Alma Mater Studiorum · Università di Bologna
Campus di Cesena

SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

UN FRAMEWORK PER SIMULAZIONE E SVILUPPO DI SISTEMI AGGREGATI DI SMART-CAMERA

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Presentata da
FEDERICO PETTINARI

Correlatore
Ing. DANILO PIANINI

Seconda Sessione di Laurea
Anno Accademico 2019-2020

Abstract

Le smartcam sono telecamere capaci di rilevare entità di interesse, e sono generalmente impiegate nella sorveglianza. Con l'avanzamento della tecnologia si è creata la possibilità di renderle mobili e dotarle di mezzi trasmissivi. Questo ha aperto grandi opportunità per la coordinazione di reti di smartcam con lo scopo di aumentarne l'efficienza. Lo sviluppo di tali sistemi può beneficiare dall'impiego di specifici strumenti per la simulazione e programmazione in modo da facilitarne l'implementazione e la valutazione.

Nell'ambito di questa tesi viene adottato Alchemist, uno degli ambienti che consentono simulazioni di ≪Collective Adaptive System≫. Vengono quindi illustrate e discusse la fasi di sviluppo di un nuovo modulo software volto ad estendere Alchemist con il supporto alle smartcam. Inoltre vengono riprodotti alcuni algoritmi presenti in letteratura ed implementati di nuovi con l'ausilio dell'≪Aggregate Programming≫, un paradigma per la programmazione di sistemi distribuiti. Questi algoritmi hanno lo scopo di creare un sistema adattativo di smartcam che comunicano fra di esse per coordinarsi nei movimenti, e nella decisione degli obiettivi da raggiungere. L'obiettivo è l'incremento del livello di copertura della zona alla quale sono preassegnate, ed il raggiungimento di un certo livello di ridondanza nell'osservazione di eventuali entità di interesse. Infine viene effettuata una validazione empirica degli algoritmi attraverso la simulazione di alcuni scenari.

Indice

Elenco delle figure

Elenco dei listati

Introduzione

Un importante problema che sorge nell'ambito dell'automazione di sistemi relativi alla sicurezza, sorveglianza e riconoscimento è quello di monitorare entità di interesse situate in un determinato ambiente. Si pensi ad esempio ad eventi pubblici con grandi affluenze di visitatori dove prevenire o reagire a situazioni di pericolo (come possono essere atti di vandalismo o terrorismo) ha un'importanza vitale. Negli scenari più ristretti e semplici un preposizionamento adeguato dei sensori può già essere sufficiente a garantire una buona copertura dell'area di interesse [13]. In situazioni più complesse, invece, un certo numero di fattori potrebbe impedire questa possibilità. L'area da monitorare potrebbe essere sconosciuta a priori, o troppo ampia perchè la disposizione di telecamere fisse possa essere economicamente concretizzabile, oppure non essere fisicamente accessibile in anticipo. In questi casi si rende vantaggioso l'impiego di smartcam mobili e dinamiche.

Si intende per smartcam qualunque telecamera programmabile capace di rilevare determinate entità di interesse entro il campo visivo. In tutti gli scenari discussi sia in questa tesi, che negli articoli di riferimento, si assumerà che le smartcam siano dotate di mobilità come possibilità di rotazione entro angoli predeterminati e possibilità di libero movimento nello spazio o lungo percorsi fissi prestabiliti. Si pensi, ad esempio, a telecamere montate su droni.

Per questioni economiche si rende necessaria l'usufruizione di una piattaforma che ne possa permettere la simulazione prima di un eventuale investimento in hardware. Purtroppo le soluzioni esistenti illustrate in sezione 1.1.1 sono poche e presentano molte limitazioni. Si rende quindi conveniente l'estensione di un sistema esistente. Alchemist, descritto in sezione 1.2, si presenta come un perfetto candidato in quanto offre una piattaforma matura e scalabile per la simulazione di ≪Collective Adaptive

System≫(CAS).

I sistemi di smartcam rientrano infatti nella categoria dei CAS. Esistono diversi approcci emergenti volti all'ingegnerizzazione lato software dei CAS che sarebbe interessante applicare allo scenario delle smartcam[20]. In particolare l'≪Aggregate Programming≫[1] definisce un paradigma di programmazione piuttosto promettente basato sulla visione dei dispositivi come insieme invece che come entità singole. L'idea è quella di poter codificare algoritmi di coordinazione che siano robusti ed affidabili, avvantaggiandosi delle astrazioni fornite dalla programmazione aggregata. Questo permetterebbero di avere una solida base di partenza per produrre codice riutilizzabile ed indipendente dall'hardware, separando la logica di comportamento dei dispositivi dai dettagli implementativi quali, ad esempio, le tecnologie di comunicazione, la struttura della rete, la gestione dei sensori e degli attuatori, etc.

Nell'ambito di questa tesi si è quindi provveduto alla creazione di un framework per la simulazione e sviluppo di sistemi aggregati di smartcam. Questo contributo include alcuni esempi di algoritmi ed approcci che sfruttano la comunicazione fra smartcam per aumentarne l'efficienza per quanto riguarda la copertura di una zona arbitraria, e l'individuazione e tracciamento di entità di interesse situate al suo interno. Riassumendo, sono stati perseguiti i seguenti obiettivi:

- Creazione di un modulo per la simulazione di smartcam in Alchemist (capitolo 2).
- Riproduzioni di algoritmi esistenti in letteratura (sezione 3.1).
- Sviluppo di nuovi algoritmi (sezione 3.2).
- Valutazione sperimentale degli algoritmi tramite simulazione (capitolo 4).

Infine tutto questo ha richiesto l'apprendimento e l'impiego di due nuovi linguaggi di programmazione, Kotlin e Protelis, di un nuovo paradigma definito dall'aggregate programming, e di tecnologie di DevOps come Gradle[1] e Travis-CI[2].

[1] `https://gradle.org/`
[2] `https://travis-ci.org/`

Capitolo 1

Background

In questo capitolo viene proposta una breve descrizione dei lavori correlati, e delle principali tecnologie utilizzate.

1.1 Stato dell'arte

I primi prototipi di telecamere dotate di intelligenza furono sviluppati negli anni '80 [18], ma è solo grazie alle recenti innovazioni tecnologiche che si stanno effettuando sperimentazioni basate su «Unmanned Aerial Vehicles»(UAV)[24]. La coordinazione di smartcam mobili è quindi un argomento piuttosto giovane nell'ambito della ricerca scientifica.

Uno dei problemi che sorgono in questo contesto è il *k-coverage*. Esterle e Lewis definiscono questo problema in «Online multi-object *k-coverage* with mobile smart cameras» [7] e mostrano qualche approccio di programmazione distribuita per affrontarlo. Si intende raggiunto il livello di *k-coverage* per un determinato oggetto (o regione) se esso è coperto da almeno k sensori; l'obiettivo è massimizzare il tempo ed il numero di oggetti per i quali si è raggiunta la *k-coverage* lungo un determinato periodo. Lo scopo è quello di fornire molteplici rilevazioni di un fenomeno con una certa ridondanza. Tuttavia si noti che, con questa definizione, in uno scenario contenente un rapporto fra il numero di obiettivi e sensori minore di k, è automaticamente preferibile perdere qualche obiettivo e raggiungere la *k-coverage* negli altri piuttosto che cercare di monitorarli tutti.

Il lavoro di Esterle et al estende la ricerca sul ≪Cooperative Multi-robot Observation of Multiple Moving Targets≫(CMOMMT) [13], definito come un problema NP-Hard. In questo contesto l'obiettivo è massimizzare il tempo durante il quale ogni bersaglio è coperto da almeno un robot. I robot hanno un campo sensoriale di 360° ma limitato e soggetto a rumore.

1.1.1 Simulazione di sistemi di smartcam

Al meglio della nostra conoscenza esiste un solo tool per la ricerca scientifica in ambito smartcam. CamSim[8] è un simulatore open-source[1] per sistemi di smartcam sviluppato nell'ambito del progetto ≪EPiCS≫[2] ed impiegato per il testing e sviluppo degli algoritmi illustrati nell'articolo di Esterle et al [7].

Sebbene il progetto sia stato utilizzando in alcuni lavori [7], è frutto di un'attività di ricerca conclusa e risulta di difficile estensione. L'ultimo contributo sostanziale a CamSim risale infatti al 2015.

1.2 Alchemist

Uno degli ambienti che consentono simulazioni di CAS a larga scala è Alchemist[15]: un simulatore open-source[3] ad eventi discreti[26] orientato alla chimica. Internamente si basa su una versione ottimizata dell'algoritmo ≪SSA≫ di Gillespie[10] denominato ≪Next Reaction Method≫[9]. Il modello del dominio, seppur inizialmente basato sulla chimica, nel tempo è stato evoluto per essere estensibile. Alchemist è infatti predisposto per poter simulare qualsiasi modello compatibile con il suo meta-modello (Figura 1.1) che si basa sulle seguenti entità:

- *Molecule* - Rappresenta il nome di un dato generico. Può essere pensato come il nome di una variabile con riferimento ai linguaggi di programmazione imperativi.
- *Concentration* - Il valore associato ad una *Molecule*.

[1] `https://github.com/EPiCS/CamSim`

[2] `http://web.archive.org/web/20190920142212/http://www.epics-project.eu/`

[3] `http://alchemistsimulator.github.io/`

- *Node* - Un contenitore di *Molecule* e *Reaction* che vive all'interno di un *Environment*
- *Environment* - L'astrazione che rappresenta lo spazio. È un contenitore di *Node* e ne gestisce l'aspetto fisico come la loro posizione ed opzionalmente il movimento.
- *Linking rule* - Una funzione dello stato corrente dell'*Environment* che associa ad ogni *Node* un *Neighborhood*.
- *Neighborhood* - Dato un *Node* (il centro) ne rappresenta il vicinato sottoforma di un insieme di *Node*.
- *Reaction* - Qualsiasi evento in grado di cambiare lo stato dell'*Environment*. È caratterizzata da una lista di *Action*, una lista opzionale di *Condition* ed una distribuzione temporale. Il funzionamento è illustrato in figura 1.2. La frequenza alla quale le *Action* sono azionate dipende dalle *Condition*, che devono essere verificate ogni volta, e dalla distribuzione temporale.
- *Condition* - Un generico predicato che dipende dall'*Environment*.
- *Action* - Modella la conseguenza di un'evento, ad esempio un cambiamento all'interno dell'*Environment*.

Nonostante la nomenclatura, è evidente come i concetti sopra-riportati siano abbastanza generici da permettere la simulazioni di scenari appartenenti a domini differenti dalla chimica. Il concetto di *Incarnation* di Alchemist infatti rappresenta il punto di partenza nella definizione di uno scenario di simulazione e può essere pensato come un'implementazione concreta del modello, che deve obbligatoriamente offrire una definizione di *Concentration* ed opzionalmente definizioni specifiche di *Condition*, *Action*, *Node*, ed *Environment*. Attualmente la distribuzione di Alchemist include quattro *Incarnation*:

- *Protelis* - Ha lo scopo di simulare reti di dispositivi che eseguono programmi dell'omonimo linguaggio descritto in sezione 1.3.1.
- *SAPERE*[25] - Modella un sistema di calcolo distribuito ispirato ad ecosistemi naturali.

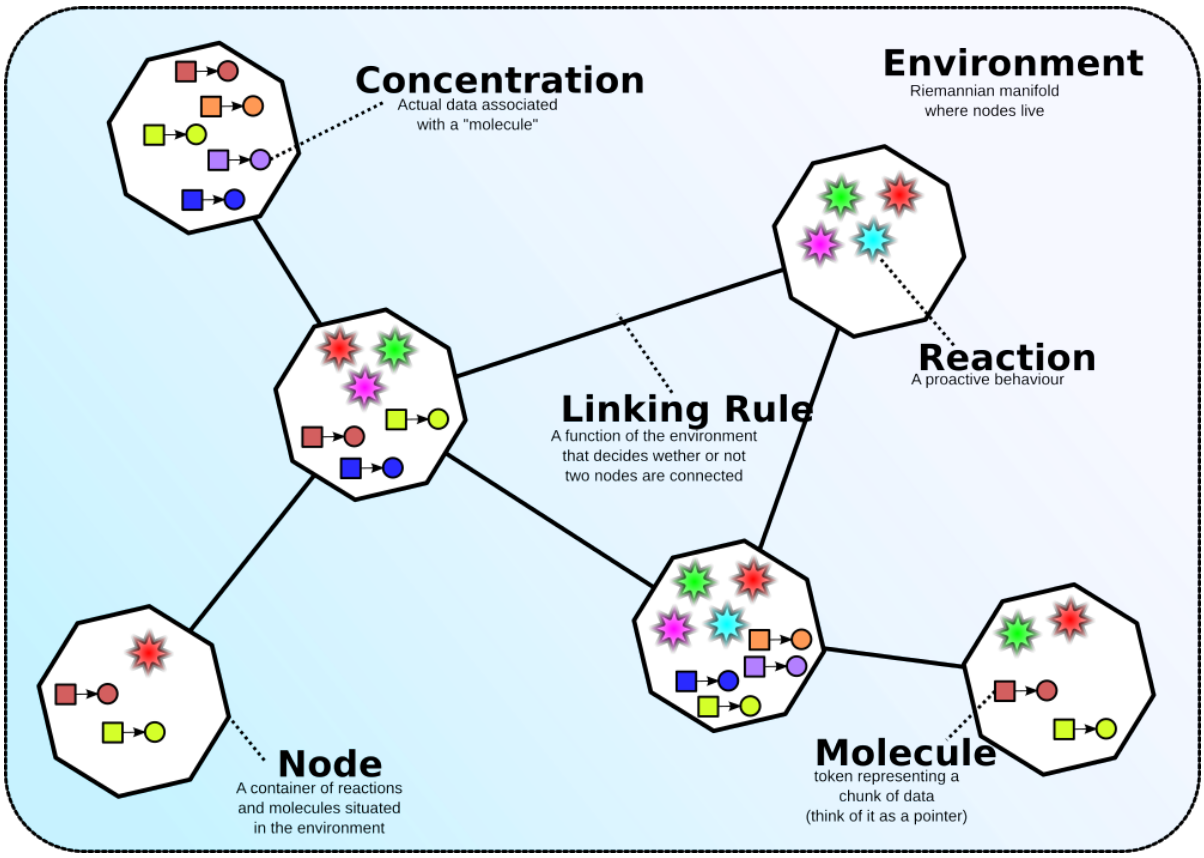

Figura 1.1: Il metamodello di Alchemist

- *Biochemistry*[11] - La prima incarnazione di Alchemist che modella ambienti multicellulari.
- *Scafi*[22] - Ha lo scopo di simulare reti di dispositivi che eseguono programmi basati sull'omonima libreria. *Scafi*[4] è una libreria Scala che offre un framework per la programmazione aggregata. Come Protelis, anch'essa è basata sul Field Calculus.

1.3 Programmazione aggregata

L'unità base della computazione è tradizionalmente identificata in un singolo dispositivo connesso col resto del mondo attraverso input ed output. Questo punto di vista

[4] `https://scafi.github.io`

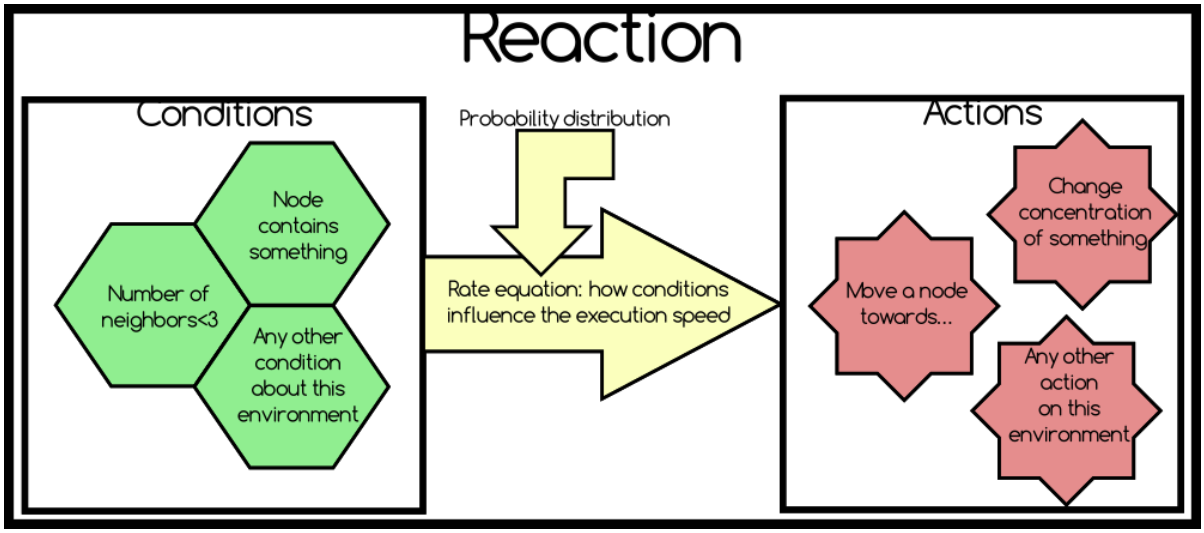

Figura 1.2: Diagramma del funzionamento delle reazioni di Alchemist

ha inevitabilmente condizionato gli strumenti e le metodologie di sviluppo moderne con la conseguenza che molti aspetti di interazione, comunicazione e coordinazione di una rete di dispositivi sono spesso strettamente legati all'implementazione dell'applicazione che ne fa uso. Questo limita la modularità, riusabilità e manutenzione del software.

L'«Aggregate Programming» [1] offre un paradigma di programmazione che consente di esprimere in modo compatto e modulare algoritmi distribuiti complessi. L'unità base della computazione non è più il singolo dispositivo ma l'insieme di dispositivi che compongono il sistema. Vengono astratte caratteristiche come la posizione, il numero di dispositivi e la configurazione della rete. Questo consente al designer di concentrarsi sugli aspetti di coordinazione senza preoccuparsi, ad esempio, dei dettagli implementativi legati all'hardware o dei protocolli di comunicazione che verranno gestiti separatamente. La computazione avviene tramite la manipolazione di strutture dati distribuite dette «Field». Queste strutture si basano sul concetto di campo magnetico nella fisica, ed associano un valore al tempo nel quale è stato rilevato, locazione nello spazio, e dispositivo che lo ha prodotto.

Esistono altri sistemi per la coordinazione di collettivi quali, ad esempio, SAPERE [25], Biochemical Tuple Spaces [21], e SwarmLinda [19]. Purtroppo questi sistemi, ed i precedenti, sono carenti negli aspetti chiave riguardanti la modularità e riusabilità.

Protelis [16] al contrario costituisce un valido approccio.

1.3.1 Il linguaggio Protelis

Protelis[5] [16] è un linguaggio di programmazione funzionale dinamicamente tipato basato sul «Field Calculus» [23] [4] e costituisce uno strumento di sviluppo per la programmazione aggregata.

In Protelis si assume che tutti i dispositivi facenti parte del CAS che si sta programmando eseguano lo stesso programma, od una versione compatibile. Tale programma viene eseguito con una certa frequenza, alternando tempi di calcolo a tempi di riposo. Ogni esecuzione è chiamata "Computational Round".

Protelis supporta un insieme ristretto di «first-class citizens» quali:

- Variabili
- Funzioni.
- Valori scalari quali:
  - Tuple. Una tupla in Protelis è una lista ordinata ed immutabile di oggetti di qualsiasi tipo ed offre funzioni come l'unione di tuple (Inteso dal punto di vista insiemistico), somma e sottrazione elemento per elemento, mappatura e riduzione.
  - Oggetti Java.
  - Booleani.
  - Stringhe.
  - Oggetti Java.
  - Numeri.
- *Neighbouring Field*. Rappresentano viste locali del "campo computazionale" generate a partire da valori scalari.

[5] `https://protelis.github.io`

Sono inoltre supportati operatori di base della programmazione aggregata quali *rep*, *nbr*, *if* e *mux* [4].

In Protelis non esiste il concetto di classe della programmazione ad oggetti tradizionale nè il valore *null*. Nonostante questo, è un linguaggio la cui esecuzione è delegata alla Java Virtual Machine (JVM), ed è in grado di interoperare con Java, che è basato su classi ed oggetti.

Capitolo 2

Design ed implementazione del modulo alchemist-smartcam

In questo capitolo verrà discusso il processo di produzione del modulo alchemist-smartcam, volto ad estendere Alchemist con il supporto alla simulazione di sistemi di smartcam.

2.1 Analisi

In primo luogo sono stati precisati i requisiti del nuovo modulo e successivamente è stato definito il modello del dominio come raffigurato in figura 2.2. In figura 2.1 è illustrato un caso d'uso raffigurante le possibili interazioni con il software.

2.1.1 Requisiti

I requisiti del nuovo modulo per la simulazione di smartcam suddivisi fra funzionali e non, sono elencati di seguito.

Requisiti funzionali

- Simulazione telecamere: possibilità di movimento, rotazione, visione di oggetti e scrittura di algoritmi custom.

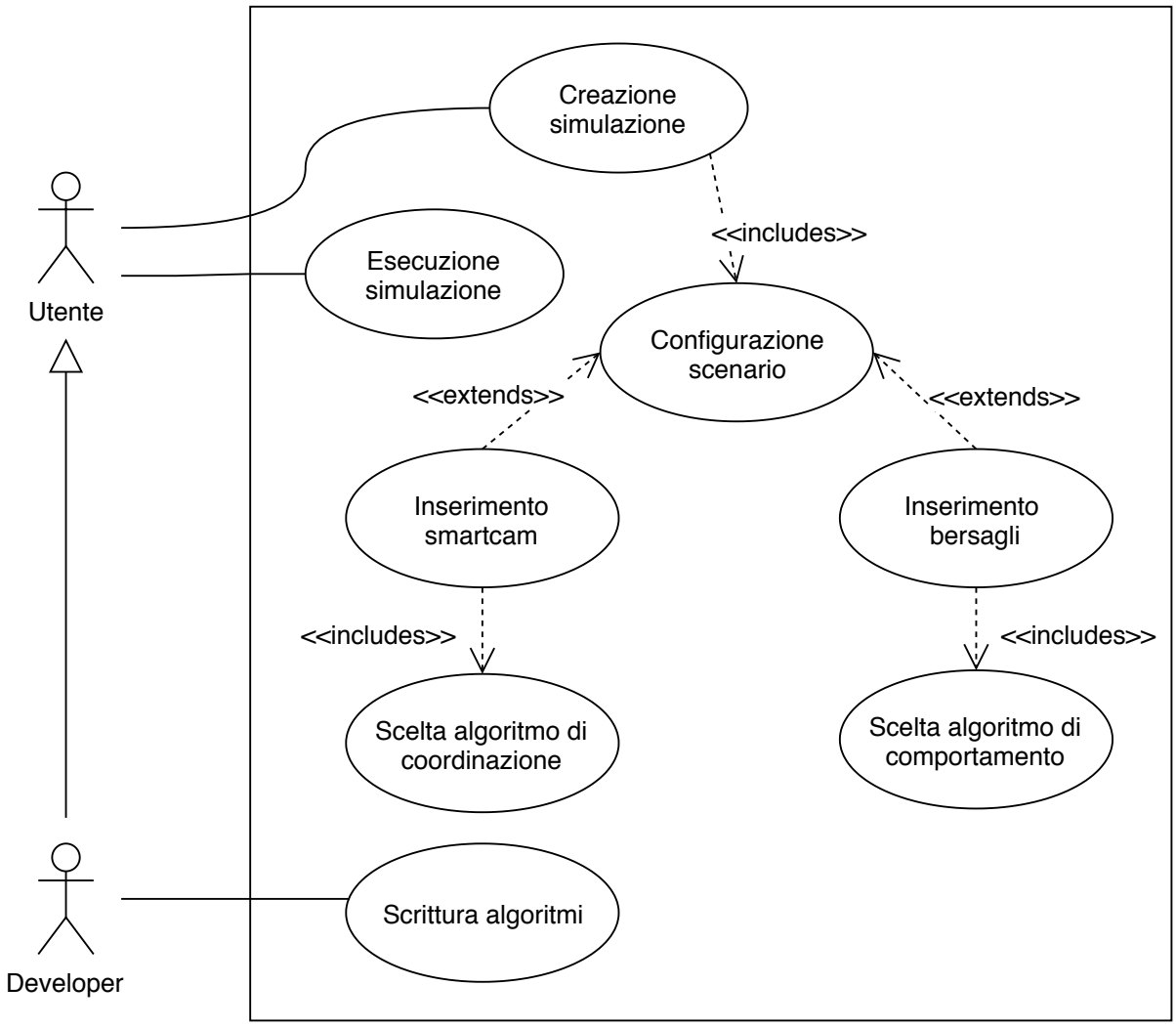

Figura 2.1: Diagramma di un caso d'uso del modulo smartcam integrato con Alchemist

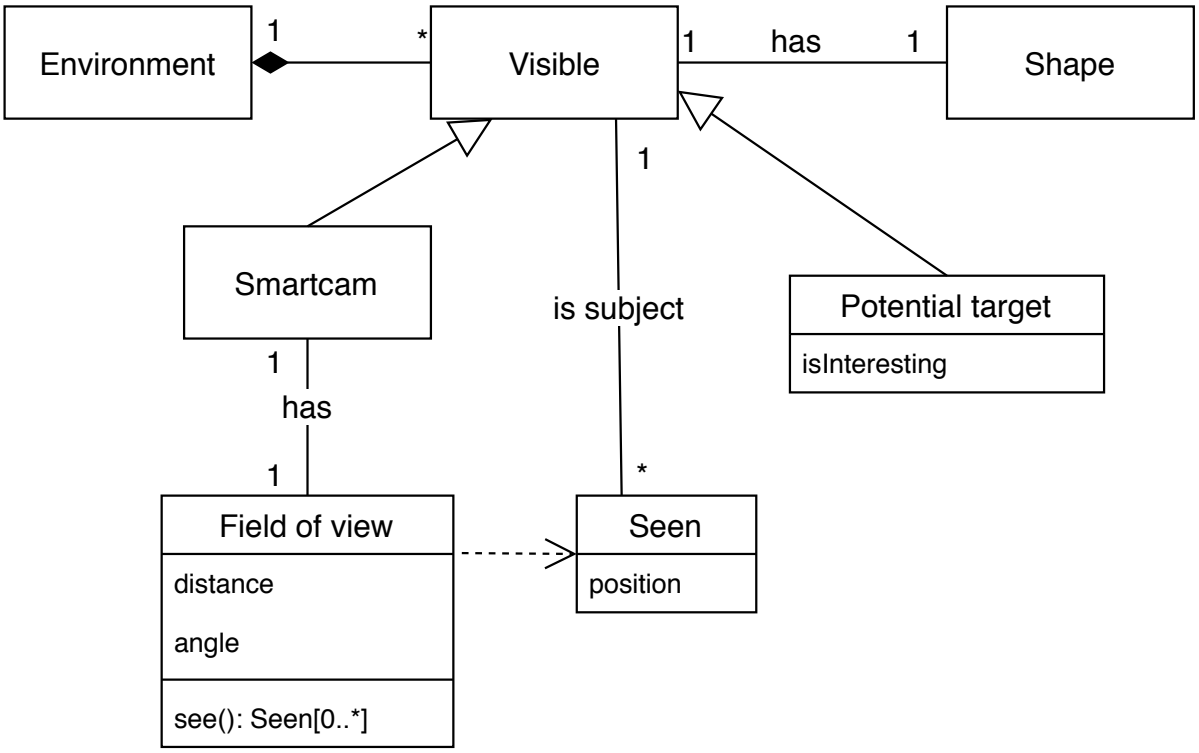

Figura 2.2: Diagramma raffigurante il modello del dominio risultato dalla fase di analisi

- Simulazione oggetti generici: potenziali bersagli delle telecamere con forme arbitrarie e possibilità di movimento.
- Supporto per la programmazione aggregata delle telecamere.

Requisiti non funzionali

- Retrocompatibilità con simulazioni esistenti.
- Estensibilità con particolare riguardo verso le caratteristiche delle telecamere ed in special modo la visione.
- Integrabilità con componenti esistenti, in particolare quelli riguardanti le simulazioni di folle e pedoni realistici.
- Mantenimento del livello attuale di performance.

2.2 Progettazione

Per favorire l'integrabilità, il ruolo di *Visible* presente nel modello del dominio (Figura 2.2) è stato assegnato al *Node* di Alchemist mentre *Environment* al suo omonimo. Purtroppo il simulatore non disponeva di alcun tipo di supporto al concetto di forma ed orientamento (rotazione) di un oggetto nello spazio e si è quindi resa necessaria l'aggiunta di queste funzionalità. La progettazione è quindi divisa in due ambiti: ciò che riguarda la geometria e le smartcam.

2.2.1 Geometria

Sono state considerate due strade principali, entrambe con importanti pro e contro: supportare le forme dei nodi e la gestione delle collisioni tramite apposite *Action* e *Reaction* oppure estendere *Environment*.

La prima opzione sarebbe stata molto semplice e veloce da implementare al rischio però di inconsistenze dovute alla mancanza di un punto centrale di coordinazione dei movimenti. Diversi sviluppatori avrebbero potuto creare diverse *Action* con assunzioni

differenti per quanto concerne le regole fisiche di una simulazione e quindi con un costo crescente per quanto riguarda la manutenzione. Oltre a questo, non avendo le *Action* una visione globale dell'ambiente, future ottimizzazioni sarebbero state molto difficili da applicare.

Estendere *Environment* avrebbe sopperito a tutti i punti deboli della prima opzione ma sarebbe stato più complesso, soprattutto riguardo al requisito di retrocompatibilità. Eventuali modifiche alle interfacce *Environment* e *Node* avrebbero comportato anche enormi operazioni di refactoring in quanto sono utilizzate in modo pervasivo in tutta la base di codice. Infine, *Environment* supporta un concetto di posizione dei nodi astratto che è indipendente dalle dimensioni dello spazio e dal tipo di geometria utilizzata, perchè non si vuole precludere la possibilità di creare simulazioni in spazi n-dimensionali ed ambienti non euclidei. Questo comporta la necessità di una struttura congegnata allo stesso livello di astrazione e genericità.

Nonostante il maggior livello di complessità è stata scelta l'estensione di *Environment* mantenendo comunque il rispetto dei requisiti non funzionali.

In primo luogo è stata definita l'interfaccia *Vector* come elemento base generico della geometria, con supporto a somme e sottrazioni fra vettori. Alchemist comprendeva già un concetto di posizione come n-upla di numeri reali, però non è stato possibile utilizzarla come vettore in quanto alcune implementazioni prevedono coordinate polari, per le quali operazioni di somma e sottrazione hanno poco senso. *Vector* è comunque stata congegnata per essere compatibile con *Position* di Alchemist, in modo che nei casi opportuni sia comunque possibile utilizzare una sola definizione di vettore senza bisogno di definirne una seconda. Ciò permette ad esempio di utilizzare *Euclidean2DPosition* di Alchemist sia come posizione dei nodi che come vettore geometrico, ma impedisce che vengano mescolati (ad esempio attraverso una soma accidentale) con posizioni non compatibili come *GeoPosition*, che è basato su uno spazio non euclideo. *Vector* è basato sul "Curiously Recurring Template Pattern"[1] sia per questioni di compatibilità con *Position* (che utilizza in origine questo pattern) che per evitare la possibilità di eseguire operazioni fra vettori di spazi vettoriali diversi a compile-time senza bisogno di effettuare ulteriori controlli a runtime.

[1] `https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern`

Successivamente sono state definite le interfacce per le trasformazioni geometriche e le forme in modo da essere generiche per quanto riguarda *Vector* e compatibili virtualmente con qualsiasi tipo di geometria e spazio n-dimensionale. A questo riguardo sono state considerate due opzioni: definire interfacce che possano supportare trasformazioni in spazi n-dimensionali oppure crearne di diverse di caso in caso e richiedere che chiunque interagisca con questo sistema sappia a priori quale geometria e dimensione stia utilizzando. La prima opzione avrebbe costituito un ulteriore aumento significativo della complessità mentre il requisito della seconda è stato giudicato ragionevole e si è quindi optato per questa scelta.

La gestione della posizione dei nodi nello spazio è originariamente demandata ad *Environment*, per coerenza quindi anche la gestione della rotazione è stata assegnata alla nuova stessa interfaccia *PhysicsEnvironment* del nuovo sistema. La *forma* è invece rimasta proprietà del nodo in modo da facilitarne anche la personalizzazione in fase di codifica delle simulazioni da parte degli utenti, senza dover modificare la struttura del file di configurazione delle stesse. Dal momento che la geometria è gestita da *PhysicsEnvironment* si è ritenuto opportuno semplificare la creazione delle forme tramite pattern "Factory"[6]. *PhysicsEnvironment* offre quindi una factory di forme che sono garantite essere compatibili con lo spazio e la geometria utilizzati.

In figura 2.3 è mostrato il modello del nuovo supporto alla geometria e fisica integrati con Alchemist. Nota: il diametro di una figura è inteso come la distanza massima fra qualsiasi coppia di vertici contenuti in essa, mentre il centroide è il centro geometrico [2]. Queste definizioni dovrebbero essere quindi compatibili con spazi n-dimensionali e geometrie non esclusivamente euclidee.

In figura 2.4 è sintetizzato il processo originale di caricamento dei nodi e dell'environment mentre in figura 2.5 è mostrato come il nuovo sistema sia stato integrato preservando la retrocompatibilità. L'operazione *getNodesWithinRange* incapsula una query ad un indice spaziale dei nodi basato su "quadtree"[12] già implementato in Alchemist. Questo garantisce una notevole riduzione nel numero di intersezioni geometriche da computare sia in fase di caricamento che a runtime durante il movimento dei nodi.

[2] `https://en.wikipedia.org/wiki/Centroid`

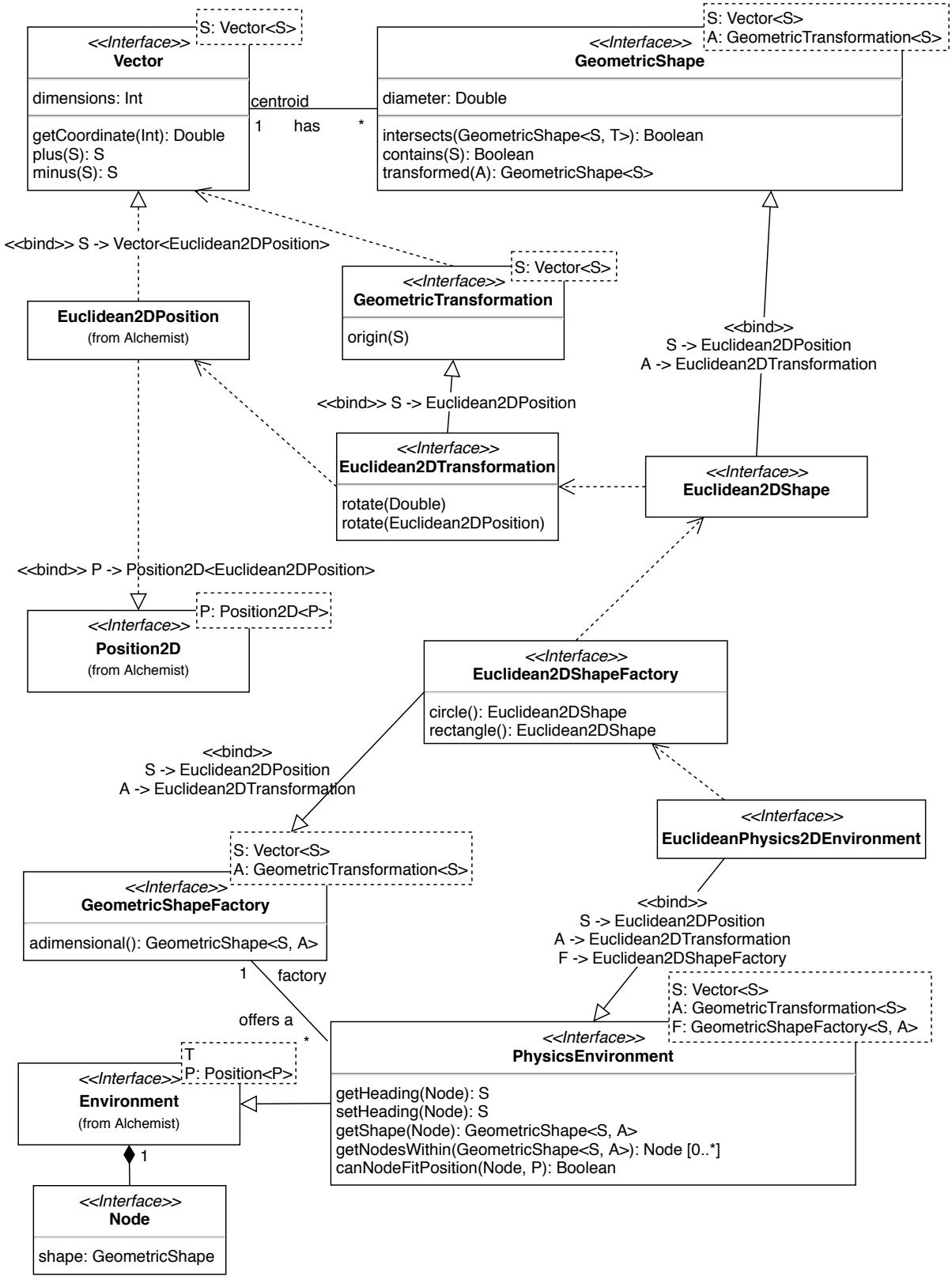

Figura 2.3: Diagramma raffigurante il modello della fisica e geometria integrati con Alchemist

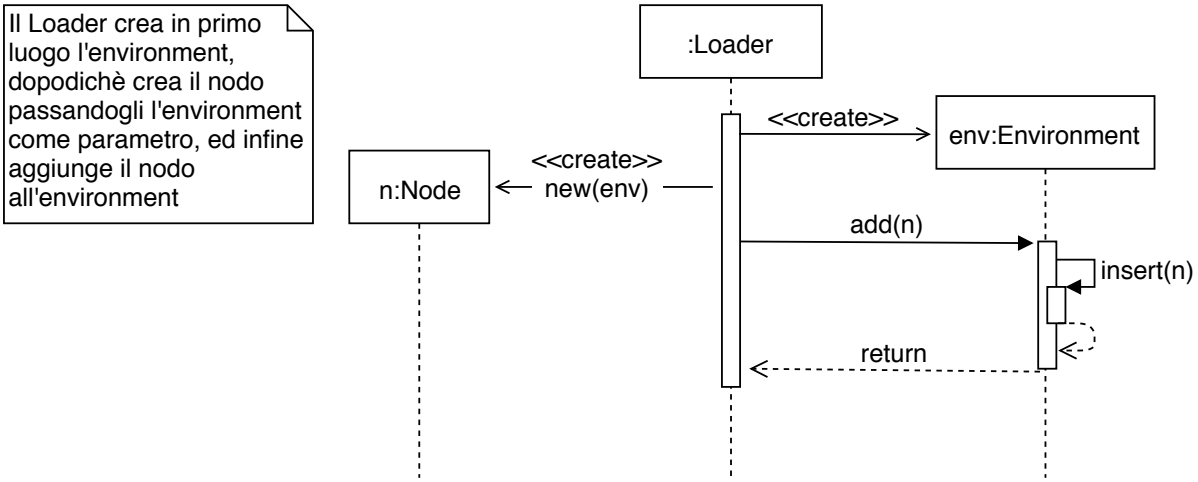

Figura 2.4: Diagramma raffigurante una sintesi della dinamica originale di caricamento dell'environment e dei nodi di una simulazione

2.2.2 Smartcam

Tutte le funzionalità richieste per simulare le smartcam sono state modellate singolarmente come componenti riutilizzabili attraverso le *Action*. Sono state sviluppate anche *Condition* come *ContainsMolecule* ed *Else* che permettono di definire comportamenti basilari senza bisogno di ulteriore programmazione, ad esempio abilitando l'esecuzione di *Action* solo in determinate condizioni.

Questo sistema garantisce l'integrabilità con tutti gli altri componenti di Alchemist ed un facile utilizzo attraverso il sistema preesistente di configurazione delle simulazioni. Inoltre ha il vantaggio di essere semplice da estendere e da riutilizzare anche in altri contesti e simulazioni di scenari di diversa natura. È infatti possibile modellare entità componendole con la selezione di *Action* e *Condition* volute. In figura 2.6 è illustrato un diagramma parziale di alcune delle componenti sviluppate e di come siano relazionate con il resto del software.

La smartcam viene rappresentata da un *Node* con le apposite *Reaction* ed *Action* per la visione, il movimento, la rotazione e la selezione di bersagli. Il campo visivo in particolare è stato modellato come un settore circolare di raggio (distanza visiva) ed angolo arbitrari ed è utilizzato internamente dall'azione *See*. Gli oggetti generici e potenziali bersagli sono anch'essi rappresentati con lo stesso sistema. Le *Molecule* di

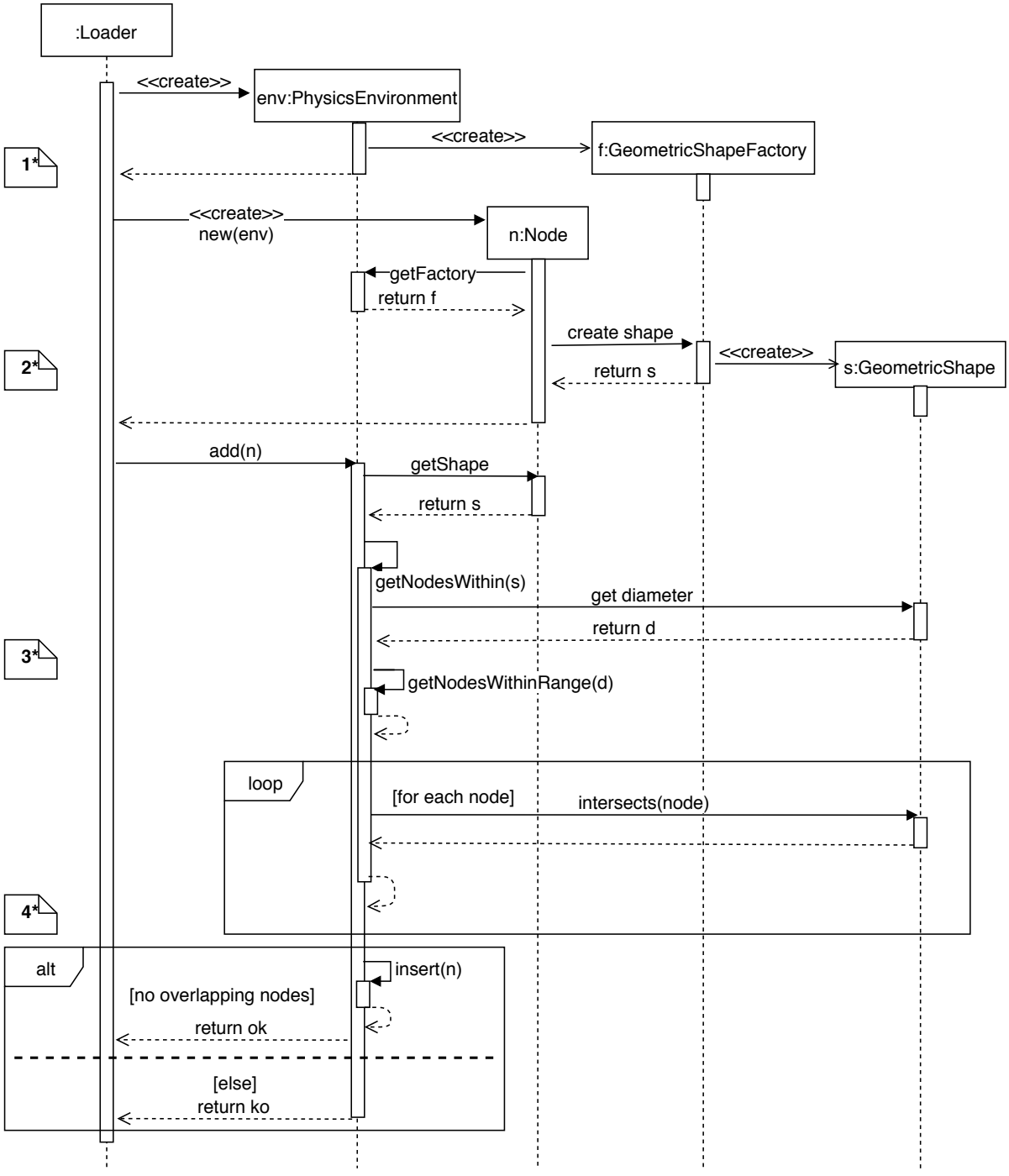

Figura 2.5: Diagramma raffigurante la nuova dinamica di caricamento di un environment che supporti la fisica. **1.** Il *Loader* crea l'*Environment* il quale si occupa della scelta della *GeometricShapeFactory*. **2.** Il *Loader* istanzia i *Node* i quali fanno uso della factory offerta dall'*Environment* per creare la propria *GeometricShape*. **3.** All'aggiunta di un *Node*, l'*Environment* esegue una query di tutti i *Node* presenti nel raggio del diametro della shape del nodo. Per ognuno di questi nodi, demanda a *GeometricShape* il vero controllo delle intersezioni geometriche. **4.** In caso non ci siano intersezioni allora il nodo viene aggiunto all'*Environment*, altrimenti il processo fallisce.

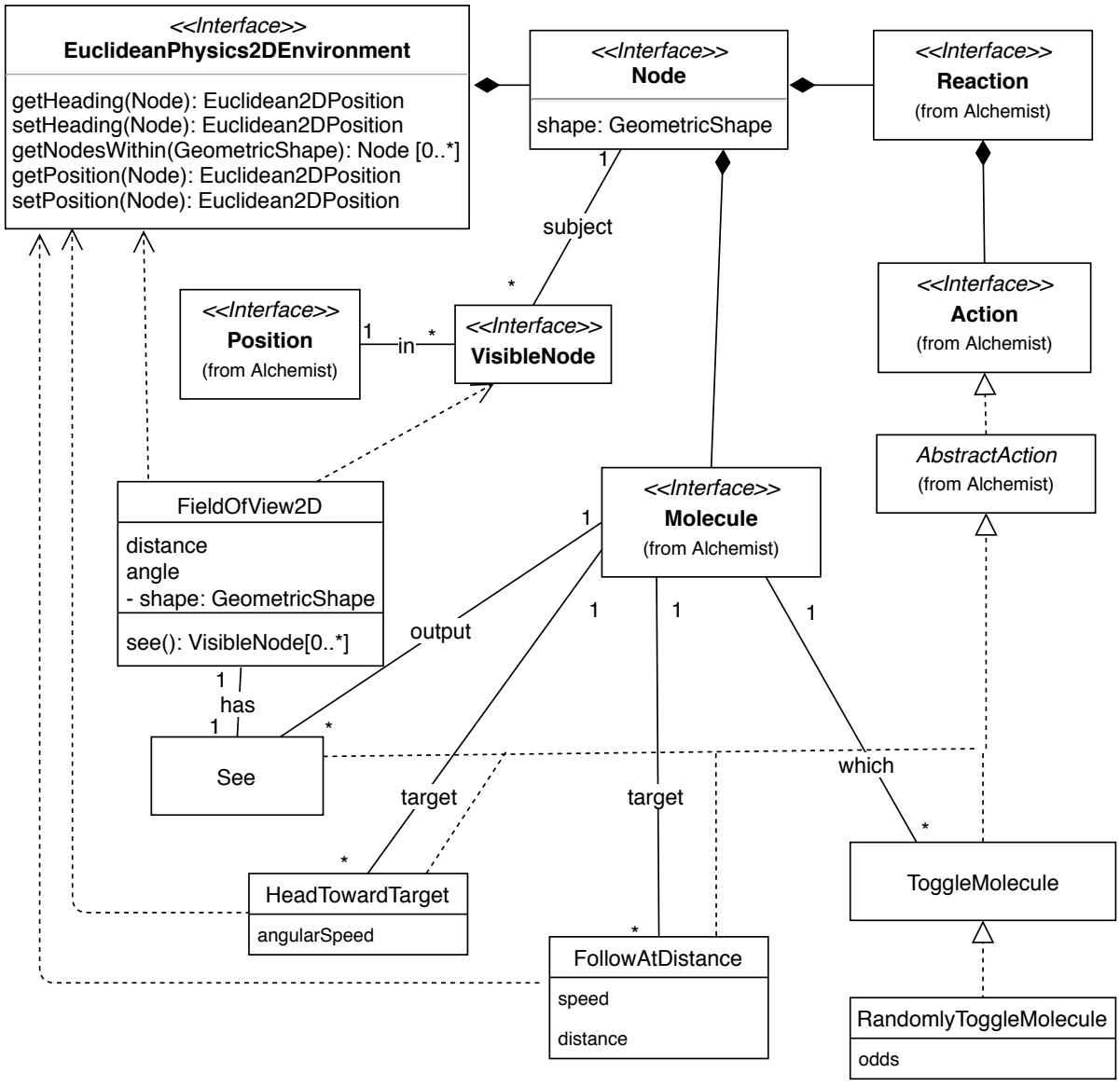

Figura 2.6: Diagramma parziale dei componenti del modulo alchemist-smartcam raffigurante come la visione ed il movimento siano stati integrati attraverso le *Action*.

Alchemist vengono utilizzate come mezzo di comunicazione fra le *Action*. In figura 2.7 è illustrato un esempio di composizione di smartcam e bersagli.

Il supporto alla programmazione delle smartcam è intrinseco al sistema, in quanto è sufficiente selezionare l'apposita *Incarnation* e comporre le entità come già menzionato.

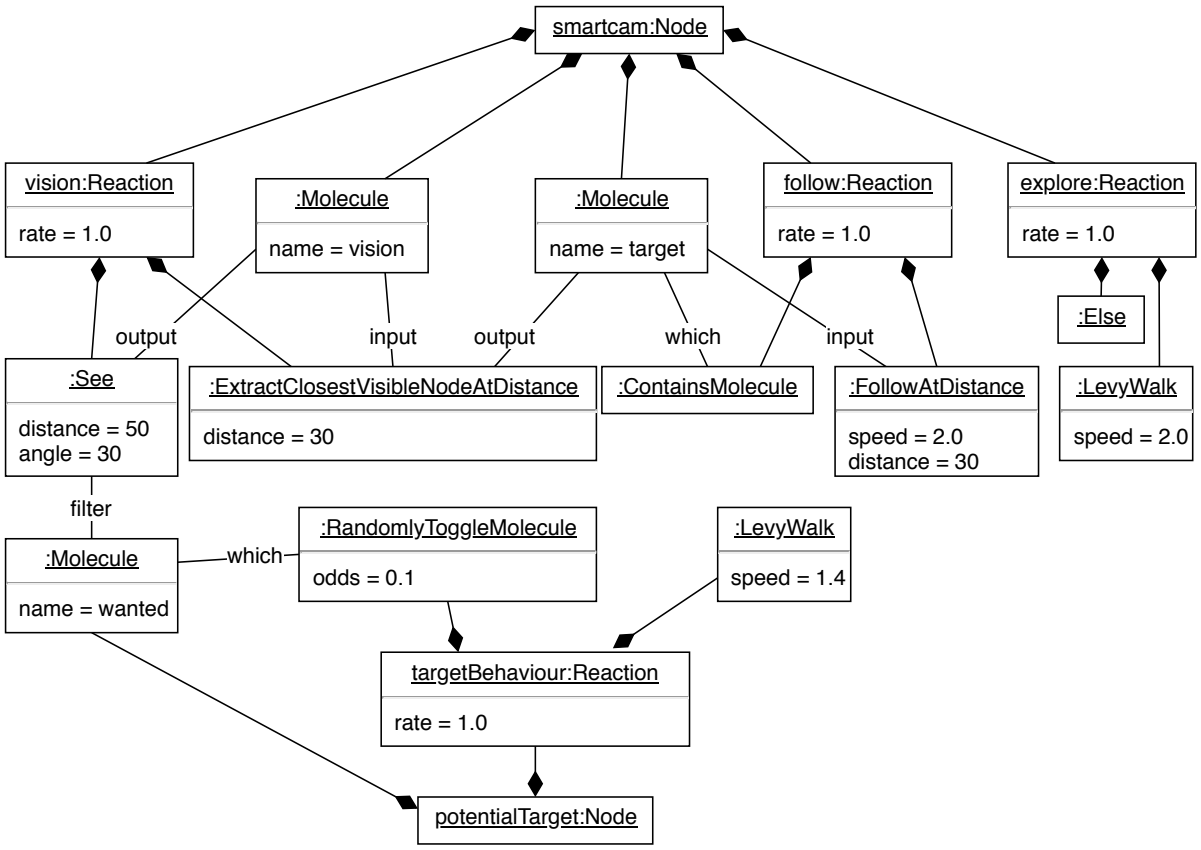

Figura 2.7: Esempio di composizione di smartcam e bersagli. *See* rileva solo oggetti contenenti la molecola *wanted* e li inserisce in *vision*. Da *vision* viene estratto il bersaglio più vicino al centro del campo visivo ed inserito in *target*. Se la smartcam contiene *target* allora lo segue a distanza, altrimenti si muove casualmente. Il potenziale bersaglio si muove casualmente ed ha il 10% di probabilità di scrivere o rimuovere la molecola *wanted* ad ogni attivazione di *targetBehaviour*.

Capitolo 3

Algoritmi aggregati per smartcam

In questo capitolo vengono discussi alcuni algoritmi già esistenti e come ne è stato effettuato il porting in Protelis. Successivamente vengono illustrati nuovi algoritmi sviluppati nell'ambito di questa tesi.

3.1 Riproduzione di algoritmi esistenti in aggregate

Sono stati scelti e riprodotti alcuni algoritmi tratti da ≪Online multi-object k-coverage with mobile smart cameras≫[7]. Nell'articolo gli autori modellano gli algoritmi basandosi sulla composizione di 3 strategie:

- *Baseline Behaviour* - Indica il comportamento base delle telecamere nei casi in cui non sia definito dalle altre 2 strategie. È definito solo un Baseline Behaviour comune a tutti gli algoritmi: le telecamere scelgono una direzione casuale che seguono fino ad un'eventuale collisione, nel qual caso scelgono un'altra direzione casuale. Alla rilevazione di un bersaglio, esso viene seguito e viene eseguita la *Communication Strategy*.

- *Communication Strategy* - Individua il modo in cui ogni telecamera sceglie a chi notificare l'eventuale rilevazione di bersagli nel campo visivo.

- *Response Model* - Individua la strategia con la quale vengono gestite le notifiche ricevute da altre telecamere.

Concettualmente gli algoritmi sono basati sulla macchina a stati mostrata in figura 3.1.

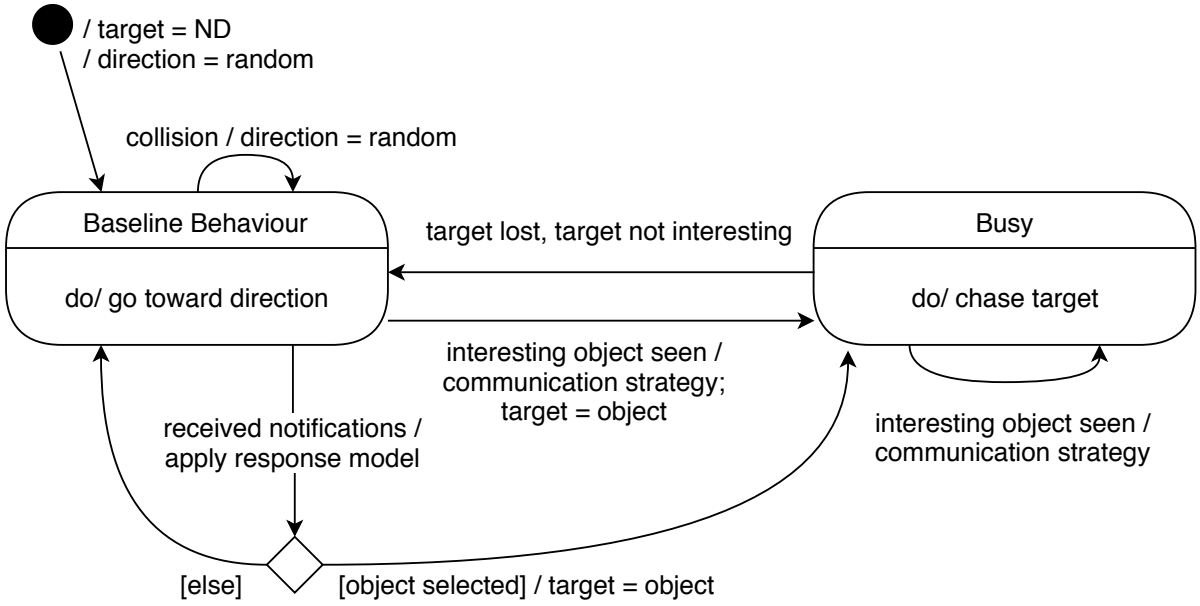

Figura 3.1: Diagramma raffigurante la macchina a stati che descrive in modo generico il comportamento degli algoritmi “Broadcast - Received Calls” e “Smooth - Available” tratti da «Online multi-object k-coverage with mobile smart cameras»[7].

3.1.1 Broadcast - Received Calls (BC-RE)

L’algoritmo si basa sulla communication strategy *Broadcast* e response model *Received Calls*.

Broadcast

Come si può intuire dal nome, questa strategia prevede la notifica di tutti i bersagli visti a tutte le telecamere.

Received Calls

Una telecamera è considerata libera se non sta seguendo nessun bersaglio. Le telecamere, se libere, si dirigono verso gli oggetti per i quali hanno ricevuto il minor numero di notifiche durante l'attuale round di computazione. Quando viene rilevato un bersaglio nel campo visivo, esso viene seguito e la telecamera viene considerata occupata.

Implementazione in aggregate

Con riferimento al listato 3.1, il broadcast è stato implementato attraverso l'istruzione *nbr* che genera un *Neighbouring Field* di bersagli. Dal field viene quindi estratto il più vicino fra i bersagli con meno sorgenti. Infine se è stato scelto un bersaglio, esso viene seguito, altrimenti viene eseguita la funzione *zigZagExploration* che implementa il Baseline Behaviour.

```
1  rep(myTarget <- noTarget()) {
2    let localTargets = getLocalTargets()
3    let allTargets = nbr(localTargets)
4    let possibleTarget = closestTarget(
5      elementsWithLeastSources(allTargets))
6    myTarget = if(myTarget == noTarget()
7      || !localTargets.contains(myTarget)) {
8      possibleTarget
9    } else {
10     // posizione aggiornata
11     localTargets.get(localTargets.indexof(myTarget))
12   }
13   followOrExplore(myTarget, zigZagExploration)
14 }
```

Listato 3.1: Implementazione di BC-RE in Protelis

3.1.2 Smooth - Available (SM-AV)

L'algoritmo si basa sulla communication strategy *Smooth* e response model *Available*. *Smooth* si basa sulla costruzione di un *Overlap Vision Graph*.

Overlap Vision Graph

Definisce un grafo i cui nodi sono le telecamere e gli archi sono valori numerici. Ad ogni round di computazione, se due telecamere stanno vedendo lo stesso oggetto allora viene aumentato il valore del rispettivo arco di una quantità $\delta = 1$. L'incremento viene eseguito per ogni telecamera e per ogni oggetto visto in comune. Ad ogni round gli archi evaporano di un fattore base pari a 0.995 ed un ulteriore 0.9 se le telecamere si sono mosse. In questo modo il valore degli archi è indice di quanto recentemente due telecamere si sono incontrate e quindi della probabilità che siano vicine. La presenza di ostacoli ai campi visivi è gestita naturalmente dal sistema. Due smartcam separate da un muro verranno infatti considerate lontane poichè, non potendo vedere gli stessi oggetti, non aumenteranno il valore dei rispettivi archi.

Smooth

Dato C l'insieme delle telecamere, per ogni telecamera $i, j \in C, i \neq j$ Smooth interpreta il valore degli archi $\lambda(i, j)$ dell'*Overlap Vision Graph* come la probabilità che i notifichi j definita secondo la seguente formula:

$$P_{Smooth}(i, j) = \frac{1 + \lambda(i, j)}{1 + \max\{\ \lambda(i, c)\ \forall\ c \in C\ \}}$$

Available

Le telecamere, se libere, si occupano delle notifiche più recenti. In caso ce ne siano più di una viene scelto il bersaglio più vicino. Se viene rilevato un bersaglio nel campo visivo esso viene seguito e la telecamera è considerata occupata.

Implementazione in aggregate

Ogni telecamera ha una visione parziale dell'*Overlap Vision Graph* che include solo gli archi per i quali è vertice. Questa parte è stata modellata come una mappa da telecamera a valore dell'arco ed implementata in Kotlin.

Per verificare quali oggetti sono in comune ai campi visivi delle telecamere è necessario che ogni telecamera esegua un broadcast locale di tutti gli oggetti che vede. SM-AV si basa infatti sull'assunzione che gli oggetti possano essere identificati univocamente e globalmente da tutte le telecamere. Inoltre l'algoritmo si basa sulla selezione semi-casuale di alcuni dispositivi a cui comunicare le notifiche dei bersagli rilevati. Dal momento che però è già stato effettuato il broadcast degli oggetti visti, l'implementazione realizzata (Listato 3.2) evita un invio duplicato di questi dati riutilizzando i primi.

3.2 Progettazione di nuovi algoritmi

In questa sezione vengono mostrati alcuni approcci ed algoritmi sviluppati nell'ambito di questa tesi. Sono state individuate due fasi principali che definiscono il comportamento delle telecamere: esplorazione ed inseguimento. Concettualmente gli algoritmi sono quindi basati sulla macchina a stati mostrata in figura 3.2. Tale macchina a stati è molto simile a quella degli algoritmi SM-AV e BC-RE (Figura 3.1), ma più flessibile, in quanto non vincola né la scelta dei bersagli né la strategia di esplorazione.

Ogni telecamera in fase di esplorazione ruota continuamente su se stessa nell'intento di estendere il limitato campo visivo sfruttando al massimo la mobilità. Questo dettaglio è stato giudicato conveniente in tutti gli scenari a campo aperto e quindi aggiunto a tutti gli algoritmi.

3.2.1 NoComm

Questo algoritmo molto semplice evita qualunque tipo di comunicazione e serve come base per poter valutare le prestazioni degli altri, ma anche per definire un comportamento da seguire in assenza di rete.

In fase di esplorazione utilizza lo stesso metodo di SM-AV e BC-RE descritto in precedenza con l'aggiunta della rotazione. Le telecamere seguono il primo bersaglio che vedono fino a quando quest'ultimo smette di essere interessante.

3.2.2 Force Field Exploration

Questo algoritmo individua un comportamento più raffinato per la fase di esplorazione. Similmente a «CMOMMT»[13] si basa sul concetto di campi di forza virtuali, che sono particolarmente facili da esprimere in aggregate programming.

Ogni telecamera produce un campo di forza repulsivo mentre ogni oggetto produce un campo di forza attrattivo. Le telecamere calcolano quindi la somma vettoriale delle forze per decidere la propria direzione di movimento. In questo modo tendono ad allontanarsi fra di loro e ad avvicinarsi a potenziali bersagli, espandendo la regione coperta e rimanendo in prossimità dei bersagli allo stesso tempo.

L'algoritmo così descritto tenderebbe però a generare situazioni statiche in cui le forze in gioco sono bilanciate ed i movimenti di conseguenza ridotti. Per questo motivo è stato aggiunto un concetto nominato "forza di volontà". Ogni telecamera ha una propria forza che applica al risultato descritto precedentemente per cercare di mantenere la direzione di movimento e contrastare gli altri campi.

Nell'implementazione sviluppata le forze f_{cam} delle telecamere e f_{ber} dei bersagli esercitate su una telecamera dipendono dalla distanza alla quale sono poste e sono state

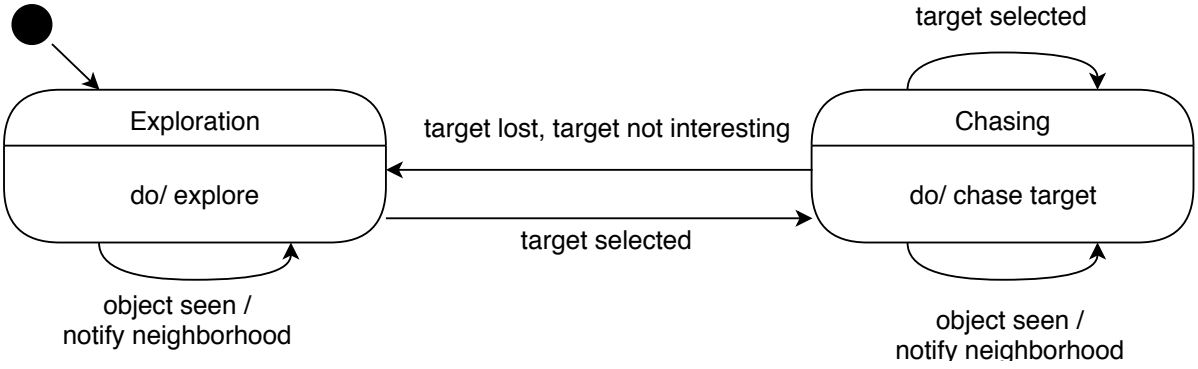

Figura 3.2: Diagramma raffigurante la macchina a stati che descrive in modo generico il comportamento dei nuovi algoritmi.

definite come segue:

$$f_{cam}(d) = \frac{w}{2}\frac{(2v)^2}{\max(1,d)^2}$$

$$f_{ber}(d) = -k\frac{4f_{cam}(d)}{\max(1,d)}$$

Dove

$k \in \mathbb{Z}$ è il numero massimo di telecamere per ogni bersaglio

$w \in \mathbb{R}$ è la forza di volontà

$v \in \mathbb{R}$ è la distanza del campo visivo

In questo modo la forza repulsiva di una telecamera diminuisce col quadrato della distanza, ed è pari alla metà della forza di volontà quando la distanza è uguale al doppio della profondità del campo visivo, ovvero quando i campi visivi di due telecamere stanno per sovrapporsi. La forza attrattiva di un bersaglio invece è molto più influente, decresce più lentamente e trova il suo massimo nell'opposto della forza di k telecamere moltiplicata per un fattore 4 scelto sperimentalmente.

Gli scenari soggetti a sperimentazione prevedono la copertura di una zona limitata. Per evitare che le telecamere escano da tale zona, ognuna di esse percepisce un campo repulsivo posizionato ai confini che segue la legge di f_{cam}.

L'implementazione è mostrata nel listato 3.3 mentre la figura 3.3 mostra come, in assenza di bersagli, questo algoritmo tenda a distribuire le telecamere quasi uniformemente nella zona e ad evitare sovrapposizioni di campo visivo, in modo da coprire contemporaneamente più territorio.

3.2.3 LinPro

Questo algoritmo si concentra sulla parte di inseguimento, ed in particolare sulla selezione dei bersagli. Il nome LinPro deriva da "Linear Programming" poichè il problema della scelta dei bersagli si può modellare tramite la programmazione lineare. L'algoritmo persegue la minimizzazione dei costi di movimento invece che la massimizzazione della copertura ed è definito come segue.
Siano:

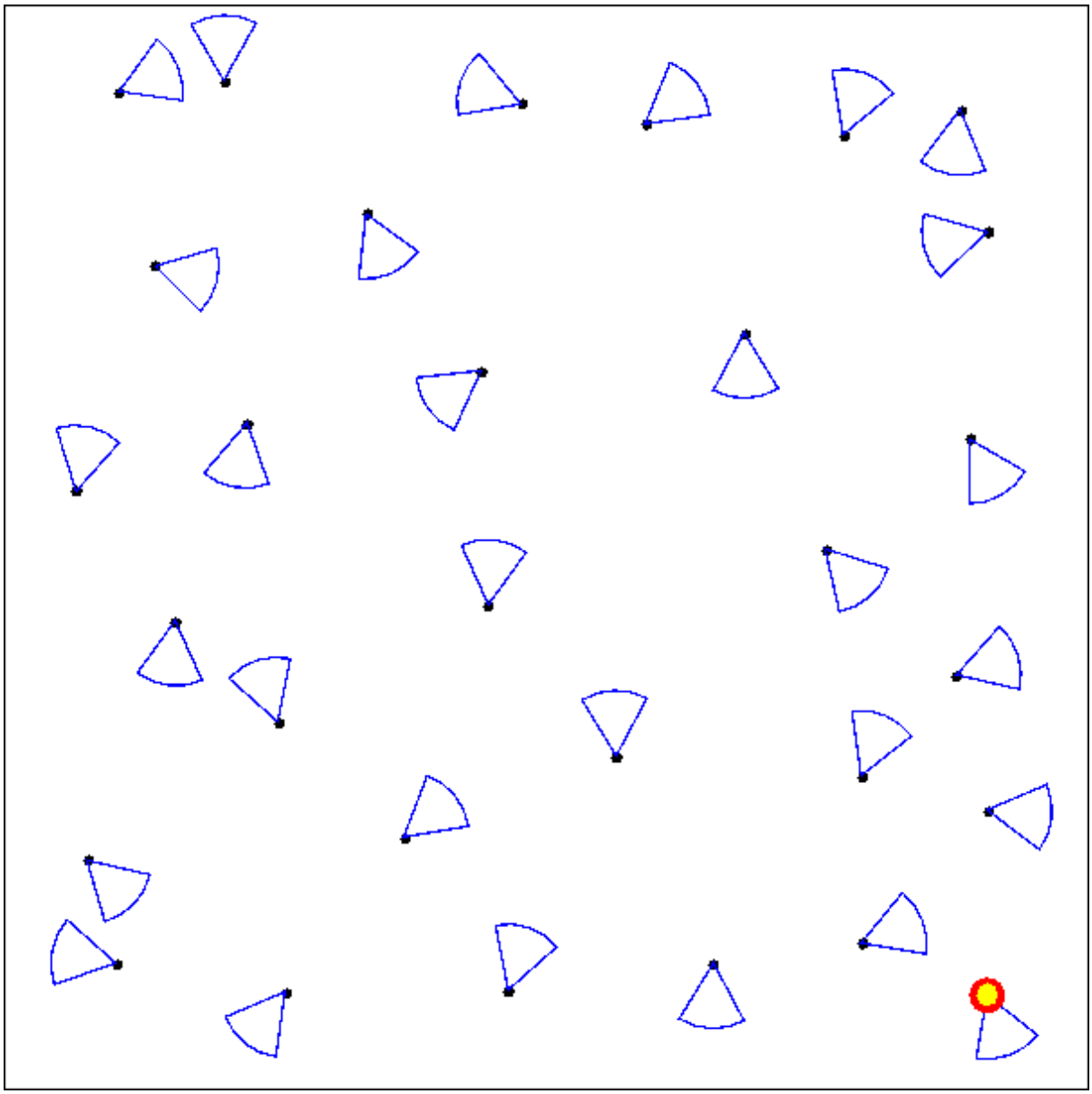

Figura 3.3: Screenshot di Force Field Exploration in azione in una zona limitata ed in mancanza di bersagli. Le telecamere tendono a distribuirsi uniformemente nel territorio.

$m \in \mathbb{N}$ il numero delle telecamere

$n \in \mathbb{N}$ il numero dei bersagli

$k \in \mathbb{N}$ il massimo numero di telecamere assegnabili ad un singolo bersaglio

$c_{ij} \in \mathbb{R}$ il costo per la telecamera i di raggiungere il bersaglio j. L'attuale implementazione utilizza la distanza euclidea dal centro del campo visivo.

Il problema che risolve è formalizzato come segue:

$$\text{Minimizza} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} q x_{i,n+1} \tag{1}$$

$$\text{Soggetto a} \quad \sum_{j=1}^{n+1} x_{ij} = 1 \qquad i = 1, \ldots, m \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} \leqslant k \qquad j = 1, \ldots, n \tag{3}$$

$$\sum_{i=1}^{m} x_{ij} \geqslant \min\left(1, \left\lfloor \frac{m}{n} \right\rfloor\right) \qquad j = 1, \ldots, n \tag{4}$$

$$x_{ij} \geqslant 0 \qquad i = 1, \ldots, m \quad j = 1, \ldots, n+1 \tag{5}$$

Dove:

- $n + 1$ individua un bersaglio fittizio di costo maggiore di tutti gli altri. Verrà assegnato alle telecamere libere in caso n sia zero oppure maggiore di $m \times k$, ovvero in caso ci siano più telecamere di quante ne servano a raggiungere la k-coverage. Una telecamera assegnata al bersaglio fittizio è considerata libera e viene dedicata all'esplorazione.

- $q = \max\{c_{ij}\} + 1$ è una costante maggiore del costo massimo, utilizzata per definire il costo del bersaglio fittizio.

- x_{ij} della soluzione ottima varrà 1 se il bersaglio j è assegnato alla telecamera i, altrimenti zero.

Le espressioni (indicate con un numero fra parentesi alla loro destra) hanno il seguente significato:

(1) L'obiettivo è minimizzare la somma dei costi di tutte le telecamere per il raggiungimento dei bersagli. Il bersaglio fittizio ha il maggior costo in assoluto in modo che venga scelto come ultima possibilità.

(2) Ogni telecamera dev'essere assegnata ad esattamente un bersaglio (che può essere quello fittizio).

(3) Non possono essere assegnate più di k smartcam ad un singolo bersaglio, eccetto quello fittizio.

(4) Il vincolo serve per evitare di perdere bersagli nei casi in cui tutte le telecamere siano già assegnate e ne vengano rilevati di nuovi. In questo modo, quando possibile, almeno una telecamera lascerà un bersaglio già osservato da altre per raggiungere il nuovo.

(5) Vincolo di non-negatività delle soluzioni.

Il modello è molto simile al "problema dei trasporti"[5] in cui le telecamere sono i magazzini, i bersagli sono le destinazioni e la quantità di merce disponibile ad ogni magazzino è pari ad uno. In particolare la matrice dei coefficienti dei vincoli ne condivide le proprietà ed i termini noti sono interi, assicurando perciò soluzioni intere. Una conseguenza di questo fatto, unito alla presenza dei vincoli (2) e (5) e considerata la funzione obiettivo, è la garanzia che ogni telecamera sarà assegnata ad esattamente un solo bersaglio invece che a frazioni di molteplici bersagli.

Nell'implementazione realizzata su Protelis (listato 3.4), ogni telecamera esegue il broadcast della posizione tutti i bersagli che vede e della propria. Con i dati raccolti procede a risolvere il suddetto problema ed all'inseguimento del bersaglio che le viene assegnato dalla soluzione ottima, o all'esplorazione nel caso le venga assegnato il bersaglio fittizio. Nell'implementazione riportata nel listato 3.4 non è mostrata la risoluzione del problema che viene affidato ad un'implementazione Java del simplesso offerto dalla libreria «apache-commons-math»[1].

[1] `https://commons.apache.org/math/`

Variante "Fair"

Questa variante mira a distribuire le telecamere in modo equo fra i bersagli rilevati, ed è realizzabile semplicemente modificando i vincoli come segue:

$$
\begin{aligned}
\text{Minimizza} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} q x_{i,n+1} & \\
\text{Soggetto a} \quad & \sum_{j=1}^{n+1} x_{ij} = 1 & i = 1,\dots,m \\
& \sum_{i=1}^{m} x_{ij} \geqslant \min\left(k, \left\lfloor \frac{m}{n} \right\rfloor\right) & j = 1,\dots,n \\
& \sum_{i=1}^{m} x_{ij} \leqslant \min\left(k, \left\lceil \frac{m}{n} \right\rceil\right) & j = 1,\dots,n \\
& x_{ij} \geqslant 0 & i = 1,\dots,m \\
& & j = 1,\dots,n+1
\end{aligned}
$$

I nuovi vincoli servono a fare in modo che il numero di telecamere assegnate ad ogni singolo bersaglio si avvicini il più possibile al rapporto telecamere / bersagli, e comunque non superi k. Il bersaglio fittizio è escluso da questi vincoli in modo che possa essere liberamente assegnato a qualsiasi numero di telecamere.

```
1  rep(info <- [noTarget(), buildOverlapRelationsGraph(1, 0.995, 0.95)]) {
2    let myTarget = info.get(0)
3    let graph = info.get(1)
4    let localObjects = getVision()
5    let globalObjects = cloneFieldToMap(nbr(localObjects))
6    graph.evaporate()
7    graph.update(globalObjects)
8    let devicesToNotify = graph.smooth()
9    let notifiedDevices = nbr(devicesToNotify)
10   let notifiers = findDevicesByData(notifiedDevices) {
11     it.contains(self.getDeviceUID())
12   }
13   // filtra i bersagli estraendo quelli che sono stati segnalati
14   let allTargets = notifiers
15     .map { id -> onlyTargets(globalObjects.get(id)) }
16     .reduce([]) { d1, d2 -> d1.union(d2) }
17     .union(onlyTargets(localObjects))
18   myTarget = if(myTarget == noTarget() || !allTargets.contains(myTarget
       )) {
19     closestTarget(allTargets)
20   } else {
21     allTargets.get(allTargets.indexof(myTarget))
22   }
23   followOrExplore(myTarget, zigZagExploration)
24   [myTarget, graph]
25 }
```

Listato 3.2: Implementazione di SM-AV in Protelis

```
def getForceOfWill() {
  10
}
def getRepulsiveForce(distance) {
  getForceOfWill() / 2 * ((getFoVDistance() * 2) ^ 2) / (max(1,
   distance) ^ 2)
}
def getAttractiveForce(distance) {
  - (getMaxCamerasPerTarget() * getRepulsiveForce(distance) * 4 / max
   (1, distance))
}
public def fieldExploration() {
  let cameraVersorsField = nbrVersor()
  let distanceFromCamerasField = self.nbrRange()
  let cameraForces = getRepulsiveForce(distanceFromCamerasField) *
   cameraVersorsField
  let envBoundariesForce = getEnvironmentBoundariesForce()
  let targetForces = unionHood PlusSelf(nbr(getVision())).map {
    getAttractiveForce(distanceFromTarget(it)) * unitVector(self.
   getCoordinates() - posToTuple(it.getPosition()))
  }.reduce([0,0]) { a,b -> a + b }
  let sumOfForces = sumHood(cameraForces) + envBoundariesForce +
   targetForces
  rep(myDirectionAngle <- randomAngle()) {
    let myDirection = angleToVersor(myDirectionAngle)
    let myForce = myDirection * getForceOfWill()
    let destination = self.getCoordinates() + sumOfForces + myForce
    let newDirectionAngle = directionToAngle(destination - self.
   getCoordinates())
    let augment = getMaxMovementSpeed() * 2 // assicura di andare alla
   massima velocita'
    destination = destination + [augment * cos(newDirectionAngle),
   augment * sin(newDirectionAngle)]
    env.put("destination", destination)
    newDirectionAngle
  }
}
```

Listato 3.3: Implementazione di Force Field Exploration

```
1 import it.unibo.smartcamexperiment.ProtelisUtils.
    CameraTargetAssignmentProblem
2
3 let localTargets = getLocalTargets()
4 let targets = unionHood PlusSelf(nbr(localTargets))
5 let cameras = nbr(getCenterOfFov())
6 let myTarget = CameraTargetAssignmentProblem.solve(cameras, targets,
    getMaxCamerasPerTarget())
7   .getOrDefault(getUID(), noTarget())
8 followOrExplore(myTarget, zigZagExploration)
```

Listato 3.4: Implementazione di LinPro

Capitolo 4

Sperimentazione

In questo capitolo vengono discusse le variabili considerate nelle simulazioni, le assunzioni di partenza, ed i risultati ottenuti dalla sperimentazione dei seguenti algoritmi:

- BC-RE
- SM-AV
- NoComm
- FF-NoComm: NoComm con Force Field Exploration.
- ZZ-LinPro: LinPro con la stessa strategia di esplorazione di BC-RE e SM-AV (Movimento a "ZigZag").
- FF-LinPro: LinPro con Force Field Exploration.
- ZZ-LinProF: Variante "fair" con esplorazione a "ZigZag"
- FF-LinProF: Variante "fair" con esplorazione a campi di forza.

4.1 Configurazione delle simulazioni

Le simulazioni si svolgono in uno spazio euclideo bidimensionale quadrato dove vengono disposti oggetti e smartcam in modo casuale ed in rapporto variabile. I requisiti

e le assunzioni di ogni algoritmo preso in analisi vengono soddisfatti in modo ideale. Vengono quindi supportati l'identificazione univoca dei bersagli, il sistema di posizionamento globale, e la comunicazione inter-smartcam senza l'introduzione di alcun errore. La tabella 4.1 mostra la lista delle variabili coinvolte ed il loro valore predefinito nel caso non sia indicato diversamente.

Durante ogni simulazione viene effettuato il sampling della *k-coverage* istantanea una volta al secondo ed alla fine di essa viene considerata come *k-coverage* finale la media dei valori ottenuti. Vengono effettuate 20 simulazioni con seed differenti per ogni combinazione di variabili prese in analisi e vengono infine estratte la *k-coverage* media e la deviazione standard.

4.2 Risultati

In primo luogo gli algoritmi sono stati simulati in scenari con diversi rapporti fra il numero di potenziali bersagli e quello delle smartcam. Si è partiti con un rapporto 1:1 per finire con 10:1 ed i risultati sono mostrati in figura 4.1. È chiaro come l'approccio di LinPro sia valido nei casi in cui si disponga di un buon numero di smartcam. Field Exploration al contrario sembra aver avuto un impatto minimale, probabilmente dovuto alla mancanza di un sistema di predizione della posizione degli oggetti una volta che essi escono dal campo visivo. NoComm ha mostrato prestazioni sorprendentemente alte, il chè fa pensare che ci sia un grande margine di miglioramento negli algoritmi che sfruttano la comunicazione. SM-AV ha mantenuto prestazioni alla pari se non peggiori di NoComm in ogni scenario, probabilmente indicando come una comunicazione basata sulla casualità non sia un buon approccio in ampi spazi. Infine BC-RE, come ci si poteva aspettare, ha mostrato le prestazioni peggiori. Il Broadcast infatti non considera che per raggiungere la *k-coverage* bastino k telecamere per ogni bersaglio, ed accoppiato con Received Calls provoca grandi gruppi di smartcam che seguono pochi bersagli.

Il secondo set di simulazioni è volto all'analisi del comportamento degli algoritmi con distanze di comunicazione decrescenti. In tutti gli scenari viene mantenuto unitario il rapporto oggetti / telecamere. I risultati sono mostrati in figura 4.2. Nel caso di LinProF, il broadcast 1-hop dei bersagli provoca delle desincronizzazioni per quanto riguarda il

problema di minimizzazione. Capita, ovvero, che le smartcam risolvino il problema con dati diversi o parziali, ottenendo soluzioni diverse e peggiorando la cooperazione. Questo risulta in un notevole calo della *3-coverage* anche se la 1 e *2-coverage* vengono mantenute inaspettatamente alte. La versione non "fair" è invece molto meno vulnerabile a questo fenomeno, in quanto l'algoritmo tende a non cambiare bersaglio alle telecamere che ne

Variabile	Valore predefinito
Durata della simulazione	2000 s
Dimensione dell'arena	500×500 m
Velocità di movimento degli oggetti	1.4 m/s [1]
Strategia di movimento degli oggetti	Lévy walk[2]
Numero degli oggetti	100
Definizione degli oggetti interessanti	([3])
Rapporto smartcam / oggetti	1
Algoritmo di coordinazione smartcam	NoComm
Velocità di movimento delle smartcam	3 m/s [4]
Distanza del campo visivo delle smartcam	30 m
Angolo del campo visivo delle smartcam	60°
Velocità di rotazione delle smartcam	$\frac{\pi}{5}\ \frac{rad}{s}$
Raggio di comunicazione delle smartcam	800 m
Frequenza dei round computazionali di Protelis	1 Hz
Numero massimo di smartcam per bersaglio (k in k-coverage)	3
Seed per il generatore di casualità	1

Tabella 4.1: Elenco delle variabili coinvolte nelle simulazioni.

[1] Velocità media di cammino preferita dai pedoni[2]

[2] Lévy walk costituisce una buona approssimazione per il movimento dei pedoni[17]

[3] Ogni oggetto, ogni 20s, ha il 5% di probabilità di diventare interessante. Gli oggetti smettono di essere interessanti con lo stesso meccanismo e probabilità.

[4] Assunzione conservativa inferiore alla velocità dei droni giocattolo odierni

stanno già seguendo uno, perciò non rischia di cadere in una soluzione parziale errata. Gli altri algoritmi non mostrano cambiamenti eccetto BC-RE che ovviamente giova dalla riduzione del raggio di broadcast.

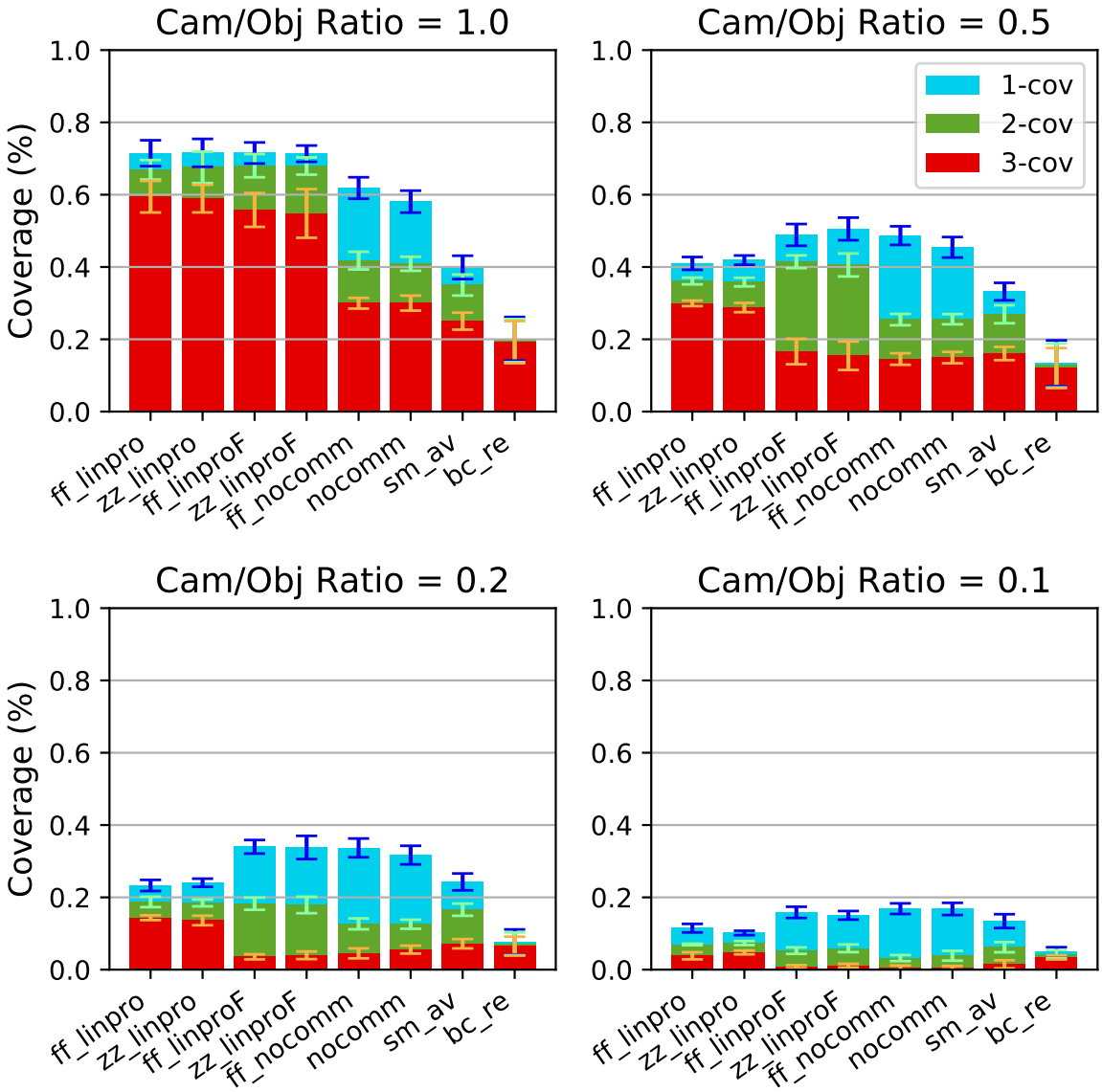

Figura 4.1: Comparazione delle performance degli algoritmi di coordinazione di smartcam al variare del rapporto smartcam / oggetti. Le barre di errore indicano la deviazione standard.

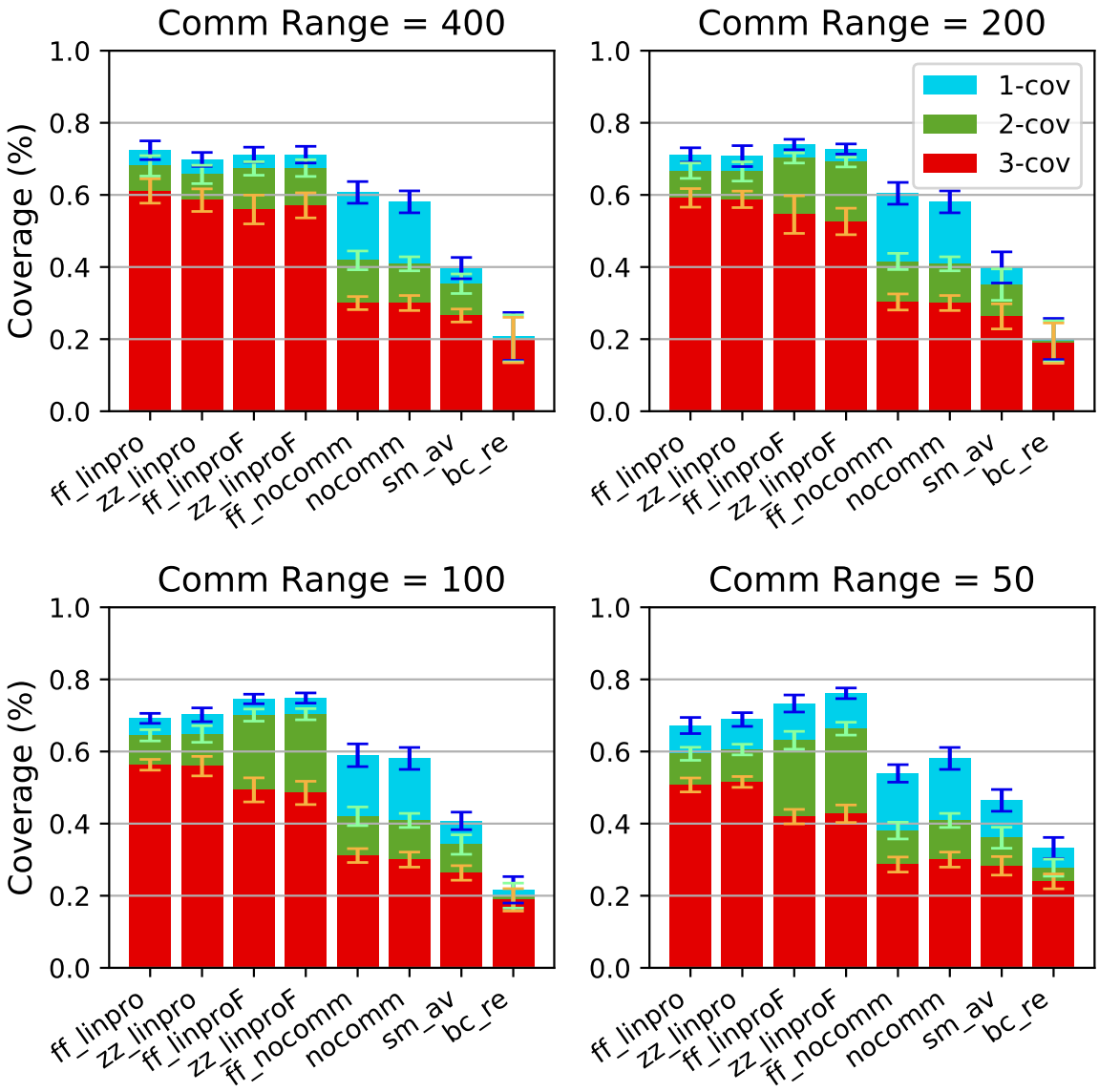

Figura 4.2: Comparazione delle performance degli algoritmi di coordinazione di smart-cam al variare del range di comunicazione. Le barre di errore indicano la deviazione standard.

Capitolo 5

Conclusioni

Nell'ambito di questa tesi è stato aggiunto un nuovo modulo al simulatore Alchemist per la modellazione di smartcam. Inoltre è stato effettuato il porting di alcuni algoritmi esistenti in aggregate, l'ideazione ed implementazione di nuovi, ed infine sono state eseguite misurazioni sperimentali.

Tutto questo ha richiesto l'apprendimento e l'impiego di due nuovi linguaggi di programmazione, Kotlin e Protelis, di un nuovo paradigma definito dall'aggregate programming, e di tecnologie di DevOps come Gradle[1] e Travis-CI[2].

5.1 Futuri sviluppi

In futuro si prevede di raffinare i nuovi algoritmi analizzando ad esempio tecniche di gossiping[14] e predizione della posizione degli oggetti usciti dal campo visivo.

Riguardo LinPro l'intento sarebbe di valutare una strategia del tipo "divide et impera". In scenari di elevate dimensioni infatti è impossibile pensare che ogni telecamera faccia il broadcast globale di tutto ciò che vede. Si potrebbe però dividere la zona in regioni più piccole ed eseguire l'algoritmo localmente. A questo riguardo esistono già alcune tecniche che potrebbero essere adottate come il ≪Self-organising Coordination Regions≫[3].

[1] `https://gradle.org/`

[2] `https://travis-ci.org/`

Inoltre, il numero di variabili in gioco (Tabella 4.1) comporta la presenza di un elevatissimo numero di diversi scenari da analizzare che sarebbe opportuno esplorare in modo più approfondito.

Infine si ritiene che il lavoro qui presentato costituisca una base sufficiente alla realizzazione di un contributo scientifico da sottoporre a revisione fra pari.

Capitolo 6

Ringraziamenti

Ringrazio la mia famiglia e gli amici che mi sono stati vicino in questo periodo. Infine non potrei non ringraziare l'Ing. Danilo Pianini, il quale mi ha gentilmente offerto più supporto di quanto avrei potuto sperare durante questa tesi.

Bibliografia

[1] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, September 2015.

[2] Raymond C. Browning, Emily A. Baker, Jessica A. Herron, and Rodger Kram. Effects of obesity and sex on the energetic cost and preferred speed of walking. *Journal of Applied Physiology*, 100(2):390–398, February 2006.

[3] Roberto Casadei, Danilo Pianini, Mirko Viroli, and Antonio Natali. Self-organising coordination regions: A pattern for edge computing. In *International Conference on Coordination Languages and Models*, pages 182–199. Springer, 2019.

[4] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 113–128. Springer International Publishing, 2015.

[5] George B. Dantzig and Mukund N. Thapa. *Linear Programming 1: Introduction (Springer Series in Operations Research and Financial Engineering) (v. 1)*. Springer, 1997.

[6] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[7] Lukas Esterle and Peter R Lewis. Online multi-object k-coverage with mobile smart cameras. In *Proceedings of the 11th International Conference on Distributed Smart Cameras*, pages 107–112. ACM, 2017.

[8] Lukas Esterle, Peter R Lewis, Horatio Caine, Xin Yao, and Bernhard Rinner. Camsim: A distributed smart camera network simulator. In *2013 IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, pages 19–20. IEEE, 2013.

[9] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, March 2000.

[10] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, December 1977.

[11] Gabriele Graffieti. Progettazione e implementazione di una incarnazione biochimica per il simulatore alchemist - tesi di laurea, 2016.

[12] Samet Hanan. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[13] Lynne E Parker and Brad A Emmons. Cooperative multi-robot observation of multiple moving targets. In *Proceedings of International Conference on Robotics and Automation*, volume 3, pages 2082–2089. IEEE, 1997.

[14] Danilo Pianini, Jacob Beal, and Mirko Viroli. Improving gossip dynamics through overlapping replicates. In *International Conference on Coordination Languages and Models*, pages 192–207. Springer, 2016.

[15] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7, 01 2013.

[16] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1846–1853. ACM, 2015.

[17] Injong Rhee, Minsu Shin, Seongik Hong, Kyunghan Lee, Seong Joon Kim, and Song Chong. On the levy-walk nature of human mobility. *IEEE/ACM Transactions on Networking*, 19(3):630–643, June 2011.

[18] B. Rinner and W. Wolf. An introduction to distributed smart cameras. *Proceedings of the IEEE*, 96(10):1565–1575, October 2008.

[19] Robert Tolksdorf and Ronaldo Menezes. Using swarm intelligence in linda systems. In *Engineering Societies in the Agents World IV*, pages 49–65. Springer Berlin Heidelberg, 2004.

[20] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(2):16, 2018.

[21] Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In *Lecture Notes in Computer Science*, pages 143–162. Springer Berlin Heidelberg, 2009.

[22] Mirko Viroli, Roberto Casadei, and Danilo Pianini. Simulating large-scale aggregate MASs with alchemist and scala. In *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*. IEEE, October 2016.

[23] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.

[24] Evşen Yanmaz, Saeed Yahyanejad, Bernhard Rinner, Hermann Hellwagner, and Christian Bettstetter. Drone networks: Communications, coordination, and sensing. *Ad Hoc Networks*, 68:1–15, January 2018.

[25] Franco Zambonelli, Gabriella Castelli, Laura Ferrari, Marco Mamei, Alberto Rosi, Giovanna Di Marzo, Matteo Risoldi, Akla-Esso Tchao, Simon Dobson, Graeme Stevenson, Juan Ye, Elena Nardini, Andrea Omicini, Sara Montagna, Mirko Viroli, Alois Ferscha, Sascha Maschek, and Bernhard Wally. Self-aware pervasive service ecosystems. *Procedia Computer Science*, 7:197–199, 2011.

[26] B.P. Zeigler. *Theory of Modelling and Simulation*. A Wiley-Interscience Publication. John Wiley, 1976.