POLITECNICO DI MILANO

Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

SAME: Un framework per la gestione dell'incertezza nei sistemi software self-adaptive.

Relatore: Prof. Carlo Ghezzi
Correlatore: Ing. Giordano Tamburrelli

Tesi di Laurea di:
Brognoli Nicolò - Mat.766095
Colombo Andrea - Mat.767456

Anno Accademico 2011-2012

Indice

Elenco delle figure

Abstract

L'aumento della dimensione dei progetti software in unione con il dominio sempre più distribuito in cui operano le applicazioni, ha portato ad un importante incremento della complessità nel design e gestione dei sistemi risultanti.
L'insieme di questi fattori introduce un notevole numero di sorgenti di incertezza che possono compromettere la capacità di un sistema software di raggiungere i propri requisiti.
La strategia più semplice per la gestione delle manifestazioni di incertezza potrebbe consistere nell'ignorarle, questo però significa che ogni qualvolta l'incertezza colpisce un sistema software potrebbe causare una deviazione del suo comportamento, con la conseguenza del mancato raggiungimento di uno o più dei suoi requisiti. Ovviamente il non soddisfacimento di requisiti in molti scenari non è tollerabile, per questo ignorare l'incertezza non è un approccio applicabile. Pertanto è necessario gestirla con lo scopo di ottenere un sistema con comportamento predicibile anche in presenza di manifestazioni di incertezza.
I sistemi self-adaptive rispondono a tale necessità adattando il proprio comportamento in risposta ai cambiamenti interni, ovvero del sistema stesso, ed esterni, ossia a modifiche nell'ambiente in cui sono eseguiti.
Lo scopo della tesi consiste nel proporre un approccio innovativo che si inserisce in quest'area di ricerca. La soluzione proposta, chiamata SAME[1], fornisce un framework per supportare la progettazione di un sistema self-adaptive. Tale framework prevede la specifica di un modello iniziale di alto livello, il quale viene trasformato in un modello più formale che contiene inoltre la codifica dei requisiti non-funzionali. Il modello finale quindi viene incluso all'interno dell'applicazione e viene sfruttato per gestirne l'esecuzione.
In particolare tale modello a runtime è in grado di mitigare le manifestazioni di incertezza scegliendo il percorso di esecuzione che massimizza le probabi-

[1]Self-Adaptive Model-based framEwork

lità di successo nel rispettare i requisiti non-funzionali.
Come caso studio per mostrare il funzionamento di SAME è stata utilizzata un'applicazione ispirata ad un sistema esistente, la quale è sfruttata per descrivere l'approccio e validarne le caratteristiche.

Capitolo 1

Introduzione

L'aumento delle dimensioni medie dei progetti software in unione con la loro crescente pervasività, ha portato ad un importante incremento della complessità dei sistemi risultanti. Inoltre la progettazione di tali sistemi è spesso condizionata da vincoli di time to market sempre più stringenti, i quali spingono sempre più spesso gli ingegneri a progettare sistemi affidandosi a componenti implementati da terze parti, per esempio utilizzando web service oppure API pubblicamente disponibili sul web.
L'insieme di questi fattori introduce un notevole numero di sorgenti di incertezza che possono compromettere l'abilità di un sistema software nel raggiungere i requisiti per il quale è stato progettato. In particolare queste sorgenti di incertezza non sono prevedibili a design time e sono la causa per cui un sistema software potrebbe non rispettare i propri requisiti a runtime. Spesso infatti un applicativo si trova ad operare in un ambiente non predicibile, dovendo far fronte a differenti problemi introdotti dalle molteplici manifestazioni di incertezza.

L'incertezza è una tematica ampiamente studiata in differenti discipline ed in varie branche dell'ingegneria, ed in generale riguarda la precisione ed accuratezza degli strumenti di misura e la stocasticità di alcuni processi.
Tornando all'ambiente software, questa incertezza colpisce i requisiti del sistema sia dal punto di vista funzionale, che dal punto di vista non-funzionale. Brevemente, i requisiti funzionali esprimono *cosa* il sistema dovrebbe compiere, mentre i requisiti non-funzionali specificano *come* un sistema deve operare. Questa distinzione viene spiegata più a fondo nel proseguo della tesi. Sottolineiamo inoltre che nel presente lavoro ci si è concentrati su quest'ultima categoria, ossia sui requisiti non-funzionali.

L'approccio più semplice ed immediato nella gestione dell'incertezza potrebbe essere ignorarla, questo però significa che ogni qualvolta si ha una sua manifestazione il sistema potrebbe avere un comportamento inatteso e quindi deviare dal raggiungimento di uno o più requisiti. Ad esempio, in un sistema in cui interagiscono differenti servizi, se uno di essi ha un tempo di risposta più lungo di quanto previsto potrebbe compromettere il response time dell'intera applicazione.
Ovviamente il non soddisfacimento dei requisiti in molti scenari non è tollerabile, per questo ignorare l'incertezza non è una strategia applicabile. Pertanto è necessario, in certi domini applicativi, gestire l'indeterminatezza per far sì che il sistema si comporti in maniera predicibile anche in presenza di manifestazioni di incertezza. La risposta alla necessità di gestire l'incertezza è rappresentata da una particolare categoria di sistemi software, ovvero i sistemi self-adaptive. I sistemi self-adaptive infatti adattano il proprio comportamento in risposta ai cambiamenti interni, ovvero del sistema

stesso, ed esterni, ossia a modifiche nell'ambiente in cui sono eseguiti. Per ottenere questo, tali sistemi sono dotati di peculiari proprietà denominate self-* property. Tali proprietà vengono discusse nella sezione 2.2, la quale introduce inoltre una tassonomia per i sistemi self-adaptive.

La ricerca nell'ambito delle self-* property è molto ampia e negli anni ha generato differenti contributi per lo studio e l'implementazione di proprietà di adattamento. Le tecniche sfruttate sono varie e principalmente possono avvalersi di:

- modelli specifici
- particolari architetture
- costrutti a livello di linguaggio
- tecniche di Artificial Intelligence (AI).

I principali approcci proposti fino ad oggi sono catalogati ed ampiamente esposti nella survey descritta nel paragrafo 2.3. In tale indagine sono stati analizzati inoltre i livelli di astrazione che possono essere colpiti nel tentativo di mitigare le manifestazioni di incertezza. In aggiunta sono stati individuati i maggiori domini di applicazione dei sistemi self-adaptive. Tali sistemi infatti trovano applicazione in un vasto insieme, grazie alla loro utilità in un ampio spettro di condizioni.

Lo scopo della tesi consiste nel proporre un approccio innovativo che si inserisce in quest'area. Abbiamo infatti sviluppato un approccio denominato SAME (Self-Adaptive Model-based FramEwork). Nello specifico, l'obiettivo del nostro approccio consiste nell'ottenere un sistema in grado di modificare la propria esecuzione in presenza di manifestazioni di incertezza su requisiti non-funzionali. Tale incertezza può essere rappresentata, ad esempio, dal tempo di risposta aleatorio dei componenti che interagiscono all'interno di un sistema software.
L'approccio prevede la delega delle scelte nel flusso di esecuzione ad un modello che ha conoscenza dei requisiti non-funzionali, per una specifica applicazione. Tale modello inoltre ha coscienza del comportamento passato ed è in grado di operare una predizione sul futuro, avendo nella sua rappresentazione una visione completa dei possibili flussi di esecuzione percorribili. Infatti il modello raccoglie informazioni sulle proprietà non-funzionali dall'avvio dell'esecuzione fino ad uno specifico istante, ed incorpora descrizioni dei successivi passi del flusso di lavoro. Il modello pertanto viene interrogato per gestire il flusso di esecuzione relativamente ai requisiti non-funzionali del sistema.

Ad esempio, un sistema implementato sfruttando il nostro approccio si baserà sul modello per decidere se eseguire una certa funzionalità oppure un'altra, oppure ancora se saltare una particolare funzionalità. Il migliore flusso di esecuzione sarà identificato cercando di massimizzare la probabilità di successo nel rispettare i requisiti non funzionali imposti.
Nello specifico, il modello utilizzato sarà formalizzato come un Markov Decision Process (MDP) [1] arricchito con reward. Una reward è un valore non negativo che può rappresentare uno certo requisito di un sistema. Queste informazioni saranno utilizzate nel calcolo della probabilità di successo nel rispettare i requisiti, per ognuna delle possibili alternative considerate. La definizione formale di un MDP è rimandata all'appendice A.

Per la costruzione di un sistema che implementa i concetti precedentemente descritti è stato realizzato un framework chiamato SAME. SAME è un framework model-based concepito per supportare lo sviluppo e la gestione a runtime di sistemi self-adaptive. Tale framework permette di creare sistemi dotati di caratteristiche a cavallo fra le proprietà di self-configuring e self-optimizing.
Brevemente, il framework prende in input il modello MDP, il quale specifica i possibili workflow, ed un modello descrittivo, che permette il mapping fra stati del modello Markoviano e la definizione delle rispettive funzionalità che si vuole implementare nell'applicazione. Partendo da queste informazioni il framework costruisce in modo automatico una gerarchia di classi e la logica necessaria per ottenere le corrette azioni di adattamento nell'applicazione.
Importante sottolineare che attraverso SAME è possibile ottenere una netta separazione fra la logica specifica dell'applicazione, che lo sviluppatore intende implementare, e la logica di adattamento autogenerata per tale applicativo. Questo risulta essere significativo poiché la netta distinzione permette una lettura più semplice del codice dell'applicazione e quindi, in caso di necessità, una più rapida e facile modifica del sistema.

Come caso studio per mostrare il funzionamento di SAME è stata utilizzata l'applicazione *ShopReview* che basa il proprio funzionamento su *ShopSavvy.*[1] ShopSavvy è un applicativo esistente, distribuito in vari marketplace, ed è stata analizzato e successivamente reingegnerizzato attraverso il nostro approccio. Brevemente, l'applicazione permette di eseguire una ricerca su di un particolare prodotto in un sito e-commerce oppure in negozi fisici nelle vicinanze dell'utente, e quindi riportare tutti i possibili negozi che dispongono di tale prodotto ad un prezzo inferiore a quello specificato.
Questa applicazione presenta quindi dei punti di ridondanza nelle funziona-

[1] http://shopsavvy.mobi/

lità che vengono sfruttati da SAME per ottimizzare l'esecuzione del sistema. Esempi di ridondanza possono essere il riconoscimento di un codice a barre che può essere fatto in locale oppure tramite un web service, oppure la funzionalità che identifica la posizione dell'utente che può essere fatta mediante tecnologia GPS o NPS.
Nel capitolo 3 viene presentato in dettaglio l'approccio SAME, attraverso l'esempio ShopReview.

Come esposto in precedenza, un sistema realizzato con l'ausilio del framework SAME è in grado di mitigare le manifestazioni di incertezza a runtime, scegliendo il migliore percorso di esecuzione per massimizzare la probabilità di successo nel rispettare i requisiti non funzionali imposti. Pertanto, prendendo come esempio il requisito non funzionale che riguarda il response time, se il sistema si accorge di essere in ritardo potrà decidere di eseguire un certo flusso di lavoro più conservativo in termini di tempi di riposta, rispetto ad uno alternativo.
Questa funzionalità sarà testata nel capitolo 4, eseguendo delle prove in cui verranno iniettati dei ritardi nel response time per valutare l'effettivo funzionamento. Ci attendiamo che il sistema risponda eseguendo dei flussi di lavoro sempre meno dispendiosi al crescere del ritardo introdotto.

Siccome un sistema self-adaptive richiede della computazione aggiuntiva per ottenere proprietà di adattamento, verrà indagato l'overhead introdotto da SAME per rispondere alle manifestazioni di incertezza. Come descritto precedentemente, l'approccio interroga un modello per prelevare le informazioni sulle quali calcolare la probabilità di successo nel rispettare i requisiti per ognuna delle alternative specificate. Questo processo introduce un overhead non eliminabile che sarà misurato in differenti condizioni nel capitolo 4. Per calcolare l'overhead si confronteranno l'esecuzione di un sistema implementato mediante SAME ed un equivalente sequenza (*i.e. hard-coded*) di chiamate a metodi alle medesime funzionalità. La differenza, in termini di performance, fra le due esecuzioni ci aspettiamo che sia contenuta grazie alla ridotta invasività del nostro approccio. Infatti, uno dei principali fattori che potrebbe degradare le prestazioni del sistema è rappresentato dal numero maggiore di chiamate a metodi, rispetto alla versione hard-coded. Questo fattore dovrebbe presentare un impatto ragionevolmente trascurabile poiché la quantità di chiamate aggiunte risulta essere molto contenuta. Ci attendiamo inoltre un buon grado di scalabilità all'aumentare della dimensione del modello oppure del numero di requisiti e alternative.

Outline

Il presente lavoro è organizzato come segue. Nel capitolo 2 vengono toccati tre principali argomenti: viene definito il problema su cui verte questa tesi, viene proposta una tassonomia delle proprietà che un sistema self-adaptive deve possedere, ed in ultimo viene presentata un'ampia indagine relativa allo stato dell'arte dei sistemi self-adaptive. Tali sistemi saranno catalogati secondo il livello di astrazione in cui intervengono, la tecnica utilizzata per raggiungere l'adattamento ed il dominio specifico di applicazione. Invece, nel capitolo 3 verrà esposto l'approccio SAME attraverso un caso studio. In ultimo, nel capitolo 4 verranno mostrati i risultati ottenuti stressando un sistema implementato attraverso l'approccio proposto.

Capitolo 2

Definizione problema e stato dell'arte

Nel presente capitolo verranno discussi tre argomenti principali: (1) la definizione problema su cui verte questa tesi; (2) una tassonomia delle proprietà che un sistema self-adaptive, ovvero un sistema che è in grado di gestire le manifestazioni di incertezza, deve possedere; ed infine (3) lo stato dell'arte degli approcci già presenti in questa area di ricerca.
Questa divisione in tre parti si riflette nelle sezioni in cui è diviso questo capitolo. In particolare nella sezione 2.1 verrà delineato il problema introdotto dall'indeterminatezza dell'ambiente nel quale il software è eseguito.
Tali problematiche possono essere affrontate mediante le self- property, per le quali nella sezione 2.2 verrà introdotta un'ampia tassonomia. Questa analisi è utile come chiave di lettura per la collocazione del nostro lavoro.*
In ultimo, la sezione 2.3 presenta una panoramica sullo stato delle tecnologie esistenti per i sistemi self-adaptive.

2.1 Definizione del problema

I moderni sistemi software sono caratterizzati da un'elevata complessità dovuta principalmente alla dimensione dei progetti, alla crescente pervasività ed al dominio sempre più distribuito in cui operano le applicazioni. Inoltre la progettazione di tali sistemi è spesso condizionata da vincoli di time to market sempre più stringenti. In conseguenza a questi aspetti, spesso gli ingegneri progettano i propri sistemi affidandosi a componenti realizzati da terze parti, per esempio utilizzando web service oppure API pubblicamente disponibili sul web. Questa crescita in termini di complessità costringe sempre più spesso a ragionare sugli eventi che possono scaturire da un ambiente non predicibile.
Tutti questi fattori introducono diverse sorgenti di incertezza che possono compromettere l'abilità di un sistema software nel raggiungere i requisiti con i quali è stato progettato. Infatti queste sorgenti di incertezza, non prevedibili in fase di progettazione, possono condurre ad un comportamento inaspettato nel momento in cui il sistema è in operazione, e quindi portare al non soddisfacimento di uno o più requisiti.

Spendiamo qualche parola riguardo il concetto di incertezza in generale. L'incertezza è un tema ampiamente studiato che attraversa diverse discipline. Ad esempio in ambito economico il concetto di incertezza è interpretato in modo letterale: non siamo semplicemente certi riguardo l'occorrenza o il risultato di un certo evento o decisione. Pertanto, tipicamente viene descritto attraverso un processo o situazione con un componente stocastico o probabilistico.
In contrasto, la disciplina della fisica classica considera l'incertezza come un artefatto introdotto da misure imperfette [6]. Ogni misura, infatti, è intrinsecamente incerta sia per limitazioni degli apparecchi di misura sia per l'applicazione di tecniche ad-hoc che potrebbero limitare la riproducibilità dei risultati. Il concetto di incertezza può essere analizzato anche in campo psicologico, riferendosi all'abilità degli esseri umani di prendere decisioni in presenza di informazioni incomplete.

Le precedenti definizioni presentano concetti distinti ma al tempo stesso correlati fra loro. Infatti, l'incertezza riguarda processi stocastici, precisione ed accuratezza degli strumenti di misura e la difficoltà nel prendere decisioni in assenza di informazioni complete. Spesso un applicativo deve far fronte a tutti questi problemi che introducono diverse sorgenti di incertezza. Ad esempio si potrebbe trovare ad operare in ambienti instabili, utilizzando misure disturbate o non affidabili e prendendo decisioni su informazioni incomplete e inconsistenti.

Seguendo quanto detto in precedenza, la natura dei moderni sistemi software introduce diverse sorgenti di incertezza, le quali possono potenzialmente inficiare la capacità di un sistema nel raggiungere i propri requisiti. Una prima semplice strategia per affrontare questo problema potrebbe consistere nell'ignorare l'incertezza, ammettendo però in questo modo la possibilità che un applicativo non si comporti come previsto. Un comportamento inatteso potrebbe quindi condurre alla violazione di uno o più requisiti. Questa situazione in molti scenari non è tollerabile e quindi si ha la necessità di avere un certo grado di controllo sull'incertezza. Gestire l'incertezza significa quindi progettare sistemi self-adaptive, ovvero sistemi capaci di avere una riconfigurazione per ognuna delle manifestazioni di incertezza.

I self-adaptive system, o dynamically adaptive system (DAS), sono sistemi che mirano a mitigare o risolvere i problemi introdotti dall'incertezza. In questa direzione, adattano il proprio comportamento in risposta a cambiamenti interni oppure dell'ambiente in cui operano. Per tali sistemi il concetto di incertezza può essere formalizzato riprendendo la definizione introdotta per i DAS in [7]: *l'incertezza è uno stato del sistema nel quale si ha una conoscenza incompleta o inconsistente, tale che non è possibile per un DAS conoscere quale delle diverse alternative ambientali o configurazioni del sistema è presente in uno specifico momento. Questa incertezza può manifestarsi a causa di requisiti mancanti o ambigui, false ipotesi, entità o fenomeni nell'ambiente operativo non predicibili, e condizioni non risolvibili causate da informazioni incomplete o inconsistenti ottenute da sensori potenzialmente inaccurati.*
Questa definizione distingue in modo implicito fra le sorgenti di incertezza che possono presentarsi nelle fasi di elicitazione dei requisiti, design o esecuzione. In ogni fase l'incertezza potrebbe essere introdotta dal DAS stesso oppure dal contesto di esecuzione. Inoltre è importante sottolineare che le sorgenti di incertezza spesso possono interessare le varie fasi in modo trasversale.
L'incertezza impatta due categorie fondamentali di requisiti:

- *Requisiti funzionali*: definiscono una funzione di un sistema software o di un suo componente. Una funzione è descritta come un insieme di input, il comportamento che assume e gli output produce. Pertanto, i requisiti funzionali possono essere calcoli, dettagli tecnici, elaborazioni di dati ed altre funzionalità specifiche che definiscono *cosa* il sistema dovrebbe compiere.

- *Requisiti non-funzionali*: specificano i criteri che possono essere utiliz-

zati per giudicare il funzionamento di un sistema, anziché uno specifico comportamento. I requisiti non-funzionali sono spesso indicati come qualità del sistema e quindi identificano *come* un sistema deve operare. La qualità del software è il grado con il quale il software possiede una combinazione desiderata di attributi, come ad esempio reliability, usability ed interoperability [9].

Riassumendo, i requisiti funzionali definiscono ciò che un sistema deve fare, mentre requisiti non-funzionali specificano come un sistema dovrebbe essere.

La tesi si concentra in particolare sui requisiti non-funzionali, focalizzandosi su quelle sorgenti di incertezza che affliggono attributi di qualità come ad esempio response time ed usability.

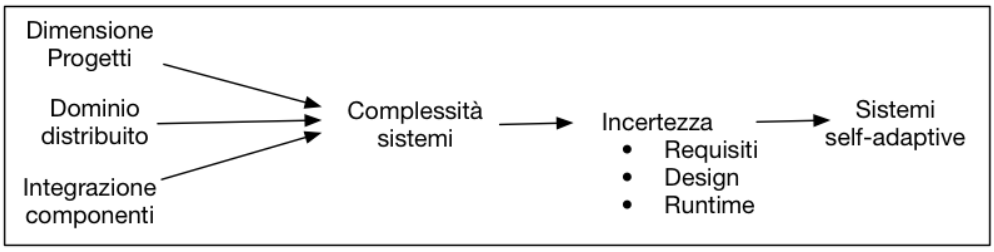


Figura 2.1: Definizione del problema.

2.2 Self-adaptation

La crescita in complessità dei sistemi software introduce molteplici sorgenti di incertezza, le quali potrebbero portare alla violazione di alcuni requisiti. La gestione di queste problematiche può essere affidata ad una particolare categoria di sistemi, ovvero i sistemi self-adaptive. Tali sistemi sono in grado di adattare il proprio comportamento in risposta ai cambiamenti, ed inoltre sono caratterizzati da una o più self-* property [10].
Queste peculiari proprietà possono essere organizzate in modo gerarchico[13]:

- General level: contiene le proprietà globali del software self-adaptive. Fra queste rientrano: self-managing, self-governing, self-maintanance, self-control e self-evaluating
- Major level: contiene le caratteristiche principali del self-management.
  - self-configuring: capacità di riconfigurarsi automaticamente e dinamicamente, in risposta ai cambiamenti.
  - self-healing: capacità di scoprire, diagnosticare e reagire alle interruzioni.

  - self-optimizing: capacità nel gestire le performance e l'allocazione delle risorse con lo scopo di soddisfare i requisiti di differenti utenti.
  - self-protecting: capacità di individuare violazioni nella sicurezza e recuperare dai loro effetti.

- Primitive level: costituito dagli attributi basilari per un self-adaptive system.
  - self-awareness: il sistema è conscio del proprio stato e comportamento.
  - context-awareness: il sistema è conscio del contesto nel quale sta operando.

2.2.1 Tassonomia Self-adaptation

Nei prossimi sottocapitoli andremo a proporre una tassonomia per il concetto di self-adaptation estendendo la gerarchia presente in [13]. Le macro aree concettuali nel quale verrà organizzata la tassonomia sono:

1. Oggetti da adattare
2. Modalità di realizzazione
3. Caratteristiche temporali
4. Problemi interazione

In figura 2.2 sono schematizzati questi aspetti.

2.2.1.1 Oggetti da adattare

Con il concetto di oggetti da adattare si intende *cosa* possiamo cambiare e *dove* si necessita un cambiamento.

- *Layer*: Individuazione del livello astratto al quale il sistema software non si sta comportando secondo i requisiti. Identificazione del livello astratto che può subire o necessitare modifiche.
- *Artefatto e Granularità*: Riconoscimento di quali artefatti e a quale livello di granularità possono/necessitano di essere modificati. Analisi di quali artefatti, attributi o risorse possono/necessitano di essere modificati per questo motivo.

- *Impatto & Costi*: L'impatto descrive lo scope dell'artefatto mentre il costo si riferisce al tempo d'esecuzione, le risorse richieste e la complessità delle azioni d'adattamento. In base a questi due fattori possiamo classificare l'adattamento come *parametrico* oppure *strutturale* [17]. Questi concetti verranno presentati ampiamente in seguito.

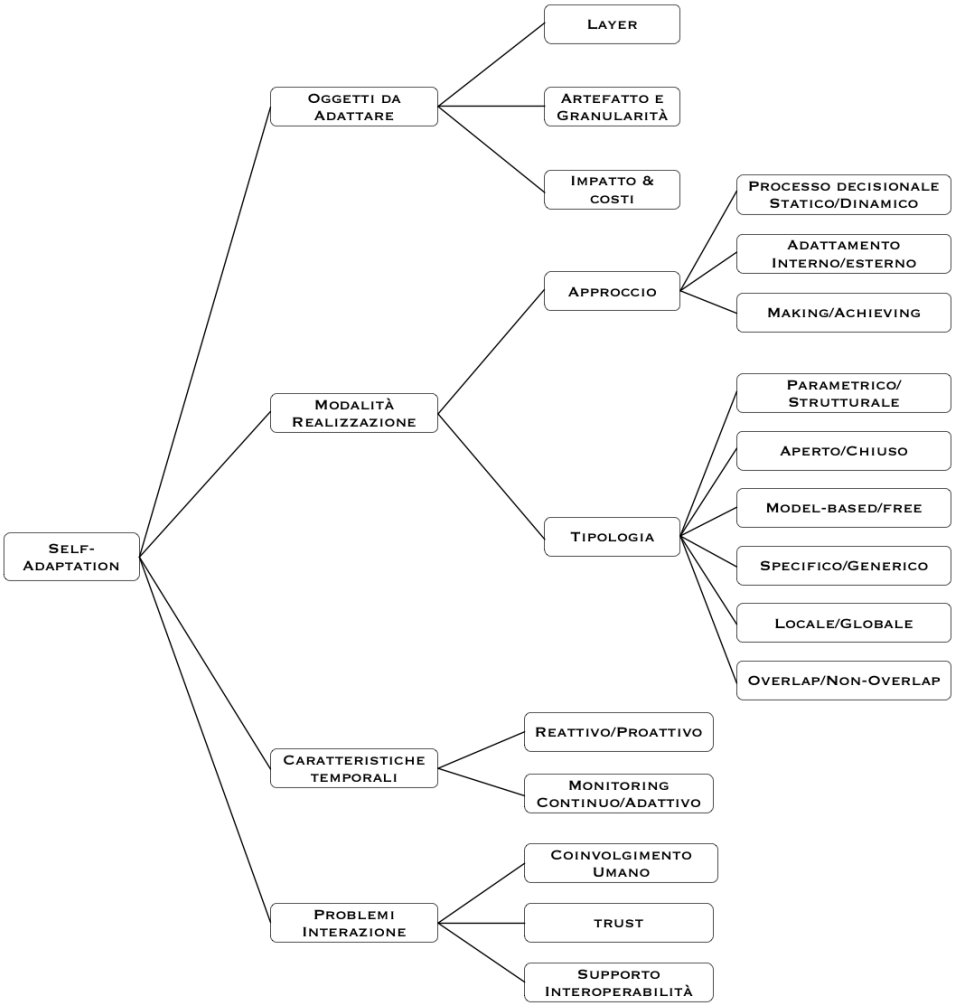


Figura 2.2: Tassonomia per self-adaptation.

2.2.1.2 Modalità di realizzazione

Le modalità di realizzazione riguardano le tematiche di *come* l'adattamento può essere applicato.
Verranno organizzati in due ampie categorie: approccio e tipologia.

Approccio
La presente classe racchiude i concetti relativi al come viene elaborata la scelta.

- *Processo decisionale statico/dinamico*: questa sottoclasse tratta come il processo decisionale può essere implementato e modificato. Nell'opzione statica tale processo è hard-coded e la sua modifica richiede la ricompilazione ed un nuovo deploy del sistema. Possibili esempi in [42, 113]. Il metodo dinamico fa leva su policy, regole e specifiche di Quality of Service (QoS) che sono definite e gestite esternamente, pertanto possono essere cambiate a run-time per creare un nuovo comportamento sia per requisiti funzionali che non e.g.[133, 123, 86].

- *Adattamento Interno/Esterno*: adattamento può essere diviso in due categorie rispetto alla separazione fra meccanismi di adattamento e logica dell'applicazione.

  – Approccio interno: questo approccio è basato sulle caratteristiche dei linguaggi di programmazione. Inoltre l'intero insieme di sensori, attuatori e processi di adattamento è intrecciato con il codice dell'applicazione. In generale, questo approccio può essere realizzato estendendo i linguaggi di programmazione esistenti oppure definendo nuovi linguaggi specifici per l'adattamento. Esempi possibili in [127, 128].
  – Approccio esterno: utilizzano un adaptation engine esterno che contiene i processi di adattamento. L'engine esterno implementa la logica adattiva che influisce sul sistema software da adattare. E.g.[50, 112].

  Questi aspetti sono presentati con una differente nomenclatura in [14]. Vengono infatti introdotti i concetti di approccio endogeno ed esogeno, che corrispondono rispettivamente ad adattamento interno ed esterno. Il focus è leggermente differente ed in particolare con endogeno si intende un adattamento cooperativo fra i componenti del sistema, mentre con esogeno si fa leva su di un control loop per la reazione ai cambiamenti.

- *Making/Achieving*[16]*:* la strategia making prevede l'ingegnerizzazione delle proprietà self-adaptive nella fase di sviluppo di un sistema. Esempi possono essere trovati in molti contributi come [69, 68, 100, 95]. Mentre la strategia achieving implica l'utilizzo di tecniche di artificial intelligence (adaptive learning) per raggiungere l'auto adattamento,

come in [107, 108, 119]. I due approcci non necessariamente devono presentarsi in mutua esclusione.

Tipologia
Le tematiche che riguardano le caratteristiche di come viene applicato il cambiamento verranno classificate secondo:

- *Adattamento parametrico/strutturale* [17]*:* per adattamento parametrico si intende la modifica delle variabili di un programma. I parametri generalmente sono tipi di dato primitivi e quindi non comportano cambiamenti particolarmente complessi nel sistema. Esempi in [77, 78]. L'adattamento strutturale implica invece cambiamenti nell'architettura software attraverso l'aggiunta, la rimozione o la sostituzione di componenti o connettori fra componenti. E.g.[136, 142].

- *Adattamento aperto/chiuso:* un sistema con caratteristiche di adattamento chiuse ha solo un numero prefissato di azioni per l'adattamento e nessun nuovo comportamento o alternativa può essere introdotta a run-time, come ad esempio in [113]. Invece, nell'adattamento aperto, il software self-adaptive può essere esteso e quindi nuove alternative possono essere aggiunte. È possibile inoltre arricchire i meccanismi di adattamento con nuove entità. E.g.[41, 119].

- *Adattamento Model-Based/Free:* nell'adattamento model-free il meccanismo non ha un modello predefinito per l'ambiente ed il sistema stesso. Infatti il meccanismo di adattamento avendo conoscenza dei requisiti, dei goal e delle possibili alternative adegua il comportamento del sistema. Ad esempio è discusso in [122, 107]. Metodologie model-based utilizzano un modello per il sistema ed il suo contesto di esecuzione, come presentato in [95, 87, 89].

- *Adattamento specifico/generico:* molte delle soluzioni esistenti sono applicabili a domini specifici. Soluzioni generiche invece possono essere configurate settando policy, alternative e processi di adattamento per differenti domini. Un'approfondita analisi dei domini di applicazione dei sistemi self-adaptive è rimandata alla sezione 2.3.

- *Adattamento locale/globale*[21]*:* con adattamento locale si intendono identificare le metodologie che non richiedono decisioni su tutto il sistema. Mentre quando le decisioni di planning riguardano il sistema nella sua interezza si parla di adattamento globale.

- *Adattamento overlap/non-overlap*[22]*:* questa sottoclasse riguarda una categorizzazione nel dominio dei sistemi distribuiti. In particolare rappresenta una classificazione delle modalità con le quali un'applicazione può compiere un aggiornamento in reazione al cambiamento. Viene utilizzato il concetto di adattamento overlap quando si ha una situazione in cui il vecchio programma ed il nuovo si sovrappongono durante le fasi di adattamento. Mentre si indica con adattamento non-overlap la modalità in cui la versione precedente del programma e la nuova non sono presenti nel sistema in modo simultaneo, durante l'adattamento. L'adattamento overlap può essere classificato a sua volta in tre principali categorie:
  - mixed-mode: permessa la comunicazione fra processo modificato e non.
  - quiescence: comunicazione delle differenti versioni non permessa
  - parallel: ogni nodo del sistema distribuito possiede sia versione modificata che quella iniziale. La comunicazione fra processi variati non è permessa ma il processo modificato (/non modificato) ad un nodo può dialogare con il processo modificato (/non modificato) di un altro nodo.

Le ultime due tipologie di adattamento sono correlate in modo trasversale fra loro e generalmente interessano sistemi molto complessi e distribuiti.

2.2.1.3 Caratteristiche temporali

La classe delle caratteristiche temporali riguarda il concetto di *quando* gli artefatti possono essere modificati o necessitano il cambiamento.

- *Adattamento reattivo/proattivo:* nella modalità reattiva il sistema risponde nel momento in cui un evento è già accaduto, mentre nella modalità proattiva il sistema predice quando un evento o cambiamento sta per avvenire, come accade in [23]. Pertanto, la strategia proattiva permette all'applicazione di individuare quando sia ha la necessità di un adattamento prima che accada una failure. In questa ottica è importante evitare azioni proattive non necessarie poiché potrebbero condurre ad un aumento dei costi, oppure a follow-up failure [24].

- *Monitoring continuo/adattivo:* questa sfaccettatura cattura la modalità con cui è effetturato il monitoring. Il monitoring è definito continuo se il processo colleziona ed elabora dati ininterrottamente, mentre

è identificato come adattivo quando viene controllato un numero ridotto di caratteristiche e, in caso di anomalia, si intensifica l'attività di raccolta dei dati. I due diversi approcci influiscono sul costo del monitoring ed il momento di rilevazione.

2.2.1.4 Problemi interazione

In ultimo viene trattata la tematica dell'interazione dei sistemi self-adaptive con gli umani e altri elementi o sistemi. Questa classe è correlata con le domande dove, quando, cosa e come avviene l'adattamento, ma soprattutto con il quesito *chi*.

- *Coinvolgimento umano:* nell'ambito del software self-adaptive il coinvolgimento di un utente umano può essere discusso in relazione a due prospettive. La prima prospettiva riguarda il grado con cui il meccanismo è automatizzato, mentre la seconda prospettiva si concentra sulla qualità di interazione fra il sistema e gli utenti o amministratori. Per la prima view si può utilizzare il *maturity model* proposto in autonomic computing [25]. I livelli in questo modello includono: basic, managed, predictive, adaptive e autonomic. Da questo punto di vista il coinvolgimento umano è meno desiderabile rispetto all'automatizzazione.
  Il concetto di qualità di interazione è inerente sia alle modalità con cui è possibile esprimere aspettative e policy, sia alle capacità di osservare cosa sta accadendo nel sistema. In questo punto di vista alternativo il coinvolgimento umano è considerato essenziale e molto prezioso per il miglioramento della gestibilità e affidabilità del sistema self-adaptive.

- *Trust:* con il concetto di trust si intende una relazione di fiducia, basata sull'esperienza passata oppure in tema di trasparenza di comportamento. Inoltre può essere definito come un livello particolare di valutazione soggettiva dove un *trustee* esibisce caratteristiche consistenti con il ruolo del *trustee* [26].
  La tematica trust può essere affrontata in ambito della sicurezza negli autonomic system, oppure in relazione a quanto un utente umano o altri sistemi possano fare affidamento al software self-adaptive per completare i propri task. Recentemente sono stati introdotti nuovi approcci per il calcolo del trust che si appoggiano ad un middleware per la valutazione dei web service [27].

- *Supporto interoperabilità*: il software self-adaptive spesso è costituito da elementi, moduli e sottosistemi. L'interoperabilità spesso rappre-

senta un problema nei sistemi distribuiti per via del mantenimento dell'integrità dei dati e del comportamento fra i vari elementi e sotto-sistemi. Nell'ambito del software self-adaptive gli elementi necessitano di essere coordinati l'un l'altro per avere le self-* property desiderate e rispettare i requisiti attesi. Pertanto i requisiti globali di adattamento sono soddisfatti se gli elementi ed i meccanismi ai vari layer di un sistema sono interoperabili.

2.3 Stato dell'arte

La proprietà di self-adaptation è ampiamente riconosciuta come una caratteristica chiave per affrontare alcune delle difficoltà nell'ingegnerizzazione e gestione di sistemi complessi. Tale proprietà conferisce al sistema software la capacità di adattare se stesso alle dinamiche interne e ai cambiamenti nell'ambiente di esecuzione, con lo scopo di soddisfare certi requisiti o goal. Possibili esempi sono sistemi in grado di riconfigurarsi automaticamente in risposta a fault, oppure sistemi capaci di ottimizzare le proprie performance a valle di cambiamenti nel contesto di esecuzione.

Differenti community debolmente legate fra loro studiano le self-* property e le loro realizzazioni. Abbiamo concentrato il nostro focus sulla community SEAMS (International Symposium on Software Engineering for Adaptive and Self-Managing Systems[1]). L'analisi condotta sui lavori pubblicati negli anni da parte della community ha portato ad una classificazione secondo tre principali view:

- livello di astrazione
- tecnica utilizzata
- dominio di applicazione

2.3.1 Livello Astrazione

La presente view approfondirà vari layer di astrazione che vengono colpiti da un approccio volto a mitigare le manifestazioni di incertezza. I livelli identificati sono:

1. Requisiti
2. Design
3. Architettura
4. Codice
5. Infrastruttura

2.3.1.1 Requisiti

Requirements engineering (RE) è un processo di ingegneria del software che copre tutte le attività coinvolte nella scoperta, documentazione e mantenimento di un insieme di requisiti per un sistema computer-based [29]. Tale

[1] http://www.self-adaptive.org/

metodologia nell'ambito dei sistemi self-adaptive appare come un ampia area di ricerca ancora aperta, nella quale solo un limitato numero di approcci è stato fino ad ora considerato. Uno dei principali problemi che introduce l'incertezza consiste nell'impossibilità, a design time, di prevedere tutte le possibili forme di adattamento. Pertanto non è possibile anticipare i requisiti per l'intero insieme di possibili condizioni ambientali, e le loro rispettive specifiche di adattamento. Ad esempio, se un sistema è chiamato a rispondere ad attacchi contro la sicurezza, non potrà conoscere tutte le tipologie di attacco in anticipo poiché nuovi attacchi sono continuamente sviluppati. Motivo per cui i requisiti per sistemi self-adaptive possono coinvolgere un certo grado di incertezza o devono essere necessariamente specificati come incompleti. L'incertezza a livello di requisiti può essere descritta seguendo la tassonomia proposta in [7]. Le principali sorgenti di incertezza identificate sono: requisiti mancanti, requisiti ambigui ed ipotesi falsificabili. In particolare, con requisito mancante ci si riferisce ad una specifica incompleta, che pertanto non copre tutti i requisiti che un sistema dovrebbe rispettare. Il concetto di requisiti ambigui viene applicato quando si ha una specifica, oppure un criterio di soddisfacimento, che può essere interpretato in differenti modi. Quindi sia i requisiti mancanti che i requisiti ambigui possono essere considerati come istanze speciali di requisiti inadeguati o insufficienti. In ultimo, con ipotesi falsificabile si intende una dichiarazione utilizzata per supportare un requisito, ma che potrebbe divenire essa stessa non valida. Queste sorgenti di incertezza possono generare requisiti insoddisfacibili oppure interazioni fra requisiti, ovvero situazioni in cui due o più requisiti interferiscono negativamente fra loro.
La specifica dei requisiti dunque deve far fronte a:

- informazioni incomplete riguardo il contesto di esecuzione
- impatto delle sorgenti di incertezza sul comportamento che il sistema deve esporre.
- evoluzione dei requisiti a runtime.

Inoltre i requisiti per un sistema software sono caratterizzati secondo la classificazione: funzionali e non funzionali. I requisiti funzionali descrivono ciò che il sistema dovrebbe fare, mentre i requisiti non funzionali indicano come e con che qualità certi compiti devono essere svolti. Nello sviluppo di un sistema self-adaptive si ha la necessità di rendere esplicite le alternative di comportamento per raggiungere uno specifico goal. Pertanto questo conduce a definire requisiti che non sono solo funzionali e non funzionali,

ma anche che includono specifiche di monitoring, criteri di valutazione e comportamenti alternativi del software [30].

Differenti approcci mirano a mitigare l'incertezza in fase di elicitazione dei requisiti. Per ogni sorgente di incertezza identificata in precedenza verranno discusse delle tecniche per attenuare o addirittura eliminare le manifestazioni di incertezza a livello di requisiti di un sistema.

Per la classe dei requisiti mancanti, ovvero per la condizione in cui la specifica risulta essere incompleta, si possono utilizzare soluzioni come: Requirements Reflection [4], Marple [31], Loki [32], KAOS Obstacle Threat Modeling [33], Partial Goal Satisfaction [34].
Requirements reflection è un paradigma che permette di rendere i requisiti disponibili a runtime, sotto la forma di runtime object. Questo approccio supporta i sistemi self-adaptive facendo divenire i requisiti delle entità utilizzabili a runtime, dotando pertanto tali sistemi delle capacità di ragionare, comprendere e modificare i requisiti a runtime. La rappresentazione dei requisiti è basata sulla goal-based RE, ed in particolare su un linguaggio che permette di rappresentare, navigare e manipolare le istanze di un meta-modello. Tale meta-modello fornisce un modo per rappresentare e mantenere le relazioni fra i requisiti ed il codice che le implementa.
Altro approccio per la gestione di requisiti mancanti fa affidamento sul tool Marple, il quale è in grado di generare proprietà che rappresentano comportamenti latenti e potenzialmente erronei. Tali proprietà nascoste sono ricercate negli UML state diagram utilizzando tecniche di evolutionary search, e modelli formali di analisi.
Loki, invece, si tratta di un approccio per la scoperta dinamica di combinazioni di condizioni ambientali, che potrebbero produrre violazioni di requisiti e comportamenti nascosti in un dynamic adaptive system (DAS). Questo permette di anticipare situazioni ambientali avverse che potrebbero sorgere a runtime.
L'esplorazione di opzioni alternative è una fase fondamentale nei processi di RE. La presenza di differenti alternative contribuisce alla generazione di differenti gradi di raggiungimento dei goal non funzionali. Spesso infatti tali goal non possono essere soddisfatti in modo assoluto. Attraverso la tecnica del partial goal satisfaction si possono definire gradi parziali di soddisfacimento ed inoltre è possibile quantificare l'impatto di ciascuna alternativa analizzata. L'approccio consiste nell'arricchimento dei modelli per la raffinazione di goal con un layer probabilistico. All'interno di tali modelli, i goal non funzionali sono specificati in maniera precisa e probabilistica, e tale specifica è interpretata in termini di misure application-specific. Inoltre l'impatto delle differenti alternative è valutato attraverso equazioni di

raffinazione che vengono applicate a variabili casuali coinvolte nei goal del sistema.

La categoria dei requisiti ambigui presenta caratteristiche in comune alla classe precedentemente analizzata. Tecniche di mitigazione come la requirements reflection può essere applicata anche in questo ambito. Un differente approccio molto utile in questo campo è rappresentato dal concetto di claims, illustrato nel NFR Framework [35]. Claims sono entità agganciate ai softgoal contribution links e sono utilizzati per registrare la logica di una scelta di una strategia di realizzazione di goal, nelle situazioni in cui è presente dell'incertezza nel determinare la scelta ottimale. I claim possono essere sfruttati come marker per l'incertezza, aiutando quindi l'analista a valutare le conseguenze di ipotesi che si dimostrano false [36].

In presenza di ipotesi falsificabili, come discusso in precedenza, una possibile tecnica di mitigazione delle manifestazioni di incertezza potrebbe essere un approccio correlato al concetto di claim. Infatti mantenendo delle rappresentazioni a runtime dei goal model e monitorando i claim attraverso delle variabili di contesto, l'effetto di un claim che si dimostra falso può essere propagato al goal model ed inoltre le strategie per la realizzazione dei goal possono essere rivalutate dinamicamente.

Requisiti mancanti o ambigui e ipotesi falsificabili portano a problematiche come requisiti insoddisfacibili ed interazioni negative fra i requisiti. Approcci come FLAGS [37], RELAX [38] e Awareness Requirement [39] possono aiutare nel contrastare l'incertezza dovuta all'incapacità di un DAS nel soddisfare un certo insieme di requisiti.

FLAGS è un innovativo goal model che generalizza le caratteristiche basilari del KAOS model [40] e aggiunge il concetto di adaptive goal. Tali goal definiscono le contromisure che devono essere eseguite se uno o più goal non sono rispettati in modo soddisfacente. Ogni contromisura produce cambiamenti nel goal model. Le contromisure possono prevenire la violazione di un requisito, dare enfasi ad un particolare goal, oppure spostarsi verso un suo sostituto. La selezione a runtime dipende dal livello di soddisfacimento del relativo goal e dalle condizioni attuali del sistema e dell'ambiente in cui sta operando.

RELAX è un linguaggio per la specifica dei requisiti, apposito per sistemi self-adaptive, nel quale espliciti costrutti sono inclusi per gestire l'incertezza. Il vocabolario è progettato per permettere all'analista di identificare i requisiti che possono essere rilassati a runtime, in occasione dei cambiamenti dell'ambiente di esecuzione. Inoltre RELAX supporta uno stile dichiarativo che consente di specificare sia le sorgenti di incertezza, che tutti i requisiti alternativi.

Gli awareness requirements (AwReqs) invece sono una nuova classe di requisiti che analizza lo stato a runtime degli altri requisiti del sistema. Pertanto sono requisiti che trattano con il successo, il fallimento oppure ogni altro possibile cambiamento di stato in altri requisiti, come ad esempio goal, vincoli di qualità ed ipotesi sul dominio.
Correlati a questa particolare categoria di requisiti si ha il concetto di evolution requirements (EvReqs), introdotto in [41]. Gli evolution requirements specificano quali cambiamenti devono essere applicati ad altri requisiti, quando si hanno particolari condizioni nell'ambiente. Tali requisiti saranno espressi come una sequenza di operazioni sugli elementi del goal model, in modo tale che possano essere sfruttati a runtime da un adaptation framework. Tale framework agirà come un controller, inviando istruzioni di adattamento verso il sistema. Pertanto EvReqs e AwReqs si completano a vicenda, permettendo all'analista di specificare i requisiti per un feedback loop che esegue l'adattamento a runtime: AwReqs indicheranno le situazioni che richiedono l'adattamento, mentre EvReqs prescriveranno quali azioni compiere in tali condizioni.

Spesso si può incappare in situazioni critiche in cui due o più requisiti interferiscono inavvertitamente fra loro, creando comportamenti negativi. Per evitare o attenuare queste problematiche possono essere utilizzati approcci come Marple[31] o Loki[32], i quali sono stati discussi in precedenza.

Gli approcci introdotti fino ad ora pongono il loro focus sull'elicitazione e specifica di requisiti. Altre tecniche si concentrano invece sulla traduzione dei requisiti in architetture software.
La metodologia introdotta in [42] prevede un modello di riferimento composto da tre layer: goal management layer, change management layer e component layer. Attraverso tale approccio, in un sistema component-based, il goal management layer fornisce al change management layer nuove configurazioni architetturali, il quale genera nuove configurazioni per i componenti. La tematica della generazione di un'architettura partendo dalle specifiche dei requisiti trova un implementazione in gocc [43]. Si tratta infatti di un compilatore architetturale per sistemi self-adaptive, il quale genera configurazioni architetturali partendo dalle descrizioni dei requisiti goal-oriented. In questa direzione il lavoro in [44] presenta uno specifico approccio in cui i requisiti sono formalizzati secondo delle specifiche scenario-based. Basandosi sui cambiamenti delle specifiche vengono quindi automaticamente sintetizzate delle estensioni della implementazione corrente. Tali estensioni avranno il compito di aggiornare il comportamento in risposta alle manifestazioni di incertezza.

In letteratura inoltre viene discusso il ruolo dell'utente umano nel monitorare un sistema self-adaptive [45]. L'input umano, nella forma di linguaggio naturale, può rappresentare una soluzione per fare il trigger dell'adattamento in quei casi in cui non può essere conseguito in modo automatico. Infatti in presenza di estrema incertezza i requisiti ed i parametri di un DAS potrebbero non catturare un aspetto chiave dell'ambiente, che invece con l'esperienza e la competenza di un essere umano potrebbe essere individuato.

2.3.1.2 Design

Il design del software è un processo di problem solving e pianificazione per una soluzione software. Tale pratica avviene a valle della fase di elicitazione e specifica dei requisiti.
Anche assumendo che i requisiti del sistema siano stati identificati in modo completo e non ambiguo, molte sorgenti di incertezza possono emergere a livello di design. Due sorgenti primarie di incertezza a design time sono le cosiddette alternative inesplorate e le decisioni di design non pertinenti ai requisiti individuati [7]. Problematiche dovute ad alternative inesplorate sorgono quando è impossibile per gli sviluppatori considerare tutte le possibili alternative di design per soddisfare un certo insieme di requisiti. Si hanno invece delle scelte di design non pertinenti ai requisiti, quando tali decisioni sono prese senza considerare o documentare uno specifico requisito. Si ottiene pertanto un design inappropriato dalla prospettiva dei requisiti.
L'incertezza introdotta nelle situazioni in cui non tutte le opzioni di design sono considerate può essere mitigata mediante soluzioni come: Avida-MDE [46], FORMS [47], Partial Models [48], Marple [31] e Adapt Cases [68], Mechatronic UML[71].
Tecniche digital evolution-based sono sfruttate per generare un insieme di modelli che rappresentano possibili implementazioni di un sistema. La digital evolution [49] è una branca dell'evolutionary computation, nella quale una popolazione di programmi self-replicating esistono in un ambiente computazionale definito dall'utente. Tali programmi detti digital organism sono soggetti a mutazioni e selezione naturale. Per generare modelli di implementazioni è stata costruita la piattaforma Avida-MDE (Avida for Model Driven Engineering), la quale permette a digital organism di generare modelli UML che rappresentano il comportamento del sistema da implementare.
FORMS (FOrmal Reference Model for Self-adaptation) consente agli ingegneri di descrivere, studiare e valutare in modo efficace alternative architetturali di design per sistemi self-adaptive. FORMS è composto da un

ridotto numero di primitive di modellazione, specificate formalmente, che corrispondono a punti chiave di variazione all'interno del sistema software. Inoltre l'approccio include un insieme di relazioni, sfruttate per guidare la composizione delle primitive di modellazione.
Altre metodologie invece adottano i partial model per ridurre il grado di incertezza in fase di design. Partial model sono astrazioni di un insieme di alternative di modellazione, e possono aiutare nella comprensione e trasformazione da modelli concreti a parziali.
Il tool Marple, introdotto in precedenza nella sezione 2.3.1.1, è in grado di generare proprietà che rappresentano comportamenti latenti e potenzialmente erronei.
L'approccio Adapt Cases estende gli UML Use Cases per tenere conto delle necessità degli adaptive systems. Infatti Adapt Cases permette la modellazione esplicita dell'adattamento, specificando formalmente tale abilità in diagrammi UML. In modo simile, Mechatronic UML permette di arricchire con la proprietà di self-optimizing i sistemi meccatronici. Consente infatti di modellare collaborazioni fra i componenti che includono adattamento strutturale, nella forma di nuovi port o multi-port.

Di fronte all'incertezza a design time, gli sviluppatori spesso accettano un certo grado di rischio, in base alla gravità delle ripercussioni nel caso in cui un requisito è violato. Tale rischio è particolarmente problematico poiché potrebbe condurre ad un analisi dei tradeoff male informata, che produrrebbe molto probabilmente una logica di scelta incorretta. Un design inadeguato spesso porta ad un implementazione che contiene degli errori causati da una progettazione non verificata, oppure che è costituita da architetture non adeguate o tecnologie abilitanti non idonee. La prevenzione di un design inadeguato può essere realizzata attraverso goal-model come FLAGS[37], linguaggi come RELAX[38], e approcci come Loki[32] e Marple[31], già ampiamente discusse nella sezione 2.3.1.1.

Mitigazioni dell'incertezza in ambito di implementazioni inadeguate possono essere rappresentate da metodologie come: Feedback Loop[50], C2[51], Rainbow[52], Chain of Adapters[69].
I feedback loop forniscono meccanismi generici per raggiungere la self-adaptation. Tipicamente coinvolgono quattro attività chiave: collect, analyze, decide e act. Sensori o sonde collezionano dati dal sistema in esecuzione e dal suo contesto. Quindi i dati accumulati sono ripuliti, filtrati ed infine memorizzati, per avere un riferimento futuro nel rappresentare un modello accurato delle esecuzioni passate e degli stati correnti. La fase di analisi esamina i dati per inferire delle tendenze ed identificare dei sintomi. Successivamente il processo di decisione tenta di predire e pianificare le scelte future sulle

azioni da applicare nel sistema in esecuzione ed il suo contesto, attraverso attuatori o effettori.
C2, invece, è uno stile component- e message-based progettato in particolare per supportare le necessità di applicazioni che hanno aspetti riguardanti una graphical user interface, ma anche per altri tipi di programmi. Lo stile permette di comporre sistemi nei quali: i componenti possono essere scritti in differenti linguaggi di programmazione e possono essere eseguiti in un ambiente distribuito ed eterogeneo; le architetture possono essere modificate in modo dinamico; differenti utenti interagiscono con il sistema e differenti tipi di media possono essere coinvolti. Pertanto lo stile C2 può essere informalmente riassunto come una rete di componenti concorrenti connessi fra loro mediante dei connettori.
Il framework Rainbow utilizza architetture software ed infrastrutture riutilizzabili per supportare le proprietà di self-adaptation nei sistemi software. Rainbow adotta una standard view per le architetture software che è tipicamente utilizzato a design time per descrivere le caratteristiche di un sistema in progettazione. Il framework include inoltre un modello architetturale del sistema nella propria implementazione. In particolare, gli sviluppatori di capacità self-adaptive utilizzano tale modello architetturale per monitorare il sistema e ragionare su di esso. Per compiere questo, Rainbow estende il concetto di stile architetturale sia in termini di operazioni primitive, che devono essere eseguite per le modifiche dinamiche del sistema, sia riguardo le modalità secondo le quali il sistema deve combinare tali azioni per ottenere un certo effetto. In aggiunta, l'utilizzo di meccanismi di adattamento esterni permettono l'esplicita specifica di strategie di adattamento per differenti concetti del sistema.
Diversamente, Chain of Adapters è una tecnica di design che semplifica la gestione dell'evoluzione delle versioni, in particolare nei web services. L'approccio prevede l'utilizzo di adattatori per rispondere alle modifiche della struttura o delle operazioni di un sistema. Tale tecnica può essere applicata per ottenere la compatibilità retroattiva fra le varie versioni dei servizi.
Un differente approccio sempre applicabile al dominio dei servizi è discusso in [70]. Questo approccio permette l'invocazione di servizi la cui interfaccia o comportamento diverge l'uno dall'altro.
Proprietà di self-healing possono essere introdotte e gestite a livello di design, nella composizione di servizi, come discusso da Chan et al. [72]. Sempre in ambito self-healing un diverso approccio per il design e la gestione di business process è descritto in [73].

L'implementazione di un sistema software self-adaptive può essere guidata da design pattern. Un design pattern rappresenta una soluzione, generale

e riutilizzabile, ad un problema molto ricorrente. Un software design pattern non fornisce del codice e non può essere direttamente trasformato in codice. Un pattern infatti identifica e modella in modo astratto gli aspetti chiave di una struttura di progettazione comune, rendendola utile per creare un design object-oriented riutilizzabile [53].
Un nuovo design pattern per l'adattamento, specifico per il dominio delle applicazioni service-oriented, è descritto in [74]. Tale pattern permette la gestione di transazioni distribuite con proprietà di adattamento dinamico.
Seguendo lo studio discusso in [54], verrà proposto un catalogo di design pattern specifici per l'implementazione di sistemi self-adaptive. Nella tabella 2.1 sono riassunti dodici design pattern per la progettazione di DAS. Tali pattern sono organizzati secondo la classificazione: monitoring (M), decision-making (DM) e riconfigurazione (R), In aggiunta, per le categorie monitoring e decision-making si può avere un ulteriore classificazione in: creazionali (C) o strutturali (S), come definito da Gamma *et al.*[53]. Per ciascun pattern descritto sono proposti degli approcci o implementazioni.

In conclusione, in letteratura sono presenti differenti framework che supportano lo sviluppatore in fase di design di un sistema self-adaptive. Esempi di implementazioni si possono trovare in [75, 76].

2.3.1.3 Architettura

Le architetture software sono state identificate come un importante elemento della costruzione di quasi tutti i sistemi ampi e complessi [93]. Suddette architetture possono essere utilizzate per forzare un sistema software nel conformarsi a specifiche regole, che risultino quindi in certe proprietà desiderate.
Differenti lavori introducono a livello architetturale caratteristiche di adattamento dinamiche per il software. Le principali tecniche utilizzate sono: model-driven, model-based, architetturali, artificial intelligence, reflection e linguaggio.

Molti approcci utilizzano un modello architetturale come rappresentazione a runtime di sistemi software, per operare il monitoring ed eseguire l'adattamento. In [17] viene proposta una metodologia che fornisce differenti modelli architetturali a runtime, a diversi livelli di astrazione. I modelli sono mantenuti a runtime attraverso tecniche di model-driven engineering. In particolare l'adattamento, sia parametrico che strutturale, è ottenuto mediante gestori automatici che modificano i modelli astratti ed i modelli a basso livello.
Sempre in ambito model-driven, la costruzione di un Organization Based

Nome	Cat.	Descrizione	Reference
Sensor Factory	M, C	Deploy di sensori e sonde nell'infrastruttura distribuita	[55, 52, 56]
Reflective Monitoring	M, S	Esecuzione di autoanalisi di un componente e alterare dinamicamente il comportamento di un sensore	[57, 58]
Content-based Routing	M, S	Informazioni di monitoring del routing basato sul contenuto dei messaggi	[52, 59, 60]
Case-based Reasoning	DM, S	Approccio rule-based per selezionare un piano di riconfigurazione	[52, 61]
Divide & Conquer	DM, S	Decomposizione sistematica di un piano di riconfigurazione complesso in piani più semplici	[52, 62]
Adaptation Detector	DM, S	Interpretazione dati di monitoring e determinazione dell'istante in cui l'adattamento è richiesto	[61, 63]
Architectural-based	DM, S	Approccio architecture-based per selezione dei piani di riconfigurazione	[55, 64]
Tradeoff-based	DM, S	Selezione sistematica di piani di riconfigurazione che meglio bilanciano diversi obiettivi	[55, 64]
Component Insertion	R, S	Inserimento e inizializzazione sicura di componenti a runtime	[55, 65, 66]
Component Removal	R, S	Rimozione sicura di un componente a runtime	[55, 65, 66]
Server Reconfiguration	R, S	Riconfigurazione sicura di un componente in una architettura client-server a runtime	[55, 65, 67]
Decentralized	R, S	Inserimento e rimozione di componenti da un'architettura decentralizzata a runtime	[65, 66]

Tabella 2.1: Catalogo di adaptation pattern

Architecture (OBAA) può essere agevolata attraverso l'utilizzo di un framework [94] concepito per sistemi multi-robot. Tale framework è basato su modelli runtime che permettono di ragionare su cosa il sistema debba compiere, in termini di goal, e riguardo come il sistema deve essere organizzato, per raggiungere suddetti obiettivi.
Un diverso framework denominato KAMI [3] si avvale anch'esso di modelli utilizzabili a runtime per gestire proprietà non funzionali, come affidabilità e performance. Nel caso di requisiti non-funzionali i modelli dipendono molto da parametri chiave, che sono forniti da esperti del dominio o prelevati da sistemi simili. Tali parametri sono continuamente analizzati con una tecnica Bayesiana, la quale produce aggiornamenti che permettono di evolvere differenti modelli formali a runtime.

Altre tecniche model-based fanno leva su metodologie di modellazione per specificare e ottenere proprietà di adattamento. Ad esempio, in [95] è proposto un approccio per la costruzione di sistemi self-adaptive che sfruttano pattern architetturali per riorganizzare la struttura di un sistema software.
Anche la gestione di web service può essere affidata a modelli che amministrano la loro composizione. In particolare in [97] è introdotto un modello matematico che formalizza le ipotesi, i principi e le regole di composizione di servizi.
Tecniche che mirano a ottenere proprietà di self-protecting, relative alla sicurezza del software, sono indagate in [96]. Viene proposta infatti un'architettura model-based che permette di individuare e mitigare attacchi DoS (Denial-of-Service) a livello web application. Questo è ottenuto aggiungendo allo stack standard di una web application un componente denominato Dynamic Firewall, il quale analizza tutte le richieste in entrata.
I modelli delle performance a livello architetturale sono utili nella gestione dell'allocazione delle risorse in ambienti virtualizzati, come mostrato da Huber et al. [87]. Sempre nel dominio dei sistemi virtualizzati, il lavoro in [86] propone un insieme di modelli per la tutela di SLA (service level agreements). Questi approcci verranno analizzati in dettaglio nella sezione 2.3.1.5, relativa al livello infrastrutturale.
In ultimo, i costrutti ed i principi individuati in framework esistenti sono concettualizzati in un modello di riferimento per sistemi decentralizzati da Weyns et al. [98].

Le proprietà self-adaptive possono essere introdotte a livello architetturale attraverso specifici stili architetturali, architetture di rifermento o tecniche architetturali.
Un noto esempio che implementa questi concetti è il framework Rainbow

[52], già introdotto nella sezione 2.3.1.2 relativa design. Rainbow infatti include il modello architetturale del sistema a runtime, con lo scopo di aggiungere capacità di adattamento dinamico.
Il progetto Fifi [99] invece propone una peculiare architettura per realizzare programmi Java dotati di caratteristiche auto evolutive. Fifi infatti simula il sistema genetico, nervoso, endocrino e immunitario con lo scopo di migliorare ed evolvere le applicazioni.
Il concetto di approccio endogeno ed esogeno, introdotto nella tassonomia proposta nella sezione 2.2.1.2, trova implementazione nell'architettura ibrida proposta in [14]. Tale architettura combina i due diversi approcci sfruttando un sistema multi-agent, esteso con un control-loop decentralizzato. Sistemi multi-agent sono impiegati anche nella strategia architetturale introdotta da Weyns e Holvoet [100]. Tale strategia consiste in un insieme di pattern architetturali che incorporano la conoscenza riguardo una particolare soluzione. Il contributo rappresenta quindi una generalizzazione delle funzioni e strutture comuni, prelevate da diverse applicazioni sperimentali studiate.
Diversamente, una particolare categoria di architetture software è rappresentata dalla cosiddetta architettura software gerarchica. Mentre l'adattamento per strutture piatte richiede solo cambiamenti nelle interazioni fra i componenti, nelle architetture gerarchiche è necessaria la mobilità gerarchica. Suddetta caratteristica è definita come la capacità di spostare i connettori fra componenti gerarchici. Questi concetti sono concretizzati nella Connection Software Architecture (CSA) [101]. CSA combina la costruzione modulare e gerarchica con lo stile architetturale object-oriented, per ottenere un nuovo stile architetturale che permette di introdurre un nuovo comportamento nel sistema senza cambiare gli elementi esistenti.
Molteplici approcci trovano applicazione nel dominio dei web service, o in generale nei sistemi component-based. In questo ambito il contributo di Martin et al. [103] analizza le relazioni fra framework esistenti come l'IBM Blueprint for Autonomic Computing ed il Web Services Distributed Management (WSDM) [102], proposto da OASIS. Viene inoltre reimplementato un framework chiamato AWSE, arricchendolo con caratteristiche dello standard WSDM. Sempre nel contesto dei sistemi component-based l'approccio in [21] presenta un'architettura, supportata da linguaggi special-purpose e tecniche aspect-oriented (AOP) [18], per il design di sistemi self-adaptive distribuiti. La metodologia è costituita da una architettura decentralizzata, in cui l'adattamento è basato su un feedback control loop. Per quanto riguarda le caratteristiche di QoS, Furtado e Santos propongono di avvalersi di architetture come Contract-Broker (C-Broker) [104]. C-Broker è un controller di QoS che espone delle API ben definite, le quali possono essere

incorporate in una applicazione oppure utilizzate come un demone esterno. C-Broker presenta un'architettura che ricalca un feedback control loop.
Altri domini come la robotica o il grid computing sono stati investigati in termini di tecniche che sfruttano architetture software. Rimandiamo una più approfondita discussione alla sezione 2.3.3.
Le tecniche architetturali vengono spesso impiegate per ottenere capacità di self-healing. Un esempio è OSIRIS-SR [105], un approccio decentralizzato per l'esecuzione di servizi compositi, arricchiti con proprietà self-healing in un ambiente distribuito. OSIRIS-SR sfrutta una topologia Safety Ring auto organizzata, per la replicazione di dati di controllo del sistema. Tali informazioni sono prelevate dai nodi appartenenti al sistema, ognuno dei quali è supervisionato da un monitor dedicato. Il contributo in [106] mira anch'esso alla diagnosi e mitigazione delle failure in un sistema software. Viene fatta leva sull'architettura Software Health Management (SHM) per individuare, diagnosticare e attenuare gli effetti di failure a livello di componenti.

Tecniche di artificial intelligence sono molto utili per ottenere proprietà self-adaptive al presente livello di astrazione.
Ad esempio, in [107] è descritto un approccio che sfrutta il reinforcement learning per ottenere on-line planning nel self-management basato su architetture. Suddetto approccio permette ad un sistema di migliorare il proprio comportamento, attraverso l'apprendimento degli esiti di alcuni comportamenti in risposta al cambiamento e mediante la modifica dei propri plan. In particolare la tecnica utilizzata è il Q-learning.
Algoritmi di reinforcement learning sono applicati anche nel contributo di Richert e Kleinjohann [108]. Viene descritta un'architettura di sviluppo che permette a robot individuali di soddisfare compiti assegnati alla società di robot. Forzando il singolo robot ad essere adaptive in direzione del beneficio del gruppo, si ottiene che la comunità intera di robot presenta caratteristiche di adattamento.

Tecniche di reflection possono essere applicate a livello architetturale. Un esempio di questo è descritto nell'approccio in [109]. Sfruttando la rappresentazione delle risorse del sistema e le loro QoS correlate, è possibile implementare un sistema service-oriented self-adaptive.

Nuovi linguaggi sono stati introdotti per costruire sistemi architecture-based self-adaptive. Cheng et al. [55] propongono un linguaggio per l'adattamento con sufficiente espressività per rappresentare la competenza umana, e buona flessibilità e robustezza per catturare preferenze complesse e potenzialmente dinamiche. Tale linguaggio permette di formalizzare azioni di adattamento e criteri di decisione che possono essere dati in input al framework Rainbow [52], per la gestione del sistema.

2.3.1.4 Codice

Il livello di astrazione del codice riguarda concetti di investigazione, modifica o creazione di codice specifico per sistemi self-adaptive.
Per raggiungere capacità di self-managing è fondamentale per un sistema l'abilità di regolare i parametri che influenzano fattori come le performance e la sicurezza. Identifichiamo con il concetto di tuning parameter un campo scalare oppure una proprietà di un campo strutturato, all'interno del codice, che misura o influisce su metriche come le performance di un sistema. L'attività di regolazione, detta anche tuning, è complessa e costosa se fatta manualmente, pertanto meccanismi di automazione di questo processo sono molto importanti.
In [77] viene proposto un approccio per automatizzare l'identificazione di tuning parameter. Dapprima vengono analizzati, in ampi sistemi open source, i tuning parameter documentati con lo scopo di classificare le tipologie e caratteristiche di tali parametri. Successivamente ogni tipo di parametro viene descritto come un pattern che cattura il contesto e le relazioni con altre parti del codice. Pertanto sia tuning parameter noti che sconosciuti vengono identificati mediante analisi statica e tecniche di pattern matching.
Un diverso approccio mira a trovare meccanismi per identificare comportamenti self-tuning e tuning parameters, ed esporli per un processo di monitoring oppure per un framework di analisi. Tali parametri sono stati dichiarati implicitamente attraverso il concetto di non-autonomic element [10]. La metodologia è in grado di localizzare tuning parameter in codice legacy, utilizzando una serie di informazioni ricavate dall'esecuzione del programma.
Una metodologia di sviluppo per la sintesi automatica di adattatori è presentata in [79]. Tali adattatori vengono generati dalla WSDL interface di web service (WS) conversazionali. Il tool a disposizione permette di costruire automaticamente script che modificano una sequenza di operazioni su un WS che deve essere sostituito, con un equivalente sequenza di operazioni sul WS sostituto.
Diversamente, il lavoro di Ke [80] mira ad ottimizzare le performance di esecuzione di un rule engine in Java. Le performance dell'engine sono migliorate eliminando il tipico accesso agli oggetti in Java (introspection), utilizzando tecniche di adaptive programming per sfruttare una diretta invocazione.

2.3.1.5 Infrastruttura

Viene proposta un analisi delle modalità con cui un sistema software self-adaptive interviene a livello infrastrutturale, ovvero nella gestione delle risorse software e hardware oppure nell'amministrazione delle comunicazioni

fra differenti sistemi.
Differenti tecniche architetturali mirano a mitigare le manifestazioni di incertezza nelle infrastrutture software.
Il concetto di Transparent Resource Management (TRM) system viene introdotto in [81]. Tale sistema può essere utilizzato per eseguire applicazioni Java dotate di policy per la gestione delle risorse, aggiunte dinamicamente come un aspetto separato. Questo approccio può essere sfruttato per implementare policy specifiche per la gestione del consumo, con lo scopo di garantire una certa quality of service (QoS) e aggiungere proprietà self-adaptive in applicazioni Java. Il metodo è abilitato da RM API, le quali forniscono interfacce per la gestione delle risorse direttamente nel codice del programma.
Spostando l'attenzione al dominio del grid computing [82], uno stile architetturale che ha come intento la risoluzione di problemi computazionalmente intensi e facilmente parallelizzabili è stato proposto da Brun et al.[83]. In tale lavoro si fa affidamento ad uno studio teorico del self-assembly e ad un modello formale chiamato Tile assemble model[84]. Sistemi che sfruttano tale modello mostrano notevoli proprietà di fault-tolerance, self-regeneration, distribuzione di informazioni e scalabilità. Pertanto queste caratteristiche possono essere ereditate da un'architettura software che implementa le regole di tali sistemi.
Sempre in ambito grid computing l'approccio descritto in [11] realizza un autonomic grid management system (AGMS) basato sull'Autonomic Toolkit [11] introdotto da IBM. AGMS esegue procedure di manutenzione preventive e correttive con l'obiettivo di assicurare operazioni affidabili ed elasticità nella job submission.

Tecniche model-based o model-driven possono includere nei sistemi software proprietà di adattamento dinamico a livello infrastrutturale.
In particolare, ha acquisito sempre più importanza l'argomento della gestione delle risorse in ambienti virtualizzati, con lo scopo di evitare violazioni di service-level agreement (SLA). La virtualizzazione infatti permette la gestione dinamica delle risorse introducendo nuovi livelli di astrazione.
Ad esempio, l'approccio in [86] analizza le modalità con cui i servizi possono essere gestiti dinamicamente in un data center virtualizzato, e mediante una tecnica model-driven cerca di evitare violazioni di SLA.
La tematica riguardante l'allocazione delle risorse è indagata anche in [87], dove viene presentato un approccio basato su modelli online delle performance ad architectural-level. Tali modelli sono sfruttati per predizioni online delle performance, che considerano gli effetti dei cambiamenti nel workload dell'utente o le conseguenze delle azioni di riconfigurazione. L'obiettivo del

metodo consiste nell'evitare l'utilizzo inefficiente delle risorse e la violazione degli SLA.
Sempre in questo ambito, una tecnica policy-based strutturata in un framework matematico analizza problemi di ottimizzazione, in sistemi self-managing, come: allocazione di risorse, gestione di QoS e ottimizzazione di SLA [88].

Capacità di adattamento dinamico possono essere introdotte anche in domini complessi come sensor network, o in generale in realizzazioni del concetto di Internet of Things (IoT). In questa direzione, l'approccio in [89] pone il proprio focus nello sviluppo di firmware self-adaptive per dispositivi con risorse limitate, facendo leva su tecniche model-driven.

In ambito service-oriented invece, il framework [91] per deployment self-adaptive basato su Disnix permette la scoperta dinamica di macchine in una rete e la generazione di un mapping fra componenti e macchine, basato su proprietà non funzionali. Disnix [90] è un tool model-driven per deployment distribuito di servizi in una rete.

In ultimo, un nuovo paradigma di programmazione, per la costruzione di processi e sistemi autonomi, è presentato da Moore e Childers [92]. Tale modello di programmazione, denominato inflate/deflate, fornisce conoscenza application-specific ad un coordinatore centrale delle risorse. Il coordinatore distribuisce e revoca risorse a runtime per raggiungere un goal del sistema.

2.3.2 Tecniche

Le tecniche utilizzate per costruire sistemi software self-adaptive sono molteplici ed ortogonali rispetto al livello di astrazione che vanno ad impattare, del quale abbiamo discusso in precedenza. Spesso inoltre non vi è una netta separazione fra le varie tipologie di approcci. In questa sezione proponiamo una possibile categorizzazione delle possibili tecniche contenute nello stato dell'arte:

1. Model-driven
2. Architettura
3. Linguaggio
4. Reflection
5. Artificial Intelligence
6. Approccio/Algoritmo specifico

2.3.2.1 Model-driven

Gli approcci di tipo model-driven tipicamente utilizzano dei modelli per supportare il design e lo sviluppo di un sistema software. Inoltre alcuni di essi esportano a runtime, consistentemente con l'implementazione, un modello che può essere utilizzato per guidare l'esecuzione. La metodologia model-driven engineering (MDE) infatti fornisce tecniche per lo sviluppo di software mediante l'utilizzo di modelli. Tali modelli sono rappresentazioni astratte della conoscenza e delle attività che controllano un'applicazione. Pertanto, metodologie che sfruttano tecniche model-driven utilizzano diversi modelli per guidare i processi di design o l'esecuzione a runtime.
Ad esempio, un possibile approccio model-driven per lo sviluppo di applicazioni component-based eseguite su dispositivi mobili, è descritto nel sommario di ricerca per il progetto IST MADAM [110].
Tecniche model-driven possono essere applicate anche per lo sviluppo di firmware con capacità adaptive. Questi concetti, già presentati nella sezione 2.3.1.5, possono essere reperiti nel lavoro di Fleurey et al. [89]. L'approccio sfrutta tre differenti modelli per rappresentare i differenti componenti del sistema: base model, cattura elementi strutturali e comportamentali; aspects model, incapsula la variabilità del sistema; adaptation model, formalizza la logica di adattamento del sistema. Sempre nell'ambito di sviluppo guidato da tecniche di modellazione, il framework in [91] si propone come un tool di sviluppo per sistemi service-oriented. Il tool fa leva su modelli per descrivere i servizi, per identificare le macchine in una rete ed il mapping delle relazioni fra quest'ultimi.

Le tecniche di sviluppo model-driven quindi possono aiutare a migliorare sia l'efficienza del processo di sviluppo software, sia la qualità del prodotto sviluppato. La modellazione di un sistema è una fase essenziale della progettazione software ed i modelli sono costruiti ed analizzati prima dell'implementazione di un sistema. Un approccio di modellazione che sfrutta questa tecnica è presente in [95]. Tale contributo, già discusso nella sezione 2.3.1.3, propone una tecnica di modellazione per il design del software con proprietà di adattamento, basato su architetture software riconfigurabili dinamicamente. Sempre nell'ambito del design di una applicazione, il lavoro in [113] introduce un modello delle risorse che è la base per un framework per lo sviluppo e deploy di applicazioni su dispositivi mobili con limitate risorse. Utilizzando tale modello è possibile ragionare riguardo le risorse richieste da una certa applicazione e le risorse a disposizione.
Un diverso esempio di gestione delle risorse è affrontato nel dominio degli ambienti virtualizzati da [87], nel quale vengono utilizzati modelli delle per-

formance a livello architetturale. Un differente modello per la gestione di servizi è descritto da Martin et al.[114]. Suddetto modello, utilizzato in un approccio goal-oriented e agent-based, consiste in un insieme di costrutti che includono risorse gestite, agent, flussi di eventi e grafi che individuano le relazioni fra le precedenti entità. Pertanto lo scopo del modello è doppio: fornire un metodo per descrivere la gestione autonoma di un servizio ed i suoi componenti, e sfruttare tale specifica come input per la generazione automatica di implementazioni agent-based.
Un differente framework multi-modello, argomentato in [116], sfrutta un modello di riferimento ed una libreria estendibile per implementare sistemi di controllo self-managing. Facendo leva quindi su tecniche di Multi Model Switching and Tuning (MMST)[115], viene gestita la QoS delle prestazioni di software systems. Modelli per la gestione delle performance sono sfruttati anche in ambito di applicazioni server, come ad esempio nel lavoro di Liu e Gorton [117].

Spesso i modelli, come indicato in precedenza, oltre che ad aiutare nello sviluppo possono essere utilizzati a runtime per ragionare sul sistema e verificare certe proprietà. Ad esempio l'approccio discusso in [17] utilizza un modello architetturale come rappresentazione a runtime di un sistema sotto controllo. Tale tecnica fornisce differenti modelli architetturali a runtime, a differenti livelli di astrazione, come base per l'adattamento.
Modelli utilizzati a runtime sono impiegati anche nel dominio della robotica. In [94] si presenta un framework per lo sviluppo di sistemi multi-robot, attraverso modelli che descrivono cosa il sistema dovrebbe fare e come organizzarsi per raggiungere certi goal. Entrambi i contributi sono stati già introdotti nella sezione 2.3.1.3.
Diversamente, il framework GRAF [111] crea, gestisce ed interpreta modelli graph-based a runtime. GRAF si affida alla lettura di modelli comportamentali per ottenere l'adattamento strutturale e parametrico.
Il contributo in [3] permette inoltre di evolvere i modelli sfruttati a runtime, mediante tecniche Bayesiane. Questo approccio discusso in precedenza nella sezione 2.3.1.3 pone il proprio focus sui requisiti non funzionali del sistema.
Correlato a questo concetto Goldsby et al. [119] propongono una tecnica per la generazione automatica di modelli, mediante digital evolution. L'idea generale consiste nell'esplorare automaticamente lo spazio delle soluzione di un sistema self-adaptive ed identificare possibili sistemi, che soddisfano certe proprietà.

Nel campo della MDE il concetto di megamodel si riferisce a modelli che hanno al loro interno modelli come elementi, e che catturano le varie relazioni fra differenti modelli nella forma di operazioni su modelli. In questo

ambito, Vogel e Giese [112] presentano un linguaggio di modellazione per megamodel esportabili a runtime, il quale supporta soluzioni di modellazione domain-specific, ed un interprete per questa parte del sistema.

In modo differente, nell'ambito della proprietà self-healing una specifica formale per modelli di gestione è proposta in [118]. Il contributo include due distinti modelli: l'organization model, che descrive le principali astrazioni di programmazione alle quali un programmatore deve conformarsi; ed il management model, che deriva specifici punti di controllo e monitoring per realizzare le self-* property.
Come visto nella sezione 2.3.1.3, particolari modelli possono essere utilizzati per soddisfare SLA in condizioni critiche come esposto in [86].
Tecniche model-based sono utilizzate anche in architetture e algoritmi per il rilevamento di attacchi DoS a livello web application, come precedentemente discusso nella sezione 2.3.1.3 in [96].

L'integrazione di tecniche di verifica formale, come ad esempio il model checking, sono solitamente applicabili in modo efficace solo in sistemi di dimensione limitata a causa del problema dovuto all'esplosione del numero di stati. L'approccio proposto in [120] mira ad alleviare questo problema mediante l'integrazione, basata sulla semantica, dello sviluppo model-based con metodologie di verifica formale, ed attraverso una tecnica per operare slicing automatico di modelli, rispetto alle proprietà da verificare.

2.3.2.2 Architettura

Tecniche che sfruttano approcci architetturali sono molto importanti poiché introducono potenziali benefici come generalità di principi, livello di astrazione per descrivere il cambiamento, scalabilità, riutilizzo di lavori esistenti e supporto all'integrazione [2].
Le differenti tipologie di tecniche possono essere riassunte come proposto in [100]:

- pattern architetturali: esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software.
- architettura di riferimento: integrano blocchi di costruzione architetturali, accumulando conoscenza di un particolare dominio di un problema.

Sempre in [100] è introdotta una panoramica per le view di una strategia architetturale per sistemi multi-agent. Mentre [121] espone una descrizione integrale della strategia architetturale, includendo una definizione dei punti di variabilità ed una specifica formale.

Per quanto riguarda invece i pattern architetturali possono essere impiegati per costruire una architettura software come ad esempio in [95], nella quale per ogni pattern è presente un pattern di riconfigurazione, che descrive le modalità di adattamento dinamico. Tale approccio rientra in differenti categorie della presente indagine, infatti è discusso nelle sezioni 2.3.1.3 e 2.3.2.1. Invece, come descritto nella sezione 2.3.1.5, uno stile architetturale applicato a livello di astrazione infrastrutturale può essere trovato in[83]. Anche lo stile C2 [51] rientra in questa classificazione, esponendo dei pattern architetturali per l'implementazione di reti di componenti concorrenti uniti da connettori.

Il design di sistemi component-based distribuiti può essere supportato ad esempio con l'architettura di riferimento proposta in [21]. L'approccio infatti è caratterizzato da un'architettura decentralizzata nella quale non sono presenti single point of failure o bottleneck, e nella quale vi è una chiara separazione fra la logica business ed i meccanismi di adattamento.
In precedenza, nella sezione 2.3.1.3 veniva introdotta una architettura di riferimento ibrida [14], ovvero che combina sia una metodologia di tipo endogena che una esogena.
Sempre in ambito di architetture di riferimento, una peculiare architettura è presente in [94]. Questo contributo coniuga una Organization Based Architecture con tecniche model-driven, pertanto anch'esso è stato presentato in precedenza nella sezione 2.3.2.1.
Una architettura generale per self-adaptive Ambient Intelligence (AmI) è analizzata da Klus e Rausch [122]. Ambient Intelligence è un paradigma che mira ad arricchire quasi ogni artefatto della vita di tutti i giorni, con certe capacità intelligenti ed abilità di comunicazione. L'architettura introdotta è utilizzata per implementare componenti in tale contesto, in particolare in sistemi self-adaptive per il riconoscimento vocale.
Il contributo in [124] innova il modello generico dell'architettura per autonomic element[12]. La nuova architettura possiede alcune delle caratteristiche esposte dall'architettura IBM, ma introduce anche nuovi elementi come un control loop intelligente: (1) monitor, (2) simulator, (3) decision maker e (4) executor. Mentre le fasi di monitor e executor sono analoghe al modello proposto da IBM, la fase decision maker è simile alle fasi sia analyzer e planner. Invece simulator è una nuova unità, senza omologhi nell'architettura IBM, che fornisce modelli di simulazione per le alternative fra i modelli di comportamento.
Una peculiare architettura, denominata Connection Software Architecture (CSA), è utilizzata per realizzare il concetto di mobility gerarchica [101]. Come presentato nella sezione 2.3.1.3, il concetto di mobility permette di

introdurre nuovi componenti in un sistema, senza la necessità di modificarlo. CSA infatti usa concetti di modularità e composizione gerarchica associati al classico stile object-oriented.
Invece, l'architettura CoBRA (Component Based Runtime Adaptable) [125] abilita l'adattamento utilizzando il modello a componenti service-oriented. Nell'approccio l'adattamento a runtime è ottenuto usando dynamic AOP.

Uno studio di fattibilità nell'applicazione di un approccio policy-based ed architecture-based, nel dominio di sistemi robotici self-adaptive è discusso in [123]. I sistemi interessati sono composti da tre parti fondamentali: un modello architetturale che specifica la struttura del sistema, un insieme di policy di adattamento, ed unità di codice eseguibile a runtime, corrispondenti ognuna ad elementi dell'architettura. Questi tre componenti sono gestite a runtime rispettivamente da Architectural Model Manager (AMM), Architectural Adaptation Manager (AAM), e Architecture Runtime Manager (ARM).

2.3.2.3 Linguaggio

Tecniche che sfruttano un nuovo linguaggio o paradigma di programmazione sono impiegate per arricchire un sistema software con self-* property, a differenti livelli di astrazione.
Un primo esempio di un linguaggio specifico per l'adattamento è introdotto da Cheng et al.[55]. Questo contributo è già stato discusso nelle sezioni 2.3.1.2 e 2.3.1.3, relative ai livelli di astrazione design e architettura. Tale approccio infatti, coniugato con il framework Rainbow [52], permette il design di sistemi architecture-based, con proprietà self-adaptive. Il linguaggio ha un grado di espressività tale da permettere la rappresentazione dell'esperienza umana, ed è caratterizzato da qualità di flessibilità e robustezza per catturare preferenze complesse e potenzialmente dinamiche. Pertanto permette la rappresentazione di azioni di adattamento e criteri di decisione pertinenti, in una forma tale che, data in input al framework, consente la gestione di un sistema.
Il contributo in [21] delinea una architettura supportata da linguaggi special-purpose e tecniche di Aspect-Oriented Programming (AOP) [18], per il design di componenti in un dominio distribuito. AOP è utilizzato per specificare ed aggiungere caratteristiche di adattamento ai vari componenti nel sistema. Nel modello presentato inoltre si distingue fra adattamento locale e globale, concetto discusso nella sezione 2.2. Inoltre, il paradigma AOP, esteso con aspetti esportabili a runtime, è utilizzato da Irmert et al. [126] per abilitare l'adattamento di servizi a runtime, in un ambito SOA. Mediante infatti la

tecnica dynamic aspect-oriented programming (d-AOP) [19] è possibile fare deploy ed attivare aspetti a runtime. Il dynamic AOP può essere realizzato ad esempio attraverso la modifica della JVM oppure con la modifica del bytecode.

Un differente paradigma di programmazione, denominato Context-Oriented Programming (COP) [20], è stato proposto come un valido approccio per la gestione del contesto. COP ha un certo grado di somiglianza con AOP. Quest'ultimo infatti può essere visto come un termine generale che indica una famiglia di approcci che supportano la modularizzazione di problematiche trasversali. Una discussione della differenza fra i due paradigmi può essere trovata in [20].
In questa direzione, il contributo nel lavoro [127] introduce tecniche COP in un linguaggio che nativamente supporta la distribuzione e la concorrenza, in particolare nel linguaggio di programmazione Erlang. Erlang è un linguaggio di programmazione funzionale general-purpose con forti caratteristiche come: strict evaluation, single assignment, e dynamic typing.
Sempre in questo ambito, un interessante valutazione delle capacità di adattamento nei linguaggi di programmazione può essere trovata in [128]. Il contributo investiga il supporto che i principali linguaggi di programmazione, object-oriented e funzionali, offrono per lo sviluppo di sistemi adaptive e context-aware. L'intento consiste nel provare come le astrazioni offerte da certi linguaggi possono essere direttamente sfruttati, senza chiamare in causa framework speciali come COP e AOP. I linguaggi scelti per l'analisi sono: Erlang, Python, Ruby, F# e Scala.
RELAX [38] invece si propone come linguaggio per la specifica dei requisiti, specifico per sistemi self-adaptive, nel quale espliciti costrutti sono inclusi per gestire l'incertezza.

Astraendo il concetto di linguaggio di programmazione, altre tipologie di approcci possono supportare lo sviluppo di un sistema. In particolare due differenti linguaggi di modellazione sono proposti in [68] e [112]. Nel primo contributo viene discussa un nuova metodologia di modellazione, chiamata Adapt Cases, che permette la modellazione specifica dell'adattamento già in fase di design, come descritto in precedenza nella sezione 2.3.1.2. Il secondo lavoro indicato presenta invece un linguaggio di modellazione completo per runtime megamodel, che semplifica in modo considerevole lo sviluppo di logica di adattamento. Questa tecnica è stata introdotta nella sezione 2.3.2.1.

2.3.2.4 Reflection

Con reflection si intende l'abilità di un programma di esaminare e modificare la struttura ed il comportamento, in particolare i valori, i meta-dati, le proprietà e le funzioni, di un oggetto a runtime [129].
La reflection pertanto può essere utilizzata per osservare e modificare l'esecuzione di un'applicazione a runtime. Un componente reflection-oriented è in grado di monitorare l'esecuzione di una porzione di codice e dunque può modificare se stesso, in accordo con un goal desiderato. Quindi tecniche di reflection possono essere sfruttate per adattare un certo sistema a differenti situazioni, in modo dinamico.
La tecnologia reflection inoltre può essere applicata a differenti livelli di astrazione, come: (1) codice, (2) architettura o (3) requisiti.

Un esempio di applicazione della reflection a livello di codice, per monitorare sistemi adaptive, è discusso da Dawson et al.[58]. In tale contributo infatti viene presentato come costruire il monitoring all'interno di programmi Java mediante reflection, con lo scopo di individuare un comportamento normale o la presenza di eccezioni in un sistema.

Nel contesto di reflection applicata a livello architetturale, l'approccio in [109] mira alla gestione delle risorse in modalità adaptive. Questa tecnica, discussa anche nella sezione 2.3.1.3, seleziona la risorsa più appropriata per eseguire un servizio, con le QoS richieste.

Reflection dei requisiti significa rendere tali specifiche disponibili a runtime. Nel contributo di Bencomo et al. [4] i requisiti sono trasformati in entità a runtime. Questo permette al sistema di ragionare dinamicamente riguardo se stesso a livello di requisiti, in modo simile a come la reflection architetturale permette di ragionare a livello di architettura software.

In generale, la reflection computazionale forma le fondamenta di un sistema self-adaptive. In [130] si sottolineano le proprietà della reflection come prerequisito alla costruzione intelligente di sistemi adaptive. Inoltre sono discusse differenti problematiche che si possono incontrare nell'applicazione di tale principio nel cosiddetto programming-in-the-large.

2.3.2.5 Artificial Intelligence

Artificial Intelligence (AI) è lo studio ed il design di intelligent agent, dove un intelligent agent consiste in un sistema che percepisce il proprio ambiente ed esegue determinate azioni che massimizzano le proprie chance di successo [131].

Tecniche che pongono le proprie radici in questo campo sono sfruttate per la realizzazione di sistemi software con capacità di adattamento auto-

matiche.
Ad esempio un approccio basato su reinforcement learning (RL) per ottenere on-line planning è introdotto in [107]. Questo approccio permette ad un sistema software di migliorare il proprio comportamento mediante l'apprendimento dei risultati delle proprie azioni, e la modifica dinamica dei propri piani in base a quanto appreso in presenza di cambiamenti nell'ambiente. Viene descritto un processo di scoperta delle rappresentazioni, fitness function, ed operatori per supportare l'on-line planning. Nello specifico è utilizzato l'algoritmo Q-learning [132], il quale è una forma di reinfocement learning privo di modello.
Reinforcement learning è solitamente una tecnica single-agent basata su unsupervised learning, in cui implicitamente viene gestita una policy singola. Dusparic e Cahill[133] propongono un estensione di questo concetto, sfruttando il Collaborative Reinforcement Learning (CRL) per supportare l'ottimizzazione in presenza di molteplici policy. CRL introduce lo scambio di action reward fra agent, in modo tale che gli agent possano anche apprendere dalle reciproche azioni. L'applicazione del concetto di multiple policy nell'algoritmo RL viene indagato in diversi lavori di ricerca, e nell'approccio in [133] si analizza la combinazione delle caratteristiche di CRL e multiple policy RL, per soddisfare i requisiti di un sistema decentralizzato su larga scala con differenti goal.

Tecniche AI possono inoltre supportare lo sviluppo di sistemi self-adaptive, come proposto dal contributo in [75]. Il framework per il design, introdotto nella sezione 2.3.1.2, sfrutta una architettura di riferimento basata su Belief-Desire-Intention (BDI) agent model [134]. Infatti per la definizione del comportamento a runtime il modello BDI fornisce le basi architetturali per un sistema agent-oriented, dotandolo di un ciclo per il ragionamento.
Una differente architettura di sviluppo denominata InFFra [108] permette a robot individuali di soddisfare task assegnati alla società di robot. Tale approccio sfrutta tecniche di RL per evolvere l'architettura del sistema, permettendo a robot di adattarsi al proprio ambiente mediante l'esplorazione autonoma.

2.3.2.6 Approccio/Algoritmo specifico

Nella presente sezione verranno raccolti gli approcci che si basano su tecniche peculiari, che non rientrano nelle categorie precedentemente discusse.
Un interessante esempio è rappresentato dallo studio in [135], che pone focus sui problemi nell'applicazione di tecniche bio-inspired a sistemi reali. La principale caratteristica di tali algoritmi è il fatto che sono basati su di un

insieme di principi ispirati al mondo naturale, ed inoltre forniscono semplici soluzioni per risolvere problemi che sarebbero molto più complessi utilizzando approcci classici. Nel contributo sono discussi i principi fondamentali di approcci di organizzazione bio-inspired, alcuni algoritmi proposti in letteratura ed i problemi che potrebbero sorgere applicando questi principi ed algoritmi a sistemi reali.
Una differente tecnica che prende però sempre spunto dal mondo naturale è la digital evolution, che viene sfruttata in [119] per evolvere state diagram noti, con lo scopo di soddisfare proprietà critiche di un sistema. Digital evolution consente di esplorare automaticamente lo spazio di soluzioni di un sistema self-adaptive, attraverso una popolazione di programmi chiamata organismi. Infatti l'evoluzione di questi organismi, guidato attraverso la definizione delle caratteristiche delle soluzioni, permette di individuare soluzioni valide e spesso inattese.

Cambiando ambito, tecniche come Coordinated Atomic Actions (CA actions) ed exception handling possono essere applicate per ottenere una riconfigurazione architetturale, come mostrato in [136]. Il concetto di CA action consiste in uno schema unificato per coordinare attività complesse e concorrenti, e per supportare recovery da errori fra molteplici componenti che interagiscono in un sistema distribuito. Il contributo fa quindi leva su CA action ed exception handling come meccanismo di strutturazione per fornire supporto nell'isolamento dei fault, riconfigurazione del sistema e resume dei suoi servizi. Riassumendo, la gestione dei fault può essere eseguita in modalità collaborativa utilizzando exception handling all'interno di una CA action.
La riconfigurazione dinamica dell'architettura di un sistema è indagato anche in [142], nel quale viene proposto il design e lo sviluppo di un meccanismo per la generazione automatica di workflow per la coordinazione dall'adattamento di sistemi software. Un workflow rappresenta un business process in termini di attività e loro relazioni, regole e dati di controllo.

Differenti tecniche invece si basano sulla process algebra CSP [137], un formalismo sviluppato per la specifica e verifica di sistemi reattivi. Tali sistemi reagiscono in risposta a stimoli esterni, regolando il proprio comportamento interno. I sistemi self-adaptive possono essere considerati come sottoclasse di suddetti sistemi, in quanto reagiscono adattandosi ai cambiamenti propagati da qualche stimolo. CSP in [138] è utilizzata per la specifica, verifica ed implementazione di sistemi self-adaptive attraverso un framework, che permette di realizzare una gerarchia di specifiche che pone una separazione fra i meccanismi di adattamento ed i comportamento funzionale di un sistema.

FlashMob [139] è processo distribuito di assemblaggio che sfrutta un protocollo di gossip aggregato per permettere ad un insieme di nodi di derivare e concordare su una configurazione globale di componenti. Protocolli gossip sono utilizzati per garantire la ricezione delle informazioni da parte di tutti i peer di una rete, senza ricorrere a broadcast affidabile, il quale può implicare un enorme numero di messaggi restringendo la scalabilità. In particolare il protocollo impiegato è il gossip aggregato, il quale attraverso la propagazione di diversi aggiornamenti calcola alcune funzioni aggregate su informazioni memorizzate ad ogni nodo. Quindi l'approccio è in grado di derivare e modificare le configurazioni di componenti distribuiti, decentralizzando il processo in modo tale che nessun nodo singolo è responsabile delle configurazioni di assemblaggio e del mantenimento di tutta la conoscenza necessaria per fare ciò.

Tecniche model-checking probabilistiche possono essere applicate nella verifica di sistemi self-adaptive, come proposto in [140]. L'approccio si affida alla stimolazione e al model-checking probabilistico per fornire livelli di fiducia riguardo la fornitura di un servizio. In particolare il focus cade sulle proprietà di elasticità, che permettono di valutare quando un sistema è in grado di mantenere una certa affidabilità a dispetto dei cambiamenti nel proprio ambiente. L'idea generale della metodologia consiste nello stimolare il contesto di un sistema con l'obiettivo di esercitare le sue capacità di adattamento, e nel collezionare dati riguardo le reazioni del sistema a tali cambiamenti nell'ambiente.

Il concetto di trust, introdotto nella sezione 2.2.1.4, può essere utilizzato per supportare un migliore decision-making in presenza di informazioni incomplete. Yew e Lutfiyya [27] presentano un middleware per il calcolo del trust da molteplici fonti di evidenze, che risolve le seguenti problematiche: mancanza di standard nella scoperta e rappresentazione delle evidenze, e difficile inserimento di tutte le evidenze nel calcolo del trust.

Diversamente, in [141] vengono analizzate due tecniche per trovare soluzioni nel search space per l'adattamento, una basata su algoritmi di planning e l'altra basata sul constraint solving. Mediante automated planning viene costruita una sequenza valida di migrazioni e quindi viene controllata se la configurazione raggiunta è quella finale. Alternativamente, attraverso constraint solving è possibile prima trovare una configurazione idonea e successivamente controllare se può essere raggiunta con una sequenza di migrazioni.

In modo differente, Fritsch et al.[143] descrivono un algoritmo per time-bounded scheduling delle azioni di adattamento. L'algoritmo pianifica le azioni di adattamento, che possono essere eseguite da un service provider, in

base a determinati vincoli. Viene tenuto conto di vincoli di tempo, cercando di massimizzare il numero di caratteristiche che possono essere adattate entro un certo limite di tempo su una specifica piattaforma software. Inoltre nella scelta vengono considerati anche vincoli come limitata memoria ed importanza delle caratteristiche.

2.3.3 Dominio

I sistemi self-adaptive trovano applicazione in differenti domini, data la loro utilità in un ampio spettro di condizioni. Nella community SEAMS vari lavori indagano l'applicazione di sistemi software self-adaptive, in particolare le aree individuate riguardano:

1. Sistemi multi-agent
2. Sistemi service oriented o component-based
3. Robotica
4. Sistemi embedded
5. Sistemi distribuiti
6. Ambienti virtualizzati
7. Grid computing

2.3.3.1 Sistemi multi-agent

Un sistema multi-agent è un sistema composto da diversi intelligent agent [131] che interagiscono fra loro in un determinato ambiente. Il coordinamento e l'interazione di tali entità autonome permette la costruzione di applicazioni distribuite e cooperative. Queste applicazioni sono caratterizzate da un'elevata dinamicità, ad esempio il ruolo o la relativa importanza di un agent può variare nel corso dell'esecuzione. Diviene importante pertanto stimare la criticità di un agent, ovvero la misura del potenziale impatto della una failure di un singolo agent sull'intera organizzazione. La stima di questo concetto può essere effettuata mediante differenti strategie, come esposto in [144]. Questi aspetti inoltre sono concretizzati in una architettura (DarX) per la replica dinamica ed il controllo di sistemi multi-agent.
L'ottimizzazione di sistemi multi-agent è indagata nel contributo presente in [133]. Tale approccio, introdotto nella sezione 2.3.2.5, propone una research agenda per estendere il Collaborative Reinforcement Learning per supportare l'ottimizzazione multiple-policy.

In [100] viene esposta una strategia architetturale che struttura il software come un insieme di agent che sono collocati in un ambiente. Una differente architettura software viene discussa da Weyns et al.[14].

2.3.3.2 Sistemi service-oriented

I sistemi service-oriented sfruttano il concetto di servizio come elemento fondamentale per sviluppare applicazioni o soluzioni. I servizi sono elementi computazionali autodescrittivi e multi piattaforma, che supportano la composizione di applicazioni distribuite in modo rapido ed a basso costo. Applicazioni basate sui servizi sono progettate come insiemi indipendenti di servizi che interagiscono fra loro e che offrono interfacce ben definite ai potenziali utenti [145].
Grazie a queste caratteristiche, in questo dominio trovano applicazione molti approcci di design ed implementazione di sistemi self-adaptive. Le tecniche utilizzate variano in modo trasversale rispetto alla categorizzazione precedentemente presentata.

Un primo esempio nel campo della rappresentazione dei requisiti è rappresentato dall'estensione del tradizionale goal model per la composizione dei servizi, proposto in [146].

Nella fase di design di un sistema self-adaptive possono essere utilizzati approcci come quelli introdotti in [69, 72, 73, 74, 79]. Alcuni di essi sono stati già argomentati nelle sezioni precedenti ed in generale si occupano della composizione di servizi, della gestione di business process oppure propongono tecniche o pattern di design.

In letteratura differenti lavori propongono una architettura di riferimento, un evoluzione di una architettura esistente oppure uno stile architetturale per arricchire il software con self-* property. Alcuni di essi si possono trovare nei contributi in [103, 104, 105, 147]. Inoltre un approccio che utilizza la tecnica della reflection a livello architetturale è proposto da Raibulet et al.[109].

Mentre tecniche model-based e model-driven applicate a sistemi component-based trovano esempio in [86, 91, 97, 125].

La valutazione e il testing di sistemi self-adaptive nel campo dei servizi viene presentata in [24, 148, 149]. Tali contributi propongono tecniche per monitoraggio e test dell'integrazione dei componenti in sistemi software con capacità di adattamento.

Altri approcci applicabili al dominio dei sistemi service-oriented sono riassunti nei lavori [23, 139, 151, 152]. In ultimo la roadmap in [150] delinea i vari contributi richiesti per ottenere sistemi self-aware sostenibili.

2.3.3.3 Robotica

La robotica è un dominio molto complesso che spesso presenta una chiara necessità di capacità self-adaptive, principalmente a causa del contesto di deploy instabile e non predicibile.
In questo ambito, un approccio modulare per l'evoluzione di società di robot mediante tecniche di artificial intelligence è proposto in [108]. Tale approccio già presentato nelle sezioni 2.3.1.3 e 2.3.2.5 descrive un'architettura di sviluppo che permette a singoli robot di completare task assegnati ad una comunità di robot.
Sistemi multi-robot possono essere riorganizzati automaticamente con l'approccio in [94] basato su modelli a runtime. Questo lavoro è stato analizzato in precedenza nelle sezioni 2.3.1.3 e 2.3.2.1.
SHAGE [153] è un framework realizzato per dare capacità di self-managing a software specifico per sistemi robot. Il framework osserva la situazione dell'ambiente circostante ad un robot, e cerca le possibili riconfigurazioni architetturali che sono adatte per la gestione della situazione incontrata.
In conclusione, uno studio di fattibilità nel dominio della robotica è esposto in [123]. Viene studiata l'applicabilità di un approccio basato su policy ed architetture per lo sviluppo di sistemi robot self-adaptive.

2.3.3.4 Sistemi embedded

I sistemi embedded sono una particolare categoria di sistemi informatici, progettati per specifici funzioni di controllo o per applicazioni special purpose. In tali sistemi le risorse sono intrinsecamente scarse e quindi spesso vanno gestite a runtime. Queste problematiche possono essere risolte mediante l'utilizzo di sistemi self-adaptive.
Nel contributo di Peper e Schneider [154] vengono descritti gli aspetti selezionati di un approccio component-oriented per lo sviluppo di sistemi ad-hoc, dotati di capacità di adattamento. Viene scelto il paradigma a componenti poiché assicura caratteristiche di information hiding e assenza di side-effect.
L'adattamento a runtime viene indagato anche nel contributo in [141], introdotto in precedenza nella sezione 2.3.2.6. Viene infatti mostrata l'applicabilità, nel dominio dei sistemi embedded connessi in rete, di differenti tecniche per trovare soluzioni in un search space.
L'analisi del comportamento di adattamento di sistemi embedded è caratterizzata da una immensa difficoltà, a causa delle interdipendenze fra i componenti. In [155] viene descritta una tecnica per generare modelli astratti del comportamento di adattamento, i quali si prestano bene per

la verifica formale di alcune proprietà come la persistenza e la correttezza nell'adattamento.

2.3.3.5 Sistemi distribuiti

I sistemi distribuiti presentano intrinsecamente un elevato grado di incertezza. Tali sistemi infatti hanno forti requisiti di coordinamento, la loro progettazione è complessa e devono essere in grado di adattarsi a frequenti cambiamenti.
Il concetto di mixed-mode introdotto nella sezione 2.2.1.2 e viene concretizzato nel contributo in [22]. Nell'adattamento mixed-mode il processo modificato può comunicare con il processo originale. Tale tecnica permette quindi di ridurre i requisiti di sincronizzazione durante il processo di adattamento, riducendo il tempo di interruzione del servizio e abbassando l'overhead di comunicazione.
Un recente contributo [156] propone un modello ed un middleware self-organizing distribuito per il design e l'implementazione di sistemi distribuiti. Questo approccio distribuito si concentra sulle necessità di coordinazione, e fornisce un chiara visione dei punti in cui è possibile includere control loop e la loro coordinazione.

2.3.3.6 Ambienti virtualizzati

L'adozione di tecnologie di virtualizzazione promette differenti benefici come: maggiore flessibilità, migliore efficienza energetica e minori costi operativi per sistemi IT. Tuttavia, workload molto variabili rendono difficile la garanzia di Quality od Service (QoS) e allo stesso tempo un utilizzo efficiente delle risorse. Per evitare violazioni di Service Level Agreements (SLA) l'allocazione delle risorse deve essere adattata in modo continuo, per riflettere i cambiamenti nel workload delle applicazioni.
Huber et al. [87] presentano un approccio per l'allocazione self-adaptive in ambienti virtualizzati basata su modelli online delle performance, a livello architetturale. Modelli di controller dell'adattamento a differenti livelli di astrazione sono invece proposti in [86]. L'introduzione di modelli a differenti livelli permette di supportare la valutazione dell'impatto delle azioni di adattamento sul sistema e sugli SLA. Entrambi i contributi sono stati argomentati anche in precedenza nella sezione 2.3.1.5.

2.3.3.7 Grid computing

Con grid computing [82] si intende l'associazione di risorse informatiche provenienti da differenti domini amministrativi, con lo scopo di raggiungere un obiettivo comune. Le caratteristiche che distinguono tali sistemi sono il debole accoppiamento, l'eterogeneità e la dispersione geografica. Il grado di complessità di sistemi grid è molto alto, e la loro gestione risulta molto complicata. Per facilitare questo processo, in [85] viene proposto un autonomic grid management system (AGMS) che si poggia sull'IBM Autonomic Toolkit [11]. L'AGMS applica procedure di gestione preventive e correttive, per assicurare il completamento dei workflow nel grid.
Diversamente, Brun e Medvidovic [83] espongono uno stile architetturale software che permette la distribuzione dei problemi su di un'ampia rete con proprietà di fault tolerance, discretezza e scalabilità.
Entrambi i precedenti contributi sono stati già presentati anche in altre sezioni.

I sistemi di grid computing manifestano inoltre problematiche relative alla sicurezza. La progettazione di sicurezza adaptive non è per nulla un processo semplice. Tipicamente un insieme di contromisure di sicurezza devono essere identificate. Come argomentato in [157], queste azioni dovrebbero essere applicate sia ai primary asset, ovvero le risorse che esplicitamente l'utente vuole proteggere, sia ai secondary asset che potrebbero essere sfruttati dagli aggressori per danneggiare i primary asset. Nel lavoro indicato in precedenza viene pertanto proposto una notazione per la rappresentazione di modelli di primary e secondary asset. Tali modelli vengono successivamente sfruttati per estendere i requisiti di sicurezza e per il design delle funzionalità di monitoring richieste dal sistema.

2.4 Summary

Nel presente capitolo sono state discusse le problematiche che affliggono i moderni sistemi software. Tali sistemi infatti sono caratterizzati da una crescente complessità che introduce notevoli sorgenti di incertezza. Come approfondito, l'incertezza riguarda processi stocastici, precisione ed accuratezza di strumenti di misura e difficoltà nel prendere decisioni in assenza di informazioni complete. L'incertezza inoltre complica notevolmente il processo di design e gestione a runtime dei sistemi software. In questo ambito, l'incertezza colpisce i requisiti dei sistemi sia funzionali che non funzionali. Come abbiamo visto si possono avere due strategie per affrontare l'incertezza:

- ignorarla, ovvero non prendersi cura delle eventuali manifestazioni e quindi ammettere un comportamento inatteso che potrebbe esporre un sistema al rischio di violazione di certi requisiti; oppure
- gestirla mediante l'implementazione di sistemi self-adaptive, ovvero sistemi in grado di avere una riconfigurazione per ognuna delle manifestazioni di incertezza.

I sistemi self-adaptive sono caratterizzati da peculiari proprietà, ovvero le self-* property. Tali proprietà possono arricchire un sistema software, dotandolo di efficienti capacità di adattamento in risposta ad un contesto non predicibile. Queste proprietà possono intervenire in vari livelli di astrazione, e nella categorizzazione proposta si è scesi in profondità partendo dal livello astratto rappresentato dai requisiti, fino a giungere al grado di astrazione corrispondente all'infrastruttura, sulla quale poggia ogni sistema software.
Inoltre gli approcci sono stati catalogati anche secondo la tecnica utilizzata per raggiungere l'adattamento autonomo di un sistema software. Principalmente le tecniche utilizzate si basano su modelli, speciali architetture software o specifici linguaggi programmazione. Altre tecniche sfruttano concetti di artificial intelligence, i quali si prestano in modo naturale nell'ambito delle proprietà self-adaptive.
In ultimo sono stati individuati i principali domini in cui i sistemi self-adaptive vengono impiegati.

Nel proseguo esporremo il nostro contributo costituito da un framework per lo sviluppo di applicazioni self-adaptive, le cui scelte di adattamento sono basate su un modello processato a design time.

Capitolo 3

Approccio SAME

In questo capitolo presenteremo l'approccio SAME, un framework che permette di supportare lo sviluppo di sistemi self-adaptive nei confronti dell'incertezza non funzionale. Per supportare la descrizione utilizzeremo un'applicazione d'esempio, chiamata ShopReview che verrà presentata all'inizio di questo capitolo. Tale applicazione, anche se molto semplice, presenta delle sorgenti d'incertezza che se non vengono gestite possono portare alla violazione di requisiti non funzionali.

3.1 ShopReview

Per presentare il nostro lavoro ci baseremo su ShopReview (SR), che è ispirata da ShopSavvy[1], un'applicazione mobile che permette agli utenti di condividere dati su prodotti commerciali e consultare le informazioni prodotte da altri utenti. Il workflow dell'applicazione è mostrato con l'UML Activity Diagram [159] in figura 3.1. Gli utenti di SR possono pubblicare il prezzo di un prodotto che hanno trovato in un certo negozio e, come risposta, l'applicazione ritorna prezzi più convenienti offerti da: (i) posti vicini all'utente (i.e. *Local Search*) o negozi online (i.e. *Web Search*). La mappatura univoca tra prezzo e prodotto è data dal codice a barre. All'avvio l'applicazione richiede di fotografare il codice a barre del prodotto interessato con la fotocamera dello smartphone e di inserire il prezzo relativo a quel codice a barre. Con questi input l'applicazione: (1) riconosce il codice a barre nella foto, (2) controlla online il prodotto associato a quel codice a barre, (3) ritorna la posizione dell'utente, (4) esegue la Local Search e la Web Search, (5) mostra i risultati ottenuti, e (6) permette all'utente di pubblicare il prezzo del prodotto per essere usato dalle ricerche degli altri utenti. Sottolineiamo che

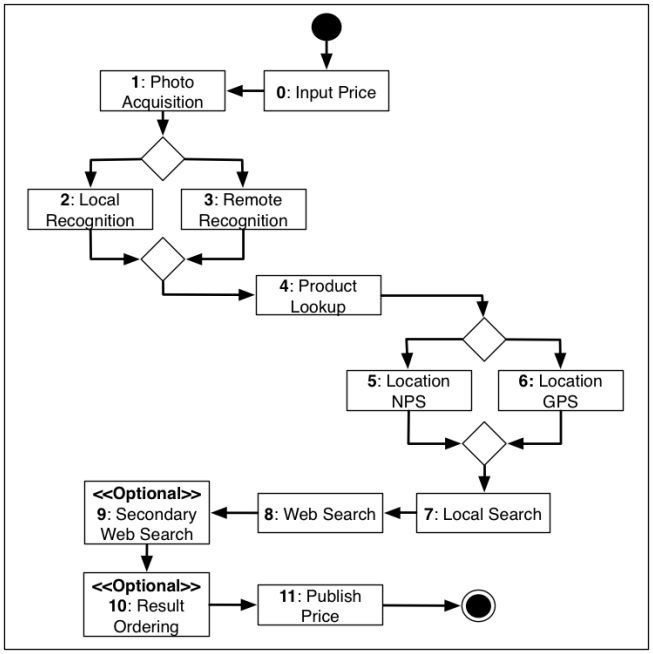


Figura 3.1: Workflow dell'esempio basato su ShopReview.

possono esistere implementazioni alternative di una stessa funzionalità, per

[1]http://shopsavvy.mobi/

esempio per fare la *Recognition* è possibile usare *Local Recognition* o *Remote Recognition*. In aggiunta possono esserci attività opzionali, cioè che possono essere eseguite o meno, ad esempio *Secondary Web Search*.
Assumiamo che SR debba rispettare dei requisiti non funzionali elencati in tabella 3.1. Un requisito è caratterizzato da una metrica non funzionale, che può essere tempo di risposta o usability, e da un classe. Per esempio R1 è un requisito relativo alle performance che di solito viene richiesto da molti market-place per applicazioni mobile. Inoltre le attività contribuiscono in modo diverso al raggiungimento della usability. Per esempio, per ritornare la posizione dell'utente, è preferibile utilizzare il GPS invece che l'NPS[2] perché più preciso.
SAME supporta due classi di requisiti: *Max* e *Min*. Con il primo si indica che $m > th$, mentre con il secondo $m < th$, dove m è una metrica non funzionale (e.g. response time) e th è la soglia da rispettare.

	Descrizione	**Metrica**	**Classe**
R1	Dopo l'input da parte dell'utente l'applicazione deve rispondere entro 3s.	Tempo di Risposta (RT)	Min
R2	L'applicazione deve fornire almeno 5 punti di Usability.	Usability (U)	Max

Tabella 3.1: Requisiti non funzionali di SR.

SR contiene sorgenti di incertezza che manifestandosi a runtime possono impedire di rispettare i requisiti non funzionali. Per esempio, i programmatori possono progettare SR in modo tale da rispettare R1, stimando o misurando sperimentalmente il tempo di risposta della funzionalità Web Search, che invoca un back-end. Tuttavia, questo tempo di risposta può aumentare inaspettatamente a causa, per esempio, della latenza della rete causando la violazione di R1. Questo comportamento inaspettato, se non venisse gestito, potrebbe causare un'insoddisfacente user experience, oppure, nel peggiore dei casi, il rigetto dal market-place. Quindi, anche in questo semplice esempio, gestire l'incertezza è fondamentale per progettare un soddisfacente sistema, ovvero SR deve essere adaptive. Le diverse implementazioni e le attività opzionali sono ciò che sfrutta il nostro approccio per ottimizzare e rendere possibile l'adaptation. È importante notare che i concetti introdotti in questa tesi possono essere applicati anche a sistemi più grandi e complessi dove i benefici del nostro approccio sono molto più rilevanti.

[2]Network Positioning System

3.2 L'approccio SAME

3.2.1 Introduzione

L'obiettivo del nostro approccio è quello di ottenere un sistema in grado di modificare la propria esecuzione tenendo conto delle manifestazioni di incertezza non funzionale (ad esempio tempi risposta incerti per alcuni componenti del sistema) delegando la scelta del workflow da seguire ad un modello che è aware dei requisiti non-funzionali. Tale modello ha conoscenza del comportamento passato, ovvero ha informazioni sulle proprietà non funzionali dall'inizio fino all'istante attuale d'esecuzione, inoltre è in grado di operare una predizione sul futuro avendo una visione globale del workflow del programma. Il modello quindi gestisce lo svolgimento dell'applicazione relativamente ai suoi requisiti non funzionali, cioè verrà interrogato per decidere se eseguire una certa funzionalità piuttosto che un'altra oppure se saltare una determinata attività. Tale modello è rappresentato formalmente da un MDP (*Markov Decision Process*, vedi appendice A). In questo progetto faremo riferimento all'implementazione Java anche se i concetti sono generalizzabili, con leggere modifiche tecnologiche, ad altre piattaforme.

3.2.2 Dettagli della soluzione

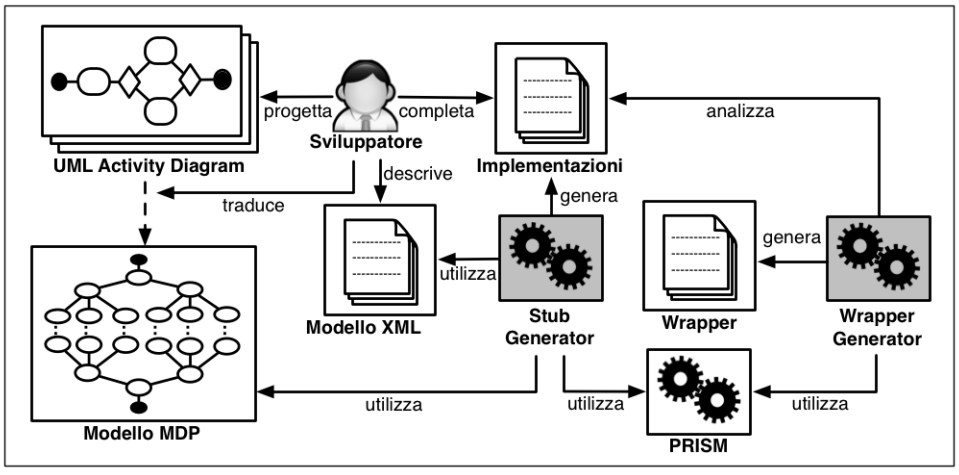


Figura 3.2: Approccio SAME.

SAME è un framework model-based concepito per supportare lo sviluppo e le operazioni a runtime di sistemi self-adaptive, reagendo all'incertezza manifestata dall'ambiente in cui operano tali sistemi.

Partiamo dando una visione generale dell'approccio proposto, come mostrato in figura 3.2. Prima di tutto SAME richiede allo sviluppatore di fornire un modello, in termini di funzionalità astratte, del sistema che intende andare a sviluppare. Questo modello è rappresentato da un Activity Diagram che esprime il workflow dell'applicazione. Da questo diagramma il programmatore, attraverso un processo di traduzione, ricava un MDP. Inoltre un mapping esplicito fra gli stati MDP e le funzionalità astratte del modello viene definito mediante un file di supporto in formato XML. Da questi input viene generato del codice eseguibile, cioè degli *stub* delle classi per ognuna delle funzionalità astratte nell'Activity Diagram. Quindi lo sviluppatore completa tali classi con il codice necessario per l'implementazione effettiva e da queste vengono generati, attraverso un tool, i wrapper. Tali wrapper includono l'implementazione fornita e si occuperanno di gestire il meccanismo di adattamento a runtime. L'inizializzazione e la chiamata a ciascun wrapper viene gestita attraverso un Factory Design pattern [53].
Per spiegare l'approccio lo schematizziamo con una serie di passi come rappresentato in figura 3.3:

1. **Definizione dell'Activity Diagram**: identificazione del workflow dell'applicazione
2. **Traduzione dell'Activity Diagram in un MDP**: creazione di un Markov Decision Process corrispondente all'Activity Diagram
3. **Mapping tra stati MDP e funzionalità astratte**: dichiarazione delle relazioni tra Activity e MDP attraverso un file XML
4. **Esecuzione *StubGenerator***: generazione degli stub
5. **Completamento degli stub**: implementazione con la logica della specifica applicazione
6. **Esecuzione *WrapperGenerator***: generazione wrapper e pattern factory.

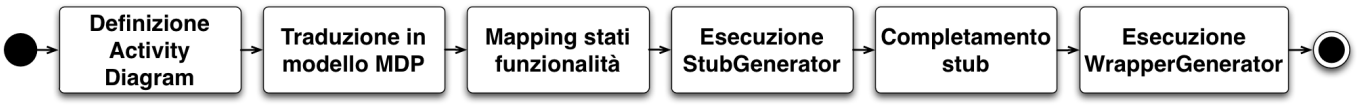
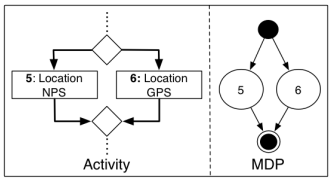


Figura 3.3: Passi SAME

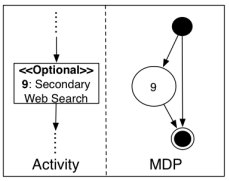
Definizione dell'Activity Diagram In questa fase lo sviluppatore modella la propria applicazione come una serie di funzionalità astratte organizzate in un workflow, rappresentato come un UML Activity Diagram da cui dovrà ricavare un MDP. Questa trasformazione è tutta a carico dello sviluppatore, anche se ci sono proposte per passare da un workflow ad un MDP in modo automatico [28].
In questo workflow potremmo avere delle funzionalità opzionali, che possono essere più o meno eseguite (e.g. Secondary Web Search nel caso di SR). Inoltre per ogni funzionalità astratta i programmatori possono fornire una o più implementazioni alternative. La metodologia per derivare l'insieme delle funzionalità concrete per ognuna di quelle astratte, dati i requisiti totali e le policy di mitigazione dell'incertezza, non rientra negli argomenti discussi in questa tesi. Tuttavia progettare sistemi con implementazioni alternative è un approccio già usato per sistemi software complessi, anche se in modo informale. Per esempio, in applicazioni mobile la posizione dell'utente si ottiene attraverso due alternative: GPS o NPS.

Traduzione dell'Activity Diagram in un MDP Non avendo un tool automatico per la trasformazione è compito dello sviluppatore creare l'MDP che rispecchi esattamente l'Activity Diagram dell'applicazione.
L'utente deve tradurre ogni funzionalità astratta in un semplice MDP con uno stato iniziale e uno stato finale, e con tanti stati intermedi quante sono il numero di implementazioni associate alla funzionalità astratta. Per esempio, sempre facendo riferimento a SR, se vogliamo tradurre la funzionalità per rilevare la posizione dell'utente, che ha due implementazioni *Location GPS* e *Location NPS*, si otterranno due stati intermedi che rappresentano tali alternative, come mostrato in figura 3.4(a).
Per le funzionalità opzionali va aggiunta una transizione diretta dallo stato iniziale allo stato finale. Ad esempio tale modifica dovrà essere inserita se vogliamo creare l'MDP dell'attività opzionale *Secondary Web Search*, come si può osservare in figura 3.4(b).
Una volta creati gli MDP per ognuna delle funzionalità astratte verranno composti come spiegato nel proseguo.
Due MDP corrispondenti a due nodi A e B consecutivi nell'Activity Diagram vengono composti eliminando lo stato finale di A e lo stato iniziale di B. Le transizioni entranti nello stato finale di A termineranno negli stati di destinazione delle transizioni uscenti dallo stato iniziale di B. Per chiarire questo concetto prendiamo come esempio la composizione degli MDP che rappresentano le funzionalità *Web Search* e *Secondary Web Search*. Eliminiamo lo stato finale di *Web Search* e lo stato iniziale di *Secondary Web Search*. Visto

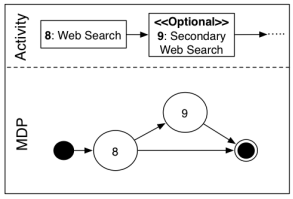
che il primo nodo ha soltanto una sola transizione che termina nello stato finale appena eliminato, questa verrà divisa in due transizioni singole che termineranno negli stati di destinazione delle transizioni del secondo nodo il cui stato iniziale è stato eliminato, come mostrato in figura 3.4(c).
Applicando questo processo di trasformazione iterativamente a tutte le funzionalità specificate nell' Activity Diagram, otteniamo l'MDP in figura 3.4(d).
È importante sottolineare che tale modello è decorato con l'impatto che ogni stato ha su ognuna delle metriche dei requisiti. Per esempio, sempre dalla figura 3.4(d), vediamo che lo stato *Location NPS* ha un impatto sul tempo di risposta di 2 secondi e fornisce un guadagno di usability di 1 punto. Questa decorazione è a carico dello sviluppatore che specificherà tali valori in fase di design. La rappresentazione del modello MDP si basa sulla sintassi di un



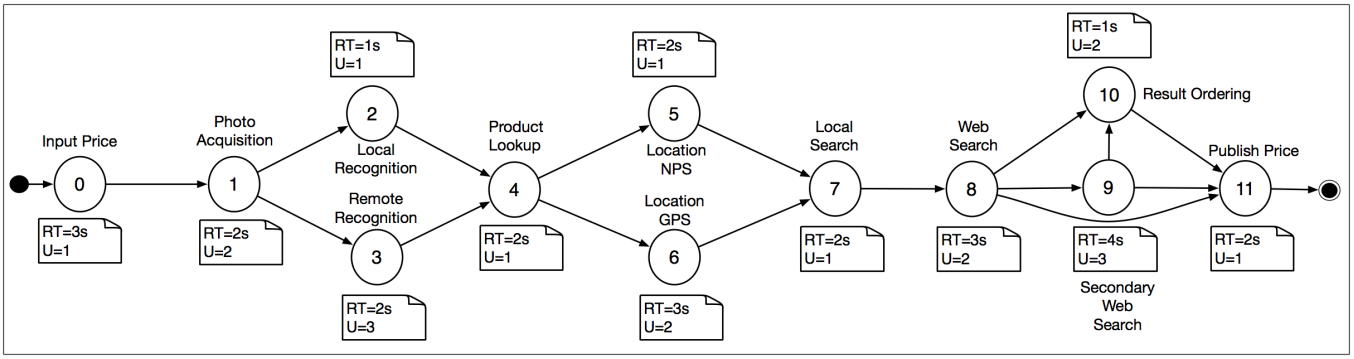
(a) Alternative



(b) Opzionali



(c) Composizione di MDP



(d) Modello ShopReview

Figura 3.4: Processo Traduzione

probabilistic model checker. Nello specifico è stato utilizzato il tool PRISM [158]. Attraverso un file di testo, nel formalismo di PRISM, lo sviluppatore può quindi esprimere il proprio MDP.
Il listato 3.1 è la rappresentazione, nella sintassi di PRISM, dell'MDP in figura 3.4(d). Al suo interno possiamo osservare la specifica dell'impatto di ogni stato su ognuna delle metriche, che corrisponde, nel formalismo MDP, ad una reward (vedi appendice A). Per esempio lo stato 5, che rappresen-

ta la traduzione della funzionalità *Location NPS*, presenta una valore per la reward *time* pari a 2 mentre per la *usability* pari a 1. Questi sono gli impatti su tempo di risposta e usability da parte di *Location NPS* di cui avevamo parlato in precedenza.
Sempre nel listato 3.1 osserviamo che vengono specificate le transizioni che collegano gli stati. Per esempio con la scrittura *[] s = 0 -> (s'= 1);* viene rappresentata la transizione tra lo stato 0 e lo stato 1.

```
module shopreview

        s : [0..12] init 0;

        [] s = 0 -> (s'= 1);
        [] s = 1 -> (s'= 2);
        [] s = 1 -> (s'= 3);
        [] s = 2 -> (s'= 4);
        [] s = 3 -> (s'= 4);
        [] s = 4 -> (s'= 5);
        [] s = 4 -> (s'= 6);
        [] s = 5 -> (s'= 7);
        [] s = 6 -> (s'= 7);
        [] s = 7 -> (s'= 8);
        [] s = 8 -> (s'= 9);
        [] s = 8 -> (s'= 10);
        [] s = 8 -> (s'= 11);
        [] s = 9 -> (s'= 10);
        [] s = 9 -> (s'= 11);
        [] s = 10 -> (s'= 11);
        [] s = 11 -> (s'= 12);
        [] s = 12 -> (s'= 12);
endmodule

label "final" = s = 12;

rewards "time"
        [] s=0 : 3;
        [] s=1 : 2;
        [] s=2 : 1;
        [] s=3 : 2;
        [] s=4 : 2;
```

```
        [] s=5 : 2;
        [] s=6 : 3;
        [] s=7 : 2;
        [] s=8 : 3;
        [] s=9 : 4;
        [] s=10 : 1;
        [] s=11 : 2;
endrewards
rewards "usability"
        [] s=0 : 1;
        [] s=1 : 2;
        [] s=2 : 1;
        [] s=3 : 3;
        [] s=4 : 1;
        [] s=5 : 1;
        [] s=6 : 2;
        [] s=7 : 1;
        [] s=8 : 2;
        [] s=9 : 3;
        [] s=10 : 2;
        [] s=11 : 1;
endrewards
```

Listato 3.1: File PRISM modello Markoviano

Mapping tra stati MDP e funzionalità astratte In questa fase viene chiesto allo sviluppatore di fornire un mapping, tramite un file XML, tra gli stati nel modello PRISM e le funzionalità astratte del workflow. Tale mapping serve per dare dei nomi, che verranno usati dai tool, alle funzionalità che il programmatore intende andare a sviluppare. Per farlo utilizzerà un modello XML creato da noi per tale scopo. Tale modello è racchiuso dal tag $< model > ... < /model >$, che si compone al suo interno dai seguenti sottotag:

- $< interface > ... < /interface >$: all'interno lo sviluppatore specificherà il nome dell'interfaccia, che verrà creata per rappresentare una funzionalità astratta, per esempio per determinare la posizione dell'utente si indicherà $< interface >< name > LocationInterface < /name > ... < /interface >$. I tag da utilizzare sono i seguenti:

  - $< name > ... < /name >$: lo sviluppatore scriverà il nome da dare all'interfaccia, come mostrato sopra per *LocationInterface*.
  - $< class > ... < /class >$: per ogni alternativa dovrà inserire questo tag con l'attributo *type=optional* se tale alternativa è opzionale. All'interno di questo tag vanno inoltre inseriti quelli per rappresentare il mapping tra funzionalità e numero dello stato PRISM corrispondente:
    * $< name > ... < /name >$: qui il programmatore scriverà il nome dell'alternativa, per esempio per l'implementazione di *Location GPS* scriverà $< class >< name > LocationGPS < /name > ... < /class >$
    * $< state > ... < /state >$: invece qui il numero dello stato PRISM associato con tale alternativa, l'esempio appena citato si completerà inserendo $< class > ... < state > 5 < /state >< /class >$.

- $< totalReward > ... < /totalReward >$: lo sviluppatore specificherà il valore che vuole andare a rispettare. Per esempio, riprendendo la tabella 3.1 sui requisiti non funzionali, se il programmatore vuole specificare R1 scriverà $< totalReward >< time > 3 < /time >< /totalReward >$. È importante che per la scelta dei nomi delle reward da inserire tra i tag, si scelga quelli specificati nel file PRISM e se il programmatore vuole gestire il tempo d'esecuzione, specifichi come nome *time*, ovviamente sia nel file PRISM che tra i tag della *totalReward*, come detto in precedenza.

- $< weight > ... < /weight >$: grazie a questo può specificare se prioritizzare una reward rispetto ad un'altra, assegnandole un peso. Per esempio, volendo assegnare un peso maggiore al tempo di risposta rispetto all'usability il programmatore scriverà $< weight >< time > 0.7 < /time >< usability > 0.3 < /usability >< /weight >$.

- $< policy > ... < /policy >$: con questo tag si va a indicare a quale classe appartiene la reward, cioè se *Max* o *Min*. Riprendendo ancora il caso del tempo di risposta e usability, per indicare che la prima appartiene alla classe *Min* mentre la seconda a *Max*, lo sviluppatore specificherà $< policy >< time > lower < /time >< usability > upper < /usability >< /policy >$.

In questi ultimi due tag è necessario inserire il nome della reward per cui si specifica la policy e il weight (anche per questi due campi bisogna rispettare

le regole scritte qui sopra per la *totalReward*).
Nel listato 3.2 è riportato il file XML usato nel nostro esempio basato su SR.

```
<model>
        <interface><name>InputPriceInterface</name>
                <class><name>InputPrice</name><state
                    >0</state>
                </class>
        </interface>
        <interface><name>PhotoAcquisitionInterface</
            name>
                <class><name>PhotoAcquisition</name><
                    state>1</state>
                </class>
        </interface>
        <interface><name>RecognitionInterface</name>
                <class><name>LocalRecognition</name><
                    state>2</state>
                </class>
                <class><name>RemoteRecognition</name><
                    state>3</state>
                </class>
        </interface>
        <interface><name>ProductLookupInterface</name>
                <class><name>ProductLookup</name><
                    state>4</state>
                </class>
        </interface>
        <interface><name>LocationInterface</name>
                <class><name>LocationGPS</name><state
                    >5</state>
                </class>
                <class><name>LocationNPS</name><state
                    >6</state>
                </class>
        </interface>
        <interface><name>LocalSearchInterface</name>
                <class><name>LocalSearch</name><state
                    >7</state>
```

```
            </class>
        </interface>
        <interface><name>WebSearchInterface</name>
            <class><name>WebSearch</name><state
                >8</state>
            </class>
        </interface>
        <interface><name>SecondaryWebSearchInterface</
            name>
            <class type="optional"><name>
                SecondaryWebSearch</name><state>9</
                state>
            </class>
        </interface>
        <interface><name>ResultOrderingInterface</name
            >
            <class type="optional"><name>
                ResultOrdering</name><state>10</state>
            </class>
        </interface>
            <interface><name>PublishPriceInterface
                </name>
            <class><name>PublishPrice</name><state
                >11</state>
            </class>
        </interface>
        <totalReward><time>22</time><usability>17</
            usability></totalReward>
        <weight><time>0.7</time><usability>0.3</
            usability></weight>
        <policy><time>lower</time><usability>upper</
            usability></policy>
</model>
```

Listato 3.2: File XML esempio basato su SR

È importante sottolineare che nel listato 3.2 il requisito non funzionale sul tempo di risposta (R1) è di 22 secondi, invece che di 3, come riportato in tabella 3.1. La motivazione è data dal fatto che questo file è quello che verrà utilizzato per la validazione di SAME.

Esecuzione *StubGenerator* Per l'esecuzione del primo tool è necessario indicare:

- il path del file PRISM
- il path del file XML
- la cartella dove si vuole che vengano creati gli stub delle classi
- il package del progetto, così che il tool lo inserisca in tutte le classi che crea, evitando di lasciare tale operazione allo sviluppatore che usa il framework.

Da questa esecuzione vengono create una gerarchia di interfacce e di classi che corrispondono all'architettura definita inizialmente dall'Activity Diagram e codificata più formalmente dall'MDP. Per esempio nell'Activity Diagram iniziale erano presenti *Location NPS* e *Location GPS*, che corrispondono agli stati 5 e 6 dell'MDP in figura 3.4. Pertanto lo StubGenerator crea l'interfaccia *LocationInterface* (listato 3.3) e le classi che la implementano *LocationGPS* (listato 3.4) e *LocationNPS* (listato 3.5).
Questi file sono stati creati eseguendo lo *StubGenerator* fornendo in ingresso il file PRISM (listato 3.1), il file XML (listato 3.2) e come package *it.polimi.package*.

```
package it.polimi.simulation;
public interface LocationInterface {

}
```

Listato 3.3: Interfaccia per prendere la posizione dell'utente

```
package it.polimi.simulation;
public class LocationGPS implements LocationInterface
    {

}
```

Listato 3.4: Classe che implementa LocationInterface per l'alternativa tramite GPS

```
package it.polimi.simulation;
public class LocationNPS implements LocationInterface
    {

}
```

Listato 3.5: Classe che implementa LocationInterface per l'alternativa tramite NPS

Completamento degli stub Questa operazione viene eseguita dallo sviluppatore. Inserisce le signature dei metodi che intende definire all'interno dell'interfaccia, poi per tutte le classi che la implementano scriverà il codice di tali metodi. Quando le interfacce e le classi sono tutte complete si può passare all'esecuzione del *WrapperGenerator*.
Nel listato 3.6 si può osservare una possibile implementazione dell'interfaccia *LocationInterface*, mentre per *LocationGPS* e *Location NPS* si può far riferimento ai listati 3.7 e al listato 3.8. Tali implementazioni potranno essere poi elaborate dal *WrapperGenerator*. Ovviamente all'interno dei metodi, al posto dei puntini, ci sarà il codice per ottenere la posizione con il GPS in un caso e con il metodo NPS nell'altro.

```
package it.polimi.simulation;
public interface LocationInterface {
        public void getLocation();
}
```

Listato 3.6: Aggiunto il metodo *getLocation*

```
package it.polimi.demo;

public class LocationGPS implements LocationInterface
   {
        @Override
        public void getLocation() {
        ...
        }

}
```

Listato 3.7: Aggiunto *getLocation* a LocationGPS

```
package it.polimi.demo;

public class LocationNPS implements LocationInterface
   {
        @Override
        public void getLocation() {
        ...
        }
}
```

Listato 3.8: Aggiunto *getLocation* a LocationNPS

Esecuzione *WrapperGenerator* I parametri necessari in ingresso al WrapperGenerator sono:

- path file XML
- path file PRISM
- package del progetto
- il percorso dove risiedono i file compilati dallo sviluppatore

Grazie a questi input, il WrapperGenerator, invocando il tool PRISM, modifica l'MDP sostituendo, per ogni stato, l'impatto su ogni requisito con un intervallo $\langle minR(s), maxR(s)\rangle$ (vedi appendice A). Questo intervallo rappresenta una predizione sugli impatti necessari per completare l'esecuzione partendo da uno specifico stato *s*. Per esempio consideriamo lo stato 5 a cui PRISM ha assegnato $\langle 9, 14\rangle$ per il tempo di risposta, come mostrato in figura 3.5. Questo indica che l'impatto sul tempo di risposta per completare l'esecuzione è compreso tra i 9 e 14 secondi. Quindi il nuovo modello MDP,

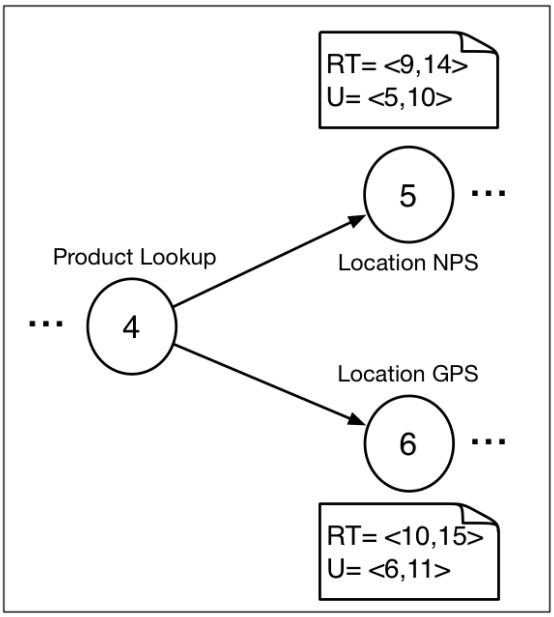


Figura 3.5: Scelta alternativa della funzionalità *Location*.

ottenuto con l'esecuzione del WrapperGenerator, decorato con gli intervalli per ogni requisito non funzionale è quello mostrato in figura 3.6.
Inoltre il WrapperGenerator crea una factory e un wrapper per ogni interfaccia, prelevando i metodi inseriti dal programmatore. La factory creata

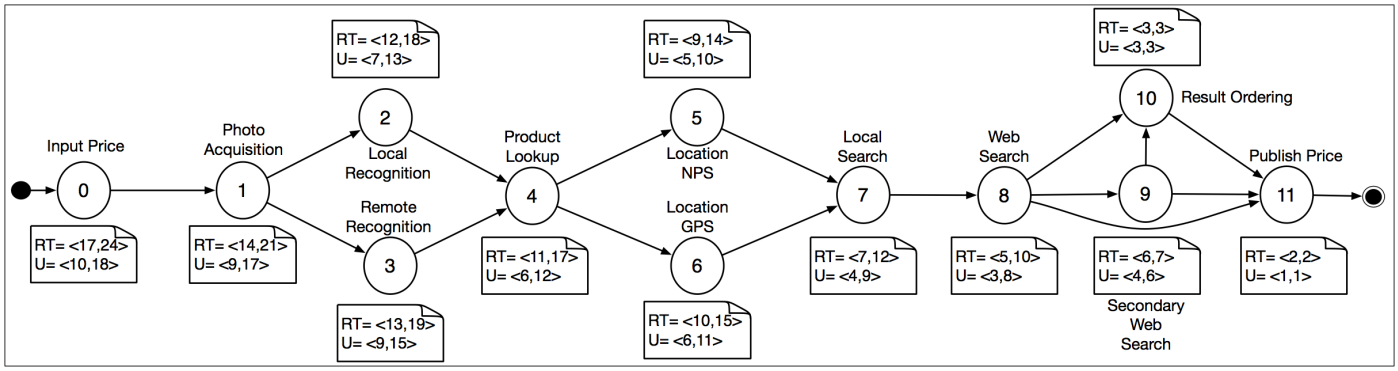


Figura 3.6: Modello Markoviano dell'esempio con intervalli.

serve per evitare allo sviluppatore di dover inizializzare tutti i wrapper e quando avrà bisogno di una di queste classi, dovrà soltanto chiamare il metodo corrispondente. I wrapper, che implementano le interfacce, conterranno tutti i metodi specificati nei quali sarà contenuta la logica di scelta tra un'alternativa piuttosto che un'altra. Riprendendo l'esempio citato precedentemente, WrapperGenerator crea, dall'interfaccia di *Location*, il wrapper *LocationInterfaceWrapper* (listato 3.9) e all'interno della factory *FactoryWrapper* un metodo (listato 3.10) per inizializzare e ritornare un'istanza di *LocationInterfaceWrapper*. Inoltre all'interno di *LocationInterfaceWrapper* vengono inseriti i valori degli intervalli calcolati da PRISM, di cui abbiamo parlato precedentemente, che serviranno per la logica di scelta. Per esempio i valori relativo allo stato 5, cioè alla funzionalità *Location NPS*, vengono inseriti con le istruzioni *mapReward.put(LocationNPStimeMin, 9.0f); mapReward.put(LocationNPStimeMax, 14.0f);*.

```
package it.polimi.simulation;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import java.util.ArrayList;
public class LocationInterfaceWrapper implements
    LocationInterface {

private List<String> alternatives = new ArrayList<
    String>();
private Map<String,Float> mapReward = new HashMap<
    String,Float>();
```

```
private List<String> rewardNameList = new ArrayList<
    String>();

        public LocationInterfaceWrapper(){
                rewardNameList.add("time");
                mapReward.put("totaltime", 22.0f);
                mapReward.put("weighttime", 0.7f);
                mapReward.put("LocationGPStimeMin",
                    10.0f);
                mapReward.put("LocationGPStimeMax",
                    15.0f);
                alternatives.add("LocationGPS");
                mapReward.put("LocationGPStime", 3.0f)
                    ;
                mapReward.put("LocationNPStimeMin",
                    9.0f);
                mapReward.put("LocationNPStimeMax",
                    14.0f);
                alternatives.add("LocationNPS");
                mapReward.put("LocationNPStime", 2.0f)
                    ;
                mapReward.put("policytime",0.0f);
                rewardNameList.add("usability");
                mapReward.put("totalusability", 17.0f)
                    ;
                mapReward.put("weightusability", 0.3f)
                    ;
                mapReward.put("LocationGPSusabilityMin
                    ", 6.0f);
                mapReward.put("LocationGPSusabilityMax
                    ", 11.0f);
                mapReward.put("LocationGPSusability",
                    2.0f);
                mapReward.put("LocationNPSusabilityMin
                    ", 5.0f);
                mapReward.put("LocationNPSusabilityMax
                    ", 10.0f);
                mapReward.put("LocationNPSusability",
                    1.0f);
                mapReward.put("policyusability",1.0f);
```

```
        }

        public void getLocation(){
                String choice = AlternativeUtility.
                    getAlternative(alternatives,
                    rewardNameList,mapReward);
                LocationInterface obj = null;
                if(choice.equals("LocationNPS")){
                        obj = new LocationNPS();
                }
                if(choice.equals("LocationGPS")){
                        obj = new LocationGPS();
                }
                obj.getLocation();
                AlternativeUtility.updateContext(
                    rewardNameList, choice, mapReward);
        }

}
```

Listato 3.9: Wrapper per le alternative di *LocationInterface*

```
package it.polimi.simulation;
public class FactoryWrapper{
...
        public static LocationInterface
            getLocationInterface(){
                return new LocationInterfaceWrapper();
}
...
}
```

Listato 3.10: Factory creata

Inoltre viene inserita anche una lista dei nomi dei requisiti non funzionali, esempio per inserire la usability viene usata l'istruzione *rewardNameList.add(usability);*. Dal file XML, discusso in precedenza, vengono estrapolate le policy per ogni requisito che vengono poi inserite a loro volta nel codice del wrapper. Ad esempio per la usability, la cui policy è *Max*, viene inserito *mapReward.put(policyusability,1.0f);*.
Tutte queste informazioni, insieme agli intervalli sui requisiti, vengono utilizzati dalla logica di adaptation per compiere la scelta tra un'alternativa piut-

tosto che un'altra. Tale logica è contenuta all'interno del wrapper nella classe *AlternativeUtility* che attraverso la chiamata *String choice = AlternativeUtility.getAlternative(alternatives,rewardNameList,mapReward);* ritorna l'alternativa da eseguire. Alla fine avviene un aggiornamento di contesto, sempre da parte dell'AlternativeUtility, per aggiornare il valore dell'impatto sui requisiti non funzionali, aggiungendo l'impatto dato dall'alternativa scelta. Per esempio se venisse scelta l'esecuzione di *LocationNPS*, il contesto verrebbe aggiornato aggiungendo 1 punto di usability, mentre il tempo invece viene gestito in modo automatico da SAME.
Come funziona la logica di adaptation verrà spiegato nel seguente paragrafo.

3.2.3 Logica di scelta

In questa sezione andremo ad analizzare in maggiore dettaglio come viene presa la decisione di adaptation, ovvero la logica contenuta in *AlternativeUtility*. Per spiegarla andremo a sfruttare un esempio, ovvero la scelta tra *Location GPS* e *Location NPS* una volta terminata l'esecuzione di *Product Lookup*, come mostrato in figura 3.7 già mostrata precedentemente.

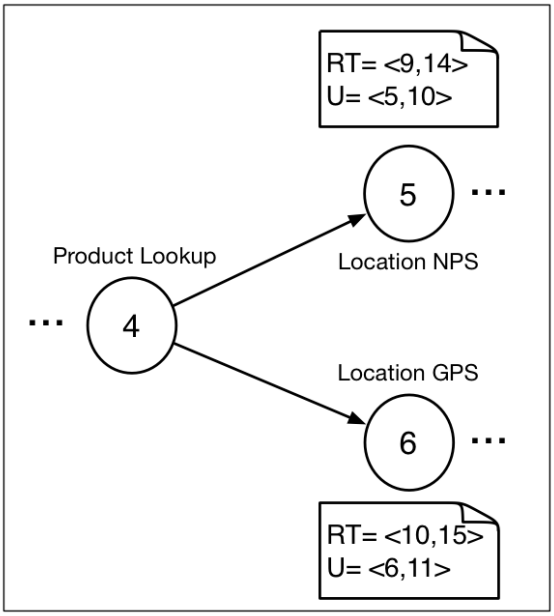


Figura 3.7: Scelta alternativa della funzionalità *Location*.

Al momento della chiamata il wrapper conosce quale è il contesto dei requisiti non funzionali, che sono stati specificati all'inizio dallo sviluppatore, cioè l'impatto complessivo al momento dell'invocazione. Nel nostro esem-

pio, avendo *response time* e *usability*, conosceremo quanto tempo è passato dall'inizio dell'applicazione, lo indicheremo con c_{RT}, e quanto abbiamo guadagnato fino ad ora in termini di usability lungo tutta l'esecuzione, c_U.
Oltre al contesto, SAME conosce anche il valore dei requisiti non funzionali che si vogliono rispettare (ad esempio la nostra applicazione deve avere un requisito sul tempo globale di risposta), questi valori vengono indicati con *th* (threshold), quindi per la usability abbiamo th_U mentre per il response time th_{RT}. Inoltre è a conoscenza, grazie al modello MDP, della predizione dell'impatto sui requisiti non funzionali per completare l'esecuzione dell'applicazione, per esempio in termini di response time avrò un impatto dai 9 ai 14 secondi partendo da *Location NPS*, come si può osservare in figura 3.7. Tale impatto, per arrivare alla fine dell'applicazione, viene modellizzato con una variabile aleatoria distribuita uniformemente nell'intervallo $< 9, 14 >$.
Grazie alla threshold, alla variabile uniformemente distribuita all'interno dell'intervallo e al contesto attuale d'esecuzione, SAME è in grado di calcolare la probabilità che l'applicazione rispetti il requisito non funzionale che ci siamo posti, per esempio terminare l'esecuzione entro un certo limite di tempo, eseguendo lo stato considerato. Tornando all'esempio, lo stato che avevamo considerato era quello relativo a *Location NPS*, quindi SAME calcolerà la probabilità di rispettare il response time eseguendo appunto *Location NPS*.

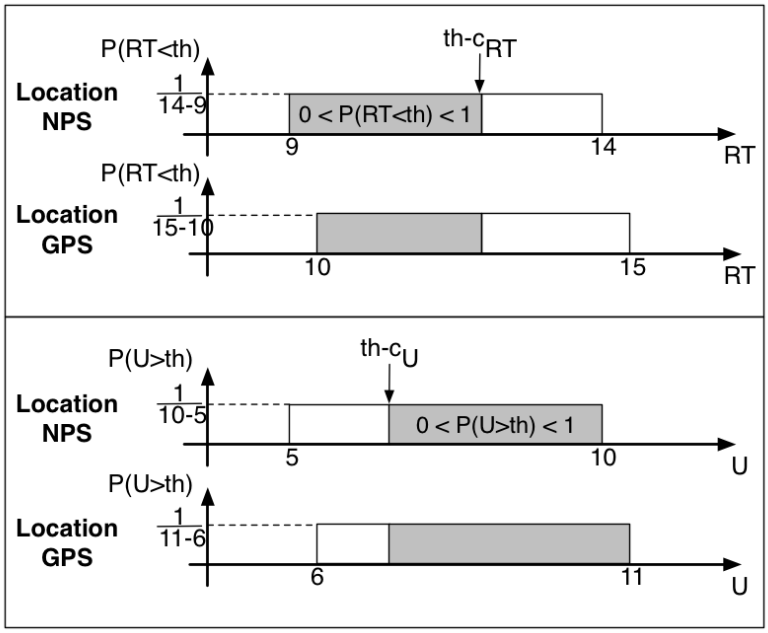


Figura 3.8: Rappresentazione della probabilità di scelta.

Per aiutare nella spiegazione vengono mostrate le formule per il calcolo di questa probabilità facendo riferimento alla scelta tra *Location NPS* e *Location GPS*, come mostrato in figura 3.8. Presentiamo solo il caso relativo al requisito non funzionale response time.

$$Diff_{RT} = th_{RT} - c_{RT}$$

Questo valore viene utilizzato insieme all'intervallo degli stati considerati, in questo caso *5* e *6*, per ottenere la differenza tra $Diff_{RT}$ e il bound inferiore dell'intervallo, perché il response time deve essere minimizzato, mentre se la policy era quella di massimizzare il requisito allora si utilizzava il bound superiore.

$$Base_{RT,NPS} = Diff_{RT} - l_{b,NPS} = Diff_{RT} - 9$$
$$Base_{RT,GPS} = Diff_{RT} - l_{b,GPS} = Diff_{RT} - 10$$

Questa base calcolata viene moltiplicata per l'altezza del rettangolo.

$$Height_{RT,NPS} = \frac{1}{u_{b,NPS} - l_{b,NPS}} = \frac{1}{14-9}$$
$$Height_{RT,GPS} = \frac{1}{u_{b,GPS} - l_{b,GPS}} = \frac{1}{15-10}$$

$$p_{RT,NPS}(RT_{tot} < th_{RT}) = Base_{RT,NPS} \times Height_{RT,NPS}$$

Una volta ottenuta questa probabilità viene sommata alla probabilità della usability. Abbiamo optato per la somma delle probabilità di ciascun requisito non funzionale perché nel caso entrambe le alternative hanno probabilità uguale a zero per una determinata reward, viene scelta quella che ha maggiori chance di rispettare l'altro requisito. Se avessimo scelto la moltiplicazione entrambe le alternative sarebbero state equivalenti. Questo valore viene confrontato con quello dell'altra alternativa e il più alto determinerà la scelta vincente.
Nel caso invece la reward fosse stata da massimizzare, i calcoli sono gli stessi, tranne nel caso del calcolo della base.

$$Base = u_b - Diff$$

3.3 Implementazione

Nella presente sezione verrà analizzata la struttura del framework SAME esposto in precedenza, il quale facilita gli sviluppatori nella progettazione di applicazioni self-adaptive. Il framework è implementato con un paradigma Object-Oriented, nello specifico attraverso il linguaggio Java.
Nella sezione 3.2.2 è stata mostrato come i tool messi a disposizione dal

framework aiutano nella definizione di un sistema software con proprietà di adattamento. Tali tool sfruttano principalmente una classe denominata *Utility*, la quale contiene le funzionalità per la costruzione del sistema che lo sviluppatore intende implementare.
Le funzionalità più interessanti sono:

- *processPrismModel:* analizza e processa il modello MDP fornito dal programmatore attraverso il tool PRISM Model Checker [158].
- *processXMLModel:* preleva e processa le informazioni di mapping fra stato e funzionalità specificati nel modello XML dato.
- *generateEmptyInterfaces, generateEmptyClasses:* generazione automatica di interfacce e classi secondo la gerarchia specificata nel modello descrittivo.
- *generateWrappers:* generazione automatica dei wrapper rispettando l'implementazione della logica dell'applicazione definita dallo sviluppatore.

La classe *Utility* pertanto viene sfruttata unicamente in fase di design, nella quale viene definita la struttura e l'implementazione del sistema software.

Diversamente, la classe *AlternativeUtility*, come mostrato in precedenza, viene utilizzata durante l'esecuzione dell'applicazione. Tale classe infatti raccoglie le funzionalità e la logica self-adaptive per ottenere l'adattamento a runtime. La logica self-adaptive esposta nella sezione 3.2.3 è contenuta nel metodo *getAlternative*, il quale durante l'esecuzione dell'applicazione verrà invocato dai wrapper per consentire la mitigazione delle manifestazioni di incertezza incontrate.

3.4 Summary

Nel presente capitolo è stato mostrato come il nostro approccio supporta la progettazione ed implementazione di un sistema con capacità di adattamento. Sono stati spiegati in dettaglio i passi fondamentali e gli strumenti che permettono la generazione automatica della struttura dell'applicazione e della logica self-adaptive.
Mediante SAME pertanto è possibile costruire un sistema che presenta una netta distinzione fra la logica self-adaptive, che permette le azioni di adattamento, e la logica specifica dell'applicazione sviluppata. In questo modo, è

possibile ottenere un implementazione più chiara che consente quindi un'eventuale modifica del sistema software risulta essere più semplice.
Nel prossimo capitolo valideremo l'approccio SAME testandone l'effettivo funzionamento e le performance a runtime.

Capitolo 4

Validazione

In questo capitolo ci occuperemo della validazione dell'approccio proposto. Nella prima parte andremo a verificare con un esempio come SAME gestisce a runtime l'incertezza non funzionale e il conseguente impatto sui requisiti. Nella seconda parte andremo a misurare, in vari scenari, l'overhead introdotto da SAME rispetto al caso in cui viene disabilitato.

4.1 Scenari d'uso

In questa sezione andremo a vedere SAME in esecuzione sull'applicativo SR ingegnerizzato come mostrato nella descrizione dell'approccio. L'implementazione dei metodi, inglobata dai wrapper, non contiene altro che una logica che permette di avere un tempo di esecuzione fisso, ma che sia settabile a nostro piacimento. Questo ci permette di verificare il comportamento di SAME in vari scenari, illustrati nel seguito di questo paragrafo.
Gli scenari che abbiamo ottenuto variando appunto questo parametro di ritardo nelle varie implementazioni che stimolano i wrapper a utilizzare la propria logica adaptive in modo da scegliere una volta un'implementazione e una volta un'altra, verranno commentate utilizzando il modello in figura 4.1. Il percorso d'esecuzione verrà colorato di rosso in modo che sia immediato per il lettore quale scelta è stata compiuta dai wrapper. In questo modello notiamo che ogni stato ha associato tutti i requisiti non funzionali, questi valori sono quelli che l'utente inserirà all'interno del file PRISM. Come accennato nel capitolo precedente, il requisito non funzionale R1 sul tempo di risposta, non è più di 3 secondi, come mostrato in tabella 3.1, ma di 22 secondi.

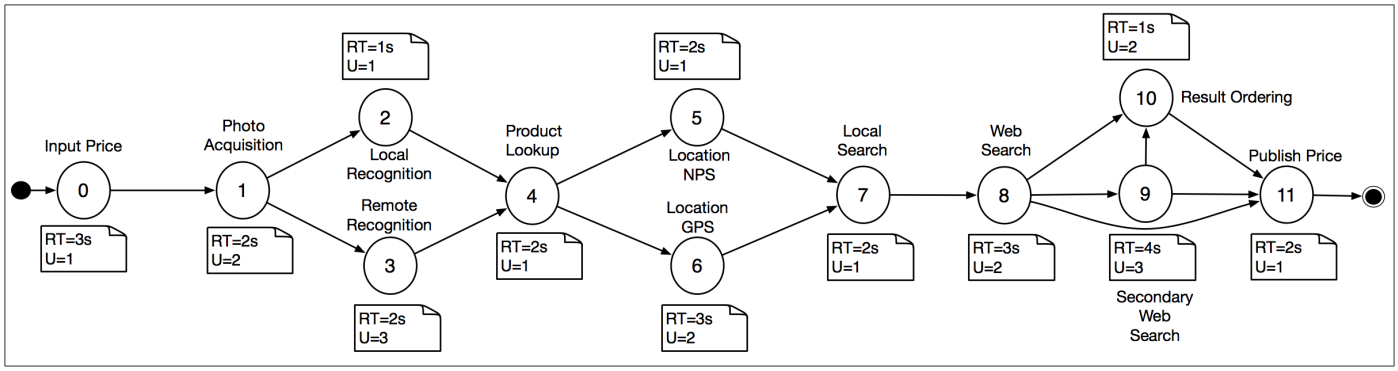


Figura 4.1: Modello Markoviano dell'esempio con intervalli.

4.1.1 Scenario 1

Nel primo scenario i tempi utilizzati sono quelli mostrati nella tabella 4.1, dove troviamo anche i tempi attesi, ovvero quelli che devono essere inseriti nel modello Markoviano dal programmatore.
Visto che i tempi di risposta delle prime due attività (*Input Price* e *Photo Acquisition*) sono vicini a quelli attesi, quando il wrapper relativo alla scelta tra *Local Recognition* e *Remote Recognition* verrà chiamato ci ritornerà il

Attività	Tempo di risposta impostato[s]	Tempo di risposta atteso[s]
Input Price	2	3
Photo Acquisition	0.5	2
Local Recognition	1	1
Remote Recognition	2	2
Product Lookup	2	2
NPS	2	2
GPS	3	3
Local Search	2	2
Web Search	3	3
Secondary Web Search	4	4
Result Ordering	1	1
Publish Price	2	2

Tabella 4.1: Tempi di risposta inseriti nello scenario 1.

secondo perché ha una usability maggiore rispetto alla prima. Il tempo di risposta più alto di *Remote Recognition*, rispetto a *Local Recognition*, non ci preoccupa perché siamo molto in anticipo rispetto a quanto avevamo progettato e quindi non rischiamo di violare il requisito non funzionale *Response Time*.
In figura 4.2 vediamo che è appunto quello che succede, vengono prese sempre le scelte con tempo d'esecuzione maggiore, perché possiamo permettercelo, ma che ci danno una usability maggiore. Sempre per lo stesso motivo vengono eseguite le attività opzionali.

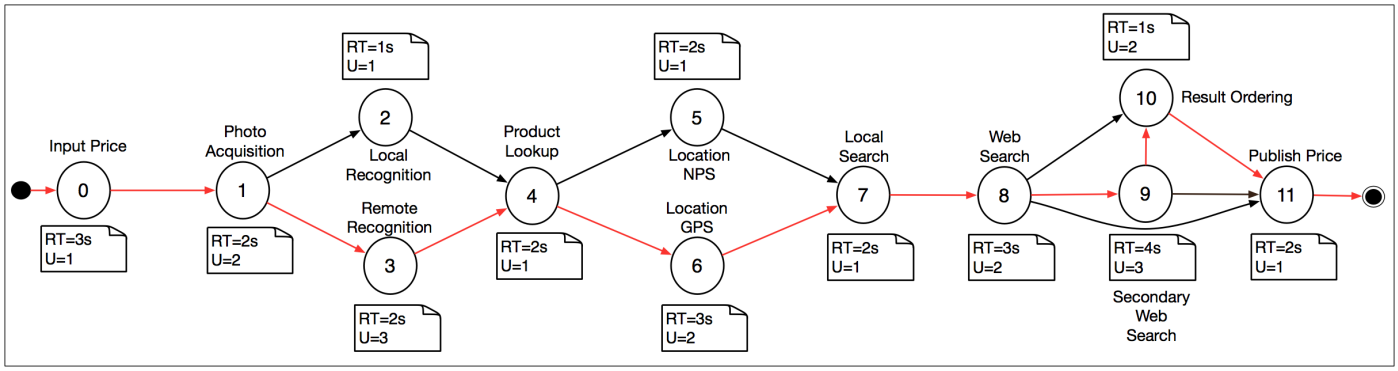


Figura 4.2: Path eseguito nello scenario 1.

4.1.2 Scenario 2

Nel secondo scenario andremo a modificare i tempi rispetto al caso precedente e come il nostro sistema reagisce. I tempi modificati rispetto a prima verranno segnati di rosso, come si può vedere dalla tabella 4.2. Come si vede

Attività	**Tempo di risposta impostato[s]**	**Tempo di risposta atteso[s]**
Input Price	2.5	3
Photo Acquisition	1	2
Local Recognition	1	1
Remote Recognition	2	2
Product Lookup	1	2
NPS	2	2
GPS	3	3
Local Search	2	2
Web Search	3	3
Secondary Web Search	4	4
Result Ordering	1	1
Publish Price	2	2

Tabella 4.2: Tempi di risposta inseriti nello scenario 2.

nella tabella 4.3 se non gestissimo questo aumento del tempo, cioè scegliendo ancora *Remote Recognition*, il tempo totale dell'applicazione diventerebbe 22.5, che è maggiore di un mezzo secondo rispetto al requisito che vogliamo rispettare. Quindi alla chiamata del primo wrapper ci viene ritornato *Local Recognition*, come si può vedere nel path d'esecuzione mostrato in figura 4.3, che ha usability minore rispetto a *Remote Recognition*, ma anche un tempo di risposta minore che ci permette di rimanere al di sotto del requisito non funzionale che ci siamo posti. Infatti scegliendo questa alternativa il tempo atteso per arrivare alla fine dell'applicazione è di 21.5 secondi, che è inferiore rispetto al requisito che ci siamo prefissati.

Requisito non funzionale (R1)[s]	22
Tempo totale atteso scegliendo *Remote Recognition*[s]	22.5
Tempo totale atteso scegliendo *Local Recognition*[s]	21.5

Tabella 4.3: Differenze tempo d'esecuzione non gestito e gestito.

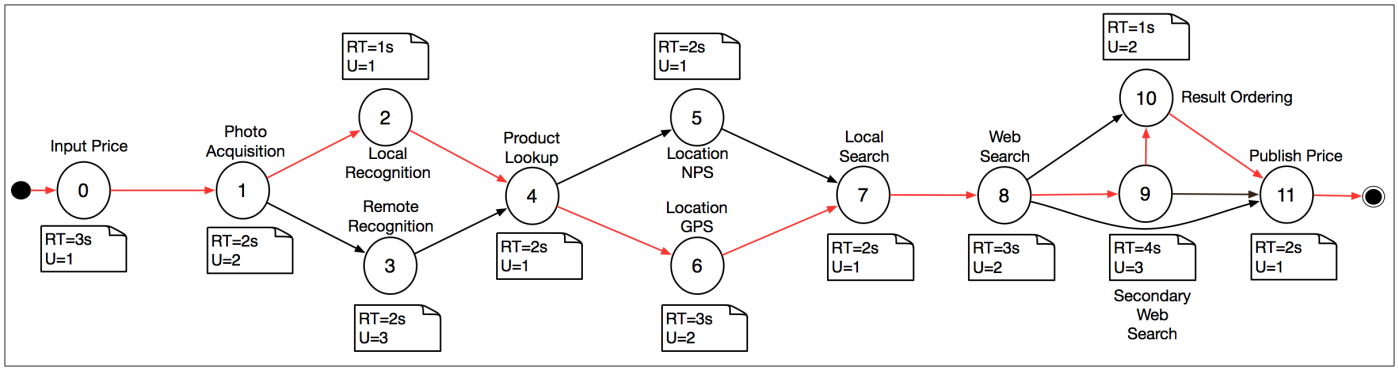


Figura 4.3: Path eseguito nello scenario 2.

4.1.3 Scenario 3

Continuando ad aumentare i tempi notiamo che il sistema d'esempio modifica a sua volta l'esecuzione. Guardando la tabella 4.4 notiamo che abbiamo incrementato ulteriormente i tempi delle prime due attività, il secondo wrapper reagirà ritornando la scelta che impiega meno tempo, come si può vedere dalla figura 4.4. L'aumento non è così significativo da dover saltare anche le attività opzionali.

Attività	Tempo di risposta impostato[s]	Tempo di risposta atteso[s]
Input Price	4.5	3
Photo Acquisition	2	2
Local Recognition	1	1
Remote Recognition	2	2
Product Lookup	1	2
NPS	2	2
GPS	3	3
Local Search	2	2
Web Search	1	3
Secondary Web Search	4	4
Result Ordering	1	1
Publish Price	2	2

Tabella 4.4: Tempi di risposta inseriti nello scenario 3.

Dalla tabella 4.5 possiamo notare che se il sistema scegliesse il calcolo della posizione tramite *GPS*, il tempo atteso totale della nostra applicazione diventerebbe di 22.5 secondi, violando di mezzo secondo il requisito che

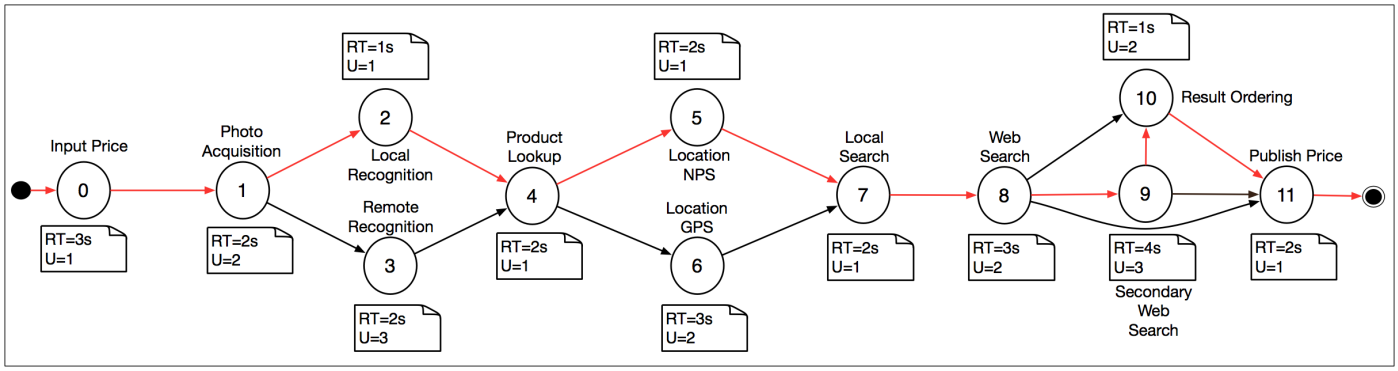


Figura 4.4: Path eseguito nello scenario 3.

ci siamo prefissati. Scegliendo invece il calcolo tramite *NPS* riusciamo a rispettare il requisito non funzionale che ci siamo posti.

Requisito non funzionale (R1)[s]	22
Tempo totale atteso scegliendo *GPS*[s]	22.5
Tempo totale atteso scegliendo *NPS*[s]	21.5

Tabella 4.5: Differenze tempo d'esecuzione non gestito e gestito.

4.1.4 Scenario 4

Dalla tabella 4.6 vediamo che abbiamo aumentato i tempi di *Product Lookup* e *Web Search* per andare ad influire sulle attività opzionali. Infatti, come mostrato in figura 4.5, il path d'esecuzione salta la funzionalità *Secondary Web Search*.

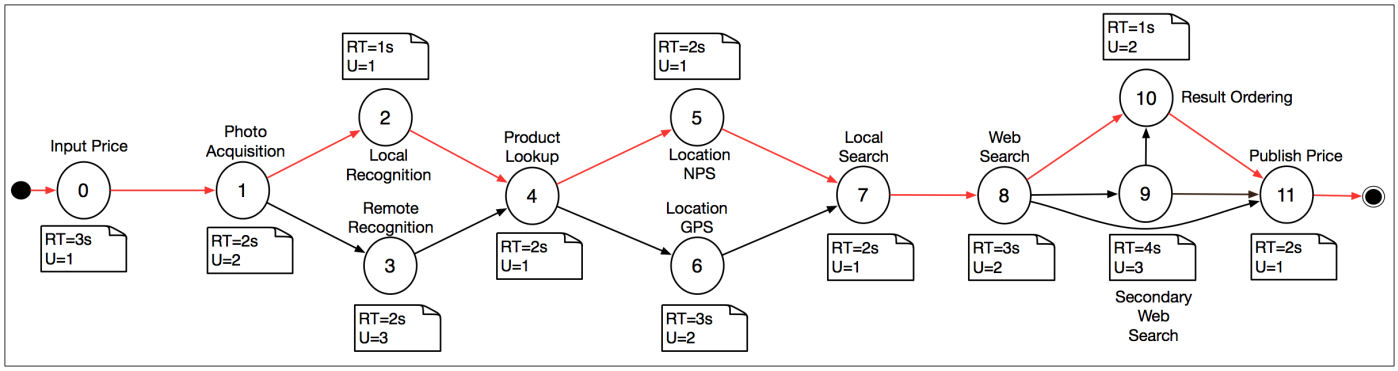


Figura 4.5: Path eseguito nello scenario 4.

Attività	**Tempo di risposta impostato[s]**	**Tempo di risposta atteso[s]**
Input Price	4.5	3
Photo Acquisition	2	2
Local Recognition	1	1
Remote Recognition	2	2
Product Lookup	2	2
NPS	2	2
GPS	3	3
Local Search	2	2
Web Search	3	3
Secondary Web Search	4	4
Result Ordering	1	1
Publish Price	2	2

Tabella 4.6: Tempi di risposta inseriti nello scenario 4.

L'attività opzionale *Secondary Web Search* viene saltata perché, come possiamo vedere dalla tabella 4.7, eseguendo tale funzionalità andremo a superare di 1.5 secondi il requisito non funzionale che ci siamo posti. Saltando tale operazione torniamo a rispettare i 22 secondi che abbiamo imposto come limite superiore del tempo d'esecuzione per la nostra applicazione.

Requisito non funzionale (R1)[s]	22
Tempo totale atteso eseguendo *Secondary Web Search*[s]	23.5
Tempo totale atteso saltando *Secondary Web Search*[s]	19.5

Tabella 4.7: Differenze tempo d'esecuzione non gestito e gestito.

4.1.5 Scenario 5

Come scenario finale abbiamo deciso di aumentare il tempo di *Web Search*,come si può vedere in tabella 4.8, portandolo a 6 secondi così da modificare il comportamento del wrapper di *Result Ordering*. Dalla tabella 4.9 possiamo capire che se il wrapper non intervenisse il tempo totale d'esecuzione andrebbe sopra i 22 secondi che ci siamo prefissati, mentre saltando l'esecuzione di *Result Ordering* riusciamo a rispettare questo limite.

Attività	**Tempo di risposta impostato**[s]	**Tempo di risposta atteso**[s]
Input Price	4.5	3
Photo Acquisition	2	2
Local Recognition	1	1
Remote Recognition	2	2
Product Lookup	2	2
NPS	2	2
GPS	3	3
Local Search	2	2
Web Search	6	3
Secondary Web Search	4	4
Result Ordering	1	1
Publish Price	2	2

Tabella 4.8: Tempi di risposta inseriti nello scenario 5.

Requisito non funzionale (R1)[s]	22
Tempo totale atteso eseguendo *Result Ordering*[s]	22.5
Tempo totale atteso saltando *Result Ordering*[s]	21.5

Tabella 4.9: Differenze tempo d'esecuzione non gestito e gestito.

Quindi per questa ultima esecuzione il path d'esecuzione ottenuto è quello ottenuto in figura 4.6.

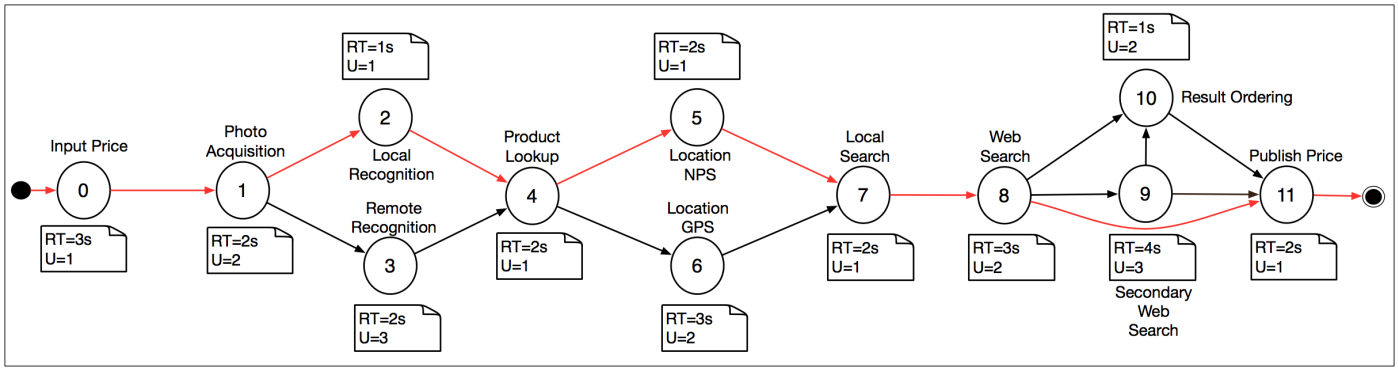


Figura 4.6: Path eseguito nello scenario 5.

4.2 Overhead dell'approccio

In questa sezione andremo ad analizzare l'overhead introdotto dal nostro framework in funzione del numero di alternative e del numero di requisiti che SAME dovrà gestire.
L'obiettivo degli esperimenti di questa sezione è quello di comparare l'overhead introdotto da SAME con quello di altre possibili implementazioni. In tutti gli scenari i metodi chiamati sono delle attività fittizie di durata 50 ms. Di seguito verranno riportati soltanto i grafici, per osservare i valori numerici si può fare riferimento alla tabelle riportate nell'appendice B.

4.2.1 Overhead rispetto alla versione hard-coded

Il primo scenario riguarda la comparazione dell'esecuzione SAME quando ogni attività ha soltanto una sola implementazione, quindi la scelta dei wrapper è deterministica, rispetto ad una sequenza equivalente di chiamate dirette alle stesse implementazioni. Ci riferiamo a questo scenario con la sequenza di chiamate senza SAME come versione *hard-coded*.
In questo scenario misuriamo l'overhead introdotto solo dal wrapping, senza includere il tempo per il calcolo dell'alternativa da utilizzare.
Come si vede dalla figura 4.7 la versione di SAME introduce un ritardo, per esempio con 50 attività viene introdotto un overhead del 2.13%, come riportato in tabella B.1.

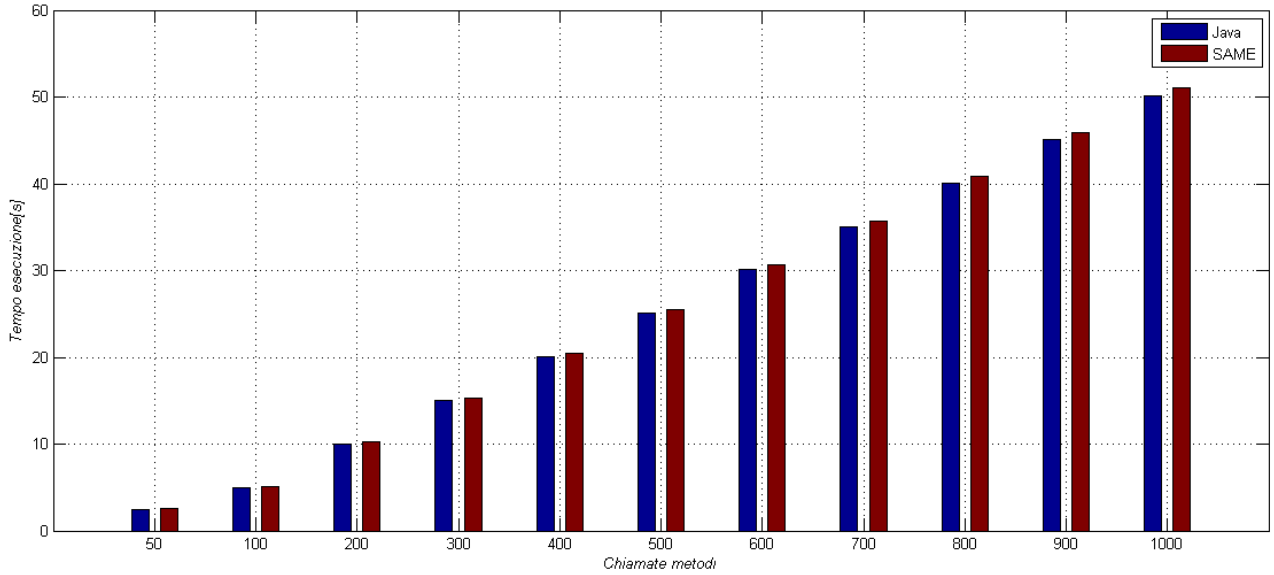


Figura 4.7: Overhead introdotto con SAME rispetto alla versione hard-coded

4.2.2 Overhead rispetto alla versione hard-coded con scelta tra più alternative

In questo scenario ripetiamo quello appena fatto introducendo delle alternative all'interno di SAME, per essere più precisi useremo 3 alternative per ogni attività, e confrontiamo i risultati sempre con la versione hard-coded usata in precedenza. Nella figura 4.8 si nota che con SAME viene introdotto sempre un ritardo, che rispetto al caso precedente è aumentato. Infatti per il caso con 50 attivitià, l'overhead introdotto rispetto alla versione hard-coded è del 3.01%, come riportato in tabella B.2.

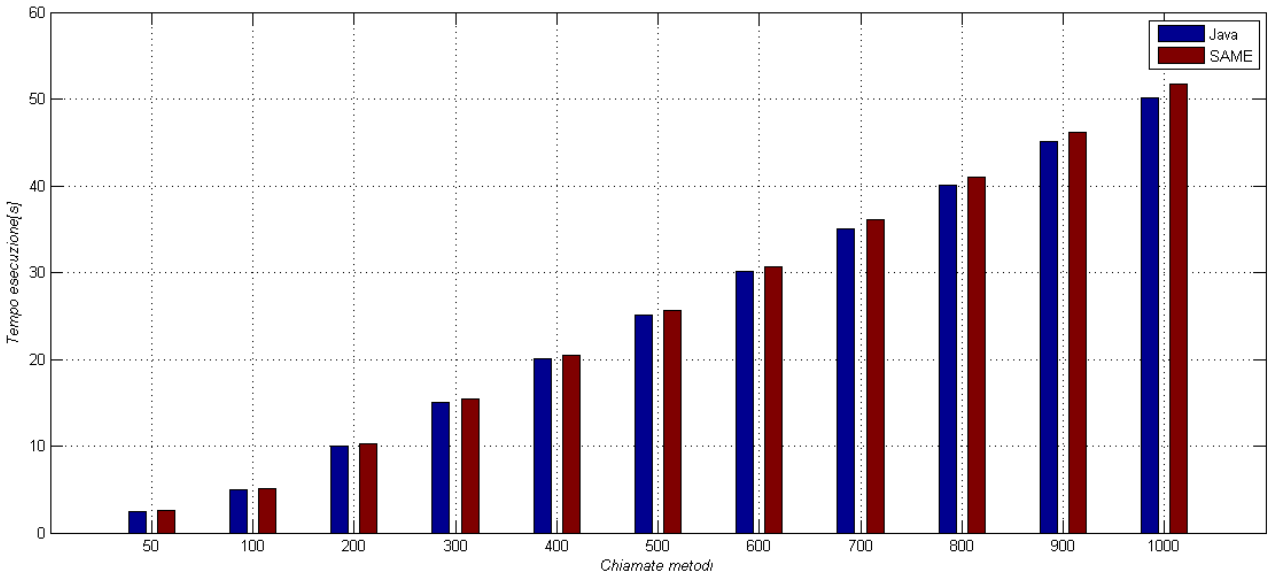


Figura 4.8: Overhead introdotto con SAME rispetto alla versione hard-coded con scelta tra più alternative

4.2.3 Variazione numero alternative

In questa scenario riprendiamo le prove fatte in precedenza, ma ci focalizziamo sul ritardo introdotto da SAME aumentando il numero di alternative tra cui un wrapper può scegliere. Il nostro setting di base sarà formato da 2000 attività e da 2 requisiti non funzionali da gestire.
Le differenze di tempo tra le singole di prove si possono osservare in figura 4.9, in particolare l'overhead introdotto ogni volta che aggiungo una nuova alternativa è in media di 7.5 ms.

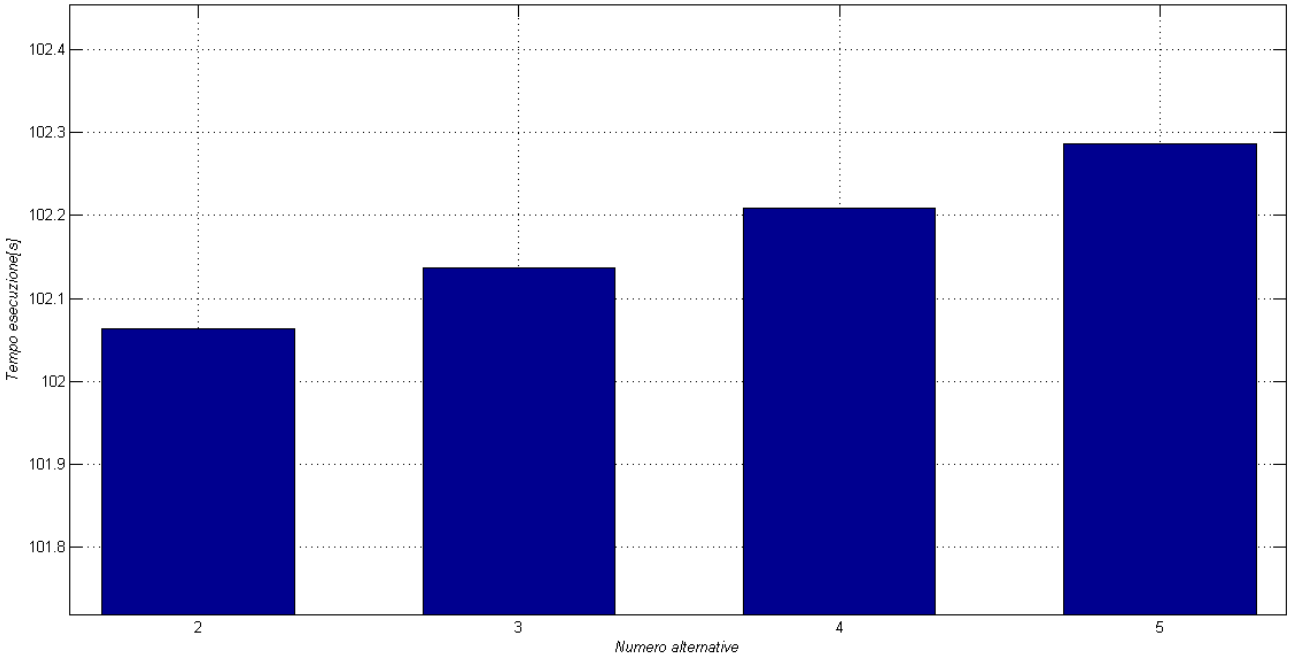


Figura 4.9: Overhead introdotto con SAME aumentando il numero di alternative tra cui scegliere.

4.2.4 Variazione numero requisiti non funzionali

In questo ultimo scenario vogliamo vedere come varia l'overhead modificando il numero di requisiti non funzionali che SAME dovrà gestire. Il nostro setting di base si compone di 2000 attività, ciascuna con 3 alternative.
Le differenze di tempo tra le prove si possono osservare in figura 4.10, in particolare ogni volta che aggiungiamo un nuovo requisito non funzionale da gestire, l'overhead introdotto è in media di 15.4 ms.

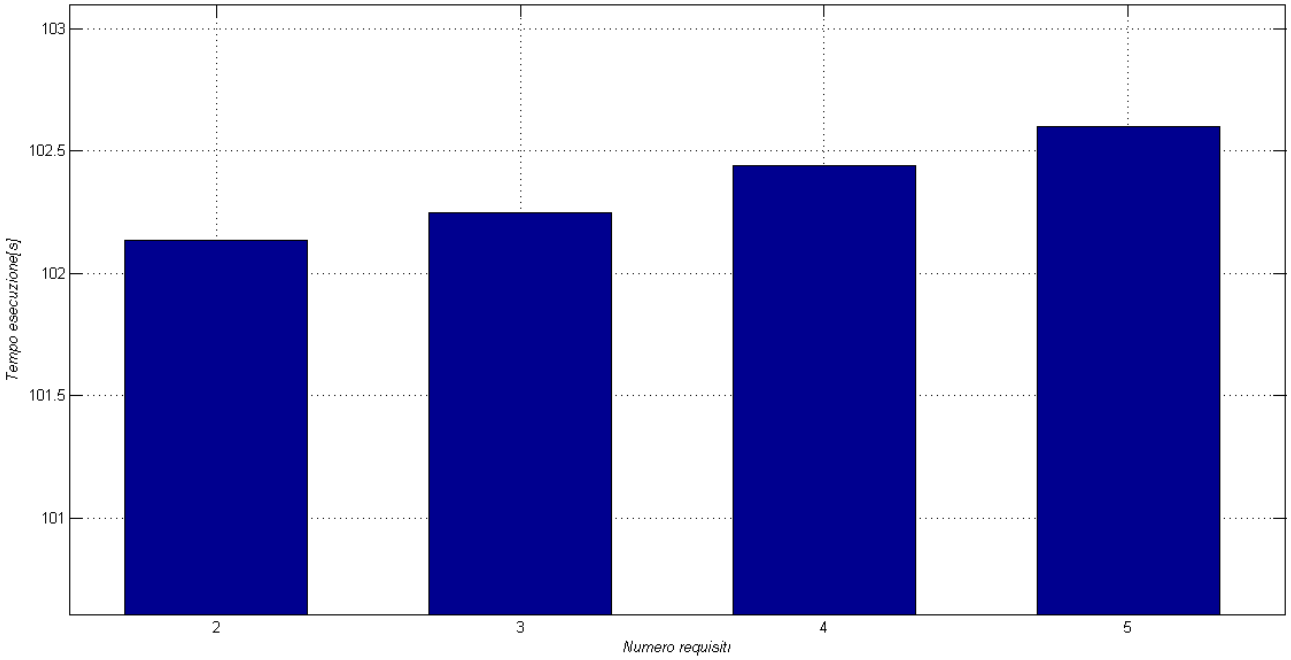


Figura 4.10: Overhead introdotto con SAME aumentando il numero di requisiti non funzionali.

4.3 Conclusioni

Come mostrato nell'analisi condotta attraverso differenti scenari, un sistema implementato attraverso l'approccio SAME è in grado di rispondere alle manifestazioni di incertezza scaturite dall'ambiente non predicibile in cui operano le applicazioni.
I sistemi self-adaptive tipicamente introducono un overhead dovuto alle computazioni aggiuntive introdotte per permettere le azioni di adattamento. Le precedenti misure mostrano come SAME introduca un limitato overhead sia rispetto al caso hard-coded sia all'aumentare del numero di requisiti e alternative. I risultati quantitativi si possono trovare in forma tabellare nell'appendice B.

4.4 Summary

In questo capitolo abbiamo verificato come SAME gestisce l'incertezza a runtime, andando a scegliere l'alternativa che ci permette di massimizzare la probabilità di rispettare i requisiti non funzionali che vogliamo gestire. Inoltre abbiamo misurato l'overhead introdotto dal nostro framework

e abbiamo constatato che è piuttosto contenuto, permettendo al programmatore di sfruttare le funzionalità di SAME con un impatto limitato sulle performance globali dell'applicazione.

Capitolo 5

Conclusioni

Il presente elaborato di tesi si è occupato di proporre un approccio innovativo, chiamato SAME[1], per la definizione e l'implementazione di sistemi self-adaptive. SAME si concentra in particolare sui requisiti non-funzionali ed è basato sull'utilizzo di modelli con differenti gradi di astrazione.
Un modello iniziale di alto livello dei possibili flussi di esecuzione viene trasformato in un modello formale, nel quale sono inoltre codificate le informazioni che riguardano i requisiti del sistema. Tale modello viene quindi sfruttato a runtime per gestire l'esecuzione in risposta alle manifestazioni di incertezza non-funzionale. Nello specifico questo modello formale è un Markov Decision Process (MDP) decorato con reward.
Il funzionamento di SAME è stato illustrato attraverso il caso di studio ShopReview, che prende spunto dall'applicazione esistente ShopSavvy[2]. Come mostrato, tale applicazione anche se relativamente semplice presenta diverse sorgenti di incertezza e alcuni punti di ridondanza nelle funzionalità, che possono essere sfruttati da SAME per ottimizzare l'esecuzione del sistema.
L'effettiva efficacia di un sistema implementato mediante SAME è stata analizzata nel capitolo 4 ottenendo risultati positivi. Ad esempio si è mostrato come, al variare di un insieme ritardi iniettati artificialmente, il sistema risponde in modo automatico adattando il proprio comportamento per soddisfare i requisiti.
Inoltre, è stato misurato l'overhead introdotto dall'approccio ed i risultati ottenuti indicano un impatto sulle prestazioni contenuto e tollerabile nella maggior parte dei casi.
Sono possibili notevoli sviluppi futuri che verranno descritti nel successivo paragrafo.

5.1 Sviluppi futuri

Gli sviluppi futuri che estendono questo contributo riguardano innanzitutto una traduzione automatica da Activity Diagram a modello MDP, che completerebbe quindi il supporto nel processo di design. Inoltre si pensa di proseguire nella validazione dell'approccio con scenari più estesi e complessi e di introdurre un supporto ad una classe più ampia di requisiti, oltre a quelli illustrati nel presente lavoro di tesi. Infine si pianifica di completare ed irrobustire il prototipo di SAME esistente rendendolo disponibile pubblicamente.

[1] Self-Adaptive Model-based framEwork
[2] http://shopsavvy.mobi/

Appendice A

Markov Decision Process

Markov Decision Process sono modelli di Kripke non deterministici con transizioni probabilistiche fra gli stati. Rappresentano la combinazione di processi stocastici discreti con la proprietà di Markov e il non determinismo. Formalmente, un MDP è una tupla $\langle S, s_0, F, Steps \rangle$ dove:

- S è un insieme finito di stati, $s_0 \in S$ è lo stato iniziale e $F \subseteq S$ è l'insieme degli stati finali;
- *Steps* rappresenta la funzione transizione, tale per cui $\forall s \in S$, $Steps(s) = Dist(s)$ oppure $Steps(s) \in 2^S$, dove $Dist(S) = \{(s_1, p_1), ..., (s_k, p_k)\}$, con $\sum_{j=1}^{k} p_j = 1$, è la distribuzione di probabilità discreta su S.

Notare che se per uno stato finale $s_F \in F$ $Steps(s_F) = \{(s_F, 1)\}$, l'autoanello è spesso non rappresentato graficamente. MDP possono essere arricchiti con reward, attraverso le quali si può quantificare un beneficio (oppure perdita) causato dalla permanenza in uno specifico stato oppure dallo spostamento lungo una certa transizione.
Una reward $\rho : S \to R_{\leq 0}$ è un valore non negativo assegnato ad uno stato. Possono rappresentare informazioni come il tempo medio di esecuzione, il consumo di potenza oppure il numero di operazioni I/O. Inoltre negli MDP ogni stato s può essere anche annotato con reward cumulate a partire da s fino allo stato finale s_F. Dato uno stato s, possono esistere molti path che lo connettono allo stato finale s_F. Ognuno di essi accumula come reward la somma delle reward degli stati presenti nel path. Se s è tale per cui $Steps(s) = Dist(S)$, la reward totale cumulata per s è data dalla somma pesata rispetto a $Dist(s)$ delle reward cumulate per tutti i path che hanno origine da s.
Tuttavia tale quantità diviene un intervallo a causa delle transizioni non

deterministiche. Infatti, tutte le scelte non deterministiche hanno associato una reward cumulativa. Per rappresentare tutte queste alternative, ogni stato s è annotato con l'intervallo $\langle minR(s), maxR(s)\rangle$ nel quale sono comprese tutte le reward non deterministiche. Tali valori sono calcolati come segue. Se s è uno stato finale, $minR(s) = \rho(s)$, mentre se $s \in S - F$, quando $Steps(s) = \{(s_1, p_1), ..., (s_k, p_k)\}$, ovvero le transizioni uscenti da s formano una distribuzione di probabilità, $minR(s) = \sum_{j=1}^{k} p_j \times minR(s_j)$, e quando $Steps(s) = \{s_1, ..., s_m\}$, ovvero le transizioni uscenti da s rappresentano una scelta non deterministica, $minR(s) = min_{i\in\{1,...,m\}} minR(s_i)$. Se uno dei path sul quale la reward cumulativa è calcolata contiene cicli, le equazioni precedenti richiamano se stesse ricorsivamente. Metodi iterativi numerici sono utilizzati per risolvere tali equazioni. La condizione di terminazione di tali equazioni è ottenuta quando la massima differenza delle soluzioni con le successive iterazioni scende al di sotto di una soglia fissata. Il limite superiore dell'intervallo, $maxR(s)$, è calcolato in modo analogo. Per una visione più approfondita si rimanda a [15].
L'esempio seguente può aiutare ad avere una più chiara comprensione delle precedenti nozioni.

A.1 Esempio

Data la situazione iniziale seguente:

- Stati: 7
- Stato iniziale: S_0
- Stati finali: $\{S_6\}$
- Step:
  - S_0:{(S_1,0.2),(S_2,0.8)}
  - S_1:{((S_3,1))}
  - S_2:{((S_3,1))}
  - S_3:{((S_4,1)),(S_5,1)}
  - S_4:{((S_6,1))}
  - S_5:{((S_6,1))}
  - S_6:{((S_6,1))}

- Reward:
  - S_0:1
  - S_1:2
  - S_2:1
  - S_3:2
  - S_4:3
  - S_5:4
  - S_6:1

La situazione descritta è rappresentata nella figura A.1

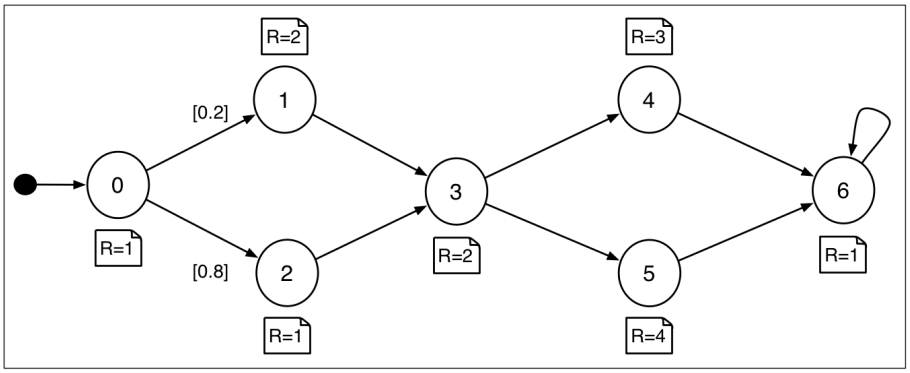


Figura A.1: MDP

Procediamo quindi con il calcolo degli intervalli attraverso la metodologia descritta nella precedente sezione.
S_6 rappresenta uno stato speciale, ovvero lo stato finale. Il suo intervallo pertanto sarà $\langle 1, 1 \rangle$, poiché i suoi valori di minimo e massimo coincidono con il valore della reward per tale stato.
S_5 non rappresenta un caso speciale, quindi il valore minimo del suo intervallo è calcolato come:

$$minR(S_5) = \sum_{j=1}^{k} p_j \times minR(s_j)$$

A valle dello stato S_5 l'unico stato presente è S_6, la cui probabilità di raggiungerlo sarà pari a 1. Quindi applicando la formula utilizzata in precedenza si ottiene:

$$minR(S_5) = minR(S_6) = 1$$

Sempre per lo stesso stato, la formula per il calcolo del valore massimo risulta essere analoga e si ottiene considerando $maxR(S_5)$. Essendo quindi $maxR(S_5) = minR(S_5)$, l'intervallo risultante per S_5 sarà: $\langle 1, 1\rangle$.
Il calcolo per S_4 è numericamente identico a quello effettuato per l'intervallo S_5 ed il suo risultato sarà: $\langle 1, 1\rangle$.
Passando al calcolo dei valori minimo e massimo per lo stato S_3 si incontra una situazione interessante. Infatti da tale stato è possibile raggiungere tramite le funzioni di transizione gli stati: S_4 ed S_5. La scelta presenta una componente non deterministica e pertanto per calcolare il valore minimo di S_3 verrà applicata la formula:

$$minR(S_3) = min_{i\in\{1,\dots,m\}} minR(s_i) = \\ min\{minR(S_4), minR(S_5)\} = min\{4, 5\} = 4$$

Ripetendo il calcolo anche per il massimo si avrà:

$$maxR(S_3) = max_{i\in\{1,\dots,m\}} maxR(s_i) = \\ max\{maxR(S_4), maxR(S_5)\} = min\{4, 5\} = 5$$

Quindi l' intervallo calcolato per lo stato S_3 sarà: $\langle 4, 5\rangle$.
Gli stati S_1 e S_2 presentano un unica transizione uscente e pertanto il calcolo dell'intervallo risulta semplice. Applicando quindi le formule, per entrambi gli stati si ottiene come risultato l'intervallo $\langle 6, 7\rangle$.
Più interessante è invece il calcolo dell'intervallo dello stato S_0. Ad ognuna delle due transizioni presenti è associata la probabilità con la quale tale transizione può essere selezionata nel momento della scelta.
Quindi il calcolo per $minR(S_0)$ sarà:

$$minR(S_0) = \textstyle\sum_{j=1}^{k} p_j \times minR(s_j) = \\ p(S_1) \times minR(S_1) + p(S_2) \times minR(S_2) = \\ 0.2 \times 8 + 0.8 \times 7 = 1.6 + 5.6 = 7.2$$

Allo stesso modo è possibile calcolare l'estremo superiore dell'intervallo:

$$maxR(S_0) = \textstyle\sum_{j=1}^{k} p_j \times maxR(s_j) = \\ p(S_1) \times maxR(S_1) + p(S_2) \times maxR(S_2) = \\ 0.2 \times 9 + 0.8 \times 8 = 1.8 + 6.4 = 8.2$$

L'intervallo per lo stato S_0 sarà quindi $\langle 7.2, 8.2\rangle$.
Applicando in questo modo le nozioni teoriche esposte nella sezione precedente si ottiene la situazione finale rappresentata in figura A.2.

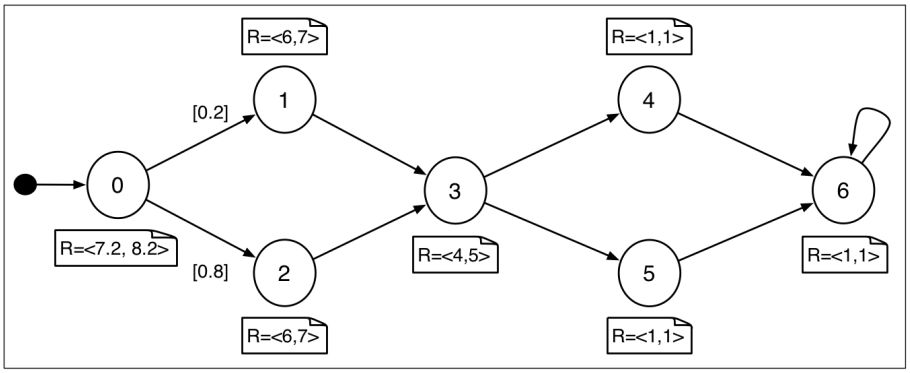


Figura A.2: MDP con intervalli.

Appendice B

Risultati sperimentali

Numero attività	**Hard-coded[s]**	**SAME[s]**	**% ritardo**
50	2.50691	2.56032	2.13
100	5.01368	5.11282	1.98
200	10.0274	10.21322	1.85
300	15.0409	15.31497	1.82
400	20.054	20.41402	1.8
500	25.0672	25.5094	1.76
600	30.0805	30.60656	1.75
700	35.0941	35.71246	1.76
800	40.1086	40.80033	1.72
900	45.1221	45.89202	1.71
1000	50.1359	51.00975	1.74

Tabella B.1: Differenze tempo d'esecuzione con SAME e codice hard-coded

Numero attività	Hard-coded[s]	SAME[s]	% ritardo
50	2.50691	2.58227	3.01
100	5.01368	5.15704	2.86
200	10.0274	10.28532	2.57
300	15.0409	15.40528	2.42
400	20.054	20.50808	2.26
500	25.0672	25.6136	2.18
600	30.0805	30.71647	2.11
700	35.0941	35.90001	2.3
800	40.1086	40.93117	2.05
900	45.1221	46.1791	2.34
1000	50.1359	51.22224	2.17

Tabella B.2: Differenze tempo d'esecuzione con SAME(con alternative) e codice hard-coded

Numero alternative	Tempo esecuzione[s]
2	102.062664937
3	102.137171192
4	102.208837687
5	102.286622239

Tabella B.3: Differenze tempo d'esecuzione con SAME aumentando il numero di alternative tra cui scegliere.

Numero requisiti	Tempo esecuzione[s]
2	102.137265793
3	102.249563245
4	102.440595216
5	102.599621317

Tabella B.4: Differenze tempo d'esecuzione con SAME aumentando il numero di requisiti non funzionali.

Bibliografia

[1] R. Bellman. *A Markovian Decision Process.* Journal of Mathematics and Mechanics 6, 1957.

[2] J. Kramer, J. Magee. *Self-Managed Systems: an Architectural Challenge.* FOSE, IEEE, 2007.

[3] I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli. *Model Evolution by Run-Time Parameter Adaptation.* ICSE, IEEE, 2009.

[4] N. Bencomo, J. Whittle,P. Sawyer,A. Finkelstein,E. Letier. *Requirements Reflection: Requirements as Runtime Entities.* Proc. 32nd International Conference on Software Engineering 2010 (ICSE 2010) New Ideas and Emerging Results (NIER) track, Cape Town, South Africa, 2010

[5] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap. Software Engineering for Self-Adaptive Systems.* pages 1-26, 2009.

[6] B.N. Taylor, C.E. Kuyatt. *Guidelines for Evaluating and Expressing the Uncertainty of NIST Measurement Results.* Gaithersburg, MD, USA: National Institute of Standards and Technology, 1994.

[7] A.J. Ramirez, A.C. Jensen, B.H.C. Cheng. *A Taxonomy of Uncertainty for Dynamically Adaptive Systems,* SEAMS, 2012.

[8] M.Glinz. *On Non-Functional Requirements.* RE, IEEE, 2007.

[9] *IEEE Standard 1061-1992 Standard for a Software Quality Metrics Methodology.* Institute of Electrical and Electronics Engineers, New York, 1992.

[10] J.O. Kephart, D.M. Chess. *The vision of autonomic computing.* Computer, vol.36, no.1, pages 41- 50, 2003.

[11] Autonomic Toolkit. IBM's Autonomic Toolkit, 2006. http://www.ibm.com/developerworks/autonomic/r3/overview.html

[12] IBM Corporation. *An architectural blueprint for autonomic computing.* White Paper, 4 edizione, 2006.

[13] M. Salehie, L. Tahvildari. *Self-Adaptive Software: Landscape and Research Challenges.* ACM, 2009.

[14] D. Weyns, R. Haesevoets, B. Van Eylen. *Endogenous versus Exogenous Self-Management.* SEAMS, ACM, 2008.

[15] C. Baier, J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[16] R. Sterrit. *Autonomic computing: the natural fusion of soft computing and hard computing.* ICSMC, IEEE, Vol. 5 pages 4754-4759, 2003.

[17] T. Vogel, H. Giese. *Adaptation and Abstract Runtime Models.* SEAMS, 2010.

[18] G.J. Kiczales, J.O. Lamping, C.V. Lopes, J.J. Hugunín, E.A. Hilsdale, C. Boyapati. *Aspect-oriented programming.* ECOOP, 1997.

[19] A. Popovici, T. Gross, G. Alonso. *Dynamic weaving for aspect-oriented programming.* AOSD, ACM, 2002.

[20] R. Hirschfeld, P. Costanza, O. Nierstrasz. *Context-oriented programming.* Journal of Object Technology, 7(3), 2008.

[21] L. Baresi, S. Guinea, G. Tamburrelli. *Towards Decentralized Self-adaptive Component-based Systems.* SEAMS, 2008.

[22] K.N. Biyani, S.S. Kulkarni. *Mixed-Mode Adaptation in Distributed Systems: A Case Study.* SEAMS, 2007.

[23] R.R. Aschoff, A. Zisman.;*Proactive adaptation of service composition.* SEAMS, IEEE, 2012.

[24] A. Metzger, O. Sammodi, K. Pohl, M. Rzepka. *Towards pro-active adaptation with confidence: augmenting service monitoring with online testing.* SEAMS, 2010.

[25] R. Murch. *Autonomic Computing.* Prentice Hall, 2004.

[26] T. Grandison, M. Sloman. *Specifying and analysing trust for internet applications.* I3E, pages 145-157, 2002.

[27] C.H. Yew, H. Lutfiyya. *A Middleware and Algorithms for Trust Calculation from Multiple Evidence Sources* SEAMS, 2012.

[28] S. Gallotti, C. Ghezzi, R. Mirandola and G. Tamburelli. Quality prediction of service compositions through probabilistic model checking. *Quality of Software Architectures. Models and Architectures*, 2008.

[29] G. Kotonya, I. Sommerville. *Requirements Engineering: Processes and Techniques.* Chichester, UK: John Wiley & Sons, 1998.

[30] N.A. Qureshi, A. Perini. *Engineering Adaptive Requirements.* SEAMS, IEEE, 2009.

[31] A.C. Jensen, B.H. Cheng, H.J. Goldsby, E.C. Nelson. *A toolchain for the detection of structural and behavioral latent system properties.*, MoDELS, pages 683-698, 2011.

[32] A.J. Ramirez, A.C. Jensen, B.H. Cheng, D.B. Knoester. *Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems.* ASE, 2011.

[33] A. van Lamsweerde, E. Letier. *Handling obstacles in goal-oriented requirements engineering.* IEEE, pages 978-1005, 2000.

[34] E. Letier, A. van Lamsweerde. *Reasoning about partial goal satisfaction for requirements and desing engineering.* ACM, pages 53-62, 2004.

[35] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos. *Non-Functional Requirements in Software Engineering.* Springer, 1999, vol. 5.

[36] K. Welsh, P. Sawyer, N. Bencomo. *Towards Requirements Aware Systems: Run-time Resolution of Design-time Assumptions.* ASE, IEEE, 2011.

[37] L. Baresi, L. Pasquale, P. Spoletini. *Fuzzy Goals for Requirements-driven Adaptation.* RE, IEEE, 2010.

[38] J. Whittle, P. Sawyer, N. Bencomo, B.H.C. Cheng, J. Bruel. *RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems.* RE, IEEE, 2009.

[39] V. E. Silva Souza, J. Mylopoulos. *From Awareness Requirements to Adaptive Systems: a Control-Theoretic Approach.* RE@RunTime, IEEE, 2011.

[40] A. vanLamsweerde. *RequirementsEngineering:FromSystem Goals to UML Models to Software Specifications.* John Wiley, 2009.

[41] V.E. Silva Souza, A. Lapouchnian, J. Mylopoulos. *(Requirement) Evolution Requirements for Adaptive Systems.* SEAMS, IEEE, 2012.

[42] D. Sykes, W. Heaven, J. Magee, J. Kramer. *From Goals To Components: A Combined Approach to Self-Management.* SEAMS, IEEE, 2008.

[43] H. Nakagawa, A. Ohsuga, S. Honiden. *gocc: a configuration compiler for self-adaptive systems using goal-oriented requirements description.* SEAMS, IEEE, 2011.

[44] C. Ghezzi, J. Greenyer, V. Panzica La Manna. *Synthesizing dynamically updating controllers from changes in scenario-based specifications.* SEAMS, IEEE, 2012.

[45] J. Whittle, W. Simm, M. Ferrario. *On the role of the user in monitoring the environment in self-adaptive systems: a position paper.*

[46] H.J. Goldsby, B.H.C. Cheng. *Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty.* MoDELS, 2008.

[47] D. Weyns, S. Malek, J. Andersson. *FORMS: a FOrmal Reference Model for Self-adaptation.* ICAC, 2010.

[48] M. Famelis, S. Ben-David, M. Chechik, R. Salay. *Partial models: A position paper.* MoDeVVa, ACM, 2011.

[49] C. Ofria, C.O. Wilke. *Avida: A software platform for research in computational evolutionary biology.* Journal of Artificial Life, ISAL, 2004.

[50] Y. Brun, G.M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Muller, M. Pezze, M. Shaw. *Engineering Self-Adaptive Systems through Feedback Loops.*

[51] N. Medvidovic, P. Oreizy, J.E. Robbins, R.N. Taylor. *Using Object-Oriented Typing to Support Architectural Design in the C2 Style.* SIGSOFT, ACM, 1996.

[52] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, P. Steenkiste. *Rainbow: Architecture- Based Self-Adaptation with Reusable Infrastructure.* Computer, vol. 37, no. 10, pages 46-54, 2004.

[53] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professionals, 195.

[54] A.J. Ramirez, B.H.C Cheng. *Design Patterns for Developing Dynamically Adaptive Systems.* SEAMS, 2010.

[55] S.-W. Cheng, D. Garlan, B. Schmerl. *Architecture-based self-adaptation in presence of multiple objectives.* SEAMS, 2006.

[56] F.Fock. *The SNMP API for Java.* http://www.snmp4j.org/.

[57] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

[58] D. Dawson, R. Desmarais, H.M. Kienle, H.A. Muller. *Monitoring in adaptive systems using reflection.* SEAMS, 2008.

[59] D. Heimbigner, A.Wolf. *Definition, deployment and use of gauges to manage reconfigurable component-based systems.* Technical Report A082924, University of Colorado, 2004.

[60] A. Zeidler, L. Fiege. *Mobility support with REBECA.* ICDCS, IEEE, 2003.

[61] G. Kaiser, P. Gross, G. Kc, J. Parekh. *An approach to autonomizing legacy systems.* Proceedings of the first Workshop on Self-Healing, Adaptive, and Self-MANaged Systems, 2002.

[62] M. Hans. *The control architecture of care-o-bot ii.* E. Prassler et al (Eds.): Advances in Human-Robot Interaction, pages 321-330, 2004.

[63] A. Lau. *Design patterns for software health monitoring.* ICECCS, pages 467-476, 2005.

[64] M. Mikalsen, N. Paspallis, J. Floch, E. Stav, G. A. Papadopoulos, and A. Chimaris. *Distributed context management in a mobility and adaptation enabling middleware (madam).* SAC, ACM, 2006.

[65] H. Gomaa and M. Hussein. *Software reconfiguration patterns for dynamic evolution of software architectures.* WICSA, 2004.

[66] J. Kramer and J. Magee. *The evolving philosophers problem: Dynamic change management.* IEEE Transactions on Software Engineering, 16(11):1293-1306, 1990.

[67] T. Kindberg. *Reconfiguring client-server systems.* CDS, 1993.

[68] M. Luckey, B. Nagel, C. Gerth, G. Engels. *Adapt Cases: Extending Use Cases for Adaptive Systems.* SEAMS, ACM, 2011.

[69] P. Kaminski, H. Muller, M. Litoiu. *A Design for Adaptive Web Service Evolution.* SEAMS, ACM, 2006.

[70] L. Cavallaro, E. Di Nitto. *An Approach to Adapt Service Requests to Actual Service Interfaces.* SEAMS, ACM, 2008.

[71] M. Hirsch, S. Henkler, H. Giese. *Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML.* SEAMS, ACM, 2008.

[72] K.S. May Chan, J. Bishop. *The Design of a Self-healing Composition Cycle for Web Services.* SEAMS, IEEE, 2009.

[73] T. Haupt. *Towards Mediation-based Self-healing of Data-driven Business Processes.* SEAMS, IEEE, 2012.

[74] H. Gomaa, K. Hashimoto. *Dynamic Self-Adaptation for Distributed Service-Oriented Transactions.* SEAMS, IEEE, 2012.

[75] M. Morandini, L. Pensierini, A. Perini. *Towards Goal-Oriented Development of Self-Adaptive Systems.* SEAMS, ACM, 2008.

[76] R. Asadollahi, M. Salehie, L. Tahvildari. *StarMX: A Framework for Developing Self-Managing Java-based Systems.* SEAMS, IEEE, 2009.

[77] N. Brake, J.R. Cordy, E. Dancy, M. LItoiu, V. Popescu. *Automating Discovery of Software Tuning Parameters.* SEAMS, ACM, 2008.

[78] H. Ghanbari, M. Litoiu. *Identifying Implicitly Declared Self-Tuning Behavior through Dynamic Analysis.* SEAMS, IEEE, 2009.

[79] L. Cavallaro, E. Di Nitto, P. Pelliccione, M. Pradella, M. Tivoli. *Synthesizing adapters for conversational web-services from their WSDL interface.* SEAMS, ACM, 2010.

[80] C. Ke. *Optimizing a Rule Engine using Adaptive Programming Techniques.* SEAMS, ACM, 2006.

[81] A. Janik, K. Zielinski. *Transparent Resource Management and Self-Adaptability Using Multitasking Virtual Machine RM API.* SEAMS, ACM, 2006.

[82] Berman, Fran, Geoffrey Fox, and Anthony JG Hey, eds. *Grid computing: making the global infrastructure a reality.* Vol. 2. Wiley, 2003.

[83] Y. Brun, N. Medvidovic. *An architectural style for solving computationally intensive problems on large networks.* SEAMS, IEEE, 2007.

[84] E. Winfree, R. Bekbolatov. *Proofreading tile sets: Error correction for algorithmic self-assembly.* DNA computing (2004): 1980-1981.

[85] R. Desmarais, H. Muller. *A Proposal for an Autonomic Grid Management System.* SEAMS, IEEE, 2007.

[86] A. Gambi, M. Pezzè, M. Young. *SLA Protection Models for Virtualized Data Centers.* SEAMS, IEEE, 2009.

[87] N. Huber, F. Brosig, S. Kounev. *Model-based Self-Adaptive Resource Allocation in Virtualized Environments.* SEAMS, ACM, 2011.

[88] S. Balasubramanian, R. Desmarais, H. Muller, U. Stege, S. Venkatesh. *Characterizing problems for realizing policies in self-adaptive and self-managing systems.* SEAMS, ACM, 2011.

[89] F. Fleurey, B. Morin, A. Solberg. *A model-driven approach to develop adaptive firmwares.* SEAMS, ACM, 2011.

[90] S. van de Burg, E. Dolstra. *Disnix: A toolset for distributed deployment.* WASDeTT, 2010.

[91] S. van de Burg, E. Dolstra. *A Self-Adaptive Deployment Framework for Service-Oriented Systems.* SEAMS, ACM, 2011.

[92] R.W. Moore, B.R. Childers. *Inflation and Deflation of Self-Adaptive Applications.* SEAMS, ACM, 2011.

[93] D.E. Perry, A.L. Wolf. *Foundations for the study of software architecture.* SIGSOFT Softw. Eng. Notes, ACM, 1992.

[94] C. Zhong, S.A. DeLoach. *Runtime Models for Automatic Reorganization of Multi-Robot Systems.* ICSE, ACM, 2011.

[95] H. Gomaa, M. Hussein. *Model-based Software Design and Adaptation.* SEAMS, IEEE, 2007.

[96] C. Barna, M. Shtern, M. Smit, V. Tzerpos, M. Litoiu. *Model-Based Adaptive DoS Attack Mitigation.* SEAMS, IEEE, 2012.

[97] B. Solomon, D. Ionescu, M. Litoiu, G. Iszlai. *Autonomic Computing Control of Composed Web Services.* SEAMS, ACM, 2010.

[98] D. Weyns, S. Malek, J. Andersson. *On Decentralized Self-Adaptation: Lessons from the Trenches and Challenges for the Future.* SEAMS, ACM, 2010.

[99] M. Hou, X. Liu, H. Liu. *Fifi: An Architecture to Realize Self-evolving of Java Program.* SEAMS, ACM, 2006.

[100] D. Weyns, T. Holvoet. *An Architectural Strategy for Self-Adapting Systems.* SEAMS, 2007.

[101] F.J. Barros. *Representing Hierarchical Mobility in Software Architectures.* SEAMS, IEEE, 2007.

[102] OASIS *An introduction to WSDM*, 2006.

[103] P. Martin, W. Powley, K. Wilson, W.Tian, T. Xu, J. Zebedee. *The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services.* SEAMS, IEEE, 2007.

[104] P. Furtado, C. Santos. *Extensible Contract Broker for Performance Differentiation.* SEAMS, IEEE, 2007.

[105] N. Stojnić, H. Schuldt. *OSIRIS-SR: A Safety Ring for Self-Healing Distributed Composite Service Execution.* SEAMS, IEEE, 2012.

[106] N. Mahadevan, A. Dubey, G. Karsai. *Application of Software Health Management Techniques.* SEAMS, ACM, 2011.

[107] D. Kim, S. Park. *Reinforcement Learning-Based Dynamic Adaptation Planning Method for Architecture-based Self-Managed Software.* SEAMS, IEEE, 2009.

[108] W. Richert, B. Kleinjohann. *Adaptivity at every layer: a modular approach for evolving societies of learning autonomous systems.* SEAMS, ACM, 2008.

[109] C. Raibulet, F. Arcelli, S. Mussino, M. Riva, F. Tisato, L. Ubezio. *Components in an Adaptive and QoS-based Architecture.* SEAMS, ACM, 2006.

[110] K.Geihs, R. Reichle, M.U. Khan, A. Solberg. *Model-Driven Development of Self-Adaptive Applications for Mobile Devices.* SEAMS, ACM, 2006.

[111] M. Derakhshanmanesh, M.Amoui, G.O'Grady, J.Ebert, L. Tahvildari.*GRAF: graph-based runtime adaptation framework.* SEAMS, ACM, 2011.

[112] T. Vogel, H. Giese. *A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels.* SEAMS, IEEE, 2012.

[113] F. Mancinelli, P. Inverardi. *A Resource Model for Adaptable Applications.* SEAMS, ACM, 2006.

[114] P. Martin, W. Powley, I. Abdallah, J. Li, A. Brown, K. Wilson, C. Craddock, *A model for dynamic and adaptable services management.* SEAMS, IEEE, 2009.

[115] K.S. Narendra, J. Balakrishnan, *Adaptive control using multiple models* Automatic Control, IEEE Transactions in , vol.42, no.2, pp.171-187, 1997.

[116] T. Patikirikorala, A. Colman, J. Han, L. Wang. *A multi-model framework to implement self-managing control systems for QoS management.* SEAMS, ACM, 2011.

[117] Y. Liu, I. Gorton. *Implementing Adaptive Performance Management in Server Applications.* SEAMS, IEEE, 2007.

[118] R. Haesevoets, D. Weyns, T. Holvoet, W. Joosen. *A Formal Model for Self-Adaptive and Self-Healing Organizations.* SEAMS, IEEE, 2009.

[119] H.J. Goldsby, D.B. Knoester, B.H.C. Cheng, P.K. McKinley, C.A. Ofria. *Digitally Evolving Models for Dynamically Adaptive Systems.* SEAMS, IEEE, 2007.

[120] I. Schaefer, A. Poetzsch-Heffter. *Slicing for Model Reduction in Adaptive Embedded Systems Development.* SEAMS, ACM, 2008.

[121] D. Weyns. *An architecture-centric approach for software engineering with situated multi-agent systems.* The Knowledge Engineering Review 22.4, pages 405-413, 2007.

[122] H. Klus, A. Rausch. *A General Architecture for Self-Adaptive AmI Components Applied in Speech Recognition.* SEAMS, ACM, 2006.

[123] J.C. Georgas, R.N. Taylor. *Policy-Based Self-Adaptive Architectures: a Feasibility Study in the Robotics Domain.* SEAMS, ACM, 2008.

[124] E. Vassev, J. Paquet. *Towards an Autonomic Element Architecture for ASSL.* SEAMS, IEEE, 2007.

[125] F. Irmert, T. Fischer, K. Meyer-Wegener. *Runtime Adaptation in a Service-Oriented Component Model.* SEAMS, ACM, 2008.

[126] F. Irmert, M. Meyerhöfer, M. Weiten. *Towards Runtime Adaptation in a SOA Environment.* Preprint no. 6/2007 of University of Magdeburg, 2007.

[127] C. Ghezzi, M. Pradella, G. Salvaneschi. *Programming Language Support to Context-Aware Adaptation: a Case-Study with Erlang.* SEAMS, ACM, 2010.

[128] C. Ghezzi, M. Pradella, G. Salvaneschi. *An Evaluation of the Adaptation Capabilities in Programming Languages.* SEAMS, ACM, 2011.

[129] J. Malenfant, M. Jacques, F.N. Demers. *A tutorial on behavioral reflection and its implementation.* 1996.

[130] J. Andersson, R. de Lemos, S. Malek, D. Weyns. *Reflecting on Self-Adaptive Software Systems.* SEAMS, 2009.

[131] S.J. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach.* (2nd ed.) Prentice Hall, 2003.

[132] C.J.C.H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University. 1989.

[133] I. Dusparic, V. Cahill. *Research Issues in Multiple Policy Optimization Using Collaborative Reinforcement Learning.* SEAMS, IEEE, 2007.

[134] A.S. Rao, M.P. Georgeff. *BDI agents: From theory to practice.* ICMAS, 1995.

[135] E. di Nitto, D.J. Dubois, R. Mirandola. *On Exploiting Decentralized Bio-inspired Self-organization Algorithms to Develop Real Systems.* SEAMS, 2009.

[136] R. de Lemos. *Architectural Reconfiguration using Coordinated Atomic Actions.* SEAMS, ACM, 2006.

[137] C.A.R Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[138] B. Bartels, M. Kleine. *A CSP-based framework for the Specification, Verification, and Implementation of Adaptive Systems.* SEAMS, ACM, 2011.

[139] D. Sykes, J. Magee, J. Kramer. *FlashMob: Distributed Adaptive Self-Assembly.* SEAMS, ACM, 2011.

[140] J. Cámara, R. de Lemos. *Evaluation of Resilience in Self-Adaptive Systems Using Probabilistic Model-Checking.* SEAMS, 2012.

[141] M. Zeller, C. Prehofer. *Timing Constraints for Runtime Adaptation in Real-Time, Networked Embedded Systems.* SEAMS, IEEE, 2012.

[142] C.E. da Silva, R. de Lemos. *Using Dynamic Workflows for Coordinating Self-adaptation of Software Systems.* SEAMS, IEEE, 2009.

[143] S. Fritsch, A. Senart, D. C. Schmidt, S. Clarke. *Scheduling time-bounded dynamic software adaptation.* SEAMS, ACM, 2008.

[144] J.-P. Briot, Z. Guessoum, S. Aknine, A. L. Almeida, J. Malenfant, O. Marin, P. Sens, N. Faci, M. Gatti, and C. Lucena. *Experience and prospects for various control strategies for self-replicating multi-agent systems.* SEAMS, ACM, 2006.

[145] M.P. Papazoglou. *Service-Oriented Computing: Concepts, Characteristics and Directions.* WISE, IEEE, 2003.

[146] L. Baresi, L. Pasquale. *Adaptive Goals for Self-Adaptive Service Composition.* SEAMS, IEEE, 2010.

[147] A. Carzaniga, A. Gorla, M. Pezzè. *Self-Healing by Means of Automatic Workarounds.* SEAMS, ACM, 2008.

[148] A. Kattepur, S. Sen, B. Baudry, A. Benveniste, C. Jard. *Pairwise testing of dynamic composite services.* SEAMS, ACM, 2011.

[149] C.E. da Silva, R. de Lemos. *Dynamic Plans for Integration Testing of Self-Adaptive Software Systems.* SEAMS, ACM, 2011.

[150] S. Dustdar, C. Dorn, F. Li, L. Baresi, G. Cabri, C. Pautasso, F. Zambonelli. *A roadmap towards sustainable self-aware service systems.* SEAMS, ACM, 2010.

[151] J. Cámara, C. Canal, G. Salaün. *Behavioural Self-Adaptation of Services in Ubiquitous Computing Environments.* SEAMS, IEEE, 2009.

[152] M. Garcia-Valls, P. Basanta-Val, I. Estévez-Ayres. *Supporting service composition and real-time execution throught characterization of QoS properties.* SEAMS, ACM, 2011.

[153] D. Kim, S. Park, Y. Jin, H. Chang, Y.-S. Park, I.-Y. Ko, K. Lee, J. Lee, Y.-C. Park, S. Lee. *SHAGE: a framework for self-managed robot software.* SEAMS, ACM, 2006.

[154] C. Peper, D. Schneider. *Component Engineering for Adaptive Ad-hoc Systems.* SEAMS, ACM, 2008.

[155] K. Schneider, T. Schuele, M. Trapp. *Verifying the Adaptation Behavior of Embedded Systems.* SEAMS, ACM 2006.

[156] S. Guinea, P. Saeedi. *Coordination of Distributed Systems through Self-Organizing Group Topologies.* SEAMS, IEEE, 2012.

[157] L. Pasquale, M. Salehie, R. Ali, I. Omoronyia, B. Nuseibeh. *On the role of primary and secondary assets in adaptive security: An application in smart grids.* SEAMS, IEEE, 2012.

[158] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker. *Prism: a tool for automatic verification of probabilistic systems.* TACAS, 2006.

[159] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Modeling Language.* University Video Communications, 1996.