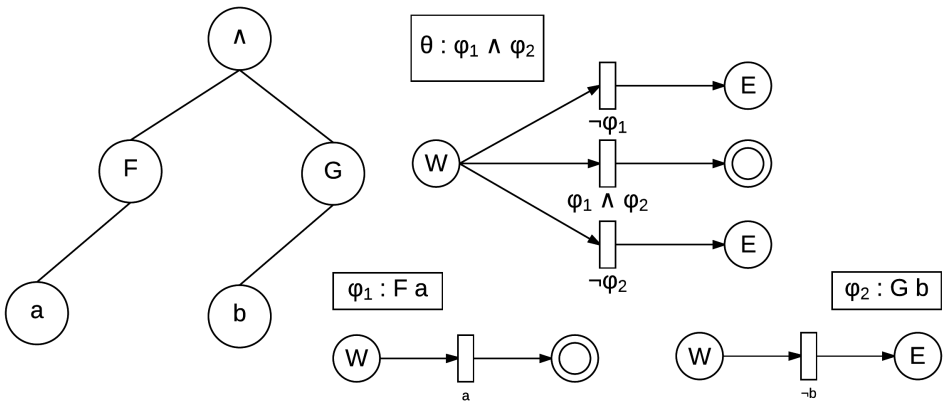


Figura 4.2: Esempio di Costruzione dell'Albero e delle Reti per la formula $\theta = (Fa) \wedge (Gb)$

Negata applicando le regole di conversione e portando tutte le negazioni fino alle foglie.

4.2.2 Reti di Petri

Una volta costruito l'Albero Binario è possibile conoscere la gerarchia di formule e sotto-formule, quindi si può passare alla costruzione delle Reti di Petri che le modellano. Ogni artefatto ProblemExploration, nel momento in cui viene istanziato, costruisce la propria struttura di Reti tramite l'Albero. In questa maniera le strutture sono identiche per tutti gli artefatti perché provenienti dallo stesso Albero, ma in ognuna può essere presente una diversa configurazione di Token che permette di operare in indipendenza.
La costruzione avviene tramite un algoritmo ricorsivo che effettua una visita sull'Albero Binario. Per ogni nodo che contiene un Operatore LTL, costruisco la Rete di Petri che modella questo operatore e come Condizioni nelle Transizioni utilizzo la formula del sotto-albero sinistro e la formula del sotto-albero destro (nel caso di formule unarie solamente il sotto-albero sinistro), quindi visito i figli. Durante questa operazione viene mantenuto anche un Dizionario, implementato tramite un HashMap, che associa una Rete alle sue sotto Reti. Infine, si arriverà alle foglie, dove sono presenti solamente proposizioni atomiche, e quindi le Condizioni che avrà il padre saranno del tipo Semplice.
Seguendo l'esempio della formula θ in Figura 4.2, come prima operazione verrà creata la Rete And con Condizioni associate φ_1 e φ_2, successivamente le Reti Finally e Globally con le Condizioni Semplici a e b. Inoltre è possibile notare come le Reti Finally e Globally (così come Until e Relase) siano state private dei cicli nell'implementazione.

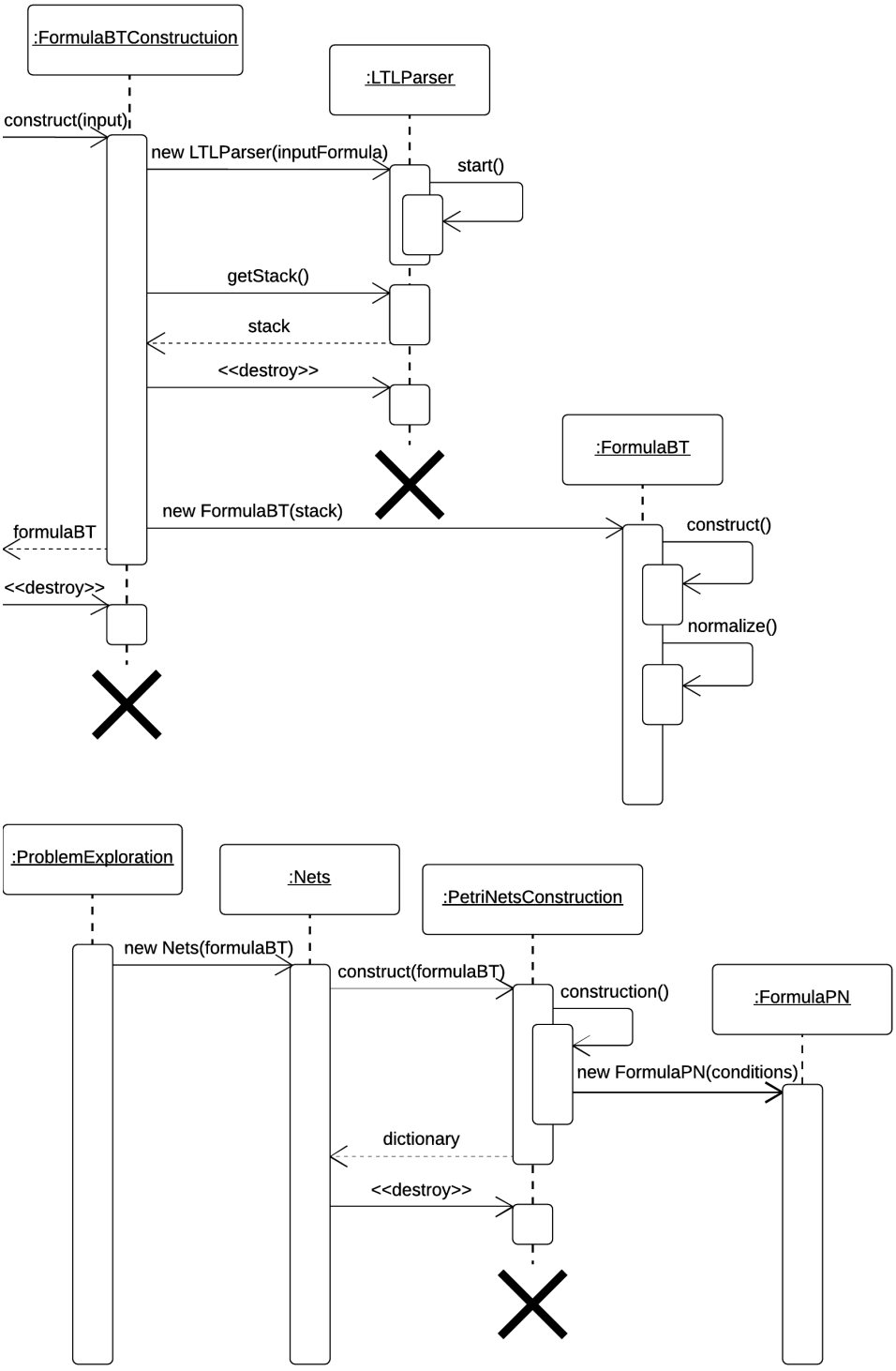


Figura 4.3: Diagrammi di Sequenza UML per le operazioni di Costruzione

4.3 Supervisione

L'operazione di Supervisione viene eseguita dal sistema una volta generato un nuovo Stato del Mondo tramite l'Evoluzione ottenuta con una Capability. In questa condizione infatti lo Stato appena creato deve essere valutato in base al contributo che offre nel soddisfacimento dei Goals. In ProblemExploration quindi, durante il processo di Espansione, viene richiamato il metodo $applyNets$ per avviare l'operazione di Supervisione "applicando", appunto, le Reti al nuovo Stato. Grazie a ciò è possibile confrontare sistema, modellato tramite lo Stato del Mondo, con la formula LTL per ottenere un grado di soddisfacibilità. Quindi uno Stato può portare ad una situazione di Errore o di Accettazione, oppure ad una situazione di Attesa. In quest'ultimo caso, attraverso un indicatore chiamato Score, viene quantificato quanto lo Stato sia vicino a raggiungere l'Accettazione. Andando a riempire il nodo che incapsula lo Stato del Mondo con queste informazioni, sarà possibile ottenere nel SolutionGraph i percorsi che dallo stato iniziale portano a Stati Accettanti (chiamati Stati di Uscita).

4.3.1 Algoritmo

L'algoritmo di Supervisione verifica ricorsivamente lo stato delle Reti delle sotto-formule per l'avanzamento dei Token. La procedura inizialmente viene invocata sulla Rete iniziale, ovvero la Rete che modella la formula presente nella radice dell'Albero Binario delle Formule. Come primo passo vengono cercate le Transizioni che possono Scattare, ovvero quelle che hanno il Posto entrante occupato da un Token. Per ogni Transizione che può scattare viene controllata la Condizione associata e vengono intraprese quattro strade diverse in base al suo tipo.
Se la Condizione è data da una sotto-formula, si controlla lo Stato in cui si trova la sua Rete: se non è ancora presente uno Stato in memoria si inizializza la Rete, se non è stata ancora visitata si richiama la procedura su di questa. La Transizione scatta se lo Stato della Rete combacia con lo Stato della Condizione (dove per Stato della Condizione si intende lo Stato di Accettazione o Errore che può questa assumere, cioè formula semplice φ o negata $\neg\varphi$).
Se la Condizione è Semplice, quindi rappresentata tramite una proposizione atomica, viene controllato se questa è verificata nel sistema ed in caso affermativo scatta la Transizione.
Se la Condizione è Combinata, ovvero formata da due Condizioni, vengono esaminate queste due, che possono essere del tipo Semplice sia Formule, e la Transizione scatta solamente se entrambe sono verificate.
Se, infine, la Condizione è del tipo sempre vera, come quella associata alla prima Transizione nella Rete Next, la Transizione scatta subito.
Di seguito viene presentato lo pseudocodice che rappresenta l'algoritmo appena descritto. La logica dell'algoritmo non impone vincoli legati all'implementazione in MUSA, ma è basata solamente sull'utilizzo delle Reti di Petri collegate tra di loro.

Algorithm 1 Supervision

```
1:  procedure SUPERVISENET(net, visitedNets)
2:      visitedNets ← net
3:      transitions ← transitionsAbleToFire(net)
4:      for t ∈ transitions do
5:          if t can fire then
6:              condition ← getCondition(t)
7:              if condition is a Formula then
8:                  result ← checkFormula(condition,net,visitedNets)
9:                  if result is TRUE then
10:                     fire(t,net)
11:             else if condition is a SimpleProposition then
12:                 result ← checkPropositionInSystem(condition)
13:                 if result is TRUE then
14:                     fire(t,net)
15:             else if condition is a Combined then
16:                 tempVector ← FALSE
17:                 combConditions ← getCombinedConditions(condition)
18:                 for combCond ∈ combConditions do
19:                     if combCond is a Formula then
20:                         result ← checkFormula(combCond,net,visitedNets)
21:                         if result is TRUE then
22:                             tempVector(i) ← TRUE
23:                     else if combCond is a SimpleProposition then
24:                         result ← checkPropositionInSystem(combCond)
25:                         if result is TRUE then
26:                             tempVector(i) ← TRUE
27:                 if tempVector(0) ∧ tempVector(1) is TRUE then
28:                     fire(t,net)
29:             else if condition is AlwaysTrue then
30:                 fire(t,net)
31:
32: procedure CHECKFORMULA(condition, net, visitedNets)
33:     condNet ← getNet(condition)
34:     condNetState ← getNetState(condNet)
35:     condState ← getCondState(condition)
36:     if condNetState is null then
37:         condNet ← init()
38:     if condNet ∉ visitedNets then
39:         superviseNet(condNet,visitedNets)
40:     if condNetState = condState then return TRUE
41:     else return FALSE
```

4.3.2 Altre operazioni in MUSA

Per la gestione del Supervisore e per l'integrazione nel sistema vengono utilizzate strutture e operazioni aggiuntive.

TokensConfiguration

In MUSA la struttura delle Reti collegate tra di loro, rappresentata dalla classe Nets, come già visto non è unica. Ogni artefatto ProblemExploration legato ad un agente, mantiene una copia della struttura in maniera tale da poterci lavorare indipendentemente. Questo comporta che le strutture di Reti per ogni artefatto siano, di base, formate da Reti vuote, dove non sono presenti Tokens. In questo modo è possibile applicare alle Reti una sovrastruttura, rappresentata dalla classe TokensConfiguration, che gestisca la configurazione dei Token e che possa essere scambiata nel Sistema tra gli artefatti.
Ogni nodo che incapsula uno Stato del Mondo contiene anche una Configurazione di Token associata a quello Stato, in modo tale da avere una Configurazione da cui partire nella Supervisione dello Stato successivo. Conseguentemente, durante l'esecuzione dell'algoritmo, i cambiamenti di stato della rete e di posizione dei Tokens vengono aggiornati nella configurazione.

Score

Un'operazione importante per la costruzione del SolutionGraph consiste nel calcolo dello Score. Nel metodo $applyNets$, una volta eseguito l'algoritmo principale per la Supervisione, bisogna considerare i cambiamenti portati dall'avanzamento dei Token nelle Reti ed indicare quanto il nuovo Stato del Mondo si sia avvicinato al soddisfacimento dei Goals. Tramite lo Score è possibile classificare gli Stati ottenuti, in modo da avere pronto lo Stato migliore da puntare nell'asta. Lo Score è calcolato in base agli stati in cui si trovano le Reti, quindi se la Rete iniziale è in uno stato di Accettazione lo Score è il migliore, ovvero 0, oppure se la Rete iniziale è in uno stato di Errore lo Score è il peggiore, ovvero 255. Invece, se la Rete iniziale è in attesa delle sotto Reti, si considera l'Albero delle Formule e si vanno a contare tutte le Formule che come figli hanno foglie. Per ognuna di queste Formule viene controllato lo stato e, rapportando il numero di quelle con la Rete in Attesa e il numero delle quelle contate, viene calcolato lo Score.

4.3.3 Esempio

Non appena ottenute le Reti per la formula θ (Figura 4.2), l'artefatto ProblemExploration è pronto per la Supervisione. Partendo dallo Stato del Mondo iniziale W_0, ipotizzando che dalla sua evoluzione durante un'Espansione sia stato generato un nuovo Stato W_1, viene avviato il Supervisore.
Come primo passo viene inizializzata la prima Rete, ovvero la Rete θ, con un Token nel Posto d'inizio. Tutte le Transizioni possono scattare, allora viene controllata la

Condizione della prima: $\neg\varphi_1$. La Rete φ_1 non ha ancora uno Stato associato, quindi viene inizializzata e deve essere controllata. A sua volta la Transizione che può scattare in questa Rete è l'unica presente, quindi viene controllata la Condizione a. Ipotizzando che in W_1 sia vera $\neg a$, la Transizione non scatta e la Rete φ_1 si trova in uno Stato di Attesa. Ritornando alla Rete θ, per essere verificata la Condizione $\neg\varphi_1$ serve che la Rete φ_1 sia in uno Stato di Errore, ma questa si trova in uno Stato di Attesa, quindi la Transizione $\neg\varphi_1$ non scatta.
Successivamente si passa alla Transizione $\varphi_1 \wedge \varphi_2$, che per scattare deve verificare le Condizioni φ_1 e φ_2. Conoscendo già lo Stato della Rete φ_1, viene subito controllata la Rete φ_2 effettuando gli stessi passaggi di inizializzazione e verifica dello Stato descritti in precedenza. Ipotizzando che $\neg b$ sia vera in W_1, la Rete φ_2 è in uno Stato di Errore. Dato che φ_1 e φ_2 sono entrambi in uno Stato di non Accettazione la Transizione non scatta.
Considerando l'ultima Transizione, $\neg\varphi_2$, lo Stato della Rete φ_2 è già in memoria e corrisponde con lo Stato della Condizione (Errore), perciò avviene uno scatto. Il Token nella prima Rete si trova adesso in un Posto di Errore dal quale non può più uscire.

4.3.4 Progetto delle Classi

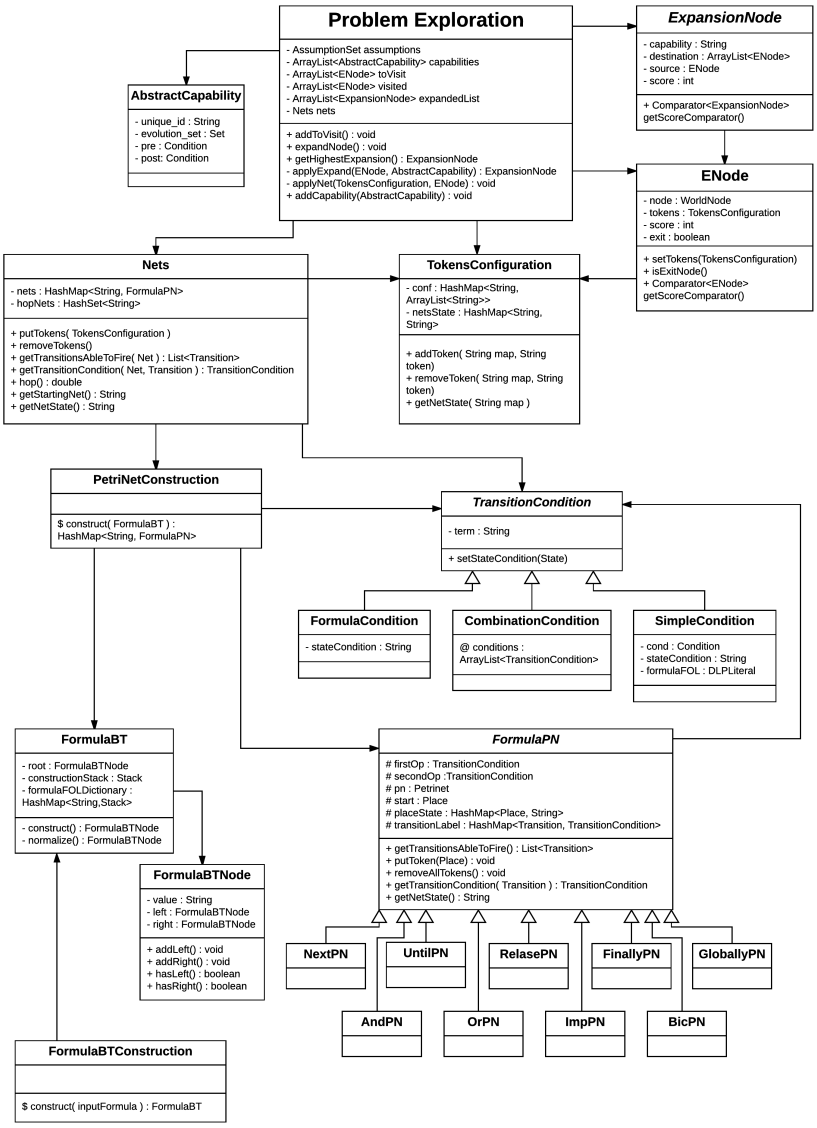


Figura 4.4: Diagramma delle Classi UML della porzione di sistema in cui opera il Supervisore

Capitolo 5

Validazione

In quest'ultimo capitolo illustro il nuovo modello di Supervisione dal punto di vista di un'applicazione ad un caso reale e metto in luce i benefici apportati rispetto allo stato dell'arte precedente.
L'applicazione ad un caso reale è affrontata considerando l'implementazione di MUSA 2.0 e delle sue funzionalità. In particolare, però, viene considerato solamente il processo di Supervisione che in MUSA avviene in fase di Configurazione. Inoltre viene comparata la potenza del nuovo Supervisore, in ambito di linguaggio e di struttura, con la potenza del precedente.

5.1 Caso di Studio - Riconfigurare il Sistema di Alimentazione a bordo di una Nave

Il caso di studio reale consiste nel problema di riconfigurare un Sistema di Alimentazione a bordo di una Nave, in particolare in seguito all'avvenimento di un guasto. Esistono diversi scenari dove il sistema elettrico può essere danneggiato in seguito ad uno o più guasti. Dato che un sistema di alimentazione moderno deve garantire efficienza e sicurezza in diverse condizioni, un sistema di controllo basato sul software può portare a termine questo compito monitorando il layer elettrico.
Nella Riconfigurazione i guasti devono essere risolti nel modo migliore, seguendo una serie di Goals, scenari e decisioni basati su requisiti funzionali e non funzionali. In particolare, in questo caso di studio bisogna considerare il problema della Riconfigurazione come un esempio Self-Adaptive. In questo modo è possibile risolvere il problema in MUSA grazie all'aiuto del Supervisore.

5.1.1 Sistema di Alimentazione

Il Sistema di Alimentazione comprende una serie di sottosistemi per la gestione di alimentazione, navigazione e altre operazioni della Nave. I componenti elettrici principali di interesse per l'esempio sono: il Generatore Principale, il Generatore Ausiliario, due Motori e due Carichi. Ogni generatore, quando acceso, porta la

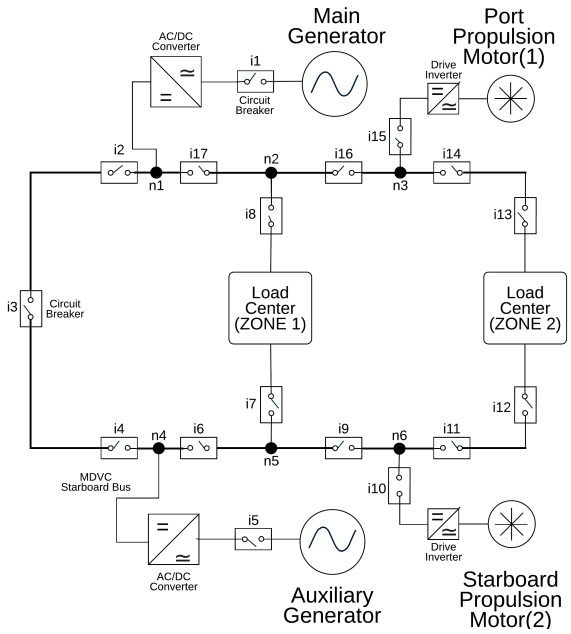


Figura 5.1: Circuito elettrico del Sistema di Alimentazione preso in esempio

corrente ai Motori ed ai Carichi e attraverso varie configurazioni di Interruttori chiusi e aperti è possibile accenderli o spegnerli.
Il sistema utilizza diverse Capabilities che permettono l'apertura e la chiusura di questi interruttori e l'accensione e lo spegnimento dei Generatori. Attraverso la combinazione di queste Capabilities è possibile generare configurazioni in maniera tale da permettere la gestione degli altri Componenti.

5.1.2 Goals

Nel Sistema di Controllo una serie di Goals rappresentano il comportamento che in generale deve essere mantenenuto. Questi sono inseriti nel sistema tramite il linguaggio LTL e nell'esempio specifico sono:

- $G(on(l1))$
  Il Carico 1 (l1) deve essere sempre alimentato.

- $G(on(l2)\ U\ off(motor_2))$
  Il Carico 2 (l2) deve essere alimentato finché si spegne il Motore 2 (nel momento in cui viene spento il Motore 2 allora non deve essere alimentato il Carico 2).

- $G(off(main) \Leftrightarrow on(aus))$
  Lo spegnimento del Generatore Principale (main) implica l'accensione del Generatore Ausiliario (aux) e viceversa (deve essere sempre attivo solo e soltanto un Generatore).

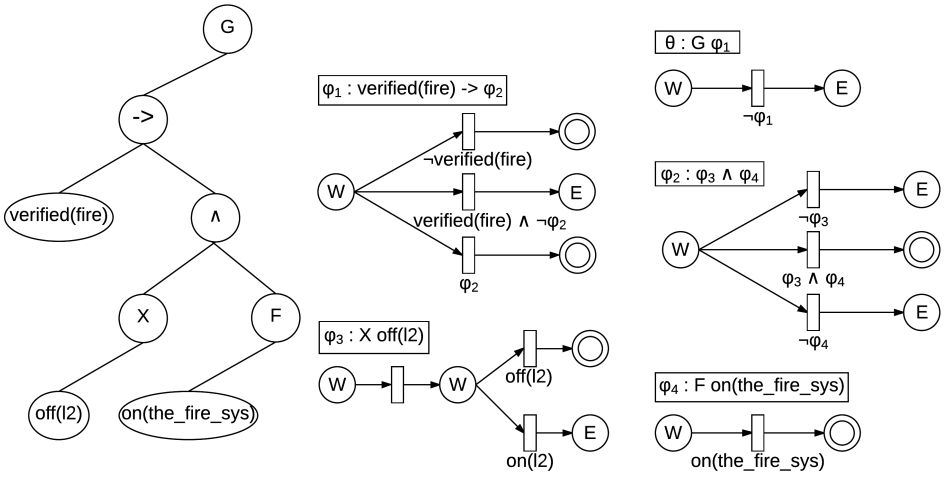
La formula LTL completa da inserire nel sistema è composta da questi tre Goals (più un Goal per la gestione di un Guasto) tramite la congiunzione :
$G(\ on(l1)\ )\ \wedge\ G(\ on(l2)\ U\ off(motor\_2)\ )\ \wedge\ G(\ off(main)\ \Leftrightarrow\ on(aus)\ )$
$\wedge\ G(\ verified(a\_fire)\ \Rightarrow\ (X\ off(l2)\ \wedge\ F\ on(the\_fire\_sys))\ )$

5.1.3 Esempio di Riconfigurazione

Uno scenario in cui avviene una Riconfigurazione del sistema di alimentazione può essere quello di un incendio a bordo. In questo caso, non appena il sistema di controllo riconosce che si è verificato un incendio ($verified(a\_fire)$ diventa vero), subito deve essere staccato il Carico 1 ed acceso il sistema antincendio. La formula che descrive questo comportamento (considerando che deve essere globalmente vero) è:

$$G(\ verified(a\_fire)\ \Rightarrow\ (X\ off(l2)\ \wedge\ F\ on(the\_fire\_sys))\ )$$

Da questa formula viene costruito il seguente Supervisore:



In questo esempio si può considerare lo Stato del Mondo iniziale W_0 dal quale parte la Supervisione, come uno Stato in cui sono verificati i primi tre Goals, in cui $l2$ è on e the_fire_sys è off e in cui $verified(fire)$ è appena diventato vero.
Le Capabilities che può utilizzare il Sistema di Controllo sono legate al circuito elettrico e comprendono l'attivazione e lo spegnimento dei Generatori e la chiusura e l'apertura degli Interruttori.
L'operazione di Supervisione porta alla creazione di un SolutionGraph in cui dallo Stato W_0, attraverso la combinazione di Capabilities che in questo caso modificano la configurazione di interruttori chiusi e aperti nel circuito, esiste un percorso che porta ad uno Stato del Mondo W_{f} in cui il Goal è soddisfatto.

5.2 Benefici del nuovo Modello

L'implementazione in MUSA di un modello che supporta il Linguaggio LTL e la costruzione tramite Reti di Petri si pone come un'evoluzione rispetto allo stato dell'arte precedente. LTL permette di ampliare il comportamento del sistema durante la fase di Supervisione e, inoltre, l'utilizzo delle Reti di Petri permette di non stravolgere la struttura precedente.

5.2.1 Maggiore Espressività

L'approccio utilizzato nel precedente modello (GoalSPEC) limita l'espressività dei Goals perché è basato sulla Logica Proposizionale. Essendo LTL un'estensione di questa logica, è facile dimostrare come venga estesa la potenza del linguaggio per definire i Goals. Alla base di ciò vi sono le formule temporali:
- Mentre la logica proposizionale è limitata nel verificare una proposizione solamente una volta, LTL esegue un ciclo infinito di verifica tramite il Globally
- L'Until e il Relase permettono dei vincoli di priorità tra due formule là dove un'Implicazione non basta
- Il Finally rende la verifica di una proposizione più flessibile perché non impone che sia verificata da subito
In generale l'utilizzo di una Logica Temporale permette di verificare dinamicamente nel tempo la veridicità di una proposizione. Questo permette una maggiore potenza di espressione dei Requisiti tramite i Goals, portando a delle configurazioni non ottenibili con il GoalSPEC.

5.2.2 Confronto con il Model Checking

Anche se l'implementazione di un Supervisore in MUSA basato sull'algoritmo classico di Model Checking era possibile, la soluzione del modello con le Reti di Petri risulta quella più conveniente. Un motivo risiede nel fatto che la Supervisione dei Goals avveniva in modalità simili a quella trattata. Per come vengono trasformate le Formule LTL in un Albero di Formule, anche nella versione precedente si otteneva un Goal Model con caratteristiche simili dalla specifica dei Goals. In entrambi i casi viene utilizzato un modello per la costruzione di una struttura basata su Reti di Petri, ma in precedenza veniva costruita solo una Rete. La logica del ProblemExploration, quindi, non cambia perché in entrambi i casi è basata sull'avanzamento del Token considerando lo Stato del Mondo.
Inoltre, in maniera ancora più importante sulla scelta, pesa la differenza di complessità e di tempi di costruzione che vi è tra l'approccio basato sulle Reti di Petri e quello basato sugli Automi. Infatti, considerando che la costruzione del Supervisore avviene con complessità di tempo lineare alla lunghezza della formula, si è intuito ma non dimostrato che l'approccio basato su Rete di Petri, rispetto a quello basato sugli Automi, porta un beneficio in termini di complessità della struttura e di tempi di costruzione.

Conclusioni

Dopo aver introdotto i concetti base della Logica Temporale Lineare tramite i suoi operatori e l'utilizzo e del Model Checking tramite l'Automa di Büchi, è stato presentato MUSA, Middleware for User-driven Service Adaptation. Il comportamento di MUSA si basa su un Reasoner che attraverso la composizione di Capabilities porta al soddisfacimento dei Goals. In un contesto di Cloud Computing, di Architettura Service Oriented e di esecuzione di Agenti distribuiti su più nodi, la soddisfazione dei Requisiti Funzionali è supervisionata tramite Formule LTL.
Il modello del Supervisore trattato in quest'elaborato è basato sulla costruzione di una Rete di Petri per ogni sotto-formula della formula LTL principale. Agli operatori del linguaggio, tra i quali i più importanti sono temporali, sono associati dei modelli di Rete che permettono la validazione della formula tramite lo Scatto dei Token. Infatti, andando ad etichettare le Transizioni di una Rete che modella una Formula con delle Condizioni, il Token passa con lo Scatto da una Posto ad un altro, stabilendo in questa maniera se la Rete si trova in uno Stato di Accettazione, di Errore o di Attesa. Una Condizione permette alla Transizione di scattare quando è verificata, ovvero quando la sotto-formula associata alla Condizione è validata.
L'implementazione in MUSA rispetta questa logica di funzionamento ed inoltre si inserisce nel contesto della costruzione del Grafo Computazionale delle Soluzioni. Sono presenti degli Agenti specializzati nella generazione di nodi per il Grafo che utilizzano il Supervisore per la soddisfazione dei Goals. Il caso di studio portato per mostrare un esempio di funzionamento consiste nel riconfigurare il Sistema di Alimentazione a bordo di una nave in seguito ad un guasto. In questo caso è possibile vedere come la composizione di Capabilities porta alla soddisfazione dei Goals definiti tramite Formule LTL.
In conclusione, il nuovo modello di Supervisione in MUSA porta una maggiore espressività nella definizione dei Requisiti e non è limitato dal metodo del Model Checking. Inoltre, questo lavoro ha messo le basi per la costruzione di un'unica Rete di Petri per la Supervisione che potrebbe portare ad ulteriori benefici, ma per motivi di tempo sarà presa in considerazione dal team di ricerca in futuro.

Bibliografia

[1] Sabatucci L., Lopes S., Cossentino M. (2017). *MUSA 2.0: A Distributed and Scalable Middleware for User-Driven Service Adaptation*. In International Conference on Intelligent Interactive Multimedia Systems and Services (pp. 492-501). Springer, Cham.

[2] Sabatucci L., Cossentino, M. (2015). *From means-end analysis to proactive means-end reasoning*. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (pp. 2-12). IEEE Press.

[3] Agnello, L., Cossentino, M., De Simone, G., Sabatucci, L. (2017). *A Self-Adaptation Exemplar: the Shipboard Power System Reconfiguration Problem*. In WOA 2017.

[4] Wolper P. (2001). *Constructing Automata from Temporal Logic Formulas: A Tutorial*. In Lectures on Formal Methods and PerformanceAnalysis (pp. 261-277). Springer Berlin Heidelberg.

[5] Katoen J. P. (1999). *Concepts, algorithms, and tools for model checking*. Erlangen: IMMD.

[6] Mayer M. C. *Logica Temporale e Verifica di Proprietà di Programmi*. Dispense Logica per l'Informatica, Università degli Studi Roma Tre.

[7] Mancini T. (2007) *Logica Proposizionale Temporale*. Dispense Metodi formali nell'Ingegneria del Software, Sapienza Università di Roma.