Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura · Campus di Cesena

Corso di Laurea Magistrale in
Ingegneria e Scienze Informatiche

INTEGRAZIONE DI PIATTAFORME D'ESECUZIONE E SIMULAZIONE IN UNA TOOLCHAIN SCALA PER AGGREGATE PROGRAMMING

Tesi di Laurea Magistrale in
Ingegneria dei Sistemi Software Adattativi Complessi

Relatore:
Prof. Ing.
MIRKO VIROLI

Correlatore:
Dott. Ing.
DANILO PIANINI

Presentata da:
SIMONE COSTANZI

III Appello - III Sessione
Anno Accademico 2014/2015

Questa pagina è lasciata intenzionalmente bianca.

Keywords:

Aggregate Programming

Pervasive Computing

Scafi

Alchemist

Questa pagina è lasciata intenzionalmente bianca.

Ai miei genitori,
Paolo e Luciana

Questa pagina è lasciata intenzionalmente bianca.

Indice

Capitolo 1

Introduzione

1.1 Scenario

Coordinare il comportamento di un elevato numero di dispositivi o servizi è un problema con importanza sempre maggiore. La densità di dispositivi mobili o "embedded" nell'ambiente in cui viviamo è in continuo aumento. Smartphone, orologi, vestiti e accessori, veicoli, segnali, edifici, display: tutto questo e molto altro sono spesso in grado di percepire, computare e comunicare. Questo scenario prende molteplici nomi, come *Pervasive Computing*, *Smart City*, e *Internet of Things* [1] [8]. Ad ogni modo, ognuno di questi è caratterizzato da computazioni che sono (i) context-dependent (molte interazioni sono opportunistiche e coinvolgono i dispositivi a livello della loro prossimità fisica, quindi le computazioni si basano principalmente su aspetti e informazioni dell'ambiente in cui lo smartphone è situato); (ii) self-adaptive e self- organizing (le computazioni devono ripristinarsi spontaneamente da possibili fallimenti e saper reagire a possibili imprevisti). Allo stesso tempo anche le reti aziendali stanno aumentando di dimensione e importanza, sia nella grande che piccola impresa. In entrambi questi casi e in altre aree che si trovano ad affrontare problematiche simili (p.es. reti di sensori su larga scala), c'è una crescente consapevolezza della necessità di un nuovo paradigma di programmazione, che permetta una programmazione di tali sistemi ad un livello aggregato. In altre parole, un linguaggio e API che permettano di programmare un insieme di dispositivi distribuiti

in termini del loro comportamento collettivo, astraendo da molti dettagli relativi all'effettiva implementazione di quest'ultimo. Questo paradigma di programmazione prende il nome di *Aggregate Programming* [3] e un possibile approccio basato su quest'ultimo si identifica in un modello eretto sul concetto di "computational field". Formalizzato come *Field Calculus*, questo linguaggio universale, sembra fornire un fondamento teorico sul quale poter costruire tutta una serie di livelli costituenti la piattaforma dell'Aggregate Programmin, fino ad arrivare ad un livello che possa offrire al programmatore delle API utili ad uno sviluppo del sistema più agile, sicuro ed elastico. Un importante linguaggio basato su questo paradigma di programmazione è costituito da **Protelis** [15] che può essere considerato come un'evoluzione del linguaggio **Proto** [2] e che rappresenta un implementazione del framework del Field Calculus. Oppure un altro importante esempio, costituente l'entità protagonista di questo elaborato, è la piattaforma *scafi* che offre tutta una serie di primitive scala per programmare il comportamento di un sistema in modo aggregato. Dopo questa breve introduzione all'aggregate programming, in quanto ambito principale in cui si calano i lavori svolti, si riporta nella prossima sezione l'obiettivo principale di questo elaborato unitamente ai scenari applicativi che emergerebbero col suo conseguimento.

1.2 Obiettivi

L'obiettivo di questo lavoro è riuscire a potenziare la toolchain per aggregate programming, attraverso l'integrazione del framework scafi con la piattaforma di simulazione Alchemist [14] e con una piattaforma di creazione ed esecuzione di sistemi in ambito *Pervasive Computing* [13] e *Spatial Computing* [9] [4]. L'unione del risultato che si otterrebbe effettuando queste integrazioni con gli sviluppi esistenti porterebbe ad un'estensione della catena di software già presente nell'ambito dell'aggregate programming, fino a raggiungere la toolchain rappresentata in figura 1.1.

In riferimento al lavoro oggetto di questa tesi, le integrazioni fra le piattaforme sopracitate darebbero vita ad una catena di software interagenti tra loro particolarmente importante e completa (evidenziata in figura col colore blu); si riporta di seguito un possibile scenario del suo utilizzo. In

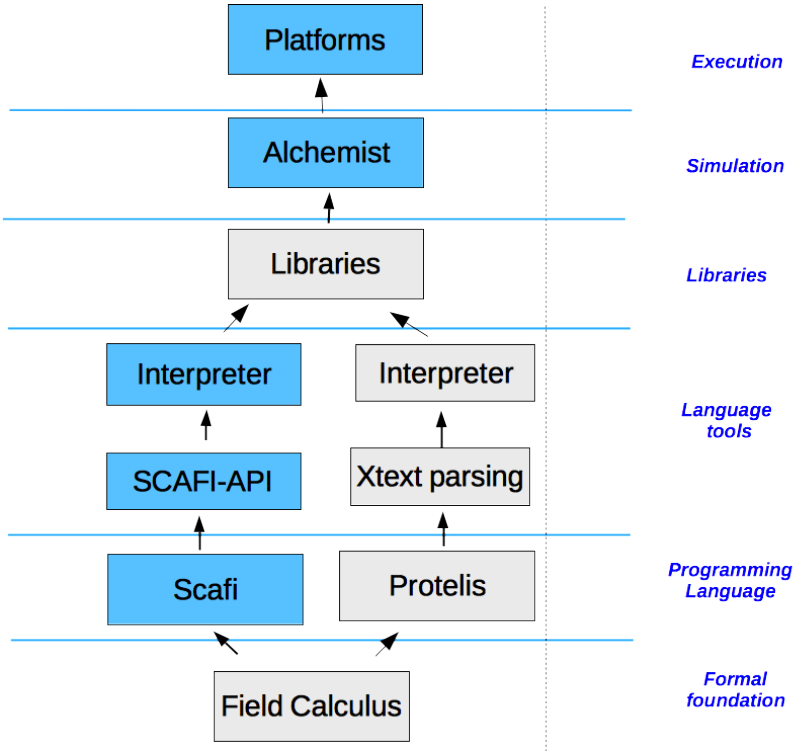

Figura 1.1: Toolchain Aggregate Programming

una prima fase grazie all'integrazione con Alchemist è possibile avvalersi delle primitive scala di scafi per programmare sistemi virtuali da eseguire in ambiente simulato; in tal modo è possibile raffinare sempre più il programma in termini aggregati fino a quando la simulazione riporta il risultato atteso per diverse configurazioni iniziali e diverse dinamiche del sistema. Una volta soddisfatti del comportamento complessivo di quest'ultimo è possibile, grazie all'integrazione di scafi con la piattaforma di esecuzione, portare il sistema virtuale in un ambiente in grado di elaborare il suo comportamento descritto in termini di primitive scafi ed eseguirlo in un contesto non più simulato ma reale. In altre parole, con questa toolchain si può usufruire di un ambiente simulato costruendo il sistema sempre tramite il framework scafi e solo dopo averne accurato il corretto funzionamento è possibile

eseguire tale sistema virtuale in un contesto reale senza dover riscrivere il suo comportamento.

La restante parte della tesi è composta dal secondo capitolo in cui si fornisce una panoramica sugli ambiti e le piattaforme oggetto degli elaborati svolti, due capitoli in cui si riporta una descrizione dettagliata di tutte le fasi del processo di sviluppo relativamente alle due integrazione effettuate, un capitolo che descrive l'esperimento svolto per l'esecuzione del framework scafi su piattaforma android, riportando nello specifico tutti i passaggi tecnici eseguiti, e infine un capitolo conclusivo nel quale sono citati anche alcuni possibili sviluppi futuri.

Capitolo 2

Aggregate Programming

2.1 Panoramica

Aggregate Programming [3] è un approccio emergente che si focalizza sulla semplificazione della progettazione, creazione e mantenimento di complessi sistemi software distribuiti grazie all'utilizzo di una unità computazionale di base non più caratterizzata dal singolo dispositivo (programmazione device-centric), ma da un insieme di questi ultimi. In tal modo il programmatore può astrarre dai singoli dispositivi e relativi dettagli, come interazione, comportamento e posizione, evitando così possibili complicazioni ad alcuni aspetti del sistema distribuito: efficacia e affidabilità delle comunicazioni, coordinamento a fronte di cambiamenti e guasti, e composizione di comportamenti attraverso differenti dispositivi e regioni. I dettagli alla base di una programmazione device-centric, che richiedono l'attenzione dello sviluppatore, minacciano inoltre la modularità dell'applicazione, la riusabilità, e la facilità di progettazione, specie dove la complessità è elevata.

L'approccio dell'Aggregate Programming è basato su 3 principi, riportati nell'articolo *Aggregate Programming for the Internet of Things* [3]:

- *the "machine" being programmed is a region of the computational environment whose specic details are abstracted away—perhaps even to a pure spatial continuum;*

- *the program is specified as manipulation of data constructs with spatial and temporal extent across that region;*
- *these manipulations are actually executed by the individual devices in the region, using resilient coordination mechanisms and proximity-based interactions.*

In tal modo, questo paradigma nasconde i complicati meccanismi di coordinazione, facilitando così la progettazione e quindi favorendo la costruzione di sistemi più modulari: questo approccio risulta molto importante nell'emergente scenario dell'IoT, caratterizzato da un continuo aumento di applicazioni per sistemi di larga scala, che sfruttano interazioni opportunistiche e di prossimità.

2.1.1 Architettura a livelli

L'Aggregate Programming nasconde la complessità di interazione che caratterizza fortemente sistemi distribuiti su larga scala, attraverso un'organizzazione costituita da una serie di livelli di astrazione (figura 2.1).

Il **Field Calculus** costituisce la base di questa architettura: un insieme di costrutti atti a modellare computazione ed interazione tra un elevato numero di dispositivi distribuiti. Il Field Calculus costituisce le fondamenta teoriche e metodologiche dell'Aggregate Programming ed inoltre essendo il livello costruito sopra le risorse software e hardware del dispositivo, costituisce il punto dove l'Aggragate Programming si interfaccia con quest'ultimo e a servizi software non aggregati. L'astrazione fondamentale del Field Calculus è il *field*, concetto ispirato ad aspetti fisici che mappa ogni dispositivo ad un qualche valore locale. Questa astrazione è utilizzata per modellare ogni aspetto della computazione distribuita, come input dei sensori, struttura della rete, interazioni e output per gli attuatori.

La costruzione e manipolazione del field, avviene attraverso cinque costrutti di programmazione: (i) operatori per la definizione e valutazione di funzioni, (ii) operatori "built-in" per computazioni locali come lettura del valore dei sensori, (iii) costrutto per computazioni relative all'evoluzione temporale, (iv) costrutto per creare un field costituente informazioni relative ai dispositivi del proprio vicinato, (v) operatore atto a selezionare

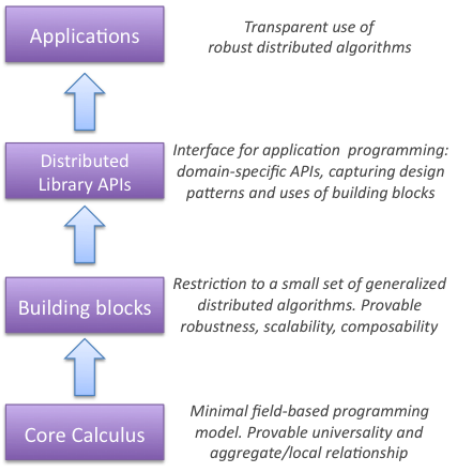

Figura 2.1: Approccio a livelli per lo sviluppo di sistemi distribuiti attraverso aggregate programming (figura ripresa da [3])

quale computazione effettuare in varie regioni dello spazio e tempo. Questi costrutti sono combinati insieme per formare un programma, dove la semantica è definita in termini di una sequenza di round di valutazione relativamente ad una rete di dispositivi. Tuttavia non è richiesta la sincronizzazione, quindi ogni dispositivo può valutare indipendentemente il suo round computazionale.

Nonostante questo primo livello rappresenti il cuore di calcolo, è solo il primo passo verso la costruzione di applicazioni distribuite basate su Aggregate Programming. Utilizzando le astrazione fornite da questo strato, si prevede la costruzione di un insieme di operatori caratterizzanti il secondo livello di astrazione: **Building blocks for resilient coordination**. Più precisamente, si tratta di un insieme di operatori di più alto livello, che forniscono un ambiente di programmazione più espressivo ed inoltre apportano al sistema da sviluppare una maggiore scalabilità, modularità, ed elasticità nella coordinazione.

Per incontrare meglio le esigenze del programmatore, si prevede inoltre la costruzione di apposite **librerie** basate sugli operatori appena esposti.

Queste librerie formano il penultimo livello di astrazione dell'architettura in questione: insieme di API costituenti per lo sviluppatore una pratica interfaccia allo sviluppo di sistemi IoT situati.

In fine l'ultimo livello è costituito dalla costruzione del **codice applicativo**.

Questi livelli, quindi, offrono un efficiente ecosistema per lo sviluppo di software relativo a servizi distribuiti IoT, analogamente a ecosistemi per lo sviluppo in ambito web o cloud.

2.1.2 Protelis

Implementazione di Field Calculus

Dato che il Field Calculus è più un framework teorico che un linguaggio di programmazione, si è progettato il linguaggio Protelis [15] come implementazione di quest'ultimo. Tale linguaggio da un lato incorpora le principali funzionalità di computazione spaziale di Field Calculus e dall'altro porta quest'ultimo all'interno di un moderno linguaggio di programmazione. Protelis può essere considerato come un'evoluzione del linguaggio Proto [2] con i seguenti miglioramenti:

- accesso ad una ricca base di API grazie all'integrazione con java;
- è un linguaggio funzionale;
- adotta una sintassi molto familiare, in quanto molto simile a quella di Java e C, riducendo eventuali barriere alla sua adozione.

Un altro aspetto molto importante di questo linguaggio è la portabilità: sia per la sua compatibilità con una grande gamma di dispositivi e sistemi, grazie alla sua implementazione in java, che per quanto riguarda eventuali passaggi da ambienti simulati ad ambienti reali.

Esempi applicativi

Nell'articolo *Protelis: Practical Aggregate Programming* [15] questo linguaggio viene sperimentato in due contesti: il primo è caratterizzato da uno

scenario di Pervasive Computing [13] e fa uso del simulatore Alchemist per la sua esecuzione, e il secondo è relativo invece alla gestione di reti di servizi in ambito aziendale.

"Rendezvous" durante un evento di massa Un classico problema negli eventi di massa, è potersi incontrare con un compagno in un determinato punto, in quanto in queste situazioni, caratterizzate da un elevato numero di dispositivi, è molto difficile se non impossibile poter accedere a servizi esterni di cloud. Tuttavia, attraverso un programma protelis di piccole dimensioni in esecuzione su ciascun dispositivo, è possibile calcolare il cammino che i due compagni dovranno seguire per incontrarsi nel luogo stabilito.

Gestione di reti di servizi Un problema comune nel gestire complesse reti di servizi aziendali, è la presenza di molte dipendenze tra diversi server e servizi (figura 2.2).

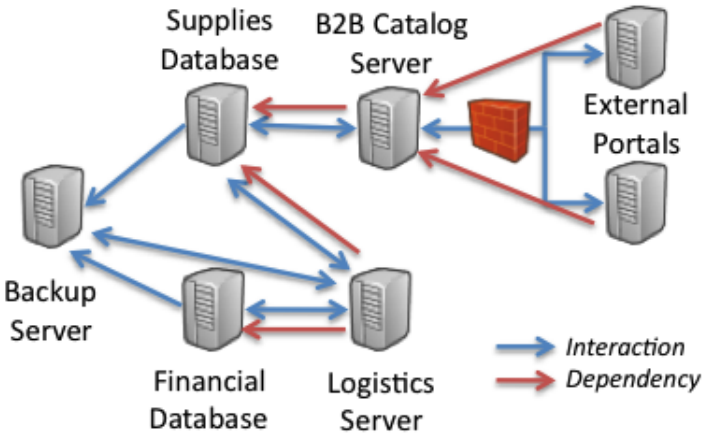

Figura 2.2: Esempio di scenario con alcune dipendenze fra servizi (figura ripresa da [15])

In particolare, in queste situazioni, qualora vi sia un problema ad un servizio, per evitare stati di inconsistenza è spesso richiesto un coordinato

arresto e riattivazione dei servizi, nell'ordine dettato dalle loro dipendenze. Protelis, in questo contesto, è stato sfruttato proprio per automatizzare la gestione di spegnimento e riaccensione. Più precisamente, si sono costruite apposite entità (daemon) costituite da un programma protelis, ognuna delle quali ha il compito di monitorare lo stato di uno specifico servizio e comunicare con altri daemon, al fine ultimo di riuscire a coordinare l'arresto e la riaccensione dei servizi in accordo con le loro dipendenze.

Questo caso d'uso di Protelis non ha solo uno scopo dimostrativo: viene infatti preso in considerazione anche nell'articolo *Distributed Recovery for Enterprise Services* [6], in cui si prevede un suo possibile utilizzo pratico nella piccola e media impresa.

2.2 Scafi

Il framework scafi è un'altra tecnologia che analogamente a protelis permette la scrittura di programmi basandosi sul paradigma di programmazione aggregata, con la differenza però che non si tratta di un vero e proprio DSL ma sono piuttosto un insieme di primitive scala che data la loro compattezza e utilità possono essere considerate come un vero e proprio linguaggio di programmazione specifico per aggregate programming. Questo framework è stato sviluppato dal professor Mirko Viroli e successivamente esteso dall'Ing. Roberto Casadei nell'ambito della sua tesi [5], dalla quale si sono reperite alcune informazioni ed immagini utilizzate poi nell'ambito di questo elaborato.

Nella figura, viene riportata l'architettura di progetto del framework scafi che mette in evidenza i suoi componenti chiave.

Il componente **Core** definisce le astrazioni base e gli elementi architetturali che saranno poi raffinati dai componenti figli. Il componente **Language** basato sulle astrazioni definite in Core, definisce i principali **Constructs** del DSL. Quest'ultimo elemento espone le primitive del *fiedl calculus* come metodi. Sopra queste primitive vi sono gli operatori **Builtins** che provvedono a rendere il linguaggio più espressivo (**RichLanguage**). Il componente **Semantics** estende la parte sintattica e strutturale di Language, raffina le astrazioni di Core e fornisce una semantica per i Constructs, inoltre il

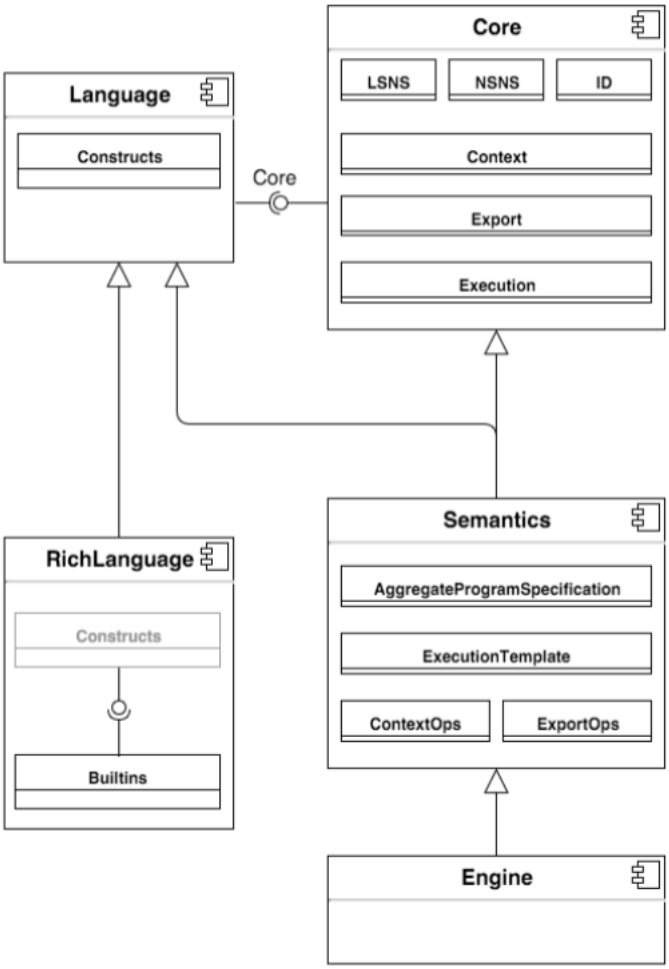

Figura 2.3: Architettura di progetto del framework scafi (figura ripresa da [5])

componente Semantics viene poi reso eseguibile dal componente **Engine**. L'implementazione di questi componenti base costituenti l'architettura è effettuata in Scala. In breve, i componenti sono rappresentati attraverso traits, classi, oggetti e così via. Questo permette di creare famiglie di tipi ricorsivi che possono essere raffinati in modo incrementale.

2.3 Piattaforma di supporto

2.3.1 Panoramica

A causa della mancanza delle comunicazioni opportunistiche ai dispositivi mobili i servizi cloud sono una valida ed efficiente soluzione per una piattaforma sulla quale eseguire applicazioni Aggregate Programming: come piattaforma, oltre a fornire i servizi di comunicazioni necessari, dovrebbe ospitare l'esecuzione di alcune computazioni locali ad ogni dispositivo, sfruttando così le maggiori risorse computazionali della piattaforma. Le applicazioni in questo contesto richiedono *location-awareness*, dal momento che le posizioni dei dispositivi sono una delle più importanti informazioni per la piattaforma: uno dei suoi principali obiettivi è infatti quello di mantenere delle strutture dati con le posizioni aggiornate di tutti i dispositivi del sistema in modo da poter comunicare a questi ultimi il loro vicinato con il quale poter interagire. Queste interazioni con i dispositivi vicini costituiscono i blocchi principali dell'Aggregate Programming: specificando un programma ad un livello aggregato, questo verrà poi "tradotto" in un programma per ciascuno dispositivo mobile che calcolano uno stato in base alle manipolazioni sui *fields* offerte da Aggregate Programming. In sintesi, ci sono diversi tipi di informazioni necessarie in questa piattaforma:

- **posizione** dei dispositivi (p.es. coordinate GPS), in modo da ricavare le relazioni di prossimità fra i dispositivi e successivamente costruire una struttura dati contenente il vicinato di ciascuno di essi; la definizione di prossimità può variare in base ad alcuni parametri (p.es. distanza fra due posizioni GPS) e può essere configurata;
- **valore dei sensori**, attraverso i quali avviene la computazione delle specifiche funzioni definite nel singolo dispositivo;
- lo **stato** dei dispositivi, rappresentante il risultato della computazione delle specifiche funzioni e dell'interazione con i dispositivi vicini.

Un semplice e comune esempio di applicazione che mostra questi concetti è la computazione della distanza di un dispositivo dalla "sorgente" (*gradiente*): in questo scenario si ha una rete di dispositivi auto-organizzanti che

computano la distanza minima dalla sorgente tramite lo scambio dell'informazione della posiziona al vicinato. Inoltre, vi sono diversi livelli con i quali i singoli dispositivi possono interagire con la piattaforma:

- mantenendo la relazione di vicinato sulla base della quale ogni singolo dispositivo effettua le sue computazioni relative alle funzioni e al calcolo del proprio stato;
- anche attraverso il calcolo dello stato dei dispositivi, questo significa che il dispositivo mobile deve solo fornire alla piattaforma la propria posizione e i valori dei sensori lasciando a quest'ultima tale computazione, il quale risultato sarà inviato appena disponibile al dispositivo.

Queste piattaforme devono inoltre fornire una buona scalabilità, in quanto sistemi di questo tipo presentano prevalentemente una natura di larga scala, e una bassa latenza nell'elaborare le informazioni pervenute dai dispositivi.

2.3.2 Piattaforma utilizzata

La piattaforma utilizzata in questo lavoro è stata realizzata dagli Ingegneri Pierluigi Montagna, Lorenzo Rocca, Matteo Bezzi nell'ambito dell'esame di Ingegneria dei Sistemi Software Adattativi e Complessi. Si riporta di seguito una breve panoramica elencando i punti fondamentali di tale architettura in modo da facilitare la comprensione del lavoro di integrazione riportato nel capitolo 3. Come si può notare dalla figura 2.4, tale piattaforma presenta esattamente le caratteristiche esposte nella sottosezione precedente, più precisamente è un framework che offre la possibilità di creare un nodo su ciascun dispositivo (sottosistema "Client") che è in grado di interagire con un server (sottosistema "SC Server"), *inviando* informazioni inerenti al proprio stato, come posizione, valore dei sensori e valori relativi ai risultati di funzioni ("sendState(NodeId, State)"), e *ricevendo* su richiesta queste informazioni relative allo stato di ciascun vicino ("fetchNeighbourhood(NodeId)").

Grazie a questo sistema, che mette a disposizione tali informazioni al singolo dispositivo, è quindi possibile effettuare una programmazione di quest'ultimo ad un livello aggregato. Dal punto di vista strutturale questa

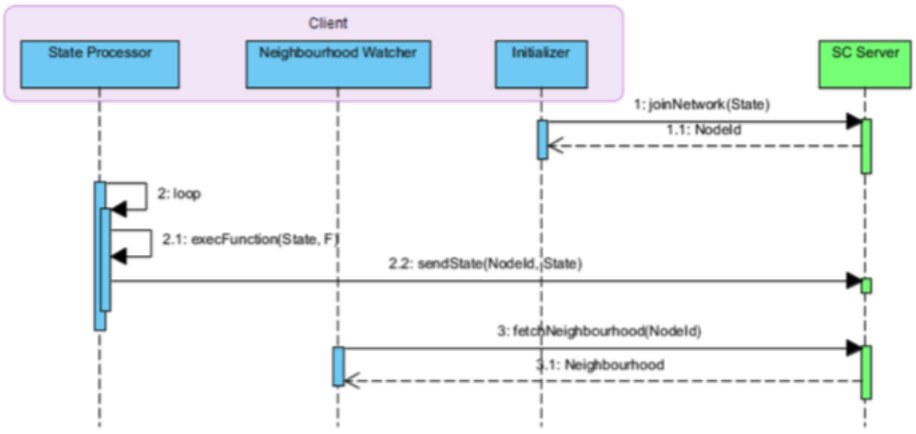

Figura 2.4: Diagramma di interazione della piattaforma

piattaforma è costituita da una parte Server e da una parte Client, esposte nelle sezioni seguenti.

Server

Questo sottosistema ha un ruolo centrale in questa piattaforma in quanto si occupa di mediare tutte le interazioni fra i vari dispositivi, più precisamente la sua funzione è quella di informare continuamente i vari dispositivi della posizione e dello stato dei loro vicini, in accordo col paradigma dell'aggregate programming. Come viene mostrato dalla figura 2.5, questo sottosistema è caratterizzato da tre parti logiche.

La **risorsa**, un'entità che modella i dati e le informazioni che possono essere acceduti anche da remoto con i metodi standard HTTP GET, POST e PUT. Alcuni esempi di risorse sono il singolo *nodo*, l'insieme dei *nodi* appartenenti alla rete oppure il *vicinato* rappresentato dall'insieme dei nodi dei propri vicini. Ciascun nodo è un tipo di dato che incapsula tutte le informazioni cruciali di ciascun dispositivo in accordo coi principi dell'aggregate programming, come la posizione, il valore dei sensori e il valore di ciascuna computazione locale.

Il **modello** che costituisce una consistente rappresentazione della rete di tutti i dispositivi storicizzata nel sistema attraverso un data-base basato

su Redis [1].

Il **connettore**, componente che abilita la comunicazione tra dispositivo e server. Più in particolare la parte di comunicazione si basa sul Framework Restlet [2] un API che aiuta la costruzione di un'applicazione web nello stile dell'architettura *ReST*

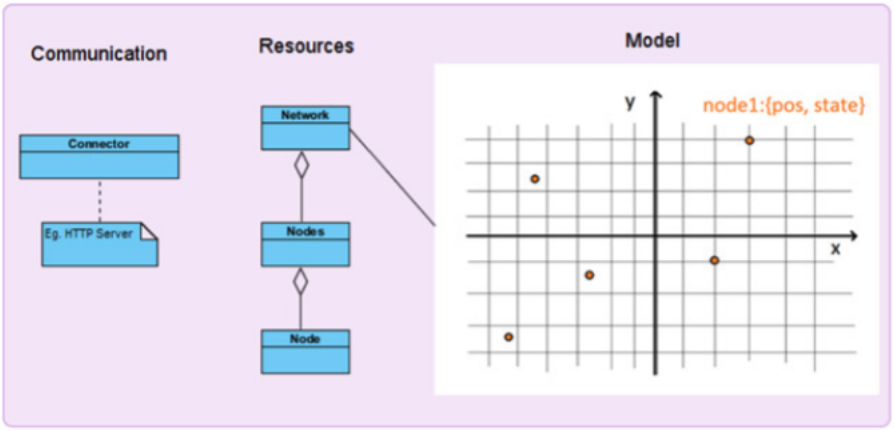

Figura 2.5: Rappresentazione concettuale del sistema che mostra i suoi principali componenti

Client

Questo sottosistema è basato su un modello dove un generico client può demandare la computazione di funzioni (in termini di aggregate programming) ad uno specifico modulo ("SC"). Questo modulo utilizza java come linguaggio di programmazione ed è caratterizzato da una specifica interfaccia che serve come entry point per interagire con questo modulo. Questo significa che una volta definita una funzione, nella forma definita dalla specifica interfaccia, l'applicazione passa questa funzione in ingresso al modulo che si occuperà della sua computazione avvalendosi delle informazioni relative al vicinato fornite dal sottosistema server. Più nel dettaglio ciascun client è

[1] http://redis.io/
[2] http://restlet.com/technical-resources/restlet-framework/guide/2.3

caratterizzato essenzialmente da un insieme di *manager* ognuno dei quali ha una specifica funzione:

- **Aggregate Facilitator**, elemento centrale per l'aggregate computing, in quanto è colui che gestisce tutti gli altri manager, avviandoli, mettendoli in pausa o stoppandoli; inoltre questo componente ha sia il ruolo di osservatore (`Observer`) dell'arrivo di nuovi risultati delle computazioni di funzioni che di Osservabile (`java.util.Observable`) in quanto deve notificare le eventuali entità interessate a questi specifici risultati;

- **Computation Manager** questo oggetto gestisce la computazione delle funzioni aggregate, calcolando un nuovo stato in accordo con lo stato precedente, il valore dei sensori e lo stato del vicinato, ed ha anche il ruolo di osservabile: ogni volta che un nuovo valore è stato computato invia una notifica ad un observer (p.es. `AggregateFacilitator`);

- **Network Manager**, colui che gestisce le parti esterne al client, in particolare l'interazione con il server, permettendo l'invio dello stato corrente e la ricezione dell'ultimo stato di tutti i vicini;

- **Sensor Manager**, analogamente ai precedenti manager questo componente gestisce tutti gli aspetti relativi ai sensori: periodicamente questo manager reperisce il valore di ciascun sensore e lo rende disponibile agli altri manager.

Un altro importante aspetto è come avviene lo scambio di informazioni fra questi manager: vi è un'entità, **Domain**, rappresentata da un plain old java object che gestisce ogni valore usato nella computazione. I manager "chiedono" a questa entità il valore corrente dei sensori, dei campi (valore delle funzioni) o dello stato del vicinato, forniti precedentemente al suo interno dallo specifico manager.

Importante citare inoltre che ciascun manager è sia un fornitore di servizi che un oggetto con un proprio flusso di controllo:

- il primo è un plain old Java object che espone delle interfacce ad esempio per l'aggiunta di funzioni da computare o sensori, ma anche per la gestione dei valori e per l'interazione col Domain;

- dall'altra parte invece sono caratterizzati da un proprio flusso di controllo per la computazione dei specifici compiti di loro competenza, p.es. la computazione di funzioni.

2.4 Simulazione

2.4.1 Importanza della simulazione

L'ambito di applicazione del paradigma dell'aggregate programming è principalmente costituito da complessi sistemi composti da molteplici parti che interagiscono tra loro fino a generare un comportamento emergente spesso imprevedibile a priori, il principale esempio di natura artificiale sono i sistemi di *pervasive computing* [13] costruiti sopra un insieme di dispositivi computazionali mobili in continuo aumento. Sistemi con queste caratteristiche sono molto diffusi anche in natura come ad esempio in ambito chimico, biologico e sociale. Entrambi i sistemi di questo tipo, sia naturale che artificiale, hanno un insieme di caratteristiche comuni:

- **situatedness** - sono immersi in un determinato spazio e dovrebbero quindi essere in grado di interagire con esso e adattare il loro comportamento di conseguenza;

- **adaptivity** - dovrebbero essere caratterizzati da proprietà di adattamento e gestione autonoma arrivando così a sopravvivere senza alcun intervento esterno o un supervisore globale;

- **self-organisation** - modelli spaziali e temporali di comportamento dovrebbero emergere da interazioni locali e senza un autorità centrale che impone piani predefiniti.

Per la realizzazione di questi sistemi è molto importante, oltre all'utilizzo di un adeguato paradigma di programmazione come l'aggregate programming e un adeguato middleware [16], avvalersi di *validi modelli*, in particolare modelli ispirati alla natura [18]: nei sistemi naturali tutte le attività dei componenti sono situate e guidate solamente da interazioni locali, e nonostante la semplicità delle leggi che regolano queste interazioni, si assiste

grazie all'auto-organizzazione all'emergere di un importante comportamento dalle dinamiche complesse.

Inoltre un'altra importante e necessaria fase per una buon sviluppo di tali sistemi è la **simulazione**, in quanto costituisce un modo di analizzare il comportamento del sistema prima della sua distribuzione; questo è un passaggio quasi obbligatorio qualora il sistema presenti un comportamento emergente.

Il framework di simulazione utilizzato in questo lavoro si chiama **Alchemist** [14], sistema inizialmente sviluppato per modellare la dinamica di soluzioni chimiche. Questo framework, di seguito esposto, è piuttosto generico e quindi può avere un ampia gamma di applicazioni, come nel pervasive computing [13], computazioni biologiche o interazioni sociali.

2.4.2 Alchemist

In questa sezione vengono riportate le principali caratteristiche di questo simulatore, soffermandoci in particolare sul suo modello computazionale.

Modello computazionale Come si può notare dalla figura 2.6 il mondo di Alchemist è costituito da un ambiente contenente dei nodi ognuno dei quali è composto da diverse molecole ed è programmato con un insieme di reazioni.

Si riporta di seguito una breve descrizione di ciascun elemento di questo framework.

- *Molecola* - astrazione per rappresentare specifiche informazioni relative al nodo;
- *Concentrazione* - valore associato a ciascuna molecola;
- *Nodo* - astrazione utilizzata per rappresentare una singola entità oggetto della simulazione, questo costituisce il contenitore di molecole e di reazioni ed è immerso nell'ambiente;
- *Ambiente* - l'ambiente è l'astrazione di Alchemist per lo spazio, è il contenitore di nodi ed è in grado di fornire informazioni circa dove si

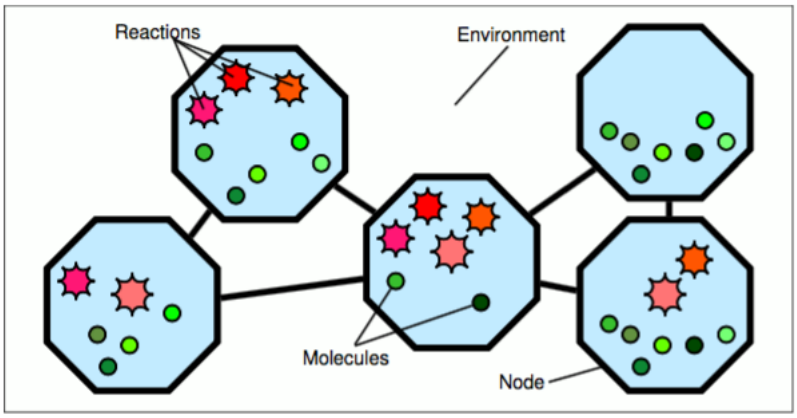

Figura 2.6: Modello computazionale di Alchemist (figura ripresa da [14])

trovano i nodi nello spazio (p.es. la loro posizione), quanto distane sono due nodi ed inoltre opzionalmente può fornire il supporto per lo spostamento di due nodi;

- *Regole di collegamento* - è la funzione dello stato corrente dell'ambiente che associa ad ogni nodo un suo vicinato;
- *Vicinato* - un entità composta da un nodo (centro) e da un insieme di nodi (vicini);
- *Reazione* - è un qualsiasi evento che può cambiare lo stato dell'ambiente, è composto da una lista di condizioni, una o più azioni e una distribuzione temporale che condiziona la sua frequenza di accadimento; ogni nodo è costituito da un insieme di queste reazioni (anche vuoto);
- *Condizione* - una funzione che prende in ingresso l'ambiente corrente e restituisce un booleano che discrimina se deve avvenire la reazione ad esso associata ed anche un numero che può influenzare la velocità della reazione;
- *Azione* - modella un cambiamento nell'ambiente.

Nella figura 2.7 viene mostrato il modello della reazione Alchemist, caratterizzato da: un insieme di condizioni dell'ambiente che determinano se una reazione può essere eseguita, un'equazione che calcola quanto veloce debba essere il cambiamento della reazione in risposta alle modifiche ambientali e, infine, un insieme di azioni che costituiscono l'effetto dell'esecuzione della reazione.

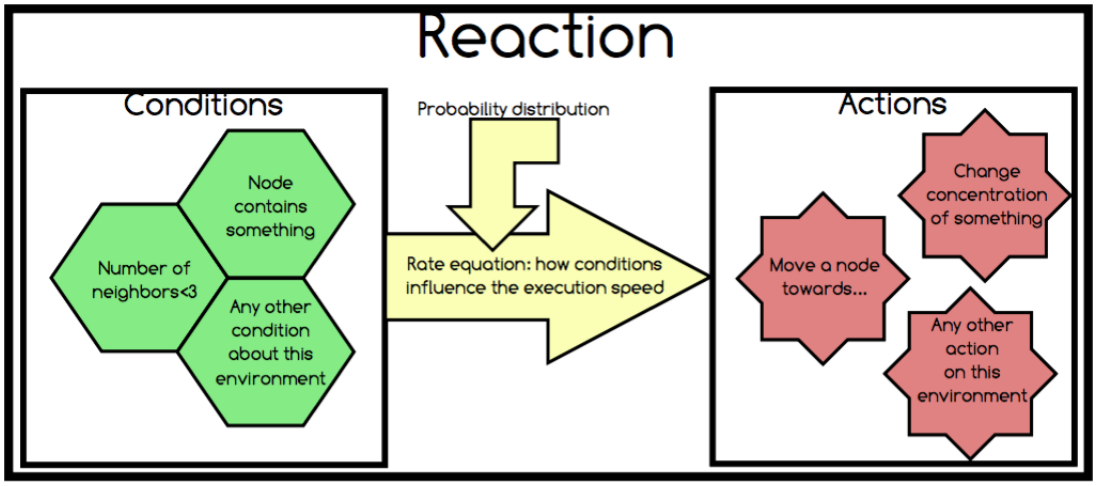

Figura 2.7: Modello della rezione di Alchemist (figura ripresa da [14])

Motore dinamico Uno dei più flessibili e veloci algoritmi per modellare sistemi chimici è il *Next Reaction Method* presentato in [12]. Il motore di Alchemist estende questo algoritmo fornendo la possibilità di aggiungere e rimuovere le reazioni dinamicamente.

Architettura L'intero framework presenta un'architettura modulare ed estendibile e in particolare attraverso la definizione di un nuovo tipo di struttura per le molecole e la concentrazione è possibile *incarnare* il simulatore per differenti e specifici usi. La figura 2.8 mostra nel dettaglio i vari componenti dell'architettura differenziandoli fra elementi comuni per ogni scenario e quelli che devono essere sviluppati per ogni specifica incarnazione.

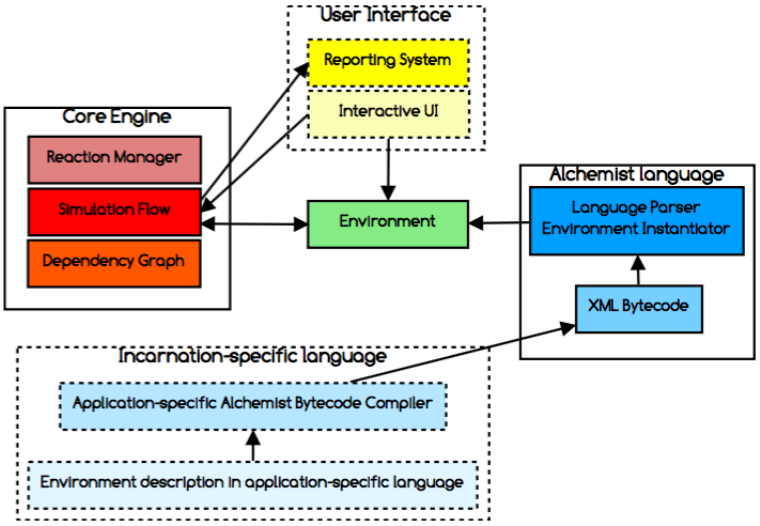

Figura 2.8: Architettura di Alchemist: gli elementi disegnati con le linee continue indicano i componenti comuni per ogni scenario e quelli con le linee tratteggiate i componenti specifici per ogni estensione (figura ripresa da [14])

Oggetto di questo lavoro è stato proprio la definizione di un'altra incarnazione per questo simulatore, in questo caso con il framework di aggregate programming *scafi* (capitolo 4).

Scrittura di una simulazione Per scrivere una simulazione prima di tutto l'utente deve implementare la specifica incarnazione, successivamente, come si può notare dalla figura 2.8, occore specificare la propria simulazione utilizzando il linguaggio XML descrivendo in maniera completa l'ambiente e le reazioni che si vuole mettere in campo. Tuttavia è comunque possibile specificare la propria simulazione anche utilizzando il linguaggio Java.

Questa pagina è lasciata intenzionalmente bianca.

Capitolo 3

Integrazione di Scafi e piattaforma AP

Questo capitolo è relativo all'integrazione del framework **scafi** con una specifica piattaforma per aggreagate programming, descritta nella sezione 2.3. Questo lavoro passa attraverso l'intero processo di produzione del software: analisi dei requisiti e del problema, fase progettuale e infine implementazione e validazione; in questo capitolo oltre ad una dettagliata descrizione di ognuna di queste fasi si riportano tutti gli artefatti e i documenti prodotti in tale processo.

3.1 Analisi dei requisiti e del problema

3.1.1 Requisiti

Il requisito dato relativo al lavoro oggetto di questo capitolo è: "effettuare l'integrazione del framework scafi con una piattaforma per aggregate programming, al fine di poter programmare in termini aggregati il comportamento di un insieme di nodi appartenenti ad un sistema pervasivo reale, avvalendosi delle primitive scala offerte da scafi".

3.1.2 Casi d'uso

I principali casi d'uso del prodotto che emergerà da questa integrazione sono riportati di seguito (figura 3.1) con un diagramma UML, che mostra le principali funzionalità del sistema che si otterrà e che rappresenta un modello dei requisiti sopraesposti.

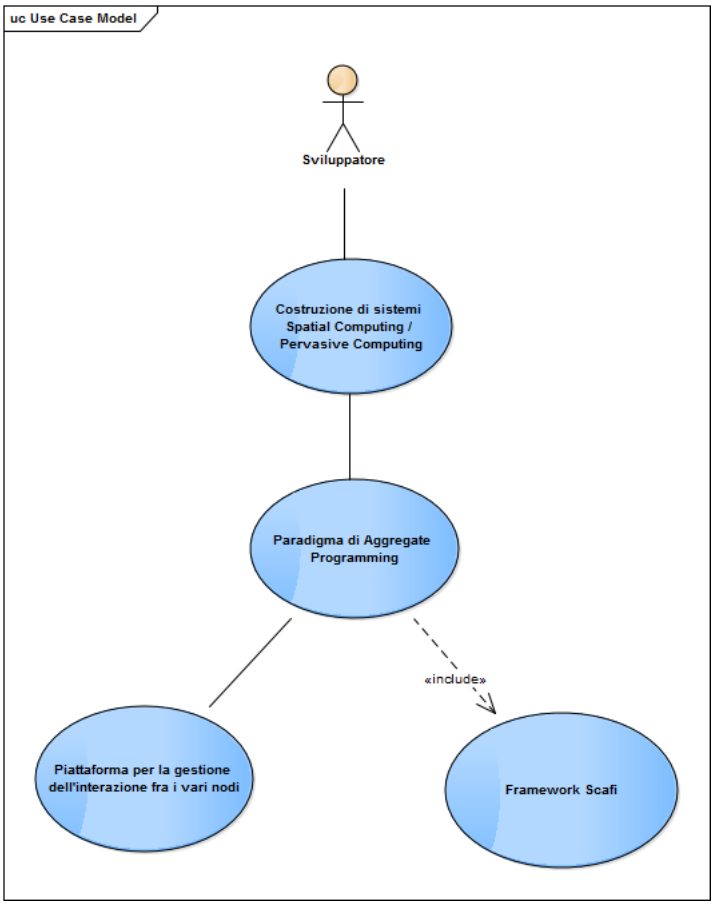

Figura 3.1: Diagramma UML dei casi d'uso dell'integrazione di scafi con la piattaforma per aggregate programming

Il principale caso d'uso del sistema risultante da questa integrazione è offrire la possibilità di poter costruire sistemi nell'ambito di Spatial

Computing [9] [4] o Pervasive Computing [13] utilizzando il paradigma dell'aggregate programming. A sua volta quest'ultimo *utilizza* un'altra funzionalità, ovvero una piattaforma che gestisce la distribuzione e le interazioni dei vari nodi, e *include* la funzionalità di poter utilizzare il framework scafi che offre una serie di primitive scala per facilitare la programmazione aggregata.

3.1.3 Architettura logica

L'architettura logica costituisce la principale parte dell'analisi del problema: rappresenta una sintesi di tutta l'analisi e una base per il successivo sviluppo. In particolare rappresenta un modello del sistema decomposto in tre dimensioni: interazione, comportamento e struttura, di seguito riportate.

Interazione Dal punto di vista dell'interazione si riporta il diagramma in figura 3.2, quest'ultimo deriva da una prima analisi effettuata astraendo da diversi dettagli e concentrandosi soprattutto sul cosa dover fare piuttosto che sul come. L'idea su cui si basa è effettuare l'integrazione attraverso una chiamata a procedura della piattaforma verso il framework scafi per demandare l'esecuzione di una determinata funzione e rimanere quindi in attesa del risultato.

Struttura Tale diagramma 3.2 offre anche una visione strutturale del sistema che è diviso nei due sottosistemi oggetto dell'integrazione:

- piattaforma di aggregate programming;
- framework scafi.

Da queste prime considerazioni si evince la necessità di utilizzare il pattern **adapter** [7] in quanto si intende usare un componente software esterno e quindi occorre adattare la sua interfaccia per motivi di integrazione con l'applicazione esistente. Come si può notare dalla figura 3.3 ogni partecipante protagonista del pattern adapter è perfettamente mappato su uno specifico componente di questo sistema:

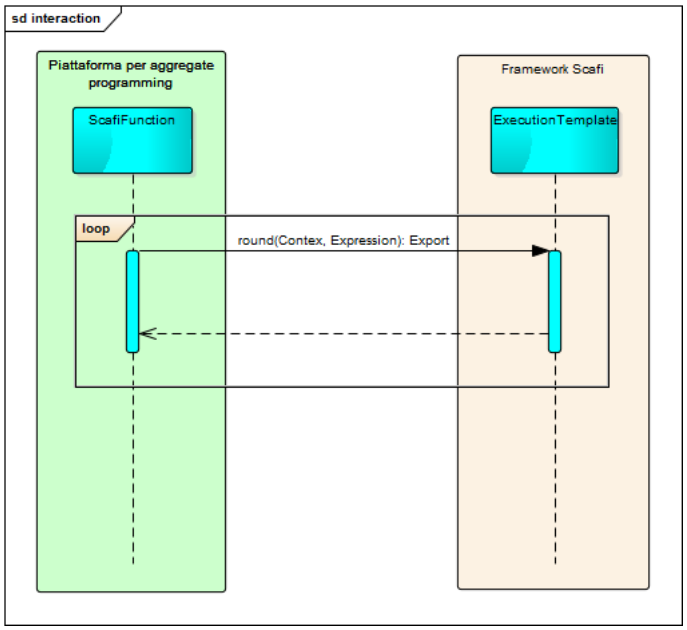

Figura 3.2: Diagramma UML di interazione

- *Client* - è rappresentato dal `ComputationWorker` ed è colui che effettua l'invocazione all'operazione di interesse, ovvero `compute()`;
- *Target* - è rappresentato dall'entità `FieldCalculusFunction` e definisce l'interfaccia specifica del dominio applicativo utilizzata dal client;
- *Adaptee* - è rappresentato dall'entità `ExecutionTemplate` e definisce l'interfaccia di un diverso dominio applicativo da dover adattare per l'invocazione da parte del client;
- *Adapter* - è rappresentato dall'entità `ScafiFunction` e definisce l'interfaccia compatibile con il target `FieldCalculusFunction` che maschera l'invocazione del metodo `round()` dell'adaptee `ExecutionTemplate`.

Da notare che dato l'utilizzo del linguaggio scala [17] come **ipotesi tecnologica** per questa integrazione, si è scelto di costruire il componente adap-

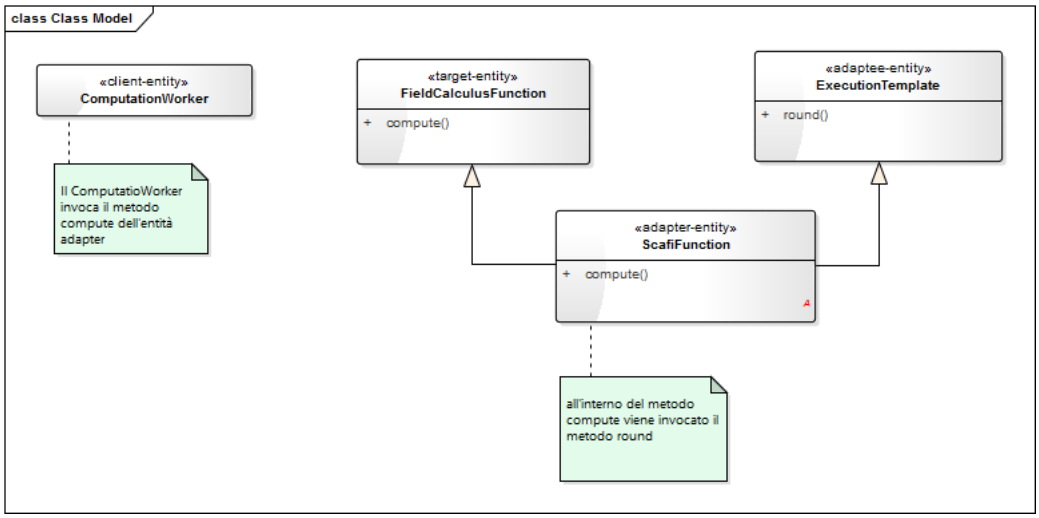

Figura 3.3: Diagramma UML del pattern adapter relativo a questo sistema

ter tramite l'estensione da entrambe le entità, sia `FieldCalculusFunction` che `ExecutionTemplate`, grazie al supporto che scala fornisce all'ereditarietà multipla. In tal caso si parla di *Class Adapter*.

Comportamento Dal punto di vista comportamentale si riporta un diagramma relativo alla piattaforma per aggregate programming ed in particolare all'entità `ComputationWorker` che è protagonista in questa integrazione, in quanto è colui che effettua con una determinata frequenza chiamate al framework scafi per l'esecuzione di funzioni.

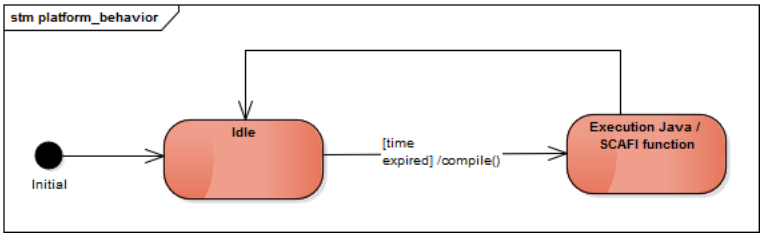

Figura 3.4: Diagramma UML di comportamento del ComputationWorker

3.2 Progetto

Questa fase si base sui risultati e i prodotti ottenuti nella precedente fase di analisi: infatti tutta la progettazione è costituita solamente dalla modifica ed estensione di concetti già emersi nell'analisi del problema. Di seguito viene formalizzata la descrizione di questa fase progettuale prendendo in considerazione sempre le tre dimensioni principali.

3.2.1 Struttura

Si riporta di seguito, prima di tutto qualche approfondimento sul progetto di entrambi i sottosistemi da integrare, approfondimenti che sono però confinati solamente alla parte che interessa a questa integrazione e in secondo luogo si descrive la struttura della soluzione messa in campo basata sul design pattern usato nell'analisi del problema.

Piattaforma: computazione di funzioni

Questa piattaforma è costituita dall'entità `ComputationManager` (sottosezione 2.3.2) in grado di calcolare diverse funzioni in modo asincrono invocando il metodo `addField()`. Il primo argomento di questo metodo è il modello della funzione da computare rappresentato da un oggetto che estende dal tipo `FieldCalculusFunction`. Questa è definita come una funzione matematica senza stato, in termini del valore precedentemente calcolato e il valore dei sensori, sia relativamente al proprio nodo che ai nodi del vicinato. Il `ComputationManager` è responsabile, attraverso i worker (oggetto con un proprio flusso di controllo e direttamente correlato ad una specifica funzione, sottosezione 2.3.2) di occuparsi della computazione di funzioni.

La figura 3.5 descrive più nel dettaglio la struttura di una Funzione.

Il metodo astratto `compute()` rappresenta il comportamento di una funzione ed è definito dall'utente attraverso l'estensione della classe FieldCalculusFunction. Questo metodo `compute()` può usare il valore dei vicini relativi ai sensori o alle funzioni e i dati dei sensori locali precedentemente aggiunti al `SensorManager`. Inoltre viene utilizzato un `ExecutionContext`

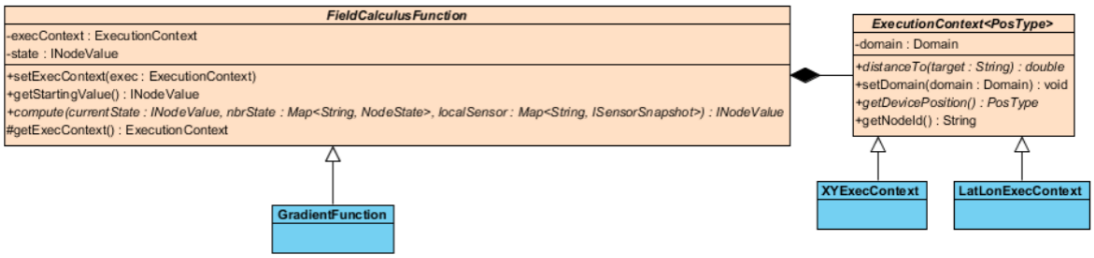

Figura 3.5: Diagramma UML di classe del modello della funzione

relativo ad un specifico tipo di coordinate (p.es. cartesiane o geospaziali) che fornisce dei metodi per il calcolo della distanza da un nodo di cui si conosce l'ID e inoltre offre la possibilità di determinare la posizione corrente.

Il valore di ritorno computato da queste funzioni viene modellato attraverso l'interfaccia `INodeValue`, che come si deduce dalla figura 3.6 l'oggetto che implementerà questa interfaccia dovrà avere un campo per mantenere una chiave, che rappresenta la funzione all'interno del dispositivo, e un campo per il valore di questa funzione.

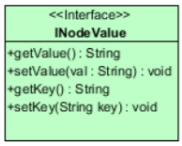

Figura 3.6: Diagramma UML della struttura del tipo del valore

Framework Scafi: ExecutionTemplate e relativo metodo round

Alcuni componenti presenti nella figura 3.7 costituiscono le entità principali di questa integrazione.

ExecutionTemplate Questa entità mantiene internamente lo stato della computazione grazie all'oggetto `Status`, che lavora come uno stack immutabile e registra i rami dell'albero del calcolo. In particolare poi all'interno dell'entità `ExecutionTemplate` è presente il metodo `round(c: CONTEXT,`

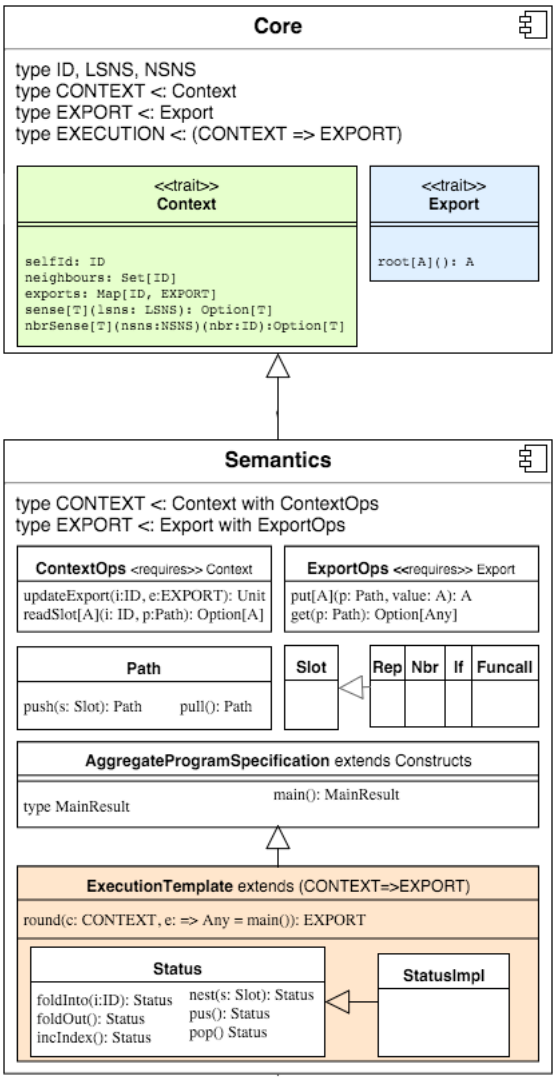

Figura 3.7: Diagramma UML di struttura delle entità interessate all'integrazione (figura ripresa da [5])

`e: => Any = main()): EXPORT` (figura 3.7) ovvero il metodo che attua la computazione di programmi aggregati scritti con primitive scala, quest'ul-

timo dovrà essere invocato dalla piattaforma per aggregate programming attuando in tal modo l'integrazione fra i due sottosistemi. Infine in questa entità è anche definito il metodo astratto `main` il quale dovrà essere implementato con il comportamento della funzione aggregata desiderato da colui che concretizzerà questa classe .

Context Il `Context` è il parametro di ingresso della funzione round e rappresenta tutte le informazioni relative all'ambiente necessarie per la computazione, più precisamente come viene mostrato dalla figura 3.7 troviamo:

- *selfId* - contiene l'id del proprio nodo;
- *neighbours* - contiene l'id di tutti i vicini;
- *exports* - contiene il risultato di tutti i miei vicini relativo alla precedente computazione della funzione in oggeto;
- *localSensor* - contiene tutti i sensori locali comprensivi del loro valore;
- *nbrSensor* - contiene il valore relativo a delle informazioni del proprio vicinato.

Export L'`Export` è la struttura dati utilizzata per rappresentare il risultato della computazione aggregata del metodo `round`; una tipica operazione su questa entità è l'estrazione del valore di root che rappresenta proprio il valore della computazione.

Integrazione dei due sottosistemi

La soluzione messa in campo come già accennato nella fase di analisi fa uso del pattern **adapter**, il diagramma UML in figura 3.8 rappresenta uno zoom di quello già riportato nella precedente fase (figura 3.3), mostrando ulteriori informazioni e dettagli progettuali.

Tale diagramma rappresenta il cuore di questa integrazione ed è basato sui seguenti concetti:

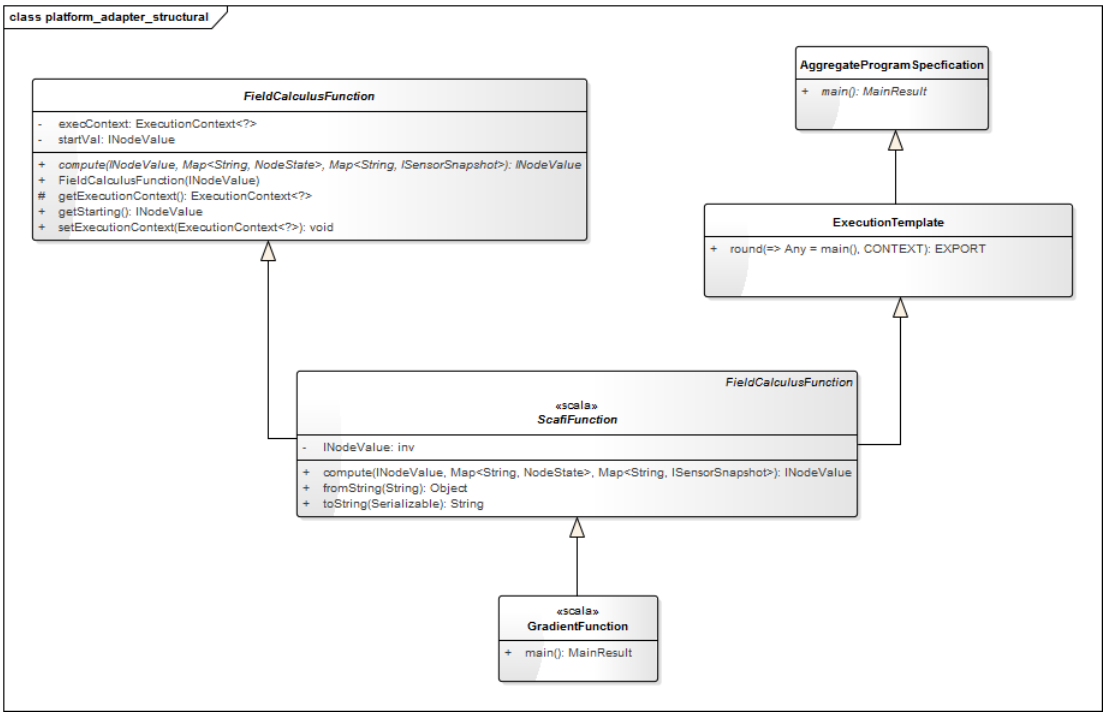

Figura 3.8: Diagramma strutturale relativo al pattern adapter

- una classe di nome `ScafiFunction`, che estendendo da entrambe le due classi da interfacciare (grazie all'ereditarietà multipla supportata in scala), prende il ruolo di *adapter*;
- in quest'ultima è stato implementato il metodo astratto `compute()` importato dalla superclasse `FieldCalculusFunction`, rimane invece astratto il metodo `main()` importato dalla classe AggregateProgramSpecification, tale metodo dovrà essere implementato dall'entità che concretizzerà `ScafiFunction`, con una specifica espressione da computare che rappresenta il comportamento aggregato desiderato (`GradientFunction`).
- l'implementazione del metodo `compute()` è caratterizzata in primo luogo dalla preparazione dell'oggetto `context` utilizzando le infor-

mazioni dei parametri di ingresso di questa funzione e infine viene effettuata l'invocazione al metodo `round()` di `ExecutionTemplate` passando in ingresso il contesto e l'espressione definita nel metodo `main`;

- per memorizzare il valore di ritorno del metodo `round()` ovvero l'`export` (sottosezione 3.2.1) necessario poi per le successive computazioni si è modificato l'interfaccia `INodeValue` aggiungendo, come si può notare dalla figura 3.9, i metodi `getContent()` e `setContent()` relativamente ad un nuovo campo atto a contenere un'altra informazione riferita al risultato che in questo caso è appunto costituita dall'`export`.

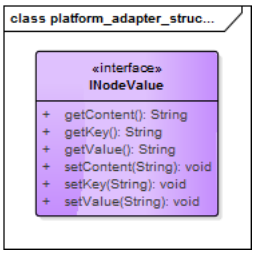

Figura 3.9: Diagramma strutturale del nuovo tipo del valore computato dalle funzioni

- data la necessità di inviare l'export in rete dal nodo al server e successivamente dal server ai nodi si è deciso di serializzare questo oggetto grazie alla sua ereditarietà dalla classe `Serializable` e salvarlo poi nello specifico campo `content` sopracitato sotto forma di stringa grazie all'utilizzo della libreria `java.util.Base64`, tale stringa sarà poi deserializzata ogni volta durante la preparazione del contesto per l'invocazione del metodo `round` nei vari nodi.

3.2.2 Interazione

La dimensione di interazione dell'architettura di progetto è rappresentata nella figura 3.10 con un diagramma UML di sequenza che estende quello

riportato nella fase di analisi (3.2) con più entità, tuttavia sempre confinate a quelle relative all'integrazione in oggetto.

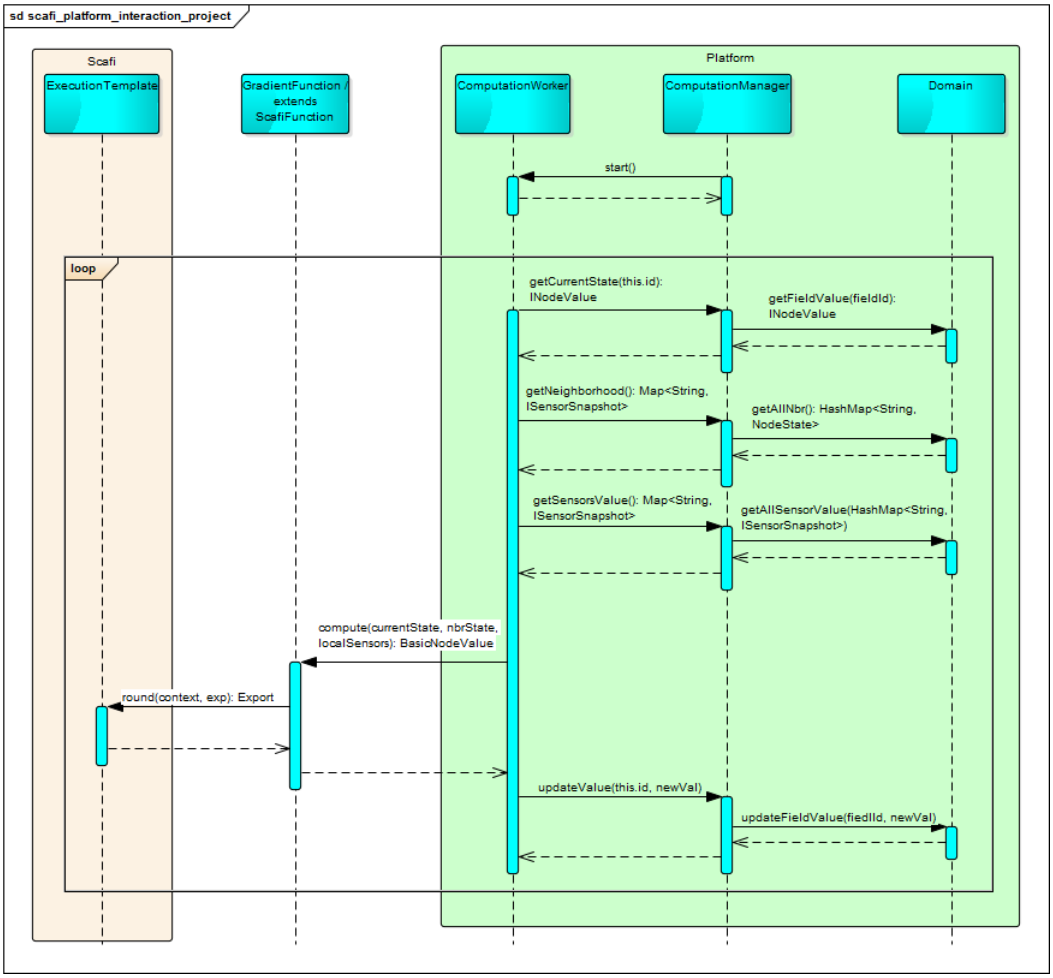

Figura 3.10: Diagramma UML di interazione dell'architettura di progetto

Questo diagramma mostra in modo chiaro tutte le azioni relative all'aggiunta di una nuova funzione e alla sua computazione mostrando la catena delle chiamate sia fra i componenti interni a ciascun sottosistema che anche fra i componenti appartenenti a diversi sottosistemi.

1. creazione della funzione desiderata estendendo da `ScafiFunction` implementando il metodo astratto `main` con primitive scafi;

2. dopo avere aggiunto questa funzione al `ComputationManager` tramite `addField()` (sottosezione 3.2.1) quest'ultimo fa partire tramite `start()` il `ComputationWorker`;

3. il `ComputationWorker` reperisce tramite il `ComputationManager` tutte le informazioni dal `Domain` che poi passerà all'entità che estende `ScafiFunction` con l'invocazione del metodo `compute`;

4. quest'ultima dopo aver opportunamente preparato l'oggetto contesto tramite le informazioni passate nel metodo `compute` effettua il "ponte" con il framework scafi invocando il metodo `round()` della classe `ExecutionTemplate`;

5. appena il framework scafi termina la computazione il risultato di quest'ultima (`Export`) ritorna alla funzione che estende l'adapter che serializza e lo inserisce in un'entità che rispetta l'interfaccia `INodeValue` in modo da poter essere compatibile in qualsiasi entità della piattaforma;

6. appena l'entità di tipo `INodeValue` torna al `ComputationWorker` quest'ultimo inserisce questo oggetto per mezzo del `ComputationManager` nel `Domain`

7. il flusso di controllo del `ComputationWorker` dopo aver aggiornato il `Domain` viene sospeso per un certo tempo prefissato e successivamente riprende dal punto 3.

3.2.3 Comportamento

Relativamente al comportamento non vi è niente da aggiungere rispetto a quanto già riportato nella fase di analisi 3.1.3.

3.3 Implementazione

Per molti elementi di questo sistema, l'**implementazione** deriva direttamente dalla fase di progetto, quindi non vi è molto da dire in questa sezione: tuttavia si riportano alcune piccole ma importanti note relative ad alcune parti d'implementazione, si riporta un confronto tra l'utilizzo del linguaggio Java e l'utilizzo delle primitive scala messe a disposizione dal framework scafi per la scrittura di un programma aggregato (calcolo del *gradiente*) e in ultimo si descrivono alcuni test effettuati al fine di validare il sistema realizzato.

3.3.1 Cenni implementativi

Per lo sviluppo di questo lavoro si è utilizzato il linguaggio **scala** [17] come tecnologia di programmazione, analogamente al framework scafi, in quanto presenta un ottima interoperabilità con Java. In particolare, grazie a questa tecnologia si è potuto giovare del suo supporto all'ereditarietà multipla nella costruzione dell'adaper, come ben riportato nelle sezioni precedenti di analisi e di progetto. Inoltre il supporto al *paradigma funzionale* e la presenza di una sintassi molto *concisa* ed *espressiva* ha facilitato l'implementazione relativa alla parte di costruzione dell'oggetto `Context` a partire dalle informazioni passate tramite i parametri della funzione `compute()`.

Un dato molto importante appartenente a questo oggetto `Context` è l'*nbrRange* che è costituito dalle distanze tra il proprio nodo e ciascun nodo del vicinato, tale informazione viene calcolata grazie all'oggetto `ExecutionContext` sopra citato (3.2.1) il quale offre la possibilità di determinare oltre alla propria posizione anche la distanza fra il proprio nodo e un altro nodo conoscendone l'ID. Il framework scafi utilizza l'informazione di nbrRange per diverse funzioni di notevole importanza come ad esempio il calcolo del *gradiente* o il calcolo del percorso più breve per arrivare in un determinato punto in uno spazio con la presenza di ostacoli (funzione *branch*).

In conclusione si vuole rimarcare la semplicità con cui il sistema ottenuto permette di aggiungere determinate funzioni aggregate scritte con primitive scafi, in quanto basta semplicemente estendere la classe adapter

`ScafiFunction` implementando il metodo astratto `main` con il comportamento aggregato desiderato, e questa è la funzione che poi sarà aggiunta al nodo nella fase di configurazione iniziale. Si riporta di seguito 3.3.1 l'implementazione di una funzione che effettua il calcolo del gradiente.

```
class GradientFun(inv: INodeValue) extends ScafiFunction(inv) {
  def gradient(source: Boolean): Double =
    rep(Double.MaxValue) {
      distance =>
        mux(source) { 0.0 } {
          foldhood(Double.MaxValue)((x, y) => if (x < y) x else
              y)(nbr { distance } + nbrvar[Double](NBR_RANGE_NAME))
        }
    }
  def main() = gradient(sense[Boolean]("source"))
}
```

Inoltre per sottolineare l'utilità della piattaforma ottenuta da questa integrazione si riportano le parti principali della stessa funzione gradiente implementata però col linguaggio java 3.3.1.

```
public class GradientFunction extends FieldCalculusFunction {
   .....
   @Override
   public INodeValue compute(INodeValue localCurrentStates,
         Map<String, NodeState> nbrState,
         Map<String, ISensorSnapshot> localSensors) {
      .....
      for (String k : localSensors.keySet()) {// find the source
         if (localSensors.get(k) != null) {
            if (localSensors.get(k).getSensorId().equals("source"))
               source = localSensors.get(k);
         } else {
            nextValue.setValue("Error");
         }
      }
      .....
```

```
    if(Boolean.parseBoolean(source.getValue())){//is source
       nextValue.setValue("0"); // distance 0
    } else { // isn't source
       .....
       for(String k:nbrState.keySet()){//find min distance node
          if (k != exec.getNodeId()) {
             NodeState ns = nbrState.get(k);
             double dist = Double.MAX_VALUE;
             for (INodeValue v : ns.getValues()) {
                if (v.getKey().equals("distGrad")) {
                   try {
                      dist = Double.parseDouble(v.getValue());
                   } catch (Exception ex) {
                      dist = Double.MAX_VALUE;
                   }
                }
             }
             if (dist < minDist) {
                minDist = dist;
                keyMinNbr = k;
             }
          }
       }
       if (keyMinNbr.isEmpty()) nextValue.setValue("inf");
       else {
          if (exec != null) nextValue.setValue(Double.toString(
             minDist + exec.distanceTo(keyMinNbr)));
       }
    }
    return nextValue;
  }
}
```

Confrontando queste due implementazioni si nota che utilizzando le primitive scala messe a disposizione dal framework scafi, si ottiene una notevole semplificazione nel descrivere e programmare il comportamento aggregato desiderato, agevolando in tal modo la costruzione dell'intero sistema.

3.3.2 Validazione

Per quanto riguarda la **validazione** del sistema, ovvero confermare che i requisiti, almeno in riferimento a determinati usi sono stati soddisfatti, si è costruita una piccola demo: grazie alla piattaforma per aggregate programming si sono creati 9 nodi virtuali e posizionati con determinate coordinate geospaziali creando una griglia 3 x 3, su ognuno di questi nodi si è predisposto un sensore sorgente, impostato a "true" per solo un nodo, e al fine di valutare la correttezza dell'integrazione si è aggiunta una funzione del gradiente scritta in scala con primitive scafi e un'altra funzione del gradiente scritta in java di cui si è già validato il corretto funzionamento. Questa demo come si può notare anche dallo snapshot di figura 3.12 ha riportato esito positivo in quanto i valori derivanti dalla computazione delle due funzioni coincidono.

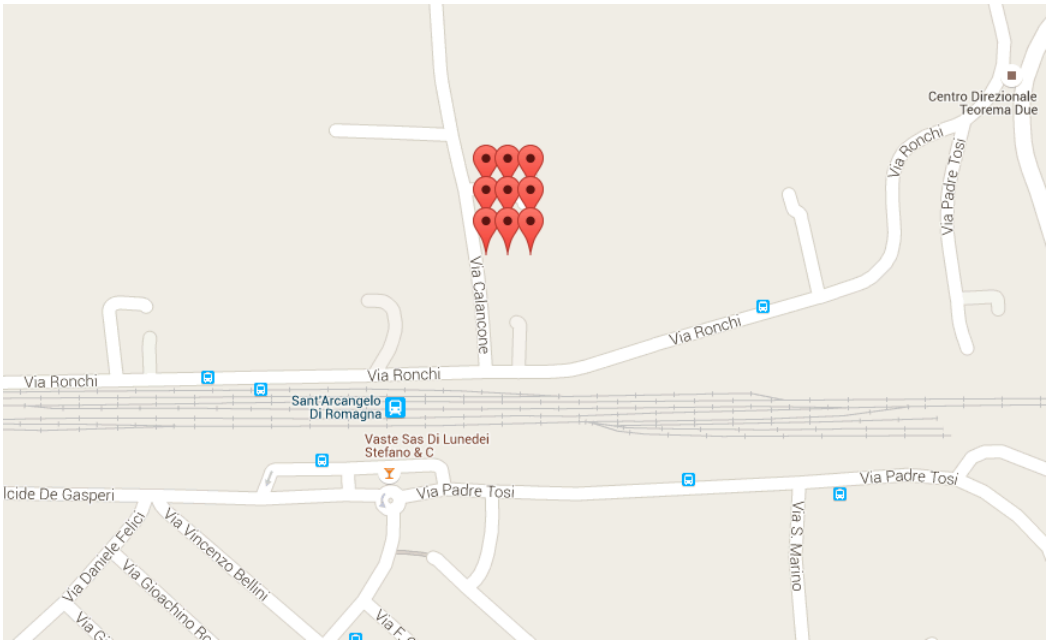

Figura 3.11: Snapshot 1 del sistema virtuale creato durante la sua esecuzione

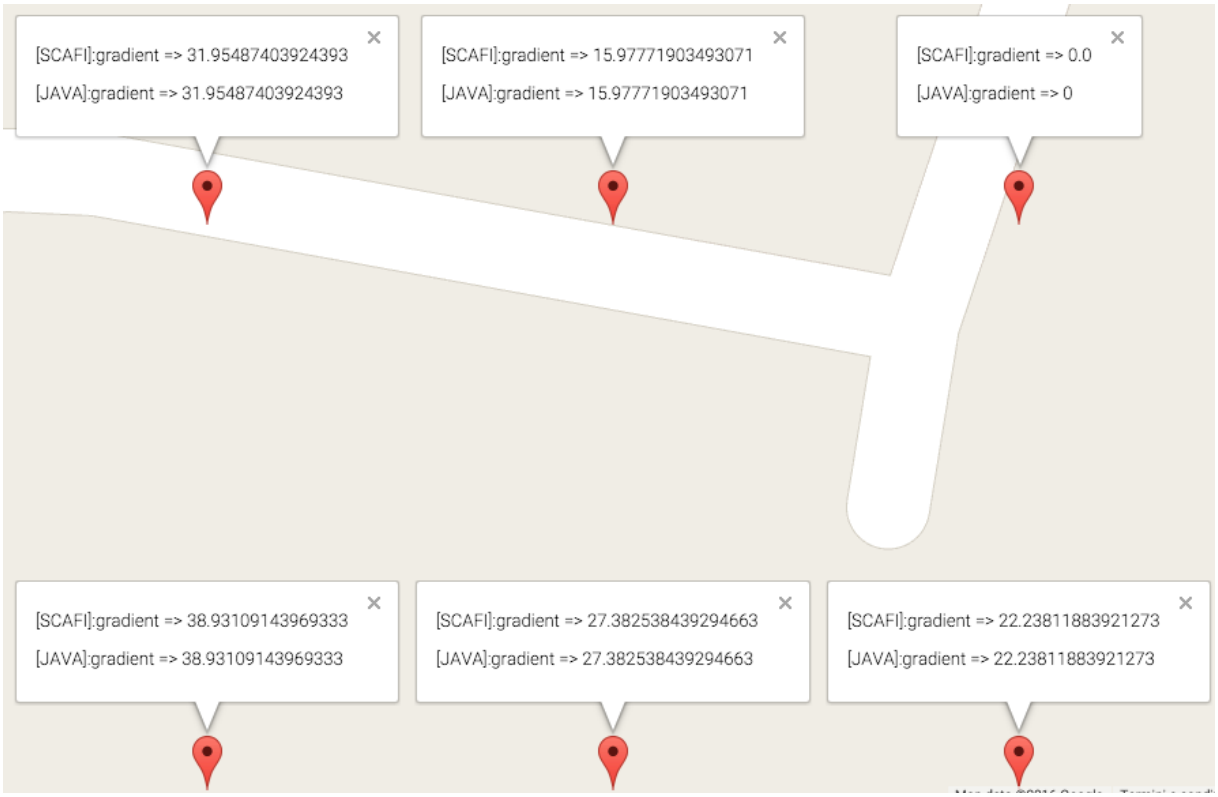

Figura 3.12: Snapshot 2 del sistema virtuale creato durante la sua esecuzione

Capitolo 4

Integrazione di Scafi e Alchemist

Questo capitolo riguarda l'integrazione del framework **scafi** (sezione 2.2) con il simulatore **Alchemist** [14] (sottosezione 2.4.2). Anche questo lavoro è stato svolto passando attraverso tutte le fasi del processo di produzione del software, di seguito riportate unitamente ai vari documenti e artefatti prodotti.

4.1 Analisi dei requisiti e del problema

4.1.1 Requisiti

Il requisito assegnato è il seguente: "effettuare l'integrazione del framework scafi con il simulatore Alchemist, in modo da riuscire a simulare il comportamento di una determinata rete di nodi scritto avvalendosi di primitive scala offerte da scafi".

4.1.2 Casi d'uso

I principali casi d'uso del prodotto emergente da questa integrazione sono riportati di seguito (figura 4.1) con un diagramma UML, che mostra le principali funzionalità del sistema ottenuto e che rappresenta un modello dei requisiti sopraesposti.

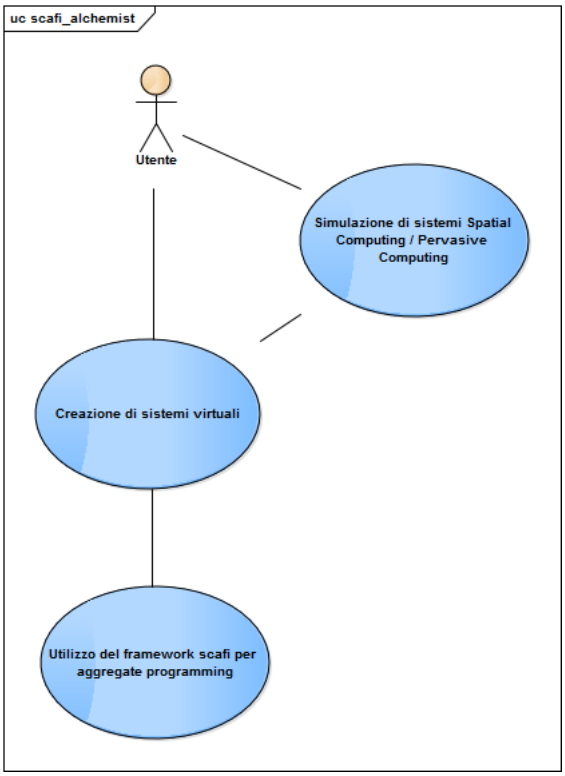

Figura 4.1: Diagramma UML dei casi d'uso dell'integrazione di scafi con il simulatore Alchemist

Il principale caso d'uso del sistema ottenuto da questa integrazione è offrire la possibilità di poter simulare sistemi nell'ambito di Spatial Computing [9] [4] o Pervasive Computing [13], secondariamente si trova la funzionalità che permette la costruzione di questi sistemi virtuali da simulare avvalendosi del framework scafi per aggregate programming.

4.1.3 Architettura logica

Si riporta in questa sezione l'architettura logica, ovvero una sintesi di tutte le considerazione e le opinioni emerse in questa fase di analisi del problema suddivise nella dimensione di interazione e di struttura.

Interazione In una prima analisi si è cercato di focalizzare l'attenzione su *cosa* deve avvenire piuttosto che sul *come*, in particolare si evince che l'idea di fondo per realizzare questa integrazione, astraendo da diversi dettagli, è come nel caso precedente riuscire ad effettuare tramite chiamata a procedura l'invocazione al metodo `round` di scafi, come si può notare dal diagramma UML sottoriportato 4.2. In tal modo Alchemist può simulare un comportamento per ciascun nodo descritto in termini di primitive scala e computato dal framework scafi.

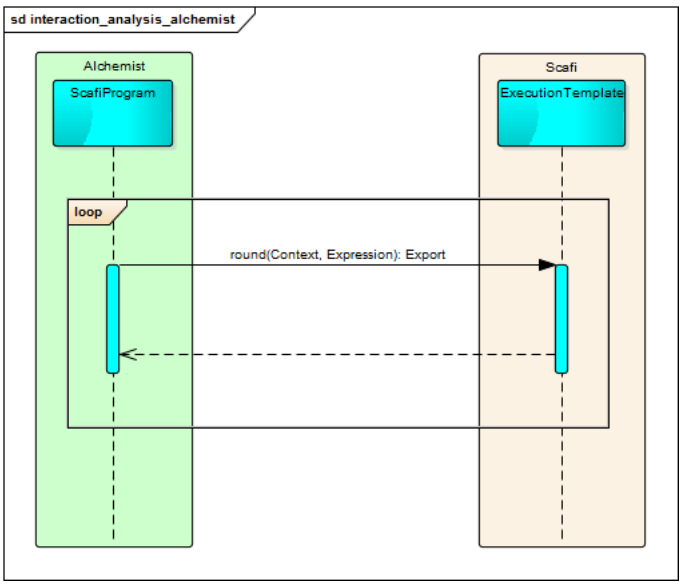

Figura 4.2: Diagramma UML dell'interazione tra i due sottosistemi

Struttura Dal diagramma sopraesposto emerge anche la struttura del sistema costituita principalmente dal sottosistema alchemist e scafi. Come nel caso precedente dovendo far interagire due componenti con interfacce diverse si desume la necessità del pattern *adapter* [7]. L'adpter adottato (`ScafiProgram`) è rappresentato nel digramma in figura 4.3 in particolare quest'ultimo estende dall'entità `Action` di alchemist e invece di estendere anche la classe da adattare viene creata un'istanza di quest'ultima diret-

tamente al suo interno previo passaggio del suo nome al momento della costruzione. Questo tipo di pattern adapter prende il nome di *Object Adapter*. L'utilizzo dell'approccio basato sulla reflection è spinto dal volere costruire questa estensione al simulatore mantenendo una certa uniformità con le metodologie già utilizzate al suo interno.

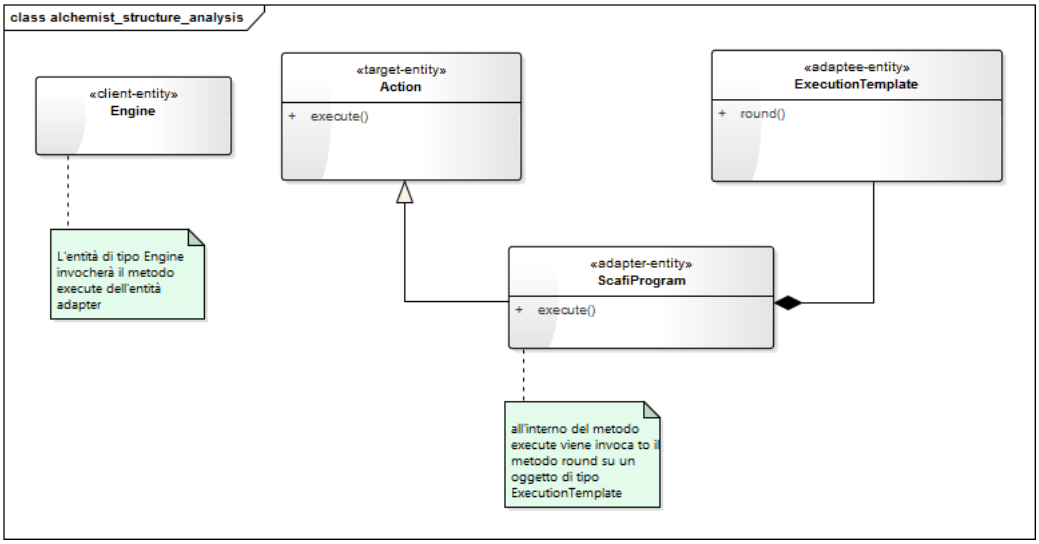

Figura 4.3: Diagramma UML del pattern adapter relativo a questo sistema

Un'altra considerazione effettuata a questo punto del processo di produzione è relativa alla natura fortemente modulare ed estendibile del simulatore alchemist: infatti nonostante questo fu inizialmente concepito per simulazioni in ambito prettamente chimico molte entità di esso possono essere opportunamente ridefinite e reimplementate al fine di effettuare simulazioni in un qualsiasi ambito. Questa sua caratteristica, come si noterà nella fase di progetto, gioverà particolarmente questo sviluppo, in quanto è sufficiente effettuare una nuova definizione di alcune entità (p.es. molecole, azioni, condizioni, reazioni o nodi) per creare una nuova **incarnazione** del simulatore.

4.2 Progetto

Il sistema è stato progettato seguendo le linee tracciate nell'analisi del problema, in particolar modo relativamente al concetto di incarnazione e al pattern adapter per effettuare l'interfacciamento dei due sottosistemi: si riporta in questa sezione una descrizione formale di questa fase progettuale concentrandosi sulla dimensione strutturale e di interazione.

4.2.1 Struttura

La soluzione adottata sfrutta il supporto all' **incarnazione** messo a disposizione da Alchemist già accennato nella fase di analisi. In particolare per questa incarnazione col framework scafi si è provveduto a definire:

- un tipo di **azione**, `ScafiProgram`, che costituisce l'entità nella quale specificare il comportamento del singolo nodo ed inoltre funge da adapter per interfacciare i due sottosistemi;
- un tipo di **nodo**, chiamato `ScafiNode` che rappresenta il singolo dispositivo;
- un nuovo tipo di **molecola**, denominata `ScafiMolecule` per rappresentare i sensori e il valore delle funzioni di ciascun dispositivo.

ScafiProgram

Il diagramma UML in figura 4.4 è rappresentativo della struttura dell'entità `ScafiProgram` e si basa sui seguenti concetti:

- `ScafiProgram` rappresenta l'entità **adapter** in quanto implementa l'entità `Action` di Alchemis e mantiene al suo interno un'istanza di un'entità di tipo `ExecutionTemplate` contente quindi il metodo `round` che deve essere invocato dal simulatore;
- quest'ultima entità deve implementare il metodo astratto `main` ereditato da `AggregateProgramSpecification` con il comportamento aggregato desiderato;

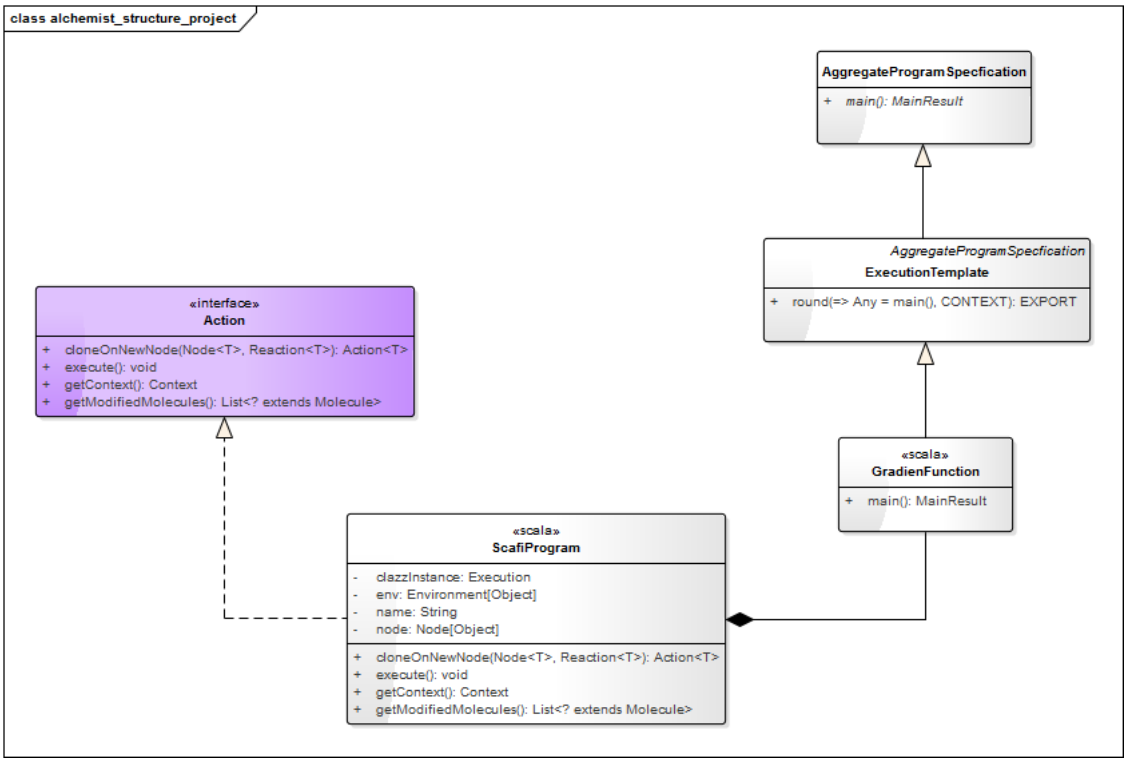

Figura 4.4: Diagramma UML strutturale dell'entità ScafiProgram

- `ScafiProgram` costituisce inoltre una nuova definizione di azione implementando l'interfaccia `Action` di Alchemist, di particolare importanza vi è l'implementazione del metodo `execute` nel quale avviene la preparazione dell'oggetto `Context` reperendo tutte le informazioni necessarie dall'ambiente di simulazione e successivamente viene effettuata l'invocazione al metodo `round`, facendo così interagire i due sottosistemi;

- il risultato della computazione della `round`, l'`export` viene salvato direttamente nel nodo.

ScafiNode

E' stato definito questo nuovo tipo di nodo, la cui struttura è raffigurata nell'immagine 4.5, allo scopo di introdurre un campo nel quale salvare il risultato di tipo `Export` della computazione della `round`: questo dato è necessario al framework scafi per la successiva computazione sia locale che quella del proprio vicinato.

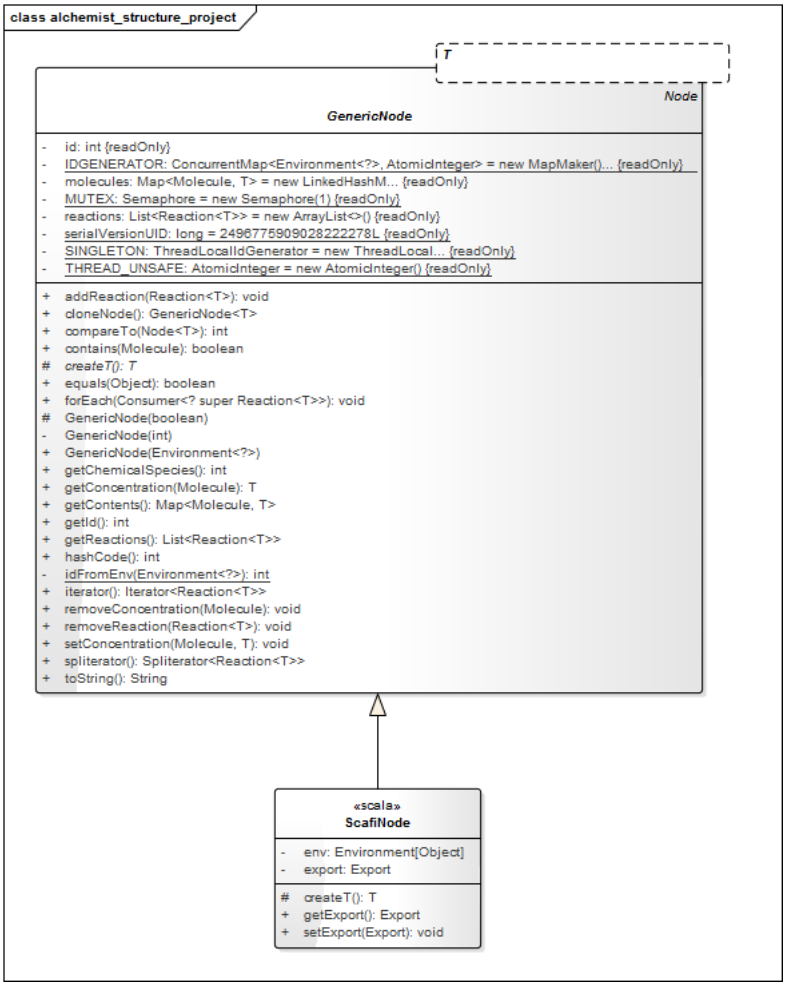

Figura 4.5: Diagramma UML strutturale dell'entità ScafiNode

ScafiMolecule

Infine si è definito anche un nuovo tipo di molecola (4.6) per rappresentare i sensori in scafi e il valore di ritorno delle funzioni. Non è stato possibile mappare il concetto di sensore sulla molecola già esistente in Alchemist, `SimpleMolecule`, in quanto non aveva un attributo nel quale poter mantenere il tipo del sensore (p.es. sensore di distanza, temperatura, ecc.), astrazione invece indispensabile nel framework scafi.

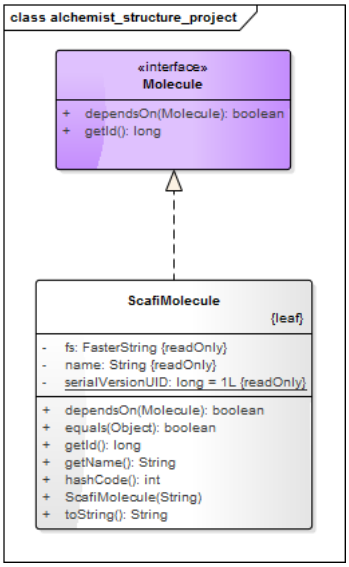

Figura 4.6: Diagramma UML strutturale dell'entità ScafiMolecule

ScafiSimulation

Si è creato inoltre l'entità `ScafiSimulation` che permette di facilitare la creazione delle simulazioni con l'incarnazione in scafi. Come si nota dal diagramma UML in figura tale entità dispone di tutta una serie di primitive atte a sgravare lo sviluppatore della simulazione da diverse operazioni e passaggi che a lui non interessano (p.es. alcuni aspetti legati all'ambiente di simulazione, azioni, reazioni, ecc.) senza comunque perdere il controllo

sulla personalizzazione dei vari nodi relativamente alla loro posizione e alle loro molecole.

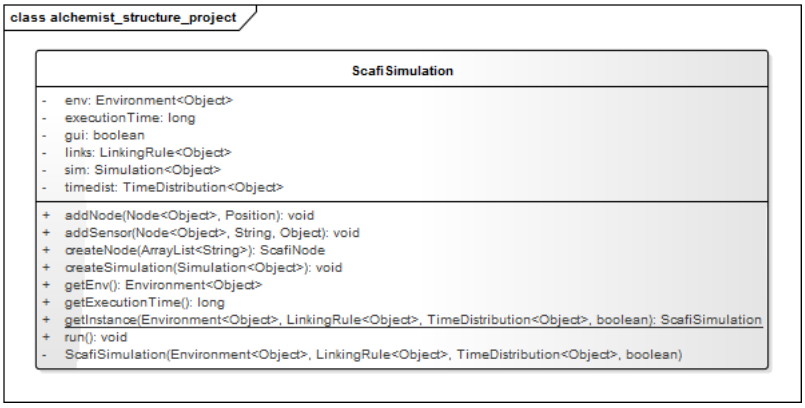

Figura 4.7: Diagramma UML strutturale dell'entità ScafiSimulation

4.2.2 Interazione

Il diagramma in figura 4.8 mostra la sequenza di azioni che avvengono durante l'esecuzione di un'azione di un nodo, avviate dall'entità `Engine` di Alchemist. Si sono riportati solamente le entità strettamente connesse al lavoro effettuato ovvero al punto di unione fra i due sottosistemi. Da notare che in questo primo approccio di integrazione si è scelto di salvare localmente a ogni nodo il risultato della propria computazione in tal modo ogni nodo al momento della preparazione del `context` per reperire i risultati delle computazioni di tutti i vicini è sufficiente che reperisca per mezzo dell'`Environment` i nodi del proprio vicinato dai quali ottenere poi il relativo oggetto `Export`.

4.3 Implementazione

Anche in questa integrazione, grazie alla svolgimento di un'accurata fase di progettazione, l'implementazione è risultata piuttosto immediata . Si

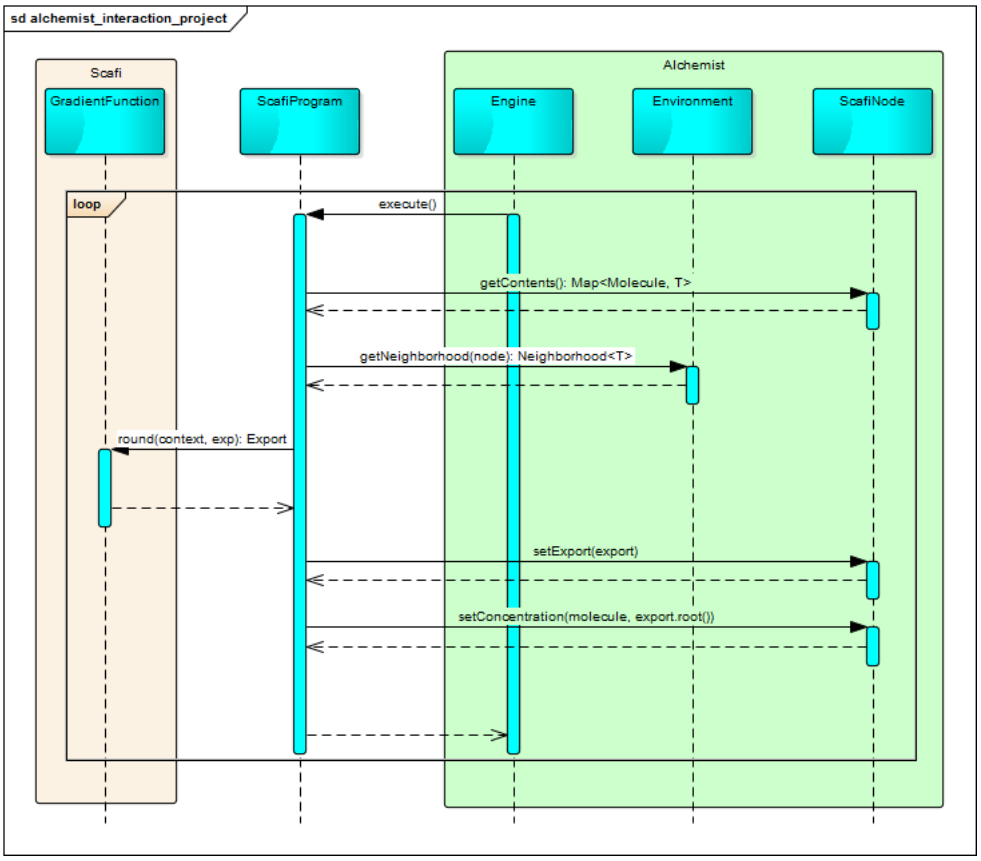

Figura 4.8: Diagramma UML d'interazione della fase progettuale

riportano in questa sezione alcune note riguardanti il linguaggio utilizzato, qualche demo effettuata al fine di assicurarsi del corretto funzionamento del sistema ed infine si riporta un test sulle performance e la scalabilità, confrontando poi i risultati ottenuti con l'incarnazione di protelis.

4.3.1 Cenni implementativi

Data l'analogia con la precedente integrazione e i vantaggi riscontrati grazie all'utilizzo di scala [17], si è mantenuto anche per questo sviluppo tale

linguaggio come tecnologia di programmazione, usufruendo sempre della sua sintassi espressiva e concisa. Per sottolineare queste caratteristiche si riporta di seguito il codice scala inerente alla costruzione dell'oggetto `Context`.

```
val ctx = new ContextImpl(
        selfId = ""+node.getId,
        neighbours = nbr.map(n => ""+n.getId).toSet,
        exports = {
            var nbrExp = nbr
                .filter(n => n.asInstanceOf[ScafiNode].getExport
                    != null)
                .map(n => ""+n.getId ->
                    n.asInstanceOf[ScafiNode].getExport).toMap
            if(node.asInstanceOf[ScafiNode].getExport != null){
                nbrExp + (""+node.getId ->
                    node.asInstanceOf[ScafiNode].getExport)
            }else{
                nbrExp
            }
        },
        localSensor = node.getContents
            .map(sv => sv._1.asInstanceOf[ScafiMolecule].getName
                -> sv._2),
        nbrSensor = Map("nbrRange" -> (nbr
            .map(n => ""+n.getId ->
                env.getDistanceBetweenNodes(node, n))
            .toMap + (""+node.getId -> 0.0))))
```

In questa integrazione però, non si è usufruito del suo supporto all'ereditarietà multipla nella realizzazione del pattern *adapter*: più precisamente l'adapter estende solo da una delle due entità da interfacciare, l'altra viene instanziata al suo interno attraverso un meccanismo basato sulla *reflection*. Come già riportato nelle sezioni precedenti questa scelta è dovuta al mantenimento di una certa uniformità con le metodologie adottate nel simulatore alchemist.

Ad ogni modo anche questa progettazione porta ad una elevata semplicità

di utilizzo, in quanto è sufficiente definire una nuova classe che estende da `ExecutionTemplate` implementando solamente il metodo `main` con il comportamento desiderato e passare il nome di questa classe al costruttore dell'adapter `ScafiProgram`.

4.3.2 Validazione

Per validare il sistema ottenuto si sono effettuate alcune simulazioni, ognuna delle quali si basa su una rete di nodi costituita da una griglia bidimensionale, dove ogni nodo occupa una posizione casuale all'interno della sua specifica cella; questo garantisce una configurazione del sistema diversa per ogni simulazione. Nell'immagine 4.9 riportata di seguito, viene mostrato un esempio di queste reti con i relativi collegamenti di vicinato per ogni nodo.

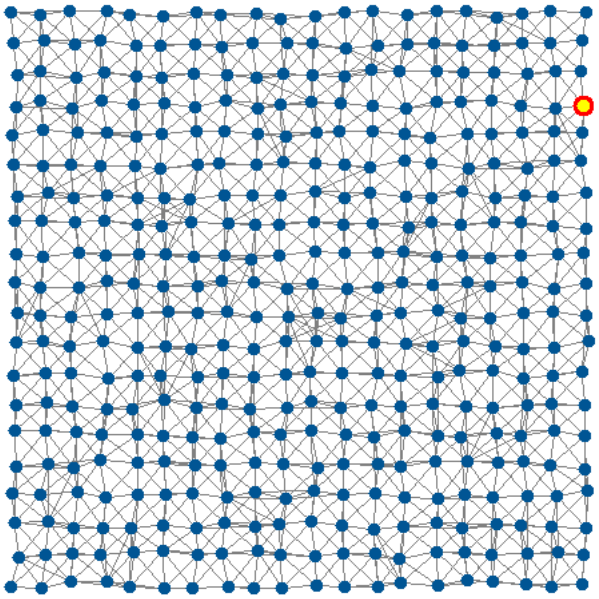

Figura 4.9: Rete di nodi con i relativi collegamenti di vicinato

Le simulazioni sono state eseguite avvalendosi della GUI `SingleRunGUI` grazie alla quale si è potuto appurare, anche se solo qualitativamente, del loro corretto funzionamento. Si sono provati diversi programmi aggregati,

ad esempio il calcolo del numero dei propri vicini, il calcolo della distanza fra due dispositivi oppure il calcolo del gradiente. Si riporta di seguito qualche snapshot effettuato durante una simulazione relativa al calcolo del gradiente in cui la sorgente è posizionata nel nodo nell'angolo in basso a sinistra. Nell'immagine 4.10 si riprende il sistema al tempo di simulazione 8 durante la sua fase transitoria iniziale, dove ancora ai nodi più lontanti dalla sorgente non si è propagata l'informazione per poter computare il proprio valore.

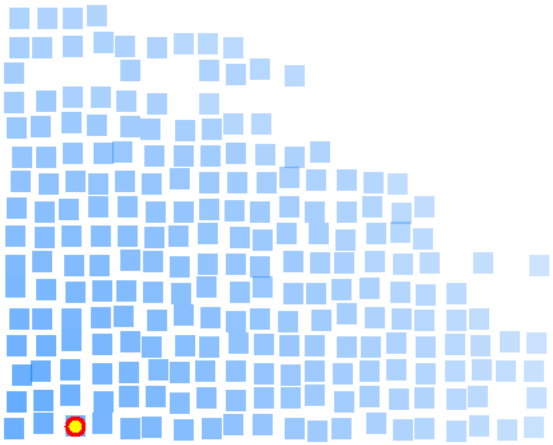

Figura 4.10: Snapshot 1 della simulazione del calcolo del gradiente

Lo snapshot in figura 4.11 scattato al tempo di simulazione 16 è relativo invece al sistema a regime, ovvero quando si è stabilizzato, in particolare in questa immagine l'effetto di colori prodotto dalla GUI mette in evidenza il corretto funzionamento della simulazione, in quanto l'intensità del colore che è inversamente proporzionale al valore del gradiente diminuisce con l'allontanarsi dalla sorgente.

Un altro test di validazione concluso con esito positivo è stato eseguito confrontando i valori del gradiente computati da ogni nodo fra una simula-

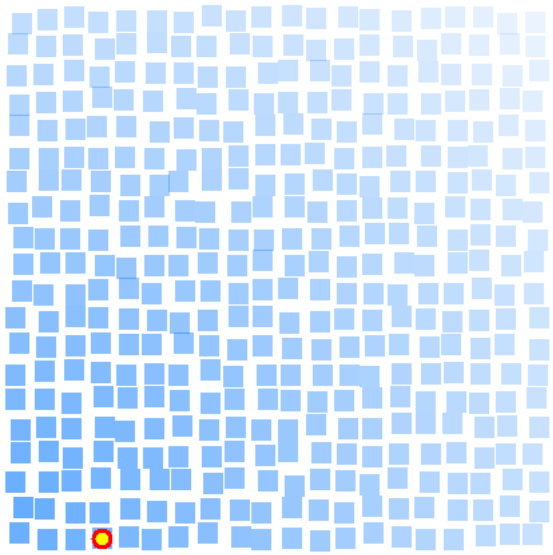

Figura 4.11: Snapshot 2 della simulazione del calcolo del gradiente

zione effettuata con questa incarnazione in scafi e una con l'incarnazione in protelis, di cui si è certi del corretto funzionamento.

4.3.3 Test di performance

Infine si sono sviluppati due test di performance che mettono a confronto il framework scafi con protelis. Questi test sono stati svolti su una macchina con due processori da 2.53GHz.

Confronto variando la cardinalità del vicinato Il primo test è basato sul programma aggregato relativo al calcolo del gradiente, l'idea è quella di fissare un determinato tempo di simulazione (100) e confrontare i tempi di esecuzione impiegati da entrambe le piattaforme per compiere una simulazione di tale lunghezza. Ovviamente entrambe le simulazioni sono svolte sulla medesima configurazione della rete dei nodi, dove ognuno di essi viene disposto in maniera casuale in una griglia di dimensioni 20 x 20. Tale

confronto viene ripetuto per diverse cardinalità del vicinato dei nodi, in modo da avere delle informazioni circa la scalabilità delle due piattaforme.

Si riporta nell'immagine 4.12 un grafico a barre riassuntivo dei risultati ottenuti.

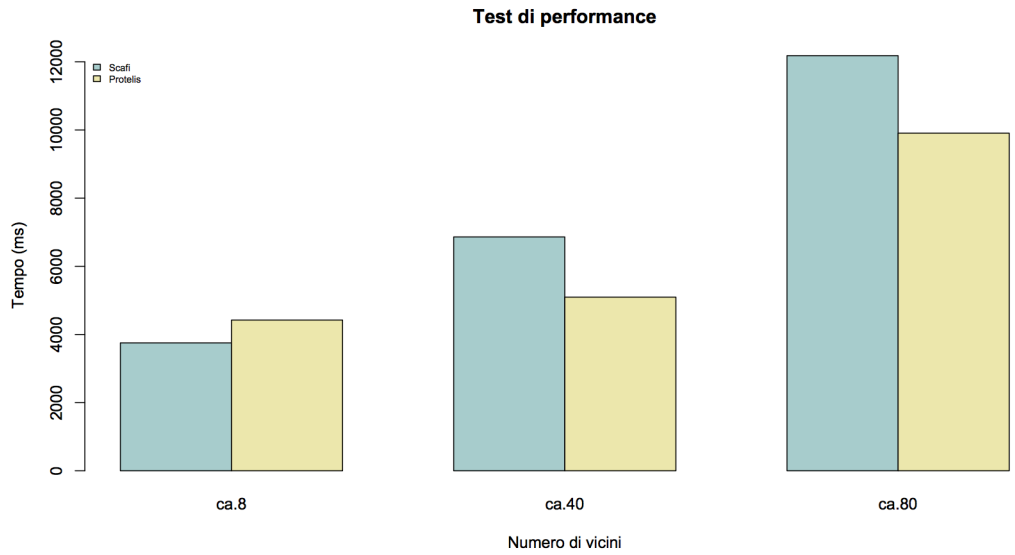

Figura 4.12: Confronto tra scafi e protelis al variare della cardinalità del vicinato dei nodi

Dal grafico si nota che per un numero di vicini circa pari a 8 le due piattaforme si equivalgono, aumentando considerevolmente il numero dei vicini fino a 40 possiamo notare che per entrambe non vi è stato un grosso aumento di tempo impiegato, quindi con questo aumento di vicini sia Scafi che Protelis presentano una buona scalabilità, con un piccolo vantaggio di Protelis di circa 2 secondi. Raddoppiando il numero dei vicini, quindi da 40 a 80, viene invece raddoppiato anche il tempo impiegato dalle piattaforme, da notare però che la differenza tra le due è rimasta di circa 2 secondi, questo significa che a fronte di tale aumento della cardinalità dei vicini Scafi e Protelis hanno scalato esattamente nello stesso modo.

Confronto variando la complessità dell'espressione da computare Questo secondo test è sempre incentrato sul voler calcolare il tempo impiegato per l'esecuzione di una simulazione di una determinata lunghezza (100) utilizzando le due piattaforme; il test è sempre basato su più coppie di simulazioni di Scafi e Protelis, tenendo fisso però, a differenza del primo test, la cardinalità del vicinato e variando la dimensione dell'espressione che queste due piattaforme dovranno computare ad ogni reazione del simulatore. Più precisamente si è fissato il numero dei vicini circa pari a 12 per ogni nodo e l'espressione da computare è costituita dalla somma del calcolo di un numero ("n") di gradienti preimpostato.

Si riporta nell'immagine 4.12 un grafico a barre riassuntivo dei risultati ottenuti anche per questo test.

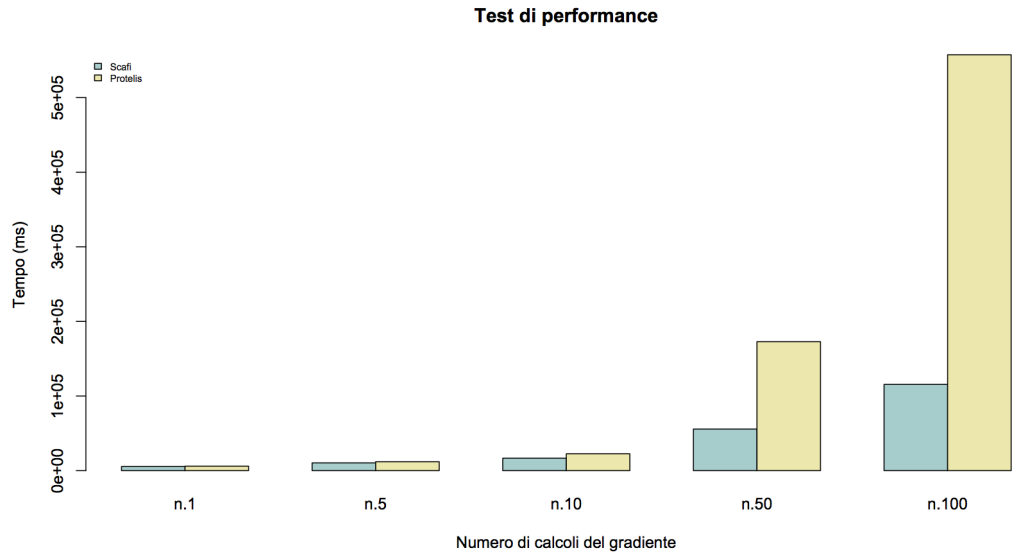

Figura 4.13: Confronto tra scafi e protelis al variare della complessità dell'espressione da computare

Sin da subito tale diagramma mette in evidenza una maggior scalabilità da parte del linguaggio scafi: fino al calcolo e la relativa somma di 10 gradienti per ogni reazione i due linguaggi si equivalgono, ma già con un

valore di "n" pari a 50 si nota che la piattaforma scafi ottiene un vantaggio di circa 2 minuti, fino ad un vantaggio di circa 7 minuti con "n" pari a 100.

Tuttavia i test riportati sono piuttosto indicativi in quanto entrambe le piattaforme sono in continua evoluzione e quindi soggette a future ottimizzazioni.

Questa pagina è lasciata intenzionalmente bianca.

Capitolo 5

Esecuzione di Scafi in Android

5.1 Introduzione

Le applicazioni Android sono di solito sviluppate utilizzando il linguaggio di programmazione *Java* e *Android Software Development Kit* (*Android SDK*). Il codice del programma viene compilato in bytecode java e successivamente tradotto nel così detto bytecode *Dalvik* (specifico formato di Android) per poi essere eseguito sul dispositivo tramite *Android Runtime (ART)*.

Nonostante java è l'unico linguaggio ufficialmente supportato e documentato per la creazione di applicazioni *SDK*, l'implicita compatibilità di Android con il bytecode java apre la porta a una varietà di altre tecnologie *Java Virtual Machine (JVM)*. Infatti, ogni linguaggio di programmazione JVM che è in grado di usufruire della *API Java* può essere usato come un'alternativa a quest'ultimo.

Il linguaggio di programmazione *scala* è una delle importanti tecnologie *JVM* che soddisfa tali requisiti. Questo linguaggio copre la maggior parte delle caratteristiche java e aggiunge una serie di funzionalità molto utili, come ad esempio: l'inferenza dei tipi, paradigma di programmazione funzionale, impliciti e traits (miglioramento delle interfacce Java) giusto per citarne alcuni.

Nella restante parte di questo breve capitolo vengono riportati i prerequisiti per poter sviluppare con scala su android, alcuni possibili modi volti alla creazione della configurazione necessaria allo sviluppo di un'applicazione

scala e infine si riportano tutti i passaggi volti alla creazione dell'ambiente nel quale si è effettuato un test di esecuzione del framework scafi unitamente a qualche cenno implementativo.

5.2 Prerequisiti

Sviluppare in Scala sulla piattaforma Android necessita di un grande impiego di risorse computazionali. Dal punto di vista della memoria RAM, il cui consumo può variare notevolmente a seconda dell'ambiente di sviluppo utilizzato e dall'uso di eventuali emulatori, il livello minimo consigliato anche per applicazioni di piccole dimensioni è 4GB. Inoltre, per quanto riguarda la CPU, considerando che il processo di compilazione del codice scala è piuttosto tedioso, una buona frequenza di clock agevolerebbe sicuramente questa fase. Dal punto di vista software invece si possono elencare i seguenti prerequisiti:

- *Java Development Kit (JDK)* - Android supporta java 7 quindi è richiesto il JDK7, qualora si disponesse di una versione più recente occorre specificare il target di compilazione a 1.7 e funzionerà senza problemi;

- *Android Software Development Kit (Android SDK)* - Android SDK è un pacchetto di strumenti che ci permetterà di sviluppare in android: compilatore per codice Java in bytecode Android, sistema di debugging delle applicazioni, emulatore Android e molto altro ancora;

- *Build tool* - occorre decidere quale build tool utilizzare, utile per la gestione delle dipendenze, del processo di compilazione, distribuzione e infine per la fornitura dell'APK; i principali tool sono *sbt* e *Gradle* caratterizzati entrambe dal relativo plugin per lo sviluppo in scala su Android;

- *Editor / IDE* - non ci sono particolari vincoli sulla scelta dell'IDE tuttavia quello consigliato è *IntelliJ*.

5.3 Project Setup

Il punto chiave per programmare in scala su android è riuscire ad ottenere prima di tutto una valida configurazione di partenza, a tal scopo vi sono diverse possibilità:

- *Template preimpostato* - la via più semplice e quella fortemente consigliata è clonare manualmente dei template di progetti già esistenti e opportunamente settati per lo sviluppo in scala;
- *Gitter8* - utility specifica per il reperimento dei template di progetto da determinati repository GitHub; Gitter8 è scritto in scala e può fornire diverse funzionalità basate su sbt, tuttavia può essere generalmente utilizzato per reperire template di un qualsiasi tipo di progetto;
- *Gen-Android* l'alternativa è crearsi il proprio progetto di setup, ad esempio il plugin *Android SDK Plugin for SBT* tramite il comando `gen-android <platform-target> <package-name> <name>` permette la creazione di un progetto base costituito da una semplice Activity, una classe di test, qualche risorsa e la corrispondente configurazione sbt.

Una volta ottenuto un progetto scheletro opportunamente configurato per la programmazione in scala si può iniziare da questo lo sviluppo semplicemente modificandolo in base al proprio fine.

5.4 Framework scafi

In questa sezione si riportano nel dettaglio tutti i passaggi necessari per l'esecuzione del framework scafi unitamente alla breve implementazione del test effettuato.

5.4.1 Configurazione progetto scheletro

Dopo aver sperimentato tutti i possibili modi per configurare un progetto di partenza esposti nella sezione precedente, si è scelto per questo intento di clonare un progetto scheletro già esistente:

1. Aprire Android Studio e scegliere l'opzione "Check out project from Version Control", selezionare "Git", e inserire il seguente link `https://github.com/macroid/macroid-starter` per la clonazione del progetto;

2. nella prossima GUI selezionare nella scelta del Project SDK la piattaforma Android, p.es. Andorid API 23 Platform;

3. una volta effettuato il clone e terminato il processo di build automation aprire la GUI per l'impostazione dei moduli con "Open Module Settings", selezionare la voce "Modules" nella parte sinistra, selezionare il modulo principale appena importato e nella scheda "Dependencies" relativamente alla voce "Module SDK" impostare la piattaforma Android, p.es. "Android API 23 Platform";

4. sempre in questa GUI selezionare la scheda "Sources" e assicurarsi che vi sia il solo folder "src/main/scala" marcato come "Sources", qualora ve ne siano altri occorre togliergli questa marcatura;

5. settare la versione del bytecode del progetto alla 1.8: aprire la GUI delle preferenze, nella parte sinistra selezionare "Build, Execution, Deployment", "Compiler" e in "Java Compiler" impostare 1.8 in "Project byetcode version";

6. infine creare da "Edit Configurations" una nuova configurazione di esecuzione selezionando "Android Application" e impostando il modulo importato; a questo punto si è in grado di eseguire l'applicazione.

5.4.2 Esecuzione di scafi

Una volta eseguiti tutti i passaggi sopraesposti abbiamo a disposizione un progetto di "natura" scala da poter modificare ed evolvere a secondo dei propri fini. Nel nostro caso prima di tutto occorre importare il modulo core di scafi:

1. clonare il repository di scafi (`https://scostanzi@bitbucket.org/metaphori/scafi-tesi.git`);

2. aprire la GUI per l'impostazione dei moduli con "Open Module Settings", selezionare la voce "Modules" nella parte sinistra, selezionare l'opzione per aggiungere un nuovo modulo attraverso "Import Module", selezionare il modulo core di scafi dal file system e successivamente scegliere l'opzione "Create module from exinting sources", a questo dare conferma ad ogni prossima GUI fino al completamento dell'importazione;

3. una volta importato con successo qualora fossero presenti errori nei sorgenti accettiamo la proposta di Android Studio di settare la "natura" scala a tale mudulo;

4. infine settare al progetto principale la dipendenza da questo modulo core di scafi.

Eseguiti tutti questi passaggi si è ora operativi per poter effettuare l'esecuzione di questo framework sulla piattaforma Android. Per sperimentare questa esecuzione si è effettuato un piccolo test costituito da una modifica nella classe `MainActivity.scala` al metodo `onCreate` di cui date le piccole dimensioni se ne riporta direttamente il codice (5.4.2).

```
override def onCreate(savedInstanceState: Bundle) = {
    super.onCreate(savedInstanceState)

    val net = simulatorFactory.gridLike(
      n = 10,
      m = 10,
      stepx = 1,
      stepy = 1,
      eps = 0.1,
      rng = 1.2
    )

    net.addSensor(name = "sensor", value = 0)
    net.chgSensorValue(name = "sensor", ids = Set(1), value = 1)
    net.addSensor(name = "sensor2", value = 0)
    net.chgSensorValue(name = "sensor2", ids = Set(98), value = 1)
```

```
    net.addSensor(name = "obstacle", value = false)
    net.chgSensorValue(name = "obstacle", ids =
        Set(44,45,46,54,55,56,64,65,66), value = true)
    net.addSensor(name = "label", value = "no")
    net.chgSensorValue(name = "label", ids = Set(1), value = "go")

    var v = java.lang.System.currentTimeMillis()

    net.executeMany(
      node = NbrCount,
      size = 100000,
      action = (n,i) => {
        if (i % 1000 == 0) {
          println(net)
          val newv = java.lang.System.currentTimeMillis()
          println(newv-v)
          v=newv
        }
      }
    )
    ....
    ....
    ....
}
```

Questa implementazione utilizza un ambiente di simulazione messa a disposizione dal framework scafi, in particolare abbiamo una prima fase in cui tramite la primitiva `gridLike` si crea la rete dei nodi, poi successivamente si impostano per ciascuno di essi i vari sensori e i relativi valori. Infine grazie alla primitiva `executeMany` viene avviata la simulazione che in questo caso come si può notare dal valore assegnato al primo parametro, `NbrCount`, si basa sul calcolo del numero dei vicini di ciascun nodo. Più precisamene `NbrCount` è costituito da un oggetto scala che estende dall'entità `ExecutionTemplate` implementando il metodo astratto `main` col comportamento aggregato desiderato. Eseguendo questa piccola applicazione noteremo che il framework esegue correttamente sul dispositivo

android, in quanto appena avviata la `MainActivity` il flusso di controllo di quest'ultima viene passato alla simulazione di scafi riportando in standard output i risultati relativi alla computazione del programma aggregato.

Questa pagina è lasciata intenzionalmente bianca.

Capitolo 6

Conclusioni

In questo lavoro di tesi si è partiti con l'obiettivo ben preciso di estendere la toolchain per aggregate programming attraverso l'integrazione del framework scafi con il simulatore Alchemist e con una piattaforma di esecuzione.

La prima fase di sviluppo è stata caratterizzata quindi, dallo studio di queste tre entità al fine di calarsi nel loro contesto fino al punto da riuscire a pensare ad una possibile strategia di integrazione.

Una volta raggiunto un buon grado di conoscenza si è iniziato dall'integrazione di scafi con la piattaforma di esecuzione, passando attraverso tutte le fasi del processo di produzione del software, con particolare attenzione alla fase di analisi in quanto caratterizzata da uno studio di fattibilità sull'intento da raggiungere. L'integrazione in oggetto si è conclusa con esito positivo, dando vita ad una piattaforma che offre la possibilità di creare sistemi in ambito di Spatial Computing [9] [4] giovando dei vantaggi offerti dalle primitive scala del framework scafi, che come si è notato anche dal confronto riportato nella sezione 3.3 del terzo capitolo, facilitano enormemente la descrizione e la programmazione del comportamento aggregato desiderato.

Successivamente ci si è concentrati sull'integrazione col simulatore alchemist, che grazie alla somiglianza col caso precedente è stato possibile usare un analogo processo di sviluppo del software e riutilizzare alcune strategie risolutive accelerandone il tal modo lo costruzione.

Anche questa integrazione si è conclusa con esito positivo raggiungendo in tal modo l'obiettivo prefissato di questo lavoro di tesi.

In particolare la toolchain ottenuta offre la possibilità di programmare con primitive scala un sistema virtuale, effettuarne tutte le simulazioni necessarie ed eseguire poi tale sistema in un contesto reale lasciando invariato il programma aggregato di ogni nodo.

Per lo sviluppo di entrambi questi sistemi si è utilizzato il linguaggio **scala** [17] come tecnologia di programmazione, analogamente alla natura del framework scafi. In tal modo si è potuto giovare del suo *supporto all'ereditarietà multipla* per l'implementazione del pattern adaper, e dellla sua *sintassi espressiva* e *concisa*, che come già riportato nei precedenti capitoli, ha facilitato l'implementazione di diverse parti dei due progetti. Infine avendo effettuato in entrambi questi lavori una integrazione di un sistema di natura scala con uno di natura java si può affermare che salvo qualche piccola incompatibilità i due linguaggi presentano un ottima interoperabilità.

I sistemi sviluppati considerando anche l'ambito in cui essi si calano presentano diverse possibili espansioni, si riporta di seguito alcune considerazioni su due importanti sviluppi futuri.

Un fronte completamente inesplorato nel lavoro effettuato è quello delle comunicazioni opportunistiche. A tal proposito Android pone diverse limitazioni: per via della grande attenzione alla sicurezza dell'utente di fatto si limita molto la libertà nell'uso delle varie interfacce a bordo dei device, richiedendo, nel migliore dei casi, invasive conferme ad ogni connessione, come avviene per il bluetooth o per il wifi direct. Tuttavia non sembra essere questa la direzione intrapresa: costituisce un esempio importante il grande rivale di Android, che supporta nativamente le reti *mesh* nelle ultime versioni del proprio sistema operativo.

Un possibile sviluppo futuro è proprio relativo a questo ambito, in particolare qualora si raggiungesse una maggiore apertura e interoperabilità si potrebbe effettuare un ulteriore espansione alla toolchain scafi realizzata, attraverso i seguenti due passaggi:

1. creazione di un supporto per aggregate programming *che sfrutta le comunicazioni opportunistiche per reperire e gestire le informazioni*

necessarie a tale paradigma per effettuare la computazione;

2. successiva integrazione di questo supporto con il framework scafi, analogamente all'integrazione con la piattaforma di esecuzione effettuata in questo lavoro.

Più precisamente questo supporto si andrebbe ad inserire nel layer *Execution* della toolchain mostrata in figura 6.1 a fianco della piattaforma, in quanto costituirebbe una sua alternativa per il reperimento e la gestione delle informazioni (p.es. informazioni sul vicinato) necessarie ai singoli nodi per poter effettuare la computazione del programma aggregato.

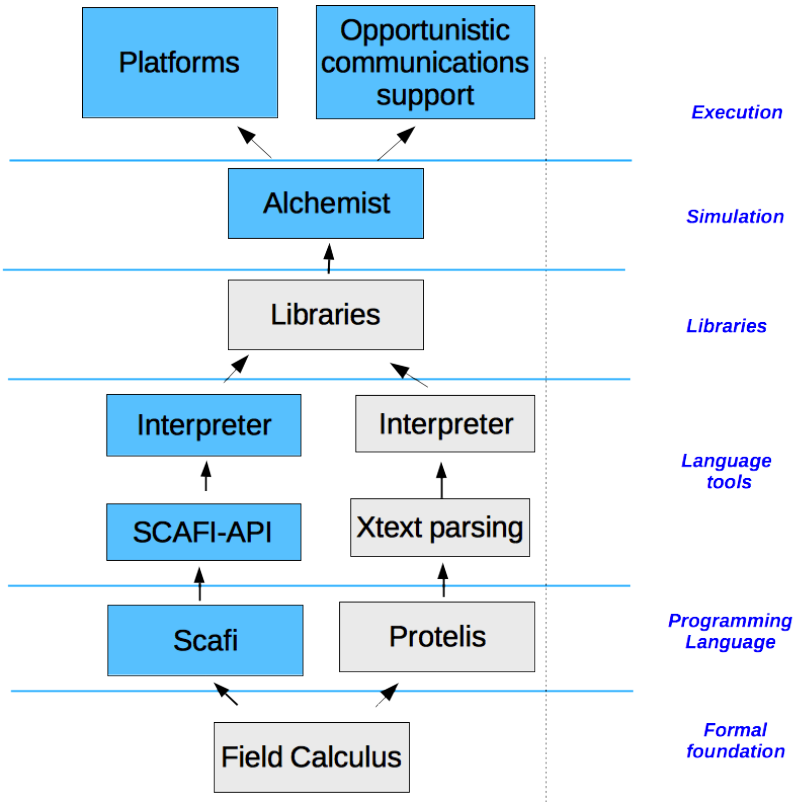

Figura 6.1: Possibile estensione alla toolchain Aggregate Programming

Infine, considerando che la piattaforma di esecuzione utilizzata è in grado di gestire lo scambio di informazioni anche relativamente a nodi risiedenti su dispositivi android e dato l'esito positivo dell'esecuzione del framework scafi sulla piattaforma android (5.4), si evince la possibilità di poter programmare in termini aggregati avvalendosi del framework scafi anche nodi destinati ad essere incarnati in dispositivi di natura android. Tuttavia, l'ambiente di lavoro necessario a questo sviluppo è piuttosto complesso in quanto costituito da diverse tecnologie, che spesso possono presentare dei punti di incompatibilità, da diverse librerie di terze parti e da molteplici dipendenze tra progetti di diversa natura. Date queste condizioni un possibile sviluppo futuro è creare tramite un tool di *build automation* [10] [11] un sistema costituito da un insieme di task atti ad automatizzare tutta la gestione delle librerie necessarie e le varie dipendenze fra i progetti, in modo che qualsiasi eventuale sviluppatore possa avere un ambiente consistente e sicuro nel quale poter programmare.

Capitolo 7

Ringraziamenti

Un sincero ringraziamento va a tutti coloro che mi hanno aiutato in vario modo a raggiungere questo importante traguardo.

Desidero ringraziare in primo luogo il prof. Mirko Viroli per la disponibilità e la cortesia con cui mi ha aiutato durante l'attività sperimentale e la stesura di questa tesi, ed inoltre l'Ing. Danilo Pianini per il supporto fornito alla realizzazione dell'elaborato finale.

Un sentito ringraziamento ai miei fratelli, Eleonora ed Emanuele, che con il loro sostegno morale ed economico mi hanno permesso di raggiungere questo importante obiettivo e che mi hanno sopportato nei momenti di difficoltà.

Grazie a tutti gli amici con cui ho condiviso questi piacevoli anni di studio, ed in particolare Simone, compagno di innumerevoli progetti.

Sinceri ringraziamenti a tutti i miei amici, in particolar modo a Sara, Alessandro e Francesco, per il sostegno dato e la fiducia mai mancata.

Questa pagina è lasciata intenzionalmente bianca.

Bibliografia

[1] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[2] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[3] J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–33, 2015.

[4] J. Beal, M. Viroli, and F. Damiani. Towards a uni ed model of spatial computing. *Proc. 7th Spatial Computing Workshop (SCW 14)*, 2014. `www.spatial-computing.org/_media/scw14/scw2014_p5.pdf`.

[5] R. Casadei. *Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields*. Tesi di laurea, Università di Bologna - Scuola di Ingegneria e Architettura Campus di Cesena - Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche, 2014/2015.

[6] S. S. Clark, J. Beal, and P. Pal. Distributed recovery for enterprise services. pages 1–10.

[7] G. Erich and Al. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[8] D. M. et al. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[9] J. B. et al. Organizing the aggregate: Languages for spatial computing. *Formal and Practical Aspects of Domain-Specic Languages: Recent Developments, M. Mernik, ed., IGI Global*, pages 436–501, 2013.

[10] M. Fowler. Continuous integration (original version). pages 1–9, 2000. `http://www.martinfowler.com/articles/originalContinuousIntegration.html`.

[11] M. Fowler. Continuous integration. pages 1–14, 2006. `http://www.martinfowler.com/articles/continuousIntegration.html`.

[12] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A*, 104:1876–1889, 2000.

[13] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol*, 18(4):1–56, 2009.

[14] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, pages 1–14, 2013.

[15] D. Pianini, M. Viroli, and J. Beal. Protelis: Practical aggregate programming. pages 1–8.

[16] M. Viroli, E. Nardini, G. Castelli, M. Mamei, and F. Zambonelli. A coordination approach to adaptive pervasive service ecosystems. *In 1st Awareness Workshop "Challenges in achieving self-awareness in autonomous systems" (AWARE 2011), SASO 2011*, 2011.

[17] D. Wampler and A. Payne. *Programming Scala*. O'Reilly Media, 2009.

[18] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.

Elenco delle figure