Università degli Studi di Torino
Dipartimento di Informatica

Corso di Laurea in Informatica

Relazione di Tirocinio

ANALISI DEL FRAMEWORK JACAMO PER LA PROGRAMMAZIONE DI UN SISTEMA MULTI-AGENTE ATTRAVERSO LA SIMULAZIONE DI UN'ORGANIZZAZIONE NO-PROFIT

Curriculum Informazione e Conoscenza

Relatore:
Prof.ssa *Cristina Baroglio*

Correlatore:
Dott. *Stefano Tedeschi*

Candidato:
Andrea Bonini

Sessione NOVEMBRE 2020
a.a. 2019/2020

Indice

Capitolo 1
Introduzione

Da molti anni lo sviluppo software si è rivolto allo studio e alla creazione di sistemi concorrenti, decentralizzati e distribuiti. Questa spinta deriva dallo sviluppo di nuove architetture hardware e da una rete sempre più accessibile; tale programmazione ormai non è più solo legata a un dominio specifico e per questo motivo sono stati introdotti nuovi livelli di astrazione per affrontare questa nuova complessità.

In quest'ottica, i Sistemi Multi-Agente (MAS) hanno ricevuto una forte attenzione in molti campi di ricerca e di sviluppo: essi hanno una vasta applicabilità in diverse aree che si estendono dal controllo dei processi industriali a quello delle scienze umane e sociali.

Un sistema multi-agente è un sistema composto da un insieme di entità software, gli agenti, capaci di interagire tra loro in uno stesso contesto per risolvere problemi complessi che vanno oltre le abilità del singolo. Un agente è in grado di percepire l'ambiente in cui si trova e le altre entità che operano in esso, è in grado inoltre di reagire a ciò che succede intorno a lui e di mettere in atto delle strategie per raggiungere i propri obiettivi. Modellare un tale sistema di conseguenza significa studiare come le diverse entità possano interagire tra loro, come gli agenti possano ragionare e agire autonomamente, come il mondo, in cui si trovano, possa essere rappresentato, come possa essere distribuita la conoscenza, le risorse o le capacità di ragionamento o ancora come si possa decentralizzare il controllo del sistema.

Sebbene la ricerca sui sistemi multi-agente si sia scissa sulle diverse dimensioni che compongono un tale sistema, quali ad esempio la definizione di un modello per gli agenti, la caratterizzazione dell'ambiente o il coordinamento degli agenti stessi attraverso norme, obblighi o divieti, c'è stato nel framework JaCaMo il tentativo e la volontà di provare a rispondere a queste domande e unire le diverse visioni in un solo strumento utile a modellare il sistema nella sua totalità.

Questa relazione ha quindi come obiettivo quello di introdurre il lettore nel mondo dei sistemi multi-agenti, in particolare a capire che cosa sia un agente, come possa essere costruito, come interagisca, come esso "ragioni" e come cooperi per risolvere problemi complessi. Tutto ciò verrà analizzato attraverso lo studio del framework JaCaMo e di ciascuna delle sue componenti. Nel Capitolo 2, si illustreranno le teorie e i modelli su cui si basano i sistemi multi-agente e il framework JaCaMo. In particolare il capitolo fornirà le nozioni di base per comprendere il funzionamento degli agenti, (Sezione 2.2), dell'ambiente (Sezione 2.3) e dell'organizzazioni (Sezione 2.4); per ciascuna dimensione, si analizzeranno i rispettivi linguaggi di programmazione e si mostrerà come essi siano stati integrati nel framework. Nel Capitolo 3, si illustrerà il caso di studio modellato e sviluppato durante il tirocinio; esso permetterà di mostrare non solo i vantaggi

pratici dell'uso di JaCaMo, ma anche i suoi limiti. Il Capitolo 4 invece conclude la dissertazione presentando le osservazioni finali e mostrando i possibili nuovi sviluppi offerti dal framework.

In questa introduzione voglio infine porgere un ringraziamento alla mia relatrice, la Professoressa Cristina Baroglio, e al mio correlatore, il Dottor Magistrale Stefano Tedeschi, per avermi permesso di effettuare questo stage: senza il loro instancabile supporto (e pazienza) questo lavoro non sarebbe stato realizzabile.

Dichiarazione di originalità

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Capitolo 2
Programmare un Sistema Multi-Agente

I Sistemi Multi-Agente (MAS) sono sistemi composti da un insieme (una organizzazione) di entità computazionali, gli agenti, capaci di interagire tra loro in un ambiente condiviso per risolvere un compito complesso. Un tale sistema permette di distribuire fra gli agenti la conoscenza, le risorse e le capacità di ragionamento e di decisione, permette di decentralizzare il controllo del sistema e di coordinare e di regolare il comportamento degli agenti tramite strutture sociali e norme.

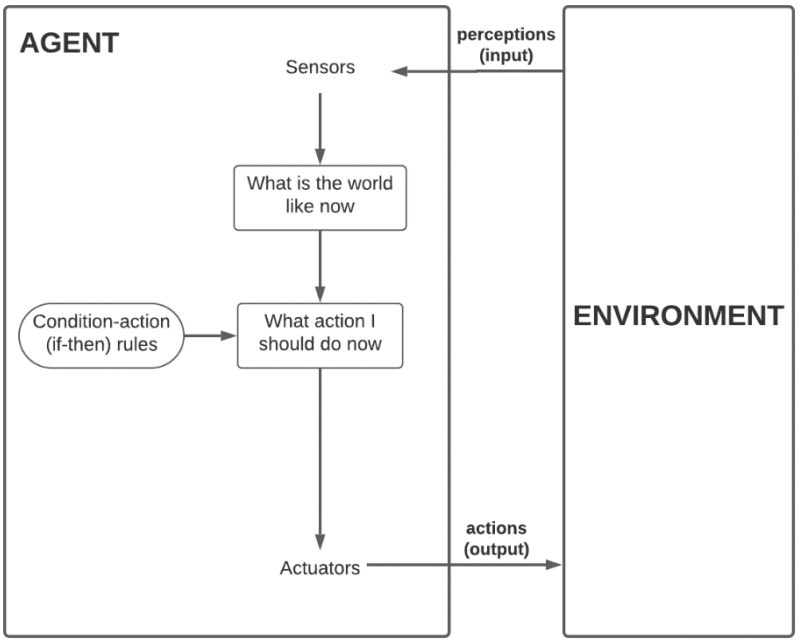


Figura 2.1: modello di interazione tra agente e ambiente

Un agente è l'unità primitiva per la computazione, un'entità collocata in un ambiente, capace di prendere decisioni arbitrarie basate sulle percezioni ambientali (*parameters*) e sugli obiettivi dell'entità stessa, capace di eseguire azioni sull'ambiente e di reagire a eventi per raggiungere i propri obiettivi progettuali [30].

Le caratteristiche principali per un agente sono [6]:

- autonomia: gli agenti operano senza intervento umano e possono controllare le loro azioni e il loro stato interno (locale);

- collocazione (situadness): gli agenti operano in un ambiente (*environment*) condiviso, dove hanno un controllo parziale su di esso; essi possono percepirlo (attraverso i propri sensori) e hanno a disposizione un repertorio di azioni per modificarlo (attraverso degli attuatori); tramite l'ambiente inoltre, gli agenti possono coordinarsi indirettamente;
- pro-attività: gli agenti sono in grado di decidere autonomamente (tramite la loro storia, le percezioni ambientali, le informazioni scambiate con altri agenti) di che cosa hanno bisogno e ciò che devono fare (si parla di *goal-directed behaviour*) per raggiungere i propri obiettivi progettuali; questa caratteristica implica che uno stesso agente possa eseguire azioni diverse se collocato in ambienti diversi;
- socialità (*social abilities and organisation awareness*): gli agenti possono interagire tra loro (o con entità esterne al sistema) attraverso un linguaggio comune (*Agent Communication Language*) all'interno di una struttura sociale che ne regola il comportamento; possono infatti condividere le informazioni in loro possesso (*knowledge*), richiederne di nuove, delegare compiti o cooperare con altri agenti per risolvere problemi complessi, non risolvibili dal singolo agente;
- reattività: gli agenti intelligenti sono capaci di percepire il loro ambiente e di rispondere in modo tempestivo ai cambiamenti che si verificano in esso.

L'ambiente invece è il medium che fornisce agli agenti le condizioni per esistere e agire. Un agente usa le informazioni percepite dall'ambiente per ragionare. L'ambiente può essere accessibile o meno (influenza il grado di accuratezza delle informazioni percepite dagli agenti), deterministico o non deterministico (una stessa azione, eseguita due volte in circostanze apparentemente identiche, potrebbe non avere l'effetto desiderato: un agente deve quindi saper gestire anche i fallimenti), dinamico o statico, continuo o discreto.

L'organizzazione infine è un'astrazione che permette di definire una struttura collettiva per far coordinare e regolare il funzionamento degli agenti. Si possono identificare quattro diversi tipi di interazione tra organizzazione e agenti [13], riassunti nella Figura 2.2. Nel caso (a) gli agenti non hanno una rappresentazione mentale dell'organizzazione, ma devono seguirne i vincoli; in (b), gli agenti hanno invece una rappresentazione interna dell'organizzazione (l'organizzazione è definita in funzione di un agente: agenti diversi hanno rappresentazioni diverse della stessa organizzazione) e sono soggetti ai suoi vincoli; in (c) invece l'organizzazione esiste, ma gli agenti possono solamente obbedire agli obiettivi dell'organizzazione in loro codificati; nei primi tre casi, dunque l'organizzazione non fornisce alcuna autonomia all'agente, in (d) infine gli agenti hanno una rappresentazione interna dell'organizzazione (l'organizzazione è definita in funzione di un agente: agenti diversi hanno rappresentazioni diverse della stessa organizzazione) e sono in grado di leggere, rappresentare e ragionare in base all'organizzazione stessa.

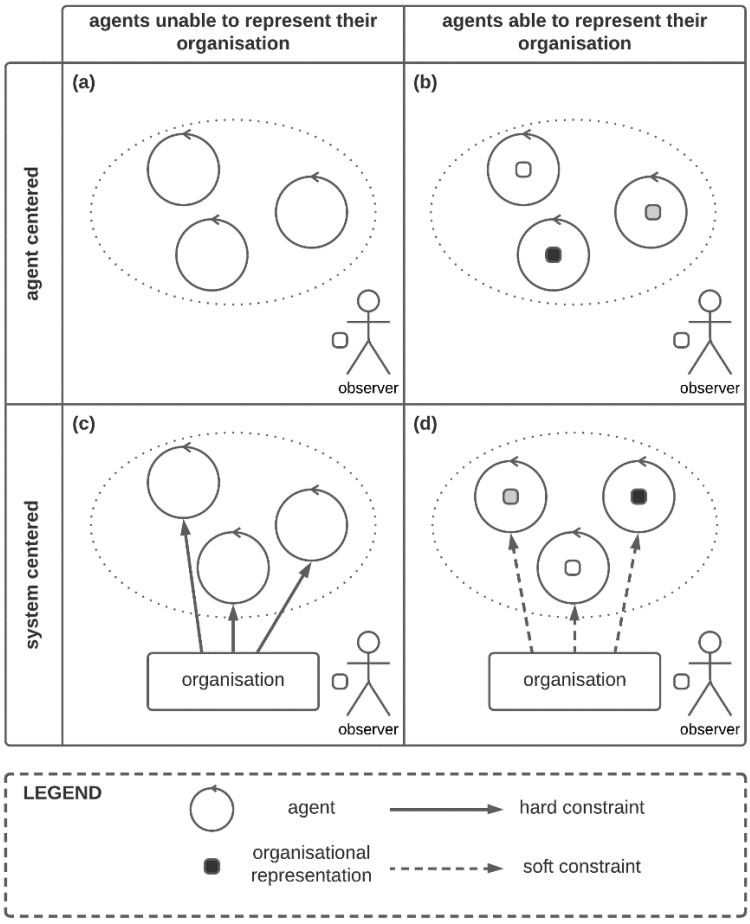


Figura 2.2: diversa rappresentazione mentale di un'organizzazione [13]

Lo studio sui sistemi multi-agente è un dominio multi-disciplinare, che ha come scopo quello di creare sistemi complessi, aperti, decentralizzati e distribuiti in ambienti complessi e dinamici. I principali domini coinvolti sono: (i) l'intelligenza artificiale distribuita (DAI) e l'ingegneria del software orientata agli agenti (AOSE), dove si definiscono le teorie, i modelli architetturali e i linguaggi di programmazione orientati agli agenti (APL); (ii) le scienze sociali, dove i MAS sono utilizzati come strumento per simulare e studiare il funzionamento delle società; (iii) la teoria dei giochi, che permette di studiare e prevedere le forme di interazione e negoziazione tra parti diverse, ciascuna con un proprio obiettivo; e infine (iv) le simulazioni.

In altre parole, i sistemi multi-agente sono un mezzo per ridurre la complessità di un sistema suddividendo i problemi in compiti specifici, sistema in cui sono presenti molte entità autonome, che devono interagire in modi unici e che sono soggette a strutture sociali e norme che ne regolano il comportamento in un ambiente condiviso [3].

Per questo motivo, la ricerca sui sistemi multi-agente si è divisa e specializzata sullo studio di diversi paradigmi di programmazione; tra questi, troviamo i) AOP: programmazione di agenti ii) EOP: modellazione e programmazione dell'ambiente; iii) IOP: programmazione delle interazioni; iv) OOP: programmazione di strutture sociali, come organizzazioni e norme.

2.1 JaCaMo

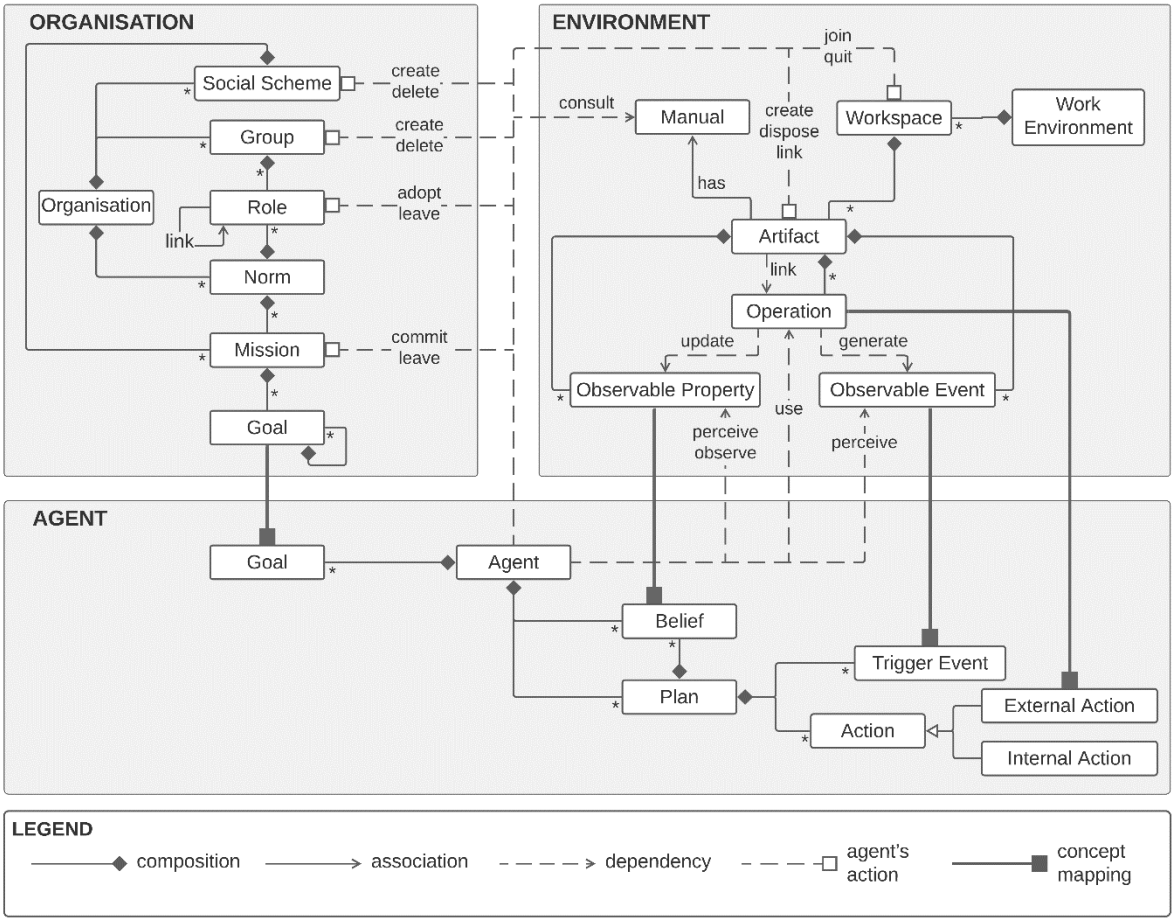


Figura 2.3: meta-modello del framework JaCaMo [3]

JaCaMo è un framework, che fornisce un supporto di alto livello per lo sviluppo di sistemi multi-agente; esso combina le dimensioni degli agenti (AOP), dell'ambiente (EOP) e dell'organizzazione (OOP) in modo sinergico, ma allo stesso tempo mantiene una forte ortogonalità (*separation of concerns*) tra le diverse dimensioni [3]. Un tale sistema multi-agente è modellato (Fig. 2.3) come una organizzazione di agenti Moise (Sezione 2.4.1) che, attraverso regole, gruppi e missioni, coordina agenti autonomi programmati in Jason (Sezione 2.2.2), che lavorano in un ambiente distribuito e condiviso basato su artefatti programmati in CArtAgO (Sezione 2.3.2). Sebbene queste piattaforme siano indipendenti, JaCaMo le integra combinando ciascun meta-modello e linguaggio di programmazione al fine di ottenere un paradigma di programmazione unificato in un'unica piattaforma volta a semplificare lo sviluppo di sistemi multi-agente complessi [16].

La Figura 2.3 mostra il meta-modello di programmazione di JaCaMo [3]. Questo meta-modello è focalizzato sulle astrazioni e sui costrutti di programmazione; non include pertanto concetti o astrazioni che non fanno parte dei linguaggi o dei framework di programmazione: ad esempio, il concetto di intenzione che è parte del runtime di Jason [3], non è incluso. Oltre a presentare i concetti principali della programmazione JaCaMo, l'obiettivo principale di questo schema è quello di mostrare esplicitamente le

dipendenze, le connessioni e il *mapping* concettuale tra le astrazioni appartenenti alle diverse dimensioni. Le astrazioni appartenenti alla dimensione degli agenti si ispirano principalmente al modello BDI su cui si basa Jason: un agente è un'entità composta da un insieme di credenze (*Belief*), un insieme di obiettivi (*Goal*), e una serie di piani (*Plan*), composti da azioni (*Action*) e innescati da eventi (come la modifica alle credenze di un agente). Nella dimensione dell'ambiente, basato su di CArtAgO e sul meta-modello A&A, ogni ambiente (*Work Environment*) è composta da uno o più *Workspace*, che contiene un insieme dinamico di artefatti (*Artifact*). Ogni artefatto fornisce un insieme di operazioni (*Operation*) e proprietà osservabili (*Observable Property* e *Observable Event*) che permettono all'agente di osservare e operare su di esso tramite un'interfaccia di utilizzo. Infine, la dimensione organizzativa, basata su Moise, descrive un'organizzazione da un punto di vista strutturale in termini di gruppi (*Group*) e ruoli (*Role*), da un punto di vista funzionale, in termini di uno schema sociale (*Social Scheme*), che scompone la struttura degli obiettivi dell'organizzazione in sotto-obiettivi (*Goal*), che sono raggruppati in missioni (*Mission*) e da un punto di vista normativo, in termini di norme che vincolano i ruoli alle missioni. Le sinergie tra le diversi dimensioni sono rappresentate dalle linee che terminano con un quadrato: un quadrato pieno rappresenta esplicitamente il *mapping* concettuale tra le entità di ciascuna dimensione (tale integrazione è totalmente trasparente e non deve essere programmata *ad hoc*); un quadrato bianco invece rappresenta l'insieme delle azioni predefinite che un agente può eseguire su ciascuna dimensione (queste azioni sono modellate come operazioni negli artefatti predefiniti di JaCaMo).

Teoricamente JaCaMo include anche la dimensione delle interazioni (IOP): gli agenti possono interagire tra loro tramite comunicazione diretta o mediata; nel primo caso gli agenti scambiano messaggi tramite atti comunicativi nel linguaggio KQML (Sezione 2.2.3), nel secondo interagiscono e si coordinano condividendo e usando artefatti

Nei prossimi paragrafi, verrà analizzata accuratamente ciascuna dimensione. La prima parte del capitolo descriverà il paradigma di programmazione per agenti, il modello BDI, e descriverà la piattaforma Jason e il linguaggio di comunicazione per agenti KQML. Successivamente si analizzerà la dimensione dell'ambiente, si approfondirà il modello A&A e si studierà la piattaforma CArtAgO, evidenziandone l'integrazione con Jason. Infine verrà introdotta la dimensione dell'organizzazione, si illustrerà il funzionamento di Moise e si mostrerà l'integrazione di questa dimensione con quella degli agenti e dell'ambiente. In questo modo è stato possibile analizzare e valutare l'efficacia e i limiti di utilizzo del modello di programmazione JaCaMo e in particolare l'integrazione sinergica delle sue dimensioni.

2.2 Programmare gli Agenti: AOP

L'Agent Oriented Programming (programmazione orientata agli agenti) nasce come astrazione per definire i processi di decisione e ragionamento degli agenti in funzione di eventi (in ambienti dinamici). L'idea di questo nuovo paradigma di programmazione orientata agli agenti è stata sviluppata da Shoham e combina l'uso di nozioni mentalistiche (come le *belief*) per la programmazione di agenti autonomi con una visione sociale del sistema [3].

Gli agenti devono elaborare costantemente nuove informazioni, reagire a eventi e modificare i propri piani nel tentativo di raggiungere i propri obiettivi a lungo termine (sotto determinate condizioni), mantenendo allo stesso tempo un comportamento reattivo e proattivo. Quello così sviluppato è un agente razionale [3], un agente, da cui ci si aspetta, che, dopo aver intrapreso una certa linea d'azione e notato che l'obiettivo non sia ancora stato raggiunto, continui a eseguire altre azioni per raggiungerlo, a meno che non ci siano prove sufficienti che tale obiettivo non sia più raggiungibile o necessario.

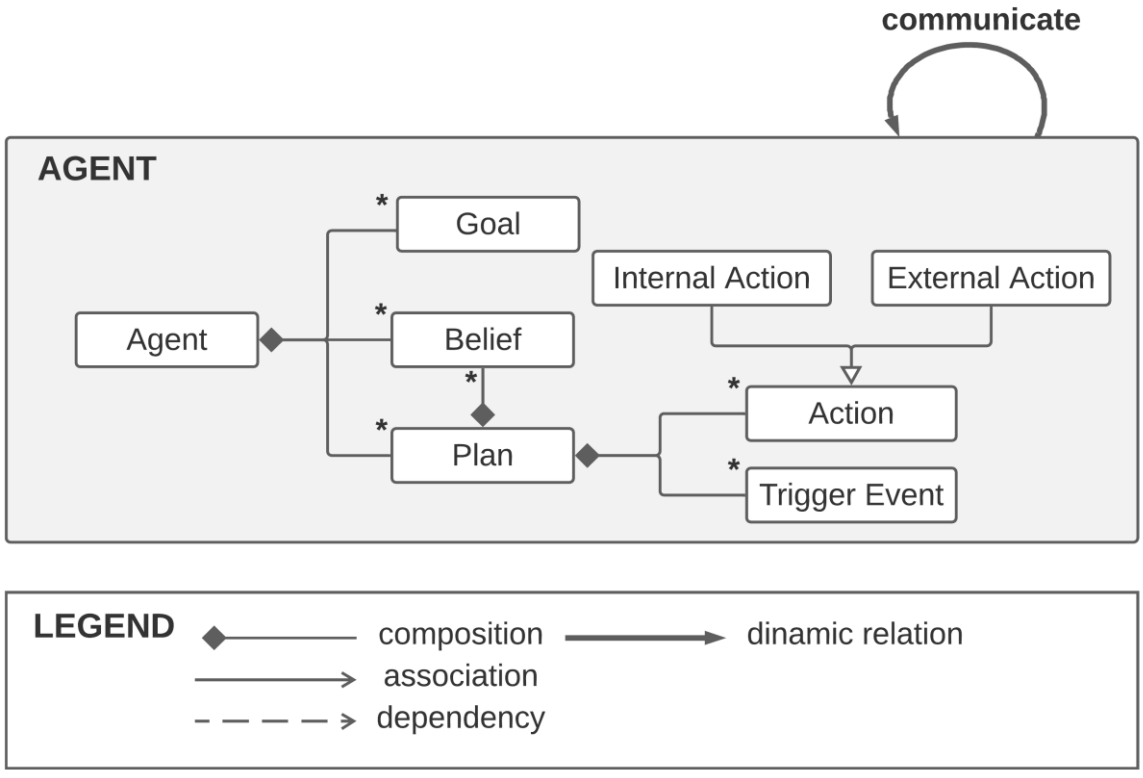


Figura 2.4: meta-modello di un Agente BDI

2.2.1 Il modello BDI

Tra i molti modelli proposti nel corso degli anni, il modello BDI (*Belief, Desire, Intention*) è diventato ben presto l'approccio principale per lo sviluppo di agenti razionali. Secondo questo modello, gli agenti sono caratterizzati da uno stato mentale (*metal state*) basato sui concetti di *Belief*, *Desire* e *Intention* [6] che possono essere descritti come:

- *Belief* (Credenze): le informazioni che un agente possiede sul mondo e sugli altri agenti del sistema;
- *Desire* (Desideri): tutte le possibili opzioni che un agente potrebbe portare a termine; tutte queste opzioni non devono necessariamente venire completate, ma sono utili per influenzare il comportamento degli agenti;
- *Intention* (Intenzioni): tutte le opzioni scelte da un agente per portare a termine qualche obiettivo (*Goal*). Questi obiettivi possono essere assegnati a un agente o possono essere il frutto di altre azioni.

In altre parole, un agente che persegue un obiettivo specifico (*Goal*), avrà solo alcune opzioni (*Desire*) compatibili con tale obiettivo; tra queste, solo una, quella selezionata, diventerà un'intenzione (*Intention*), che permetterà all'agente di raggiungere il proprio *Goal*.

Ma come si possono trasformare le *Belief*, i *Desire* e le *Intention* in azioni? Il modello BDI è basato sul *practical reasoning*, un ragionamento rivolto all'azione [6]. Esso è diviso in due fasi: nella fase deliberativa (*deliberation phase*), l'agente, date un insieme di opzioni, decide quali intenzioni portare a termine in base alla sua attuale visione del mondo (espressa in funzione di *Belief*, *Desire* e *Intention*); nella seconda fase (*means-ends reasoning phase*), l'agente decide come svolgere il compito usando i mezzi a propria disposizione (ad esempio tramite azioni sull'ambiente o messaggi inviati ad altri agenti).

Un agente BDI può essere quindi considerato come un sistema reattivo di pianificazione (*reactive planning system*) [5], un processo sempre attivo (che non termina), che reagisce a un evento (es. modifica di *Belief*) proveniente dall'ambiente e il cui comportamento segue il *pratical reasoning* [6]. Il suo funzionamento è basato sul ciclo (chiamato *reasoning cycle*), riportato in Figura 2.5. In questo algoritmo, definite le credenze e le intenzioni iniziali, si possono identificare (nel ciclo esterno) tre fasi principali: la fase di percezione, quella di ragionamento e quella di azione. Nella prima fase, la **percezione**, l'agente aggiorna la propria *Belief Base* (insieme di tutte le informazioni disponibili a un agente in un dato momento): acquisisce nuove informazioni o percependo i cambiamenti nell'ambiente [riga 4], o ricevendo nuovi messaggi da altri agenti. Nella seconda fase, il **ragionamento**, l'agente aggiorna le proprie *Belief* combinando le nuove informazioni con le conoscenze pregresse tramite la *belief revision function* (*brf*) [riga 5]; determina quindi i nuovi desideri, valutando le opzioni [riga 6] e tra queste "filtra" le intenzioni [riga 7]. Infine seleziona un nuovo piano [riga 8] per completare l'intenzione scelta. Ora l'agente entra nell'ultima fase, l'**azione**: selezionato un piano, l'agente svolge tutte le azioni presente in esso. È importante notare che dopo aver eseguito ciascuna azione (nel ciclo interno), l'agente aggiorna le proprie credenze [riga 14], e se necessario i desideri e le intenzioni: in questo

modo, l'agente è in grado di valutare se il piano risulti ancora realizzabile e di conseguenza se continuare a perseguire o meno l'intenzione corrente [riga 19].

```
1.   B ← B0      /* B0 are are initial beliefs */
2.   I ← I0      /* I0 are initial intentions */
3.   while true do
4.       get next percept ρ via sensors;
5.       B ← brf(B, ρ);
6.       D ← options(B, I);
7.       I ← filter(B, D, I);
8.       π ← plan(B, I, Ac)    /* Ac are initial belief */
9.       while not(empty(π) or succeded(I, B) or impossible(I,B))
10.          α ← first element of π;
11.          execute(α);
12.          π ← tail of π;
13.          observe environment and get next percept ρ;
14.          B ← brf(B, ρ);
15.          if reconsider(I, B)then
16.              D ← options(B, I);
17.              I ← filter(B, D, I);
18.          end-if
19.          if not sound(π, I, B) then
20.              π ← plan(B, I, Ac);
21.          end-if
22.      end-while
23.  end-while
```

Figura 2.5: reasoning-cycle di un agente BDI [6]

2.2.2 Jason

Jason è una piattaforma per lo sviluppo di sistemi multi-agente che incorpora un linguaggio di programmazione orientato agli agenti basato sul modello BDI e un ambiente di sviluppo integrato (IDE); permette di creare agenti autonomi, ambienti e sviluppare e testare sistemi multi-agente. Il linguaggio deriva da una versione estesa di AgentSpeak, un linguaggio di programmazione per agenti basato sulla logica introdotto da Rao nel 1996, che è stato successivamente ampliato da una serie di pubblicazioni di Bordini e Hübner [6].

Jason è disponibile open-source sotto licenza GNU LGPL dal sito ufficiale di Jason [21]. L'interprete AgentSpeak è stato sviluppato in modo da essere modulare e facilmente personalizzabile: è molto semplice per gli sviluppatori di un sistema multi-agente implementare le proprie personalizzazioni, come ad esempio *Belief Base* personalizzate, funzioni di aggiornamento e recupero di *belief* e modifiche al ciclo di ragionamento degli agenti. L'IDE Jason fornisce un'interfaccia grafica per la modifica di un file di configurazione del sistema multi-agente e del codice AgentSpeak. Attraverso l'IDE è possibile eseguire e controllare l'esecuzione del sistema multi-agente. Inoltre esso fornisce anche un altro strumento, chiamato "*Mind Inspector*", una sorta di debugger orientato agli agenti, che permette all'utente di ispezionare gli stati interni degli agenti quando il sistema è in esecuzione in modalità debug.

Un agente, in Jason (e di conseguenza in JaCaMo), è quindi principalmente definito in base alle sue *Belief*, ai suoi Piani (*Plan*) e ai suoi *Goal* [6]. Il ciclo di ragionamento di un agente Jason estende quello del modello BDI. La *Belief Base* è l'insieme di tutte le informazioni disponibili a un agente in un dato momento, provenienti dalle percezioni, dalle comunicazioni con altri agenti o da sé stesso (tramite *Mental Note*). I *Goal* (*Desire*) sono gli stati che un agente vuole raggiungere: sono l'impegno assunto dall'agente al fine di cambiare il mondo in uno stato in cui l'obiettivo sia vero (si parla di *Achievement Goal*); i *Goal* determinano il comportamento pro-attivo di un agente.

I cambiamenti nell'ambiente offrono nuove opportunità agli agenti, come adottare nuovi obiettivi o addirittura abbandonare quelli esistenti. Le modifiche alle *Belief* di un agente o ai suoi *Goal* generano nuovi eventi, in basi ai quali l'agente agisce; tali cambiamenti possono essere di due tipi: aggiunta o cancellazione. L'agente reagisce a tali eventi selezionando tutti i piani rilevanti (in grado di gestire quel determinato evento) e tra questi, solo alcuni saranno applicabili in base al contesto.

Il concetto di intenzione fa parte del Jason runtime e pertanto non è incluso nel linguaggio di programmazione [3]: un'intenzione è definita come un piano per raggiungere un obiettivo in risposta a un evento. Ogni intenzione è un insieme (più precisamente uno *stack*) di piani parzialmente istanziati [5].

I piani sono delle "ricette", un insieme di azioni che permettono di completare un qualche obiettivo e definiscono il comportamento pro-attivo e reattivo di un agente; essi hanno la seguente forma:

```
triggering_event: context <- body.
```

dove:

- `triggering_event`, denota l'evento scatenante per cui si può utilizzare un piano (aggiunta o rimozione di *Belief* o di *Goal*)
- `context`, il contesto, viene utilizzato per verificare se un determinato piano è in grado di gestire l'evento in base alle informazioni che un agente possiede: è un'espressione booleana per verificare le *belief* in possesso di un agente.
- `body`, il corpo del piano, è l'insieme delle azioni usate per gestire l'evento se il contesto è considerato valido nel momento in cui il piano è selezionato. Ci sono diverse formule che possono apparire in un piano: azioni interne (es. modifica dello stato interno dell'agente) o esterne (es. interazioni con ambiente), *Achievement Goal* (es. esecuzione di obiettivi secondari), *Test Goal* (es. interrogazione della *Belief Base*), *mental note* (es. aggiunta di b*elief*) ed espressioni;

Jason permette inoltre di definire anche piani per gestire il fallimento di un piano; questo evento è causato o dalla mancanza di piani applicabili per il raggiungimento di un *Goal* o dal fallimento di un *Test Goal* o ancora dal fallimento di un'azione [10].

Sebbene Jason offra un ambiente di sviluppo integrato per sviluppare un sistema multi-agente e la possibilità di creare ambienti, queste funzioni non verranno approfondite in quanto non utili all'analisi di JaCaMo.

2.2.3 KQML

L'abilità sociale rappresenta una caratteristica fondamentale per un agente: essa permette di comunicare con altri agenti per scambiare informazioni, delegare obiettivi e partecipare ai protocolli di negoziazione e di interazione.

Nei sistemi multi-agente la comunicazione tra gli agenti è tipicamente gestita utilizzando linguaggi di comunicazione per agenti (ACL, *Agent Communication Languages*), che si basano sulla teoria dello *speech-act* (in particolare sul lavoro di Austin e Searle). Questa teoria si basa sul presupposto che il linguaggio stesso è azione [6]: i messaggi sono atti comunicativi; un agente razionale fa un'affermazione nel tentativo di cambiare lo stato del mondo, nello stesso modo in cui un agente esegue azioni (esterne) per cambiare lo stato del mondo. Ciò che distingue gli *speech-act* dalle altre azioni è che il dominio del primo è tipicamente limitato allo stato(i) mentale(i) dell'ascoltatore(i) dell'enunciato, ossia alla modifica delle sue *Belief*, *Desire* e *Intention*.

Jason integra un linguaggio di comunicazione di agenti di alto livello, ispirato dal linguaggio KQML, basato sulla *speech-act theory*. In Jason, lo scambio di messaggi è asincrono: i messaggi sono conservati in una *mailbox* e all'inizio di ogni *reasoning cycle* un messaggio viene prelevato ed esaminato tramite una funzione di selezione, personalizzabile dal programmatore [6].

Knowledge Query and Manipulation Language (KQML) è stato il primo tentativo di definire un linguaggio pratico per la comunicazione come teorizzato nella letteratura sull'intelligenza artificiale distribuita. Concettualmente, è possibile identificare tre livelli in un messaggio KQML: contenuto, comunicazione e messaggio. Il contenuto contiene il messaggio nel linguaggio di rappresentazione del programma (il KQML può contenere qualsiasi linguaggio di rappresentazione personalizzato); il livello di comunicazione codifica nel messaggio una serie di caratteristiche che descrivono i parametri di comunicazione, come l'identità del mittente e del destinatario, o un identificatore univoco associato alla comunicazione; infine il livello del messaggio codifica il tipo di messaggio che un agente vuole trasmettere. Questo livello determina i tipi di interazioni (asserzione, *query* o comando) che si possono avere tra diversi agenti, definendo un insieme di atti performativi.

Gli atti performativi in Jason si basano ed estendono quelli definiti in KQML [5]: tramite un atto comunicativo, l'azione `.send`, è possibile inviare un messaggio specificando atto performativo e contenuto. Tra gli atti performativi possiamo distinguere:

1. `tell`: il mittente informa il destinatario di nuove informazioni (aggiunta o modifica di una *belief* nella *Belief Base* del destinatario);
2. `untell`: il mittente informa il destinatario che il contenute del messaggio è falso (rimozione di una *belief* nella *Belief Base* del destinatario);
3. `achieve`: il mittente chiede al destinatario di raggiungere uno stato del mondo in cui il contenuto del messaggio sia vero; in altre parole, il mittente delega un obiettivo;
4. `unachieve`: il mittente chiede al destinatario di abbandonare un obiettivo;
5. `tellHow`: il mittente informa il destinatario di un piano;
6. `untellHow`: il mittente informa il destinatario di ignorare un piano
7. `askIf`: il mittente chiede al destinatario se il contenuto del messaggio sia vero;
8. `askAll`: il mittente chiede al destinatario tutte le risposte a una domanda;
9. `askHow`: il mittente chiede al destinatario tutti i suoi piani per trattare uno specifico evento.

2.3 Programmare l'ambiente: EOP

Quello di ambiente è un concetto fondamentale nello sviluppo di un sistema multi-agente: è il luogo in cui gli agenti operano. Gli agenti sono in grado di percepire l'ambiente attraverso dei sensori e modificarlo a proprio piacimento attraverso degli attuatori. L'ambiente in un sistema multi-agente è un'astrazione di prima classe con un duplice ruolo: l'ambiente permette agli agenti di esistere, media le loro interazioni e l'accesso alle risorse del sistema [29] e fornisce un'astrazione progettuale per la costruzione di un sistema multi-agente.

Principalmente possiamo distinguere due tipi di ambiente [26]. Secondo il campo dell'intelligenza artificiale, l'ambiente identifica il mondo esterno, un contesto esogeno in cui ogni agente ha una precisa sfera di influenza, dove può operare, e solo sfere sovrapposte possono essere controllate da più agenti [6]. Programmare un tale ambiente significa modellare una scatola nera (estranea al controllo del programmatore del MAS), una interfaccia in cui sono definite le percezioni e l'insieme delle azioni che un agente può usare per ispezionare o influenzare l'ambiente. Questo approccio è adottato da Jason, ma per lo scopo di questa relazione verrà ignorato.

Nella modellazione di MAS invece, l'ambiente diventa indigeno, diventa parte integrante del MAS stesso: non è più solo il target delle azioni degli agenti o un generatore di percezioni, ma è anche il mediatore delle interazioni tra agenti diversi. Questo tipo di ambiente, chiamato *Working Environment*, si pone come un livello ulteriore nel MAS, concettualmente collocato tra gli agenti e il mondo esterno. Esso permette di migliorare la gestione della complessità del sistema e fornisce una buona ortogonalità (*separation of concerns*). Senza questa ortogonalità, ad esempio, una lavagna (che permette una comunicazione indiretta tra gli agenti) dovrebbe essere implementata come un agente, ma ciò creerebbe un errore di progettazione: la lavagna non deve raggiungere in modo proattivo e autonomo un qualche obiettivo, ma deve essere usata dagli agenti come mezzo di comunicazione e coordinazione. Adottando quindi il nuovo approccio, la lavagna è implementata come una risorsa, usabile dagli agenti in termini di azioni e di percezione.

Questa nuova prospettiva ha portato alla nascita di un nuovo campo di ricerca, i cui risultati sono stati pubblicati come atti di una serie di workshop denominata E4MAS, dove l'ambiente diventa una nuova dimensione da programmare, una dimensione ortogonale, ma allo stesso tempo integrata, a quella degli agenti: gli agenti rappresentano la parte autonoma del sistema, mentre l'ambiente la parte computazionale, funzionale al lavoro degli agenti.

2.3.1 Il modello A&A

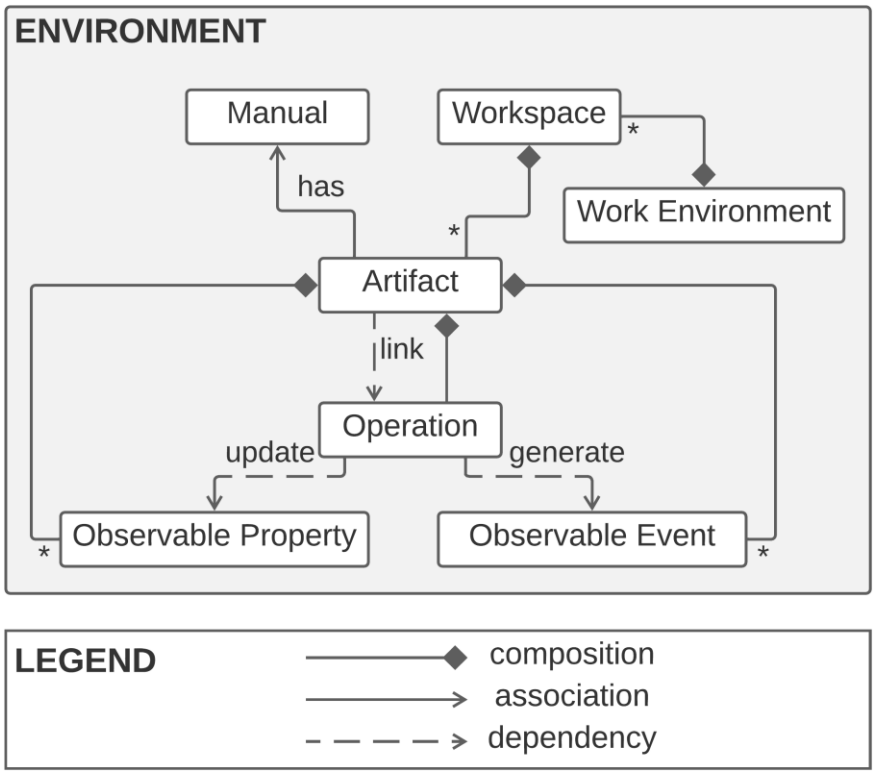


Figura 2.6: meta-modello A&A [26]

Il modello concettuale *Agents & Artifacts* (A&A) integra la nozione di agente autonomo, precedentemente definita, con le nozioni di artefatto (*Artifact*) e di spazio di lavoro (*Workspace*). Il modello definisce inoltre le possibili interazioni che si verificano tra queste astrazioni:

- comunicazione: gli agenti comunicano tra loro tramite un linguaggio di comunicazione di alto livello (Sezione 2.2.3);
- operazione: gli agenti possono interagire con gli artefatti;
- composizione: gli artefatti possono essere legati tra loro, tramite l'operazione di *link*;
- presentazione: gli artefatti si presentano agli agenti tramite un'interfaccia d'uso.

Secondo questo modello quindi, un MAS è concepito, progettato e sviluppato in funzione di agenti che operano e collaborano in un ambiente comune non solo comunicando attraverso uno specifico linguaggio, ma anche creando e utilizzando artefatti organizzati in diversi *workspace*. L'ambiente è quindi concepito come un insieme dinamico di entità computazionali chiamate artefatti, che rappresentano risorse e strumenti. L'insieme complessivo degli artefatti può essere organizzato in uno o più spazi di lavoro (*workspaces*), eventualmente distribuiti in diversi nodi della rete.

Un artefatto è quindi un'entità computazionale passiva e reattiva usata dagli agenti per collaborare, coordinare le operazioni e modificare l'ambiente del sistema: permette di incapsulare e modularizzare le funzionalità del sistema. Dal punto di vista del progettista MAS, la nozione di artefatto è un'astrazione di prima classe; essa permette di strutturare l'ambiente e modellare le funzionalità a disposizione degli agenti. Dal punto di vista degli agenti invece, gli artefatti sono entità di prima classe, che possono essere create, usate o osservate.

Gli agenti possono utilizzare un artefatto tramite un'interfaccia d'uso [26], composta da un insieme di proprietà osservabili e di operazioni (vedi Figura 2.7). Le proprietà osservabili sono le variabili di un artefatto percepibili dagli agenti che osservano l'artefatto: esse costituiscono le percezioni per gli agenti. Le operazioni invece rappresentano l'insieme di azioni esterne (eseguibili sull'ambiente), che un agente (o un altro artefatto) può eseguire su quell'artefatto: l'esecuzione di un'operazione può provocare sia il cambiamento dello stato interno dell'artefatto, sia generare un flusso di eventi osservabili (modifica di proprietà osservabili o generazione di segnali) che possono essere percepiti dagli agenti che stanno usando o semplicemente osservando l'artefatto.

L'insieme degli artefatti forniscono il repertorio di azioni a disposizione degli agenti. Questo repertorio in generale è dinamico, in quanto l'insieme degli artefatti può essere modificato dinamicamente dagli agenti [23]. Gli artefatti possono essere inoltre essere legati tra loro: un'operazione su un artefatto potrebbe innescare l'esecuzione di operazioni su altri artefatti.

Infine, un artefatto può essere dotato di un manuale, un documento consultabile dagli agenti, contenente una descrizione delle funzionalità fornite dall'artefatto e delle relative istruzioni operative. Tale caratteristica è stata concepita in particolare per sistemi aperti composti da agenti intelligenti che decidono dinamicamente quali artefatti utilizzare e come utilizzarli.

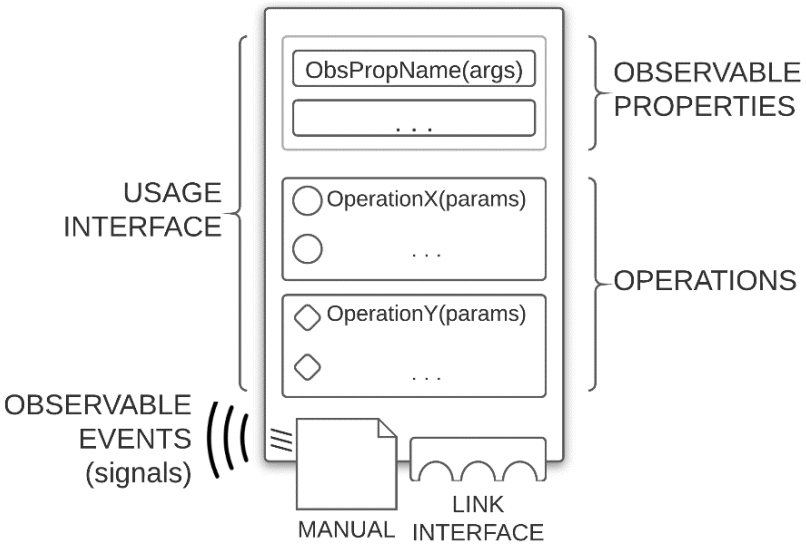


Figura 2.7: rappresentazione astratta di un artefatto nel modello A&A [26]

2.3.2 Tecnologia CArtAgO

CArtAgO (*Common ARtifact infrastructure for AGent Open environments*) è un framework e un'infrastruttura per la programmazione e l'esecuzione di ambienti basati su artefatti per MAS, che implementa il modello concettuale A&A descritto nella sezione precedente. È stato concepito in modo da essere ortogonale alla dimensione della programmazione degli agenti e pertanto è potenzialmente integrabile con qualsiasi linguaggio e piattaforma di programmazione per agenti esistente. In dettaglio, la piattaforma include:

- Un'API basata su Java per la programmazione di artefatti, per la definizione di nuovi tipi di artefatti secondo il modello di programmazione A&A (sfrutta le annotazioni Java);
- Un'API per integrare i linguaggi di programmazione orientati agli agenti con gli ambienti indigeni di CArtAgO. Questa API fornisce i mezzi per creare e interagire con gli artefatti e gestire i *workspace* locali e remoti;
- Una gestione a runtime della distribuzione e dell'esecuzione degli ambienti indigeni, dei *workspace* e del ciclo di vita degli artefatti.

CArtAgO è una tecnologia open-source implementata sulla piattaforma Java e rilasciata sotto licenza GNU LGPL [9].

2.3.3 Integrazione tra agenti Jason e ambiente CArtAgO

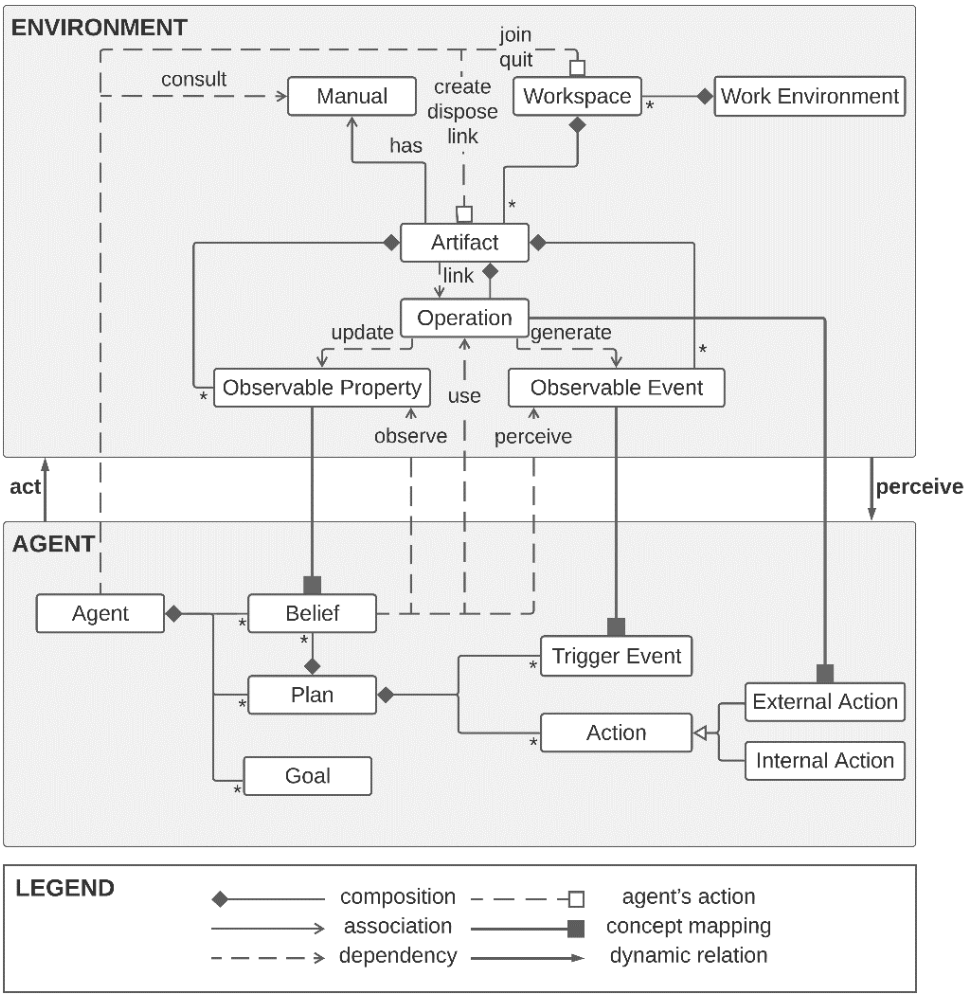


Figura 2.8: modello di interazione tra agente e ambiente

In questa sezione si analizzerà l'integrazione del modello A&A e del modello BDI, tramite l'analisi delle funzionalità principali offerte dalla piattaforma CArtAgO. Un tale programma, basato su questi modelli, è concepito come un insieme dinamico di agenti (BDI) Jason autonomi che lavorano in modo cooperativo all'interno di un ambiente di lavoro CArtAgO condiviso, la cui topologia è strutturata in termini di *workspace*, eventualmente distribuiti [3]. L'ambiente sarà composto da un insieme dinamico di artefatti, con cui gli agenti possono interagire dinamicamente.

In CArtAgO, si definisce un artefatto estendendo la classe `Artifact` [25]. Gli artefatti sono caratterizzati da un'interfaccia d'uso, contenente un insieme di proprietà osservabili e di operazioni.

Gli oggetti Java e i tipi di dati primitivi permettono di legare la dimensione degli agenti con quella degli artefatti, in particolare per codificare i parametri nelle operazioni, i campi nelle proprietà osservabili e i

segnali. Il metodo `init` è usato come costruttore di un artefatto, e permette di inizializzare il suo stato iniziale; i campi di istanza di una classe sono usati invece per implementare lo stato non osservabile dell'artefatto.

Le proprietà osservabili sono le variabili di stato di un artefatto e sono automaticamente percepite dagli agenti: sono infatti rappresentate come *belief* (dinamiche) nella *Belief Base* dell'agente che sta osservando (tramite operazione `lookupArtifact`) l'artefatto e possono essere usate per innescare l'esecuzione di un piano (`triggering_event`). Una proprietà osservabile è definita (solitamente durante l'inizializzazione di un artefatto) tramite la primitiva `defineObsProperty`, specificando il nome della proprietà e il valore iniziale; inoltre essa viene creata e aggiornata ogni qual volta un'operazione viene completata con successo, un segnale viene generato o un'operazione viene sospesa (primitiva `await`).

Le operazioni invece sono le azioni che un'agente può eseguire su un artefatto; sono implementate con metodi annotati con l'annotazione `@OPERATION`. All'interno delle operazioni, il valore delle proprietà osservabili può essere ispezionato e modificato dinamicamente (primitive `getObsProperty` e `updateObsProperty`). È possibile altresì creare operazioni a lungo termine e sospendere l'esecuzione dell'operazione, tramite la primitiva `await`, fino al verificarsi di una condizione specificata nella primitiva stessa (rappresentata da un metodo booleano annotato con l'annotazione `@GUARD`). Utilizzando invece la primitiva `signal`, un artefatto può rendere osservabile anche eventi che si verificano durante l'esecuzione di operazioni. Un'operazione può inoltre fallire (primitiva `failed`): in questo caso tutte le modifiche fatte alle proprietà osservabili verranno annullate e verrà inviato un *feedback*, contenente un messaggio d'errore, all'agente che ha eseguito l'operazione [26].

L'esecuzione di operazioni all'interno di un artefatto è transazionale (semantica *action-as-a-process*) [32]. Ciò implica che [26]:

- le modifiche allo stato osservabile di un artefatto sono fatte atomicamente e solo in caso di operazioni eseguite con successo;
- in fase di esecuzione possono essere invocate contemporaneamente più operazioni su uno stesso artefatto, ma solo un'operazione alla volta può essere eseguita (le altre azioni sono sospese). Pertanto non si possono verificare casi di *race condition* sullo stesso artefatto;
- le azioni possono essere operazioni a lungo termine il cui completamento verrà notificato asincronicamente all'agente;
- si possano implementare meccanismi di coordinamento basati sulla sincronizzazione delle azioni.

Un'operazione viene eseguita non appena un agente ne avvia l'esecuzione (l'agente esegue l'azione nel corpo di un piano se e solo se c'è almeno un artefatto che fornisce l'operazione corrispondente). Il piano dell'agente viene quindi sospeso fino al completamento dell'operazione (successo o fallimento). Sebbene un'operazione possa essere bloccata, l'agente che l'ha esegue non lo è: il suo ciclo di esecuzione prosegue, e

l'agente può eventualmente reagire a percezioni ed eseguire altri piani. In questo modo non viene limitata la reattività dell'agente.

Seguendo il meta-modello A&A, altre caratteristiche di CArtAgO includono:

- capacità di collegare (*linking*) tra loro gli artefatti: ciò rende possibile per un artefatto di eseguire operazioni su altri artefatti;
- capacità di eseguire operazioni interne (non visibili agli agenti);
- capacità di specificare per ogni artefatto un manuale per sfruttarne funzionalità.

2.4 Programmare l'organizzazione: OOP

L'organizzazione di un sistema multi-agente è l'insieme dei ruoli, delle relazioni e della gerarchia di autorità che ne governano il comportamento. È un'entità sovraindividuale, che aiuta gli agenti a cooperare e guida il modo in cui interagiscono per raggiungere un determinato obiettivo (o insieme di obiettivi) potenzialmente a lungo termine. Le specifiche di un'organizzazione sono necessarie per permettere all'organizzazione di "ragionare" su sé stessa: permettono di decidere quali agenti possono entrare o lasciare l'organizzazione, quali comportamenti siano permessi e quali siano gli obiettivi globali, i protocolli, i gruppi, i ruoli e le responsabilità di ogni agente membro (attraverso uno o più schemi di coordinamento, *cooperation scheme*).

Un'organizzazione può quindi aiutare gruppi di semplici agenti a mostrare comportamenti complessi per raggiungere un obiettivo comune, riducendo la complessità del loro ragionamento, l'incertezza e formalizzando gli obiettivi di alto livello (di cui nessun agente è a conoscenza). Allo stesso tempo però, le organizzazioni possono influire negativamente sulle interazioni computazionali, ridurre la flessibilità e la reattività degli agenti e possono aggiungere un ulteriore livello di complessità al sistema.

Inoltre in un tale sistema, l'obiettivo globale e l'autonomia degli agenti sono in conflitto. Infatti sebbene l'autonomia sia una proprietà essenziale in un MAS, essa riduce la coerenza interna nel raggiungimento dell'obiettivo globale: una soluzione a tale problema è quella di usare le norme, delle regole che influenzano il comportamento degli agenti e definiscono il loro coordinamento all'interno del sistema. Una norma quindi è l'obbligo o il permesso di compiere una qualche azione o di realizzare un qualche obiettivo; può anche avere una condizione che stabilisce quando essa è attiva o un termine da rispettare. Si possono distinguere due meccanismi per implementare le norme in un MAS [4]:

- *Regimentation*: le norme regimentate bloccano l'esecuzione delle azioni degli agenti che violano la norma; solitamente si regimentano alcune azioni per preservare le proprietà fondamentali del sistema (es. l'accesso a un computer)

- *Enforcement*: la violazione delle norme da parte degli agenti è resa possibile ma è monitorata e soggetta a incentivi; è dunque necessario rilevare le violazioni e applicare sanzioni per contenerle o premi per incentivarne il rispetto [15].

Mentre le norme regimentate sono un meccanismo preventivo, che limita l'autonomia degli agenti, l'enforcement è un meccanismo reattivo che permette agli agenti di obbedire o meno a una norma a seconda della loro attuale visione dell'organizzazione. Entrambe le tecniche devono essere usate e implementate attentamente per preservare quindi l'autonomia degli agenti e allo stesso tempo permettere loro di lavorare congiuntamente [15]. Un'organizzazione di questo tipo si dice organizzazione normativa.

2.4.1 Moise

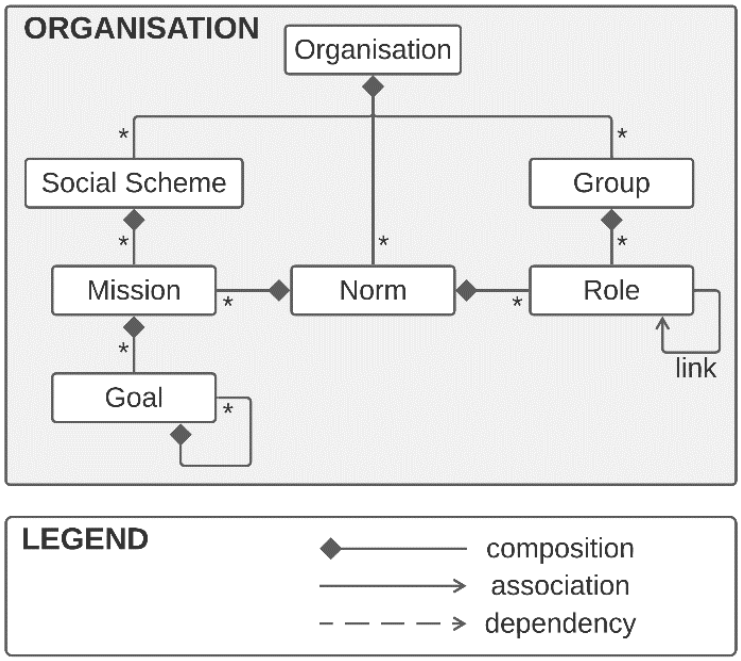


Figura 2.9: meta-modello Moise

Moise fornisce al programmatore un modello organizzativo e una infrastruttura per la programmazione e l'esecuzione di organizzazioni in un sistema multi-agente [22]. Moise usa un modello *organisation-centered*, in cui l'organizzazione esiste a priori (definita dal progettista o dagli stessi agenti) e in cui l'obiettivo principale dell'organizzazione viene raggiunto mentre gli agenti perseguono gli obblighi e i permessi specificati dai loro ruoli [12]. L'esecuzione, il funzionamento e l'evoluzione dell'organizzazione è gestita a tempo di esecuzione da un'infrastruttura organizzativa dedicata.

Il modello Moise scompone, attraverso una sintassi `xml`, le specifiche dell'organizzazione in tre dimensioni: la dimensione strutturale, la dimensione funzionale e quella deontica [12]. La dimensione

strutturale e la dimensione funzionale possono essere specificate indipendentemente l'una dall'altra, ma esse sono legate dalla dimensione deontica.

La dimensione strutturale permette di suddividere l'organizzazione sotto l'aspetto individuale, attraverso i ruoli (*Role*), sotto l'aspetto collettivo, attraverso i gruppi (*Group*), e sotto l'aspetto sociale, attraverso i collegamenti, ossia le interconnessioni tra i ruoli (*Link*). I gruppi sono un insieme di collegamenti, ruoli e relazioni di compatibilità (*compatibility constraints*); sono utilizzati per fornire agli agenti che ne fanno parte un contesto condiviso: quando un agente decide di svolgere un ruolo in un gruppo, accetta alcuni vincoli comportamentali in funzione del suo ruolo. I collegamenti invece definiscono le relazioni tra ruoli diversi (*authority*, *communication*, *acquaintance*) [14]: essi vincolano direttamente gli agenti durante le loro interazione (nota: al momento in JaCaMo, il concetto di *Link* non è sato ancora completamente implementato).

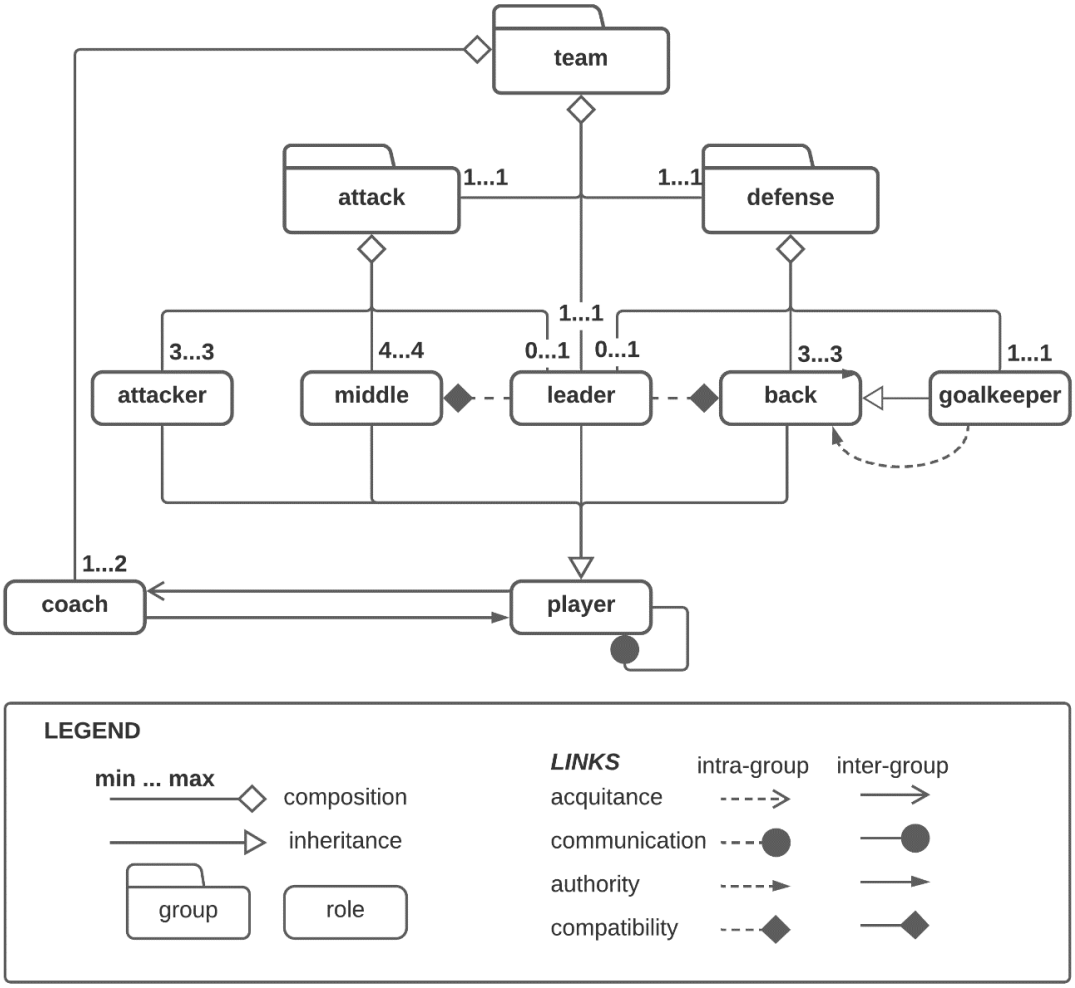


Figura 2.10: rappresentazione grafica delle specifiche strutturali di una quadra di calcio

La dimensione funzionale specifica come gli obiettivi globali debbano essere raggiunti, ossia come questi obiettivi siano scomposti, e raggruppati in insiemi di obiettivi coerenti, le missioni (*Mission*) che possono essere assegnate ai diversi ruoli. La scomposizione degli obiettivi globali si traduce in un albero degli

obiettivi, chiamato schema (*Social Scheme*), dove le foglie (i nodi terminali) rappresentano gli obiettivi (*Goal*) che possono essere raggiunti singolarmente dagli agenti. Ogni schema è assegnato a un singolo gruppo.

In uno schema ci possono essere diversi tipi di *Goal*:

- *Performance Goal* (tipo predefinito): questi obiettivi devono essere dichiarati come raggiunti (*done*) dagli agenti a cui sono stati affidati, una volta completati;
- *Achievement Goal*: questi obiettivi devono essere dichiarati come soddisfatti (*satisfied*) dagli agenti a cui sono stati affidati, una volta completati;
- *Maintenance Goal*: questi obiettivi non vengono completati in un preciso momento, ma vengono perseguiti mentre lo schema è in esecuzione. Gli agenti, a cui sono stati affidati, non hanno bisogno di dichiararli soddisfatti [14].

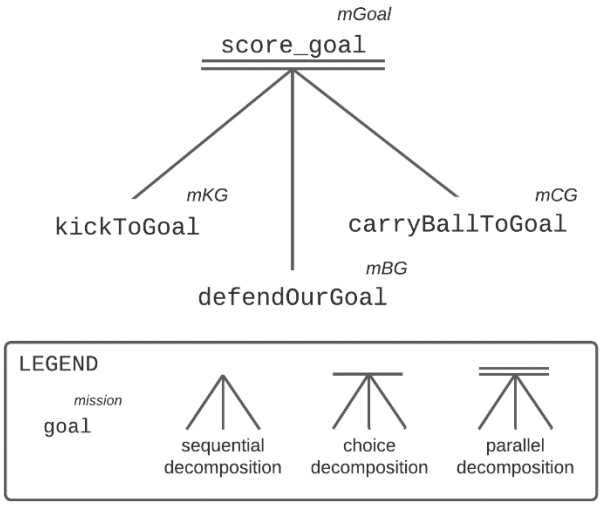


Figura 2.11: rappresentazione grafica di uno schema per una squadra di calcio

In uno schema organizzativo, un *Goal* può essere decomposto in diversi *sub-goal* che possono essere completati secondo tre modalità: in sequenza (*sequence*), in parallelo (*parallelism*) o a scelta (*choice*). Nel primo modo, un *goal* B viene affidato a un nuovo agente solo dopo che il *goal* A (che lo precede nello schema) è stato assegnato e raggiunto (*done* o *achieve*). Nel secondo caso, invece i goal possono essere assegnati in parallelo, pertanto non è importante l'ordine in cui vengono completati. Nell'ultimo caso, è necessario completare solo uno fra i *sub-goal* disponibili. In tutti i tre i casi è necessario completare uno o più *sub-goal*, affinché sia possibile completare il *goal* (padre) che li ha "generati". I *Maintenance Goal*, in quest'ottica, rappresentano ancora una sfida aperta, perché sebbene teorizzati, non sono stati ancora implementati correttamente.

ROLE	DEONTIC RELATION	MISSION
attacker	obligation	mKG
back	obligation	mBG
middle	obligation	mCG
player	obligation	mGoal

Figura 2.12: rappresentazione grafica delle specifiche deontiche di una quadra di calcio

La dimensione deontica (anche detta normativa o sociale) infine viene usata per collegare la dimensione strutturale a quella funzionale specificando, attraverso le norme (*Norm*), i permessi (*permission*) e gli obblighi (*obligation*), di ciascun ruolo per ciascuna missione. Un permesso permette agli agenti che svolgono un certo ruolo di decidere se dedicarsi o meno a una missione. Un obbligo invece costringe gli agenti che interpretano un ruolo a completare una missione.

2.4.2 Integrazione tra organizzazione Moise e ambiente CArtAgO

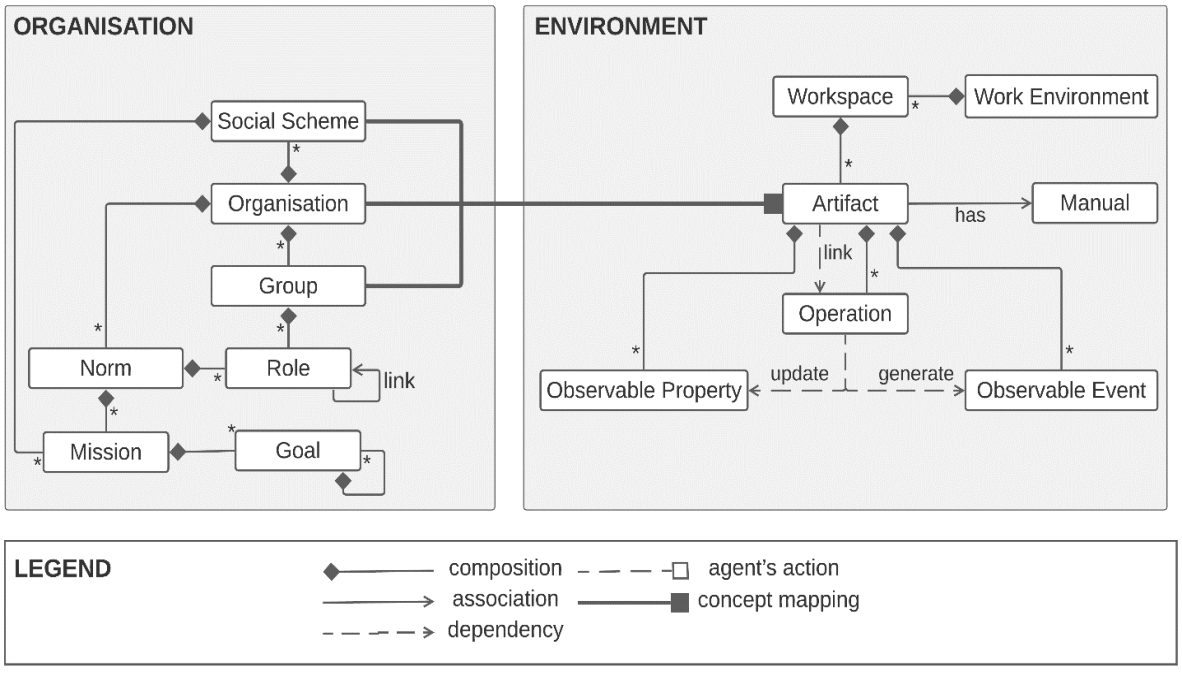


Figura 2.13: modello di interazione tra Ambiente e Organizzazione

Per integrare la dimensione organizzativa a quella dell'ambiente, si è scelto di modellare l'organizzazione come un insieme di artefatti: le entità computazionali (gruppi, schemi e stati normativi) che gestiscono lo stato attuale dell'organizzazione sono state rese come artefatti organizzativi, che incapsulano le specifiche dell'organizzazione. Inoltre per questo scopo, è stato definito il linguaggio NOPL (*Normative Organization Programming Language*) per tradurre le specifiche Moise in modo da essere interpretate dagli agenti: gli artefatti

organizzativi infatti incorporano un interprete NOPL per fornire l'infrastruttura organizzativa necessaria per gestire un'organizzazione Moise a runtime [3].

Dal punto di vista degli agenti, tali artefatti forniscono l'insieme di azioni per partecipare pro-attivamente ad un'organizzazione e forniscono le proprietà osservabili per renderne percepibile lo stato. Ciò permette agli agenti membri di ragionare sull'organizzazione stessa e di modificarla a tempo di esecuzione. Questo *mapping* fornisce al sistema uniformità (l'interazione tra agenti e organizzazione è basato sul modello A&A) e dinamismo (la forma dell'organizzazione si modifica in funzione degli artefatti organizzativi), e permette di distribuire e gestire l'organizzazione [3].

Gli artefatti organizzativi [3] adottati in un'applicazione JaCaMo sono:

- Artefatti *OrgBoard* - utilizzati per tenere traccia dello stato attuale delle entità organizzative (viene creata un'istanza per ogni organizzazione).
- Artefatti *GroupBoard* - utilizzati per gestire il ciclo di vita di specifici gruppi di agenti; tra le operazioni principali troviamo la possibilità di assumere o abbandonare un ruolo, o uno schema; tra le proprietà osservabili, l'elenco dei membri (`player`) di un gruppo e quello degli schemi affidati al gruppo.
  Esempio d'uso: se un agente sceglie di adottare un ruolo in un particolare gruppo, esso eseguirà l'azione `adoptRole` sul manufatto *GroupBoard* che rappresenta il gruppo;
- Artefatti *SchemeBoard* - utilizzati per gestire l'esecuzione degli schemi; tra le operazioni principali, troviamo la possibilità per un ruolo di dedicarsi a una missione (o lasciarla), di completare (o resettare) un obiettivo. Tra le proprietà osservabili, ci sono gli obblighi e i permessi per ciascun ruolo e lo stato dei goal.
  Esempio d'uso: (tramite questo artefatto) un agente può impegnarsi in una missione (`commitMissions`) o comunicare all'organizzazione di aver completato un obiettivo di una particolare missione (`setGoalAchieved`) sullo specifico *SchemeBoard*.

Nel contesto di un MAS, inoltre è spesso usato il meccanismo delle regimentazioni per garantire il rispetto delle norme. In Moise, gli artefatti sono responsabili di assicurare che tutte le norme vengano rispettate dagli agenti e permettono di rilevare la violazione delle norme. Tuttavia essi non applicano alcuna sanzione: tali politiche infatti sono delegate agli agenti organizzativi.

Come ultima osservazione, l'insieme degli artefatti organizzativi (di una stessa organizzazione) sono collegati tra loro tramite l'operazione di *linking* fornita da CArtAgO: ciò è necessario per mantenere coerente lo stato complessivo di un'organizzazione distribuita su molteplici artefatti.

2.4.3 Integrazione tra organizzazione Moise e agenti Jason

Per integrare la dimensione organizzativa a quella degli agenti, gli obiettivi definiti nelle specifiche dell'organizzazione sono mappati in obiettivi individuali per ciascun agente con uno specifico ruolo in uno specifico gruppo; questa assegnazione è determinata dagli obblighi e dai permessi presenti nelle specifiche deontiche dell'organizzazione. Sull'artefatto *SchemeBoard* vengono memorizzati lo stato corrente degli obiettivi: un obbligo è soddisfatto solamente se un agente raggiunge l'obiettivo corrispondente prima della sua scadenza. L'agente una volta percepiti tali obblighi tra le proprietà osservabili dell'artefatto *SchemeBoard* corrispondente, può scegliere se perseguirli o meno: se lo fa, verrà creato un corrispondente obiettivo nell'agente. Il mapping tra gli obiettivi dell'organizzazione e gli obiettivi individuali è sotto il diretto controllo del processo decisionale dell'agente: in questo modo si preserva l'autonomia degli agenti, che sono quindi liberi di decidere se perseguire o meno un obiettivo e quale linea d'azione usare per raggiungerlo [3].

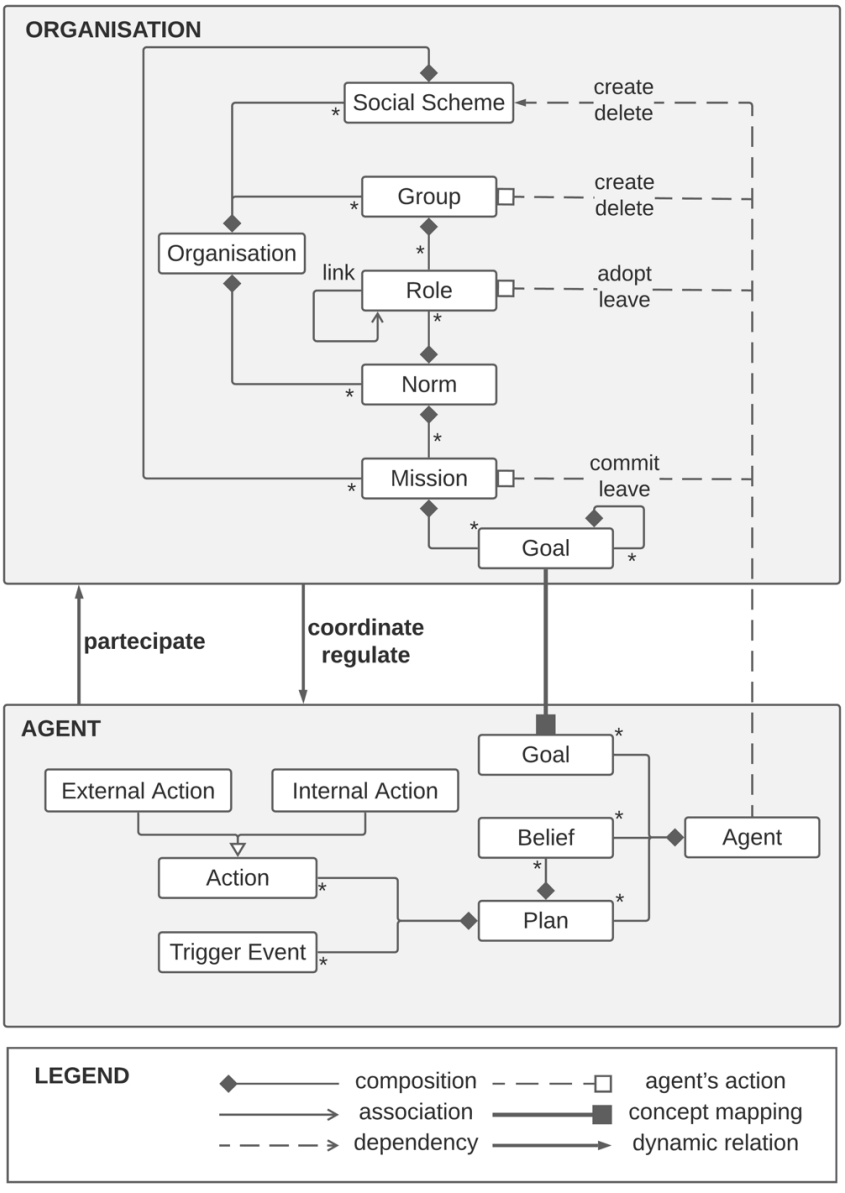


Figura 2.14: Modello di interazione tra Agente e Organizzazione

Capitolo 3
Il caso di studio

Il sistema multi-agente realizzato durante il tirocinio consiste nella simulazione di una organizzazione no-profit (ONP) in JaCaMo. Si è scelto di rappresentare il mondo popolato dagli agenti come quello di un'ipotetica società. In questa società esistono due macro-categorizzazioni: quella data dall'ente no-profit e quella composta dai cittadini comuni. L'ente è una organizzazione complessa, caratterizzata da diversi ruoli con diverse mansioni; ha come obiettivo principale quello di aiutare tutti i cittadini in difficoltà promuovendo campagne di donazione, reclutando nuovi membri e coordinando i vari progetti. Dalla parte opposta, ci sono i cittadini, gli agenti che non fanno parte dell'organizzazione, ma che lavorano, guadagnano, donano ed eventualmente possono entrare nell'ente in qualità di volontari. Tra questi inoltre sono anche presenti cittadini in difficoltà che non possono lavorare, ma che necessitano di aiuto da parte dell'ente benefico.

Nei prossimi capitoli si analizzerà l'organizzazione del caso di studio, si metteranno in evidenza le peculiarità delle specifiche strutturali, funzionali e deontiche di Moise in relazione anche al codice degli agenti membri dell'organizzazione in modo da evidenziare i meccanismi di interazione che intercorrono tra le entità; si mostrerà infine il ciclo di vita degli agenti e dei principali ruoli, e gli artefatti utilizzati.

3.1 Specifiche strutturali

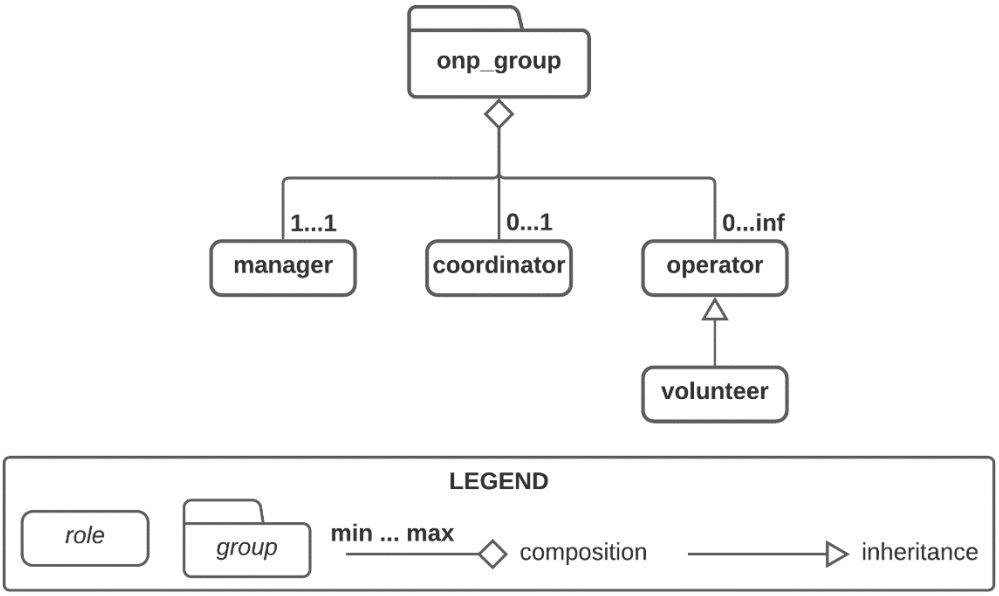


Figura 3.1: struttura dell'organizzazione

Sorge spontaneo modellare l'organizzazione no-profit come un'organizzazione Moise. Questa organizzazione è composta da un singolo gruppo, in cui sono presenti i tre ruoli principali: il manager, il coordinatore (*coordinator*) e gli operatori (*operator*). Il manager ha il compito di creare l'organizzazione, di assumere il suo staff, e di gestire i programmi (*project*) di aiuto per i cittadini in difficoltà. Il coordinatore invece stabilisce a quali volontari (eventualmente reclutandone di nuovi) assegnare le campagne di donazione; e infine gli operatori si occupano delle campagne stesse: come in un *call center*, contattano ripetutamente i cittadini per promuovere un progetto.

Come si evince dalla struttura riportata in Figura 3.1, formalmente il ruolo di volontario (*volunteer*) estende il ruolo di operatore e ne eredita tutte le proprietà (relazione di *sub-roling*). Questa scelta permette di ampliare e differenziare gli agenti operatori, così da poter affidare loro mansioni diverse, pur lasciandoli sempre soggetti alle stesse limitazioni (come ad esempio potrebbe essere un vincolo sulla cardinalità). Nella visione iniziale, erano presenti diversi tipi di operatore: sebbene avessero dovuto tutti promuovere uno stesso progetto, la fase finale dello stesso sarebbe stata completata solo dal ruolo corrispondente. Se ad esempio un progetto avesse come scopo quello di costruire una casa, solo un ipotetico ruolo costruttore potrebbe completare quell'obiettivo. Nel caso di studio tuttavia, questa funzionalità è stata tralasciata poiché poco interessante da implementare, in quanto i diversi operatori avrebbero avuto un codice del tutto simile e ciò non avrebbe portato ad alcuna miglioria significativa. La complessità tuttavia sarebbe rimasta invariata, in quanto sarebbe stato compito dell'organizzazione stessa creare e gestire i diversi obblighi (o permessi) affinché ciascun ruolo *operator* (es. costruttore) completasse il rispettivo compito. Ai fini della relazione pertanto, il termine operatore sarà usato come sinonimo di volontario, e viceversa.

Inoltre, come teorizzato nelle pubblicazioni su Moise, anche in JaCaMo è possibile specificare dei vincoli tra ruoli appartenenti ad uno stesso gruppo o a gruppi diversi: i link di *authority*, di *communication* e di *acquaintance*. Essi regolano la gerarchia e le interazioni tra ruoli diversi. Al momento tuttavia questa funzionalità non è stata implementata completamente in JaCaMo, pertanto non è stato possibile utilizzarla (sebbene potesse essere utile) ai fini della simulazione e per questo motivo non è stata riportata in Figura 3.1.

Un agente che vuole partecipare a un'organizzazione con un dato ruolo deve in primo luogo entrare nel workspace che contiene l'organizzazione (`joinWorkspace(workspace, Id)`), cercare (`lookUpArtifact(artifact, ArtId)`) e osservare (`focus(ArtId)`) l'artefatto *OrgBoard*. Una volta creato uno schema, l'artefatto OrgBoard aggiorna la proprietà osservabile `schemes([scheme_1, … , scheme_n])`; l'agente che sta osservando l'artefatto rileva la modifica e reagisce, osservando l'artefatto *SchemeBoard* appena creato. L'artefatto *SchemeBoard* crea delle nuove proprietà osservabili *obligation* che l'agente usa per dedicarsi a una missione ed eseguire gli obiettivi corrispondenti. Una volta completati, l'agente può dichiarare l'obiettivo come soddisfatto.

```
1. // obligation to commit to a mission
2. +obligation(Ag,_,committed(Ag,M,Sch),_)[artifact_id(ArtId),workspace(_,_,W)]
3.   : .my_name(Ag)
4.   <- .print("I am obliged to commit to ",M,"on",Sch");
5.      commitMission(Mission)[artifact_name(Sch), wid(W)].
6.
7. // obligation to achieve a goal
8. +obligation(Ag,Norm,What,Deadline)[artifact_id(ArtId)]
9.   : .my_name(Ag) & (satisfied(Scheme,Goal) = What
10.    | one(Scheme,Goal,Ag) = What)
11.  <- !Goal[scheme(Scheme)];
12.     goalAchieved(Goal)[artifact_id(ArtId)].
```

Figura 3.2: piani per rendere un agente "obbediente" all'organizzazione (file `org-obedient.asl`*)*

JaCaMo, per semplificare questi passaggi, offre alcuni *template* predefiniti in cui sono raccolti dei piani per automatizzare alcune operazioni. È possibile definire *workspace*, artefatti, organizzazione e ruoli in un file di configurazione `.jcm` (in cui vengono anche definiti tutti gli agenti che partecipano al MAS): i piani definiti nei *template* permettono infatti agli agenti di entrare automaticamente nei workspace e osservarne gli artefatti (`common-cartago.asl`) precedentemente definiti (e che vengono istanziati automaticamente a runtime dal *launcher* di JaCaMo), permettono di entrare in un'organizzazione e assumere un ruolo in un gruppo e rilevare i nuovi schemi per quel gruppo (`common-moise.asl`) e infine di rendere un agente "obbediente" all'organizzazione (`org-obedient.asl`), in modo che esso persegua le missioni affidate al suo ruolo e completi gli obbiettivi (*obligation*) generati dall'organizzazione.

In Figura 3.2 sono riportati alcuni piani per mostrare come un agente reagisce a un *obligation*, generata da uno *SchemeBoard*; la Figura 3.3 mostra invece l'interazione tra il manager e un agente e come quest'ultimo entri in un'organizzazione [riga 13-16] in qualità di coordinatore.

```
1.  //manager
2.  +!hire_staff[scheme(Sch)]
3.  <- .wait(500);
4.     .send(agent2, achieve, hire(coordinator));
5.     .goalAchieved(hire_staff)[artifact_name(Sch)].
6.
7.  //coordinator
8.  +!hire(Role)
9.  <- joinWorkspace(environment, Wid);
10.    lookupArtifact(yellowpages, YId);
11.    focus(YId);
12.    lookupArtifact(clock, CId);
13.    focus(Cid);
14.    joinWorkspace(ora4mas, Worg);
15.    lookupArtifact(onp_group, ArtGr)[wid(Worg)];
16.    focus(ArtGr)[wid(Worg)];
17.    adoptRole(Role).
```

Figura 3.3: interazione tra il ruolo manager e il ruolo coordinator

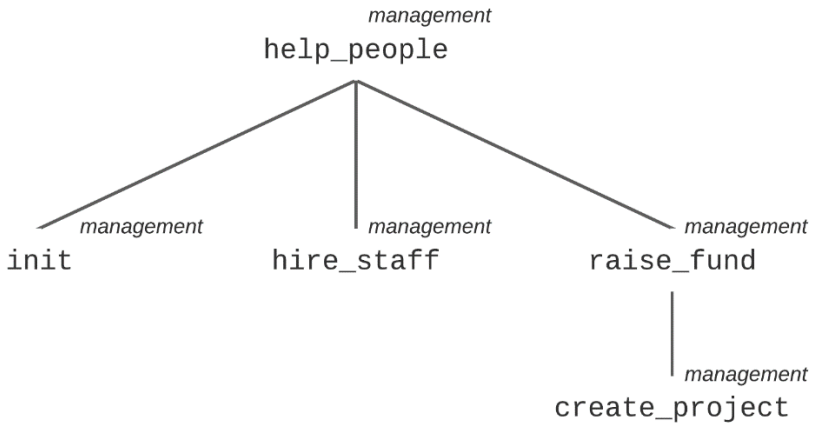
3.2 Specifiche funzionali

Per coordinare il raggiungimento dell'obiettivo dell'organizzazione (`help_people`), sono stati definiti due diversi schemi.

3.2.1 Lo schema principale

Un agente (`manager.asl`), dopo avere creato l'artefatto organizzativo *OrgBoard*, il gruppo `onp_group` e il primo schema `main_scheme` (istanziando l'artefatto *SchemeBoard* corrispondete) nel workspace `ora4mas` crea e affida lo schema all'unico gruppo dell'organizzazione `onp_group`; infine assume il ruolo di manager all'interno del gruppo stesso e persegue la missione *management*, che raggruppa tutti gli obiettivi presenti nello schema principale (`main_scheme`). In questo schema, l'obiettivo `help_people` è decomposto principalmente in tre sotto-obiettivi. Questi obiettivi sono raggruppati in una singola missione affidata all'unico agente al momento presente all'interno dell'organizzazione: il manager. Perseguendo la missione infatti, il manager, in sequenza, crea nel workspace environment gli artefatti accessori all'ente benefico (goal `init`): l'artefatto `YellowPages` e l'artefatto `Clock` (Sezione 3.6). Successivamente assume un agente con il ruolo di coordinatore (obiettivo `hire_staff`) e crea un primo progetto (goal `create_project`), il cui obiettivo sarà quello di raccogliere delle donazioni per aumentare i fondi dell'ente benefico. Il coordinatore

invece, ricevuta la richiesta di completare l'obiettivo `hire(coordinator)` da parte del manager individua e osserva gli artefatti appena creati e si unisce all'organizzazione; il codice è riportato in Figura 3.3.



mission	cardinalities
management	1...1

Figura 3.4: rappresentazione della decomposizione dello schema main_scheme in sub-goal, missioni e loro cardinalità

I progetti rappresentano il cuore dell'organizzazione e della simulazione. Essi sono entità dinamiche con un ciclo di vita complesso: vengono istanziati dal manager per aumentare i fondi dell'organizzazione (quando il budget dell'ente scende sotto una certa soglia) o in seguito al contatto di un cittadino in difficoltà con un operatore; il coordinatore, rilevato il nuovo progetto, lo assegna ad alcuni volontari, che lo promuoveranno; infine dopo aver raccolto abbastanza donazioni, un progetto entra nella fase finale, dove verrà completato (con successo); in alternativa un progetto può anche fallire qualora i volontari non riescono a raccogliere la somma desiderata entro la sua scadenza (specificata da un timer nell'artefatto progetto corrispondente); in questo caso, sarà il manager a valutare come procedere.

Nelle fasi iniziale del tirocinio, l'organizzazione era stata composta con un singolo schema organizzativo e i progetti erano stati modellati esclusivamente come artefatti; ciò tuttavia risultava essere troppo semplice e non garantiva un efficace meccanismo di coordinamento: molti obiettivi dello schema erano quasi sempre attivi e non dovevano essere mai raggiunti (erano quindi dei *Maintenance Goal*, vedi Sezione 3.2.2). In questa versione, l'organizzazione non migliorava, anzi ostacolava la modellazione del sistema. Pertanto si è scelto di plasmare i progetti non solo come artefatti, ma anche modellarne le fasi in uno schema.

3.2.2 Maintenance Goal

Uno tra gli ultimi obiettivi dello schema principale, `create_project`, è un *Maintenance Goal*. Come affermato nel Capitolo 2 (Sezione 2.4.1), un *Maintenance Goal* è un obiettivo che non deve essere necessariamente completato in un preciso momento, ma rimane attivo e deve essere perseguito per tutta la durata dello schema. In Moise (e di conseguenza in JaCaMo) tuttavia, il suo funzionamento è assai limitato e lontano dall'idea originale. Attualmente gli schemi non supportano i *Maintenance Goal* [30]: il goal transita automaticamente nello stato *enable* (viene attivato), ma non viene generato alcun obbligo e di conseguenza l'obiettivo non viene affidato ad alcun agente (l'obiettivo però potrà comunque essere completato solo dal ruolo a cui è stata affidata la missione corrispondente); se si colloca inoltre un *Maintenance Goal* in una sequenza, esso bloccherà l'esecuzione dei goal successivi, fintantoché non verrà dichiarato soddisfatto (stato *satisfied*). Ciò è in contraddizione con la definizione: un *Maintenance Goal* non dovrebbe essere considerato come pre-condizione per i goal successivi (ossia non dovrebbe essere dichiarato soddisfatto) [30]. Come affermato dagli autori di JaCaMo, "i *Maintenance Goal* non sono ancora supportati e ci sono ancora molti problemi e dubbi a riguardo"[1]. Attualmente durante l'esecuzione dello schema, dal punto di vista di Moise, essi sono considerati come *Achievement Goal* [1]. Devono essere quindi gli agenti (e non l'organizzazione) a perseguire questi goal in un modo particolare: gli agenti accettano il goal e creano un secondo goal per mantenere attivo il goal precedentemente accettato; il modo attuale per programmarlo e il seguente:

```
1. // g is the org goal as in moise scheme, mg is the agent internal goal to maintain g
2. +!g <- .print(ok);
3.        !!mg.
4.
5. +!mg <- ...;
6.         !mg.
```

Figura 3.5: codice di un agente per mantenere attivo un Maintenance Goal[1]

Nel caso di studio, come mostrato in Figura 3.6, il manager inizialmente rileva grazie alla proprietà osservabile `goalState`, generata dallo *SchemeBoard*, l'obbligo di completare il *Maintenance Goal* `create_project`: il manager istanzia quindi un nuovo progetto [riga 1-4]. Per mantenere attivo quest'obiettivo, il manager creerà un nuovo goal (`!!create_project`) al verificarsi di specifiche condizioni: esso crea un nuovo progetto ogni volta che il budget dell'ente benefico scende sotto una certa soglia (a patto che ci sia solamente un progetto in corso) per aumentare i fondi dell'ente [riga 6-10] oppure qualora un agente operatore venga a contatto con un cittadino in difficoltà [riga 12-13].

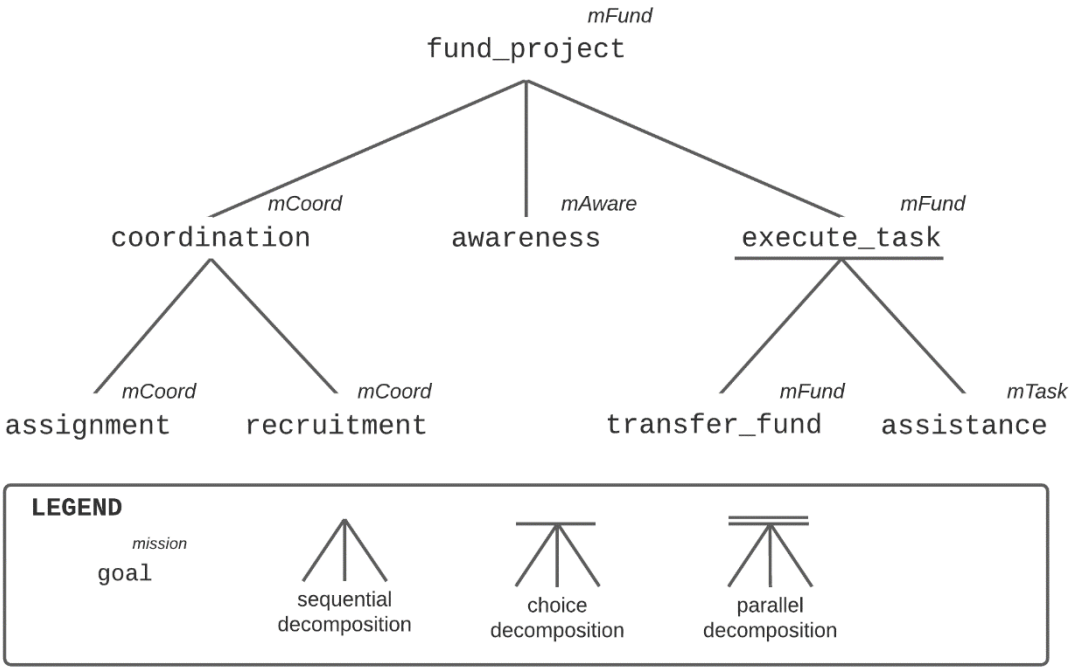
[1] Data l'assenza di documentazione, le informazioni relative ai *Maintenance Goal* sono state ottenute contattando direttamente gli sviluppatori del progetto

Lo schema principale di conseguenza non verrà mai completato dal manager, in quanto l'organizzazione rimarrà sempre in attesa del verificarsi di una delle due condizioni precedentemente descritte.

```
1. +goalState(_, create_project, _, _,enabled)
2.  : goalState(_,hire_staff, _, _,satisfied)
3.  <- .my_name(Me);
4.     !!create_project(fundraising, 0, Me, money).
5.
6. +money(M)
7.  : M < 300 & not fundraising
8.  <- .my_name(Me);
9.     +fundraising;
10.    !!create_project(fundraising, 0, Me, money).
11.
12. +candidate(Ag, Trouble)
13.  <- !!create_project(Ag, math.random * 1000 + 300, Ag, Trouble).
14.
15. +!create_project(Project, Target, Rec, Problem)
16.  : joined(ora4mas, WOrg)
17.  <- .concat(sch_, Project, Sch);
18.    makeArtifact(Project,"onp.Project",[Target,Rec,Problem,Sch],ArtPr);
19.    focus(ArtPr);
20.    createScheme(Scheme, project_scheme, SchArtId);
21.    setArgumentValue(fund_project,"project_art",Sch)[artifact_id(SchArtId)];
22.    addScheme(Scheme);
23.    focus(SchArtId)[wid(WOrg)];
24.    -+ongoing(N+1);
25.    commitMission(mFund)[artifact_id(SchArtId)].
```

Figura 3.6: codice dell'agente manager per mantenere attivo il Maintenance Goal create_project

3.2.3 Lo schema di un progetto



mission	cardinalities
mCoord	1...1
mAware	0...inf
mFund	1...1
mTask	0...inf

Figura 3.7: rappresentazione grafica dello schema di un progetto e cardinalità delle sue missioni

Lo schema di un progetto (`project_scheme`) permette di monitorarne l'evoluzione. Una volta istanziato dal manager, un progetto attraversa tre fasi (formalmente il goal `fund_project` è decomposto in tre sotto-obiettivi, Fig. 3.7).

Nella prima fase, `coordination`, il coordinatore cerca di assegnare il progetto a cinque volontari: invita in un primo momento (goal `assignment`) tre volontari a dedicarsi al progetto (un volontario può dedicarsi a più progetti contemporaneamente), e in una fase successiva (goal `recruitment`) recluta due nuovi volontari. Normalmente, completato il goal `coordination`, ci possono essere tra i due e i cinque volontari assegnati a un progetto; un volontario contattato può infatti accettare o meno di lavorare a un progetto: in entrambi i casi, esso non deve inviare alcun messaggio al coordinatore per notificare la sua decisione, ma sarà il coordinatore stesso ad accorgersene controllando gli agenti che operano nello schema del progetto: tramite la proprietà osservabile `goalState` [Fig. 3.8, riga 18], il coordinatore può ottenere la lista di tutti gli agenti che hanno effettuato il *commit* alla missione `awareness` e verificare se l'agente invitato partecipa alla missione. Una precisazione sull'obiettivo `recruitment`: l'unico caso in cui il coordinatore non può reclutare

nuovi volontari si verifica quando il numero dei volontari è superiore ai tre quarti della società (proprietà osservabile nell'artefatto `YellowPages`). Come caso limite quindi, il numero di volontari può essere uguale a zero, qualora tutti gli operatori rifiutino di partecipare e non sia possibile reclutarne di nuovi: in questo caso il progetto fallirà.

Il codice del coordinatore fa largo uso dei *contingency plan*. Essi permettono di definire molteplici piani per trattare uno stesso evento a seconda del contesto (la rappresentazione del mondo dell'agente): i piani vengono selezionati nello stesso ordine in cui sono riportati nel codice (dall'alto verso il basso).

```
1.  @org2[atomic]
2.  +!assignment[scheme(Sch)]
3.   <- .findall(Ag, play(Ag, operator, _), List);
4.      !assign(2, List)[scheme(Sch)].
5.
6.  +!assign(N, List)[scheme(Sch)]
7.   : not .empty(List) & N > 0
8.   <- .nth(0, List, Ag);
9.      .delete(Ag, List, L);
10.     .send(Ag, tell, hire_vol(operator, "mAware", Sch));
11.     waitTicks(2); //artifact Clock's operation
12.     !assign_aux(Ag, N, L)[scheme(Sch)].
13.
14. +!assign(N, _)[scheme(Sch)]
15.  <- .print("assigned to the scheme ", 2-N, " volunteers").
16.
17. +!assign_aux(Ag, N, List)[scheme(Sch)]
18.  : goalState(Sch, awareness, CommittedAgents, _, _)
19.    & .member(Ag,CommittedAgents)
20.  <- .print("assigned ", Ag, " to ", Sch);
21.     !assign(N-1, List)[scheme(Sch)].
22.
23. +!assign_aux(Ag, N, List)[scheme(Sch)]
24.  <- .print(Ag, " refused to commit to ", Sch);
25.     !assign(N, List)[scheme(Sch)].
```

Figura 3.8: piani del coordinatore per completare il goal organizzativo assignment

Come si può vedere in Figura 3.8, il coordinatore, per completare il goal `assignment`, inizializza la variabile `List` con l'elenco degli attuali operatori (ottenuti grazie alle proprietà osservabile `play(Agent, Role, Group)`) [riga 3] e crea un sotto-obiettivo `assign`, dove ricorsivamente estrae un operatore dalla lista e gli invia una richiesta di partecipare allo schema `Sch`, che identifica un progetto [riga 6-12]. Atteso un certo tempo, controlla se l'agente contattato ha scelto di perseguire la missione `awareness` (è diventato operatore) nello schema `Sch`: se ha accettato, decrementa il numero di agenti da contattare e richiama l'obiettivo `assign` [riga 21], alternativamente lascia il numero invariato [riga 25]. Da notare che il goal `assignment` [riga 2] è svolto atomicamente dal coordinatore; in questo modo il coordinatore svolge progressivamente tutte le istruzioni senza essere interrotto da altri eventi.

Nella seconda fase, in Figura 3.9, `awareness` (*awareness rising project*, opera di sensibilizzazione), i volontari assegnati a un progetto cercano di sensibilizzare la società, influenzando il comportamento dei cittadini e convincendoli a donare. In particolare, i volontari estraggono casualmente, tramite l'artefatto `YellowPages`, un nuovo agente da contattare e gli inviano un messaggio [riga 7], che modificherà la *Belief Base* dell'agente contattato (vedi Sezione 3.5). In questa fase, è probabile che un volontario, contatti un cittadino in difficoltà. In questo caso, il cittadino chiederà aiuto al volontario, il quale contatterà a sua volta il manager [riga 14-15] che creerà un nuovo progetto specificando il tipo di problema che affligge il cittadino. L'obiettivo `awareness` rimarrà attivo fintanto che un progetto non verrà concluso.

```
1.  +!awareness[scheme(PrSch)]
2.   : goalArgument(PrSch, fund_project, "project_art", Project)
3.     & status("ongoing", _, _)[artifact_name(_, Project)]
4.   <- waitTicks(2);      // artifact Clock's operation
5.      randomContact(Ag); // artifact YellowPages's operation
6.      ?goalArgument(PrSch, fund_project, "project_art", Project);
7.      .send(Ag, achieve, persuade(Project));
8.      !!awareness[scheme(PrSch)].
9.
10. +!awareness[scheme(Scheme)]
11.  <- goalAchieved(awareness)[artifact_name(Scheme)].
12.
13. /* belief from a citizen with trouble(T) */
14. +help(Trouble)[source(Ag)]
15. <- .send(manager, tell, candidate(Ag, Trouble)).
16.
17. /* belief from artifact project */
18. +problem("assistance")[artifact_name(_, Pr)]
19.  <- ?goalArgument(Sch,fund_project,"project_art",Pr)[workspace(_, WS,_)];
20.     commitMission(mTask)[wsp(WS), artifact_name(Sch)].
```

Figura 3.9: piani di un operatore

Se i volontari sono riusciti a raccogliere abbastanza denaro prima della scadenza del progetto (caso di successo), il progetto entra nell'ultima fase (`execute_task`). Questa fase è caratterizzata da una scelta (*choice*), in cui solo uno tra i goal `transfer_fund` e `assistance` deve essere completato per terminare il progetto; i goal verranno generati e assegnati ai ruoli manager e *volunteer*. In particolare questa scelta viene dettata dalla difficoltà in cui si trova il beneficiario di un progetto: se un progetto ha la proprietà osservabile `trouble` inizializzata su *money*, il progetto deve essere completato dal manager [Fig. 3.11, riga 35-43], il quale trasferisce i soldi raccolti al cittadino in difficoltà. Invece se `trouble` è stata inizializzata su *assistance*, saranno i volontari stessi a dedicarsi all'assistenza diretta del cittadino [Fig. 3.9, riga 18-20]. In entrambi i casi, una volta completato l'obiettivo, il cittadino non sarà più in difficoltà e potrà lavorare ed eventualmente entrare nell'ente benefico. Nel caso in cui i fondi raccolti siano superiori ai fondi necessari per completare il progetto, l'esubero sarà trattenuto dal manager e aggiunto ai fondi dell'ente [Fig. 3.11, riga 14], che serviranno per salvare un eventuale progetto fallito. Come ultima azione, il manager distrugge lo schema del progetto [Fig. 3.11, riga 23]. In Moise, l'eliminazione di una entità (ruolo, gruppo, schema, missione, obiettivo) deve seguire il seguente schema:

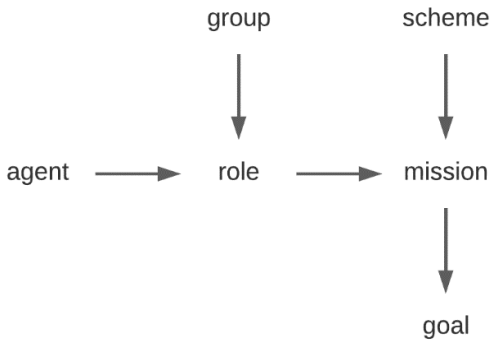


Figura 3.10: dipendenze per l'eliminazione [14]

Pertanto quando il manager deve distruggere uno schema dovrà attendere che i goal transitino in uno stato finale (in JaCaMo, l'unico stato disponibile è a*chieved*) e che le missioni vengano abbandonate. Se invece bisogna rimuovere un gruppo, è necessario che i goal siano completati, che le missioni e i ruoli siano abbandonati [14].

```
+!execute_task[scheme(Scheme)]
: status("failed", FundRaised, MissingFund)[artifact_name(_, Project)]
 <- .print("the project ", Project, " is failed");
    ?money(M); //organisation's fund
    -+money(M + FundRaised);
    getBeneficiary(B)[artifact_name(Project)];
    .send(B, tell, project_failed);
    goalAchieved(execute_task)[artifact_name(Scheme)];
    !destroy_sch[scheme(Scheme)].

+!execute_task[scheme(Scheme)]
: status("funded", FundTarget, Exceeding)[artifact_name(_, Project)]
 <- ?money(M); //organisation's fund
    -+money(M + Exceeding);
    goalAchieved(execute_task)[artifact_name(Scheme)];
    !destroy_sch[scheme(Scheme)].

+!execute_task[scheme(Scheme)]
 <- .print("mFund - Error: project is ongoing").

+!destroy_sch[scheme(Scheme)]
 <- leaveMission(mFund)[artifact_name(Scheme)];
    destroy[artifact_name(Scheme)].

// failed project management: the manager tries to save the projects
+!transfer_fund[scheme(PrSch)]
 : status("failed",FundRaised,MissingFund)[artifact_name(_,Project)]
   & money(M) & MissingFund < M
 <- !transfer_money(MissingFund, Project);
    .print("O.N.P. saves the project: ", Project)
    !transfer_fund[scheme(PrSch)].

// funded project management: the manager transfer the fund to
   the beneficiary (choice)
+!transfer_fund[scheme(Sc)]
 : goalArgument(Sc,fund_project,"project_art",Pr)[workspace(_,_, Wid)]
   & status("funded", FundTarget, Exceeding)[artifact_name(_, Pr)]
   & problem("money")[artifact_name(_, Pr)]
 <- !transfer_aux(Pr);
    getBeneficiary(B)[artifact_name(Pr)];
    .send(B, tell, bank_transfer(FundTarget))
    .print("Transferred ", FundTarget, "$ to ", B).
    goalAchieved(transfer_fund)[artifact_name(Sc), wid(Wid)].

+!transfer_fund <- true.
```

Figura 3.11: piani (parziali) per gestire il caso di fallimento o di successo di un progetto (e del suo schema)

3.2.4 Fallimento di un progetto

Qualora i volontari invece non riescano a raccogliere abbastanza donazioni, il progetto fallisce. Allo stato attuale dello sviluppo di JaCaMo, non è però possibile programmare il fallimento di uno schema o di un goal, sebbene sia puramente teorizzato nelle specifiche di Moise. Pertanto per simulare questo fallimento, si è scelto di procedere nel seguente modo: una volta che il coordinatore e gli operatori rilevano il cambiamento dello stato da "*on going*" a "*failed*" (proprietà osservabile dell'artefatto `Project`), dichiarano raggiunti tutti le missioni a cui si sono dedicati.

Quando un progetto fallisce, gli operatori, se presenti, stanno perseguendo l'obiettivo `awareness`: un operatore, prima di contattare un cittadino, può rilevare il fallimento (o il completamento) di un piano esaminando il contesto, ossia controllando se la proprietà osservabile *status* (generata dal corrispondente artefatto `Project`) sia diversa dallo stato "*on going*". Viceversa il coordinatore può perseguire uno qualsiasi degli obiettivi a lui affidati: per questo motivo, il coordinatore dovrà abbandonare a priori tutti gli obiettivi (`drop_intention(goal)`) e solo in seguito potrà lasciare la missione.

Completati gli obiettivi, vengono generati i goal presenti nella *choice* `execute_task`. Una volta fatto ciò, il manager gestirà il fallimento: tramite il piano `transfer_fund`, riportato in Figura 3.11 [riga 26-31], il manager, dopo che il progetto ha cambiato stato, verificherà se sarà possibile finanziare il progetto con i fondi dell'ente benefico [righe 28-29]. In caso di successo, il progetto potrà transitare dallo stato "*failed*" allo stato "*funded*", che rappresenta la fase finale in cui un progetto potrà venire completato normalmente (dal manager tramite il goal `transfer_fund` [riga 35-43] o dagli operatori tramite il goal `assistance`). In caso contrario, il progetto fallirà: nel piano `execute_task` [riga 1-9], i fondi raccolti verranno aggiunti a quelli dell'ente e il beneficiario sarà avvisato del fallimento del progetto [riga 7] e potrà nuovamente richiedere aiuto a un volontario che lo contatterà.

3.3 Specifiche normative

ROLE	DEONTIC RELATION	MISSION
manager	obligation	management
manager	obligation	mFund
coordinator	permission	mCoord
operator	obligation	mAware
operator	obligation	mTask

Figura 3.12: specifiche normative

Le relazioni deontiche mettono in relazioni le missioni di uno schema con i ruoli che le devono portare a termine. Le norme si distinguono principalmente in *obligation* o *permission*. Se un agente, o meglio un ruolo, è "obbligato" a raggiungere un obiettivo, esse deve effettuare il *commit* alla missione; viceversa se un agente ha il permesso di raggiungere un obiettivo, esso potrà scegliere o meno se effettuare il *commit*. Le norme, insieme alla cardinalità per i ruoli, determinano se uno schema è ben formato, e di conseguenza se può essere avviato.

3.4 Trattamento delle eccezioni

Al momento in JaCaMo, i goal e gli schemi non possono fallire. Grazie al mio correlatore, il Dottor Stefano Tedeschi, è stato possibile approfondire e studiare un progetto in sviluppo presso il Dipartimento di Informatica di Torino. Questo lavoro è stato sviluppato dalla Prof.ssa Cristina Baroglio, dal Prof. Matteo Baldoni, dal Dott. Roberto Micalizio e dal Dott. Stefano Tedeschi.

Il progetto ha come scopo quello di sopperire a questa mancanza ampliando le funzionalità di Moise: in particolare esso permette di creare e gestire all'interno dell'organizzazione il lancio di eccezioni. Siccome il progetto è ancora in fase di sviluppo, attualmente non ci sono ancora pubblicazioni in merito e di conseguenza ciò che verrà descritto nelle prossime righe potrà subire modifiche.

Le specifiche di un'organizzazione Moise vengono estese per permettere la cattura e la gestione di eccezioni e vengono ampliate le operazioni disponibili a un agente.

Nelle specifiche funzionali di un'organizzazione la definizione dello schema e delle missioni rimane pressoché invariata, ma a questi viene aggiunto un nuovo *tag* con la seguente forma:

```
 1. <recovery-strategy id="rec">
 2.     <exception id="nan">
 3.         <properties>
 4.             <property id="type" value id="goal-failure"/>
 5.             <property id="target" value id="parseAmount"/>
 6.         </properties>
 7.         <goal id="throwNan" />
 8.     </exception>
 9.     <handler id="handler_nan">
10.        <goal id="recoverFromNan" />
11.     </handler>
12. </recovery-strategy>
```

Figura 3.13: meccanismo per gestire il fallimento di un ipotetico goal `parseAmount`

Il codice sopra riportato permette di creare un nuovo obiettivo, qualora l'organizzazione rilevi il fallimento di uno specifico goal (un agente può dichiarare un goal fallito tramite la nuova operazione `goalFailed(goal)`). Questo nuovo obiettivo (in Fig. 3.13, `throwNan`) verrà assegnato e completato dal rispettivo ruolo, che a sua volta lancerà un'eccezione, tramite l'operazione `throwException(exception_type, [argument_list])`. *L'handler* della `recovery-strategy` gestirà quindi l'eccezione appena lanciata, generando un nuovo obiettivo (in Fig 3.13, `recoverFromNan`). L'operazione `throwException(exception_type, [argument_list])` genera una *belief* `exceptionThrown(Scheme, Exception, Agent)` per ogni eccezione lanciata e una *belief* `exceptionArgument(Scheme, Exception, Argument)` per ogni elemento della lista `argument_list`; queste *belief* sono proprietà osservabili dell'organizzazione (possono essere percepite da ogni membro) e sono utilizzate per differenziare il comportamento dell'*handler* e/o trasferire informazioni tra agenti diversi. Infine, tra le nuove facoltà di un agente c'è l'operazione `goalReleased(goal)` che permette di "rilasciare" un goal, in modo che possa venire completato eventualmente da un altro agente.

Questo progetto oltre a gestire il fallimento di un goal, può gestire nello stesso modo la scadenza di una *deadline* (di un goal) ed eventualmente questo meccanismo potrà essere esteso e trattare altre casistiche.

3.5 Ciclo di vita di un cittadino

I cittadini (`citizen.asl`) sono gli unici agenti esterni all'organizzazione. Hanno un ciclo di vita semplice. Tutti i cittadini si registrano inizialmente nell'artefatto `YellowPages`, che permette loro di essere contattati dai volontari dell'organizzazione. I cittadini comuni (che non possiedono la *belief* `trouble`) passano il loro tempo a lavorare: dopo una certa quantità di tempo, l'artefatto `Clock` genera un segnale che viene percepito da tutti gli agenti che osservano l'artefatto; i cittadini, percepito questo segnale, ottengono uno stipendio (casuale) che si va ad aggiungere al denaro già in loro possesso.

I cittadini possono essere inoltre contattati dal coordinatore dell'ente che gli propone di entrare nell'organizzazione e di sensibilizzare gli altri cittadini. Inoltre possono essere contattati anche dai volontari dell'organizzazione. Ogni volta che un volontario contatta un cittadino, modifica la *belief* `altruism` del cittadino di un valore fisso [Fig. 3.14, riga 11]: il cittadino ha una probabilità dell'*altruism*% di donare parte del suo capitale al progetto del volontario. Dopo aver donato, la *belief* `altruism` viene azzerata.

Un cittadino in difficoltà (*belief* `trouble`) invece non può lavorare, non può guadagnare e non può diventare un operatore. Rimane esclusivamente in attesa di essere contattato da un volontario: quando succede, il cittadino invierà una richiesta di aiuto al volontario specificando il tipo di difficoltà in cui si trova, e attenderà la fine del progetto, che verrà istanziato dal manager. Se il progetto avrà successo, il cittadino non si troverà più in difficoltà e potrà iniziare a lavorare. In caso contrario, il cittadino verrà avvisato dal manager del fallimento del progetto e potrà richiedere nuovamente a un volontario di essere aiutato.

```
@ptrouble[atomic]
+!persuade(Project)[source(Ag)]
   : trouble(X) & not contacted
  <- .send(Ag, tell, help(X));
     +contacted.

@pnormal[atomic]
+!persuade(Project)[source(Ag)]
   : not trouble(_)
  <- ?altruism(X);
     -+altruism(X+10);
     !new_donation((math.random * 100) + 1, Project).

+!persuade(_) <- true.

+!new_donation(Success, Project)
   : money(M) & M >= 1000
  <- !transfer_money(M * 0.5, Project);
    -+altruism(0).

+!new_donation(Success, Project)
   : altruism(X) & money(M) & M > 0 & Success < X
  <- !transfer_money(M * 0.2, Project);
     -+altruism(0).

+!new_donation(_, _) <- true.

/* belief from the manager for citizen in trouble */
+assistance[source(manager)]
   <- .print("I don't have any trouble. I can work!!!");
      -trouble(assistance);
      -assistance[source(manager)];
      !work.

+project_failed[source(manager)]
  <- -project_failed[source(manager)];
     -contacted.
```

Figura 3.14: estratto del codice di citizen.asl

3.6 Gli artefatti

Nella modellazione del caso di studio è stato necessario usare tre diversi artefatti (non conteggiando gli artefatti organizzativi) funzionali all'organizzazione:

- L'artefatto `YellowPages.java` permette all'ente benefico di mantenere un elenco di tutti i cittadini e di tutti i volontari. In particolare, siccome non è possibile per un agente contattarne un altro senza conoscerne il nome, è fondamentale che tutti gli agenti esterni all'organizzazioni si registrino (tramite l'operazione `register()`), in modo che sia possibile per un volontario promuovere un progetto. L'artefatto viene utilizzato anche dal coordinatore per estrarre casualmente un nuovo cittadino da reclutare (operazione `randomCitizen(OpFeedbackParam<Term> res)`).
  Lo stato interno è determinato da due liste in cui in una sono elencati tutti i contatti dell'organizzazione (cittadini e volontari) e nell'altra solamente i cittadini comuni. In questo semplice modo è possibile estrarre casualmente un contatto per il volontario o un cittadino per il coordinatore. Un cittadino, una volta diventato volontario, dovrà di conseguenza aggiornare le liste (tramite l'operazione `startVolunteering()`).
  In Fig. 3.15, è riportato il codice dell'operazione `randomContact`, ove è possibile notare che per restituire un valore all'agente che ha invocato l'azione (un agente esegue questa operazione eseguendo, nel suo codice, l'azione `randomContact(Agent)`), deve essere specificato un parametro `OpFeedbackParam<T>`, che verrà inizializzato con il valore della computazione [Fig. 3.15, riga 11].

```
 1. @OPERATION
 2. void randomContact(OpFeedbackParam<Term> res) {
 3.     if(citizens.size() == 0)
 4.         failed("YP-Error","no registered agents");
 5.     else {
 6.         Random random = new Random();
 7.         String ag = contacts.get(random.nextInt(contacts.size()
 8.         while(ag.equals(getCurrentOpAgentId().getAgentName()))
 9.             ag = contacts.get(random.nextInt(contacts.size()));
10.         Term t = new Atom(ag);
11.         res.set(t);
12.     }
13. }
```

Figura 3.15, codice dell'operazione per estrarre un contatto (cittadino o volontario) casuale nella classe YellowPages.java

- L'artefatto `Clock` permette di definire delle tempistiche comuni all'interno del MAS [25]. Attraverso l'operazione `startClock()`, il manager avvia l'esecuzione asincrona di un'operazione interna (annotata con `@INTERNAL_OPERATION`), di un conteggio a lungo termine: ogni `TICK_TIME` viene incrementato il numero di *tick* trascorsi. Ogni cinque *tick*, viene generato un segnale, catturato dagli agenti che osservano l'artefatto. Questo segnale serve principalmente per incrementare i fondi disponibili a un cittadino comune. Altre operazioni permettono di fermare il conteggio (`stopClock()`) o di resettarlo (`resetClock()`) o di attendere un certo numero di tick (`waitTicks(n)`). La primitiva `await_time` permette di sospendere l'esecuzione dell'operazione finché il tempo specificato non è trascorso. Sospendendo l'operazione, l'artefatto diventa accessibile agli agenti per altre operazioni e i cambiamenti alle proprietà osservabili vengono effettuati. Il codice dell'operazione `startClock` è molto simile a quello dell'operazione `startTimer()` riportato in Figura 3.16 [riga 15-19].
- L'artefatto progetto (classe `Project.java`) permette di modellare precisamente un progetto come entità passiva. Lo stato interno è determinato da proprietà come la somma di denaro da raccogliere (`fundTarget`), la somma raccolta (`fundRaised`), l'elenco dei donatori (`supporters`), il beneficiario (`beneficiary`), il tipo di problema (`problem`) e il tempo rimanente prima del suo fallimento (espresso in funzione del numero di *tick* mancanti). La maggior parte di queste proprietà sono statiche e sono inizializzate dal manager, quando il progetto viene creato. L'artefatto fornisce agli agenti tre proprietà osservabili: i) `sch`, indica il nome dello schema che modella l'evoluzione del progetto; ii) `problem`, indica il tipo di problema che affligge il beneficiario: può essere di tipo economico (*money*) o di tipo assistenziale (*assistance*); iii) `status`, caratterizza lo stato corrente del progetto: inizialmente settato a "in corso" (*on going*), può transitare nello stato di completamento (*funded*), qualora si raggiunga il *fundTarget*, oppure nello stato di fallimento (*failed*) in caso in cui il numero di *tick* diventi uguale a zero. Tra le operazioni principali fornite agli agenti ci sono: i)`donate`, azione che permette a un agente di donare e può settare lo stato del progetto in funded, e ii)`startTimer`, che avvia una operazione interna che decrementa ogni `TICK_TIME` il numero di *tick* del progetto. Le operazioni sono riportate in Fig. 3.16.

```
1.  @OPERATION
2.  void donate(double dollars) {
3.      String ag = getCurrentOpAgentId().getAgentName();
4.      supporters.add(ag);
5.      this.fundRaised += Math.round(dollars);
6.      display.updateFund(this.fundRaised);
7.      if(fundRaised >= fundTarget) {
8.          this.counting = false;
9.          ObsProperty p = getObsProperty("status")
10.         p.updateValues("funded", fundTarget, fundRaised - fundTarget);
11.     }
12. }
13.
14. @OPERATION
15. void startTimer(){
16.     if(!this.counting)
17.         execInternalOp("count");
18.     else
19.         failed("already counting");
20. }
21.
22. @INTERNAL_OPERATION
23. void count() {
24.     this.counting = true;
25.     while(this.counting) {
26.         await_time(Clock.TICK_TIME);
27.         ticks--;
28.         if(ticks <= 0) {
29.             System.out.println("Project Failed");
30.             ObsProperty p = getObsProperty("status").
31.             p.updateValues("failed",fundRaised,fundTarget-fundRaised);
32.             this.counting = false;
33.         }
34. }
```

Figura 3.16: codice dell'operazioni principali della classe Project.java

Capitolo 4
Conclusioni

4.1 Il lavoro

Lo sviluppo di sistemi complessi basati sugli agenti richiede strumenti di progettazione e programmazione che forniscano delle astrazioni (di prima classe) per ciascuna dimensione di un sistema multi-agente. A tal fine, in questa relazione si è descritto un approccio globale per la programmazione orientata a implementare un tale sistema. Il lavoro svolto mi ha permesso di comprendere il potenziale della tecnologia JaCaMo, che integra sinergicamente agenti, ambiente e organizzazione; nella prima parte dell'elaborato, attraverso l'analisi di ciascuna dimensione e delle relative tecnologie è stato possibile apprezzare le peculiarità, le caratteristiche chiave e lo scopo di ciascun'astrazione e tramite l'integrazione delle varie dimensioni comprendere infine il framework stesso; il caso di studio, esposto nella seconda parte, invece è un semplice *proof of concepts*, ma grazie ad esso è stato possibile studiare i limiti e i vantaggi offerti del framework e apprezzarne la vasta estensibilità.

4.2 Vantaggi e limiti di JaCaMo

Come si evince dalla relazione, la forza del framework JaCaMo risiede nell'integrazione delle diverse dimensioni e in quest'ottica ciascuna dimensione è potenzialmente integrabile o sostituibile da una qualsiasi nuova tecnologia.

I vantaggi e i limiti di JaCaMo pertanto derivano dalle tre attuali tecnologie che lo compongono e in particolare dalla programmazione orientata agli agenti [27]: il livello di astrazione *agent-oriented* aiuta infatti a ridurre la complessità nello sviluppo di sistemi concorrenti e distribuiti; l'architettura BDI permette di integrare, seppur in modo basico, il comportamento reattivo e pro-attivo; la possibilità di eseguire diverse azioni (*Plan*) a seconda delle informazioni in possesso a un agente, permette di programmare facilmente un comportamento basato sul contesto e di introdurre una forma semplice di polimorfismo; grazie alle differenti astrazioni, è possibile modellare concettualmente entità diverse e, basandosi su un repertorio di azioni e percezioni fornite agli agenti, è possibile definire le interazioni tra le diverse dimensioni concettuali del programma.

Quando però ci si allontana dal campo dell'intelligenza artificiale distribuita e si passa a un contesto più generale tuttavia, emergono debolezze e limiti: *l'agent oriented programming* soffre della mancanza di una serie di caratteristiche fondamentali rispetto ai principali linguaggi di programmazione. Manca ad esempio il concetto di tipo [27]: ciò porta all'assenza della rilevazione (statica) di errori, e porta di conseguenza a

necessarie e lunghe sessioni di debugging per rilevare gli errori a tempo di esecuzione; non è possibile rilevare staticamente neanche gli errori relativi all'integrazione delle diverse dimensione: ad esempio, un agente può adottare un ruolo sconosciuto in una organizzazione o può adottare un piano per completare un goal organizzativo non affidato al suo ruolo. Inoltre senza il concetto di tipo, non è possibile generalizzare o specializzare (ad esempio tramite la definizione di sotto-tipo) le relazioni tra concetti e entità diverse ed esplicitare il principio di sostenibilità per supportare l'estensione e il riuso del codice. Nella letteratura sugli agenti è riconosciuta l'assenza del supporto alla modularità: Jason ad esempio manca di alcuni costrutti per modularizzare e strutturare l'insieme dei piani che definiscono il comportamento dell'agente [27]. Infine, sebbene Moise semplifichi la coordinazione degli agenti, il numero di percezioni create e gestite dagli artefatti organizzativi cresce esponenzialmente all'aumentare del numero di gruppi, schemi, missioni, obiettivi, permessi o obblighi definiti nelle specifiche dell'organizzazione. Essendo proprietà associate ad artefatti, esse sono note a ciascun membro dell'organizzazione: ciò, oltre a rendere difficoltoso il debugging di un agente (tramite *Mind Inspector*), costituisce un *overhead* per gli agenti stessi.

6.3 Future estensioni

JaCaMo è una tecnologia ancora acerba, ma dalle grosse potenzialità. In particolare la sua forza risiede nella sua vasta estensibilità: è facile integrare nuove tecnologie nello strumento o sostituire quelle esistenti. JaCaMo stesso in realtà è una estensione del progetto JaCa, che combina la tecnologia per agenti Jason e quella per ambienti CArtAgO.

Essendo un progetto open-source è stato possibile assistere nel corso degli anni alla nascita di molte versioni: JaCa-Android e JaCa-Web forniscono un livello di astrazione orientato agli agenti per progettare e programmare rispettivamente applicazioni mobile sulla piattaforma Android e applicazioni web client-side; JaCA-DDM è un progetto per il data mining; JaCaMo-Sim [24], presentato durante l'E4MAS 2020 ha come scopo quello di realizzare un simulatore, il cui modello è basato sul DES (*Discrete Event Simulation*); ultimo esempio è rappresentato dal progetto in sviluppo presso il Dipartimento di Informatica di Torino per la gestione delle eccezioni.

Bibliografia

[1] Baldoni, M., Baroglio, C., & Micalizio, R. (2020). Fragility and Robustness in Multiagent Systems.

[2] Boissier O., Multi-Agent Programming Course, "https://www.emse.fr/~boissier/enseignement/maop20-winter/index.html", 2020

[3] Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Sci. Comput. Program. 78(6), 747–761 (2013)

[4] Boissier, O., Hübner, J.F., Kitio, R., Ricci, A., "Instrumenting multi-agent organisations with organisational artifacts and agents,"Autonomous agents and multi-agent systems, vol. 20, no. 3, pp. 369–400,2010.

[5] Bordini R.H., Hübner J.F. (2006), BDI Agent Programming in AgentSpeak Using Jason. In: Toni F., Torroni P. (eds) Computational Logic in Multi-Agent Systems. CLIMA 2005. Lecture Notes in Computer Science, vol 3900. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11750734_9

[6] Bordini, R., Hübner, J., Wooldridge, M. (2007). Programming Multi-Agent Systems in AgentSpeak Using Jason. 10.1002/9780470061848

[7] Bordini, R., Hübner, J., (2007), A Java-based interpreter for an extended version of AgentSpeak, "http://jason.sourceforge.net/Jason.pdf", 2020

[8] Bordini, R., Hübner, J., Tralamazza, D., (2007). Using Jason to Implement a Team of Gold Miners. 4371. 304-313. 10.1007/978-3-540-69619-3_18.

[9] CArtAgO, sito ufficiale, "http://cartago.sourceforge.net/", 2020

[10] Cossentino, M., Lopes, S., Nuzzo, A., Renda, G., Sabatucci, L.: A comparison of the basic principles and behavioural aspects of Akka, JaCaMo and Jade development frameworks. In: 19th Workshop From Objects to Agents (WOA 2018), Palermo, 28–29 June 2018

[11] Dorri, A., Salil, Jurdak, Raja. (2018). Multi-Agent Systems: A survey. IEEE Access. 1-1. 10.1109/ACCESS.2018.2831228.

[12] Hübner, J. F., Sichman, J. (2002), A model for the structural, functional, and deontic specification of organizations in multiagent systems. 118-128.

[13] Hübner, J. F., Sichman, J. (2007). Developing organised multi-agent systems using the MOISE+ model: Programming issues at the system and agent levels.. Int. J. Accounting, Auditing and Performance Evaluation. 1-1. 10.1504/IJAOSE.2007.016266.

[14] Hübner, J. F., Sichman, J., Boissier, O., Moise tutorial (2010), "http://moise.sourceforge.net/doc/tutorial.pdf"

[15] Hübner, J., Kitio, Rosine, Ricci, A. (2010). Instrumenting multi-agent organisations with organisational artifacts and agents: "Giving the organisational power back to the agents". Autonomous Agents and Multi-Agent Systems. 20. 369-400. 10.1007/s10458-009-9084-y.

[16] Hübner, J.F., JaCaMo (2016), GitHub repository, "https://github.com/jacamo-lang/jacamo", 2020

[17] Hübner, J.F., Jason (2016), GitHub repository, "https://github.com/jason-lang/jason", 2020

[18] Hübner, J.F., Ricci A., CArtAgO (2016), GitHub repository, "https://github.com/CArtAgO-lang/cartago", 2020

[19] Hübner, J.F., Ricci A., Moise (2016), GitHub repository, https://github.com/moise-lang/moise, 2020

[20] JaCaMo, sito ufficiale, "http://jacamo.sourceforge.net/", 2020

[21] Jason, sito ufficiale, "http://jason.sourceforge.net/", 2020

[22] Moise, sito ufficiale, "http://moise.sourceforge.net/"

[23] Ricci A., Viroli M., Omicini A. (2008) The A&A Programming Model and Technology for Developing Agent Environments in MAS. In: Dastani M., El Fallah Seghrouchni A., Ricci A., Winikoff M. (eds) Programming Multi-Agent Systems. ProMAS 2007. Lecture Notes in Computer Science, vol 4908. Springer, Berlin, Heidelberg. https://doi.org/10.1007

[24] Ricci, A., Exploiting simulation for MAS programming and engineering – the JaCaMo/SIM platform, "https://screencast-o-matic.com/watch/cYfv2MB3tP", EMAS@AAMAS 2020

[25] Ricci, A., Santi, A., CArtAgO and JaCa By Example (2010), "http://jacamo.sourceforge.net/cartago/doc/cartago_by_examples/cartago_by_examples.pdf"

[26] Ricci, Alessandro & Piunti, Michele & Viroli, Mirko. (2011). Environment programming in multi-agent systems: An artifact-based perspective. Autonomous Agents and Multi-Agent Systems. 23. 158-192. 10.1007/s10458-010-9140-7.

[27] Santi, A. (2013), Engineering agent-oriented technologies and programming languages for computer programming and software development, supervisor Ricci A., PhD thesis for Course in Electronics, Computer Science and Telecommunications, ALMA MATER STUDIORUM — Università di Bologna

[28] Tedeschi S. (2021) Engineering Multiagent Organizations Through Accountability. In: Novais P., Vercelli G., Larriba-Pey J.L., Herrera F., Chamoso P. (eds) Ambient Intelligence – Software and Applications. ISAmI 2020. Advances in Intelligent Systems and Computing, vol 1239. Springer, Cham. https://doi.org/10.1007/978-3-030-58356-9_36

[29] Weyns, D., Omicini, A. & Odell, J. Environment as a first class abstraction in multiagent systems. Auton Agent Multi-Agent Syst 14, 5–30 (2007). https://doi.org/10.1007/s10458-006-0012-0

[30] Wooldridge M., 2009, An Introduction to MultiAgent Systems (2nd. ed.), Wiley Publishing.

[31] Hübner, J.F., "support mantenance goals (MG)", (2016), moise issue #4, https://github.com/moise-lang/moise/issues/4

[32] Alessandro Ricci, Andrea Santi, and Michele Piunti. Action and perception in agent programming languages: from exogenous to endogenous environments. In *Programming Multi-Agent Systems*, ProMAS'10, pages 119–138, Berlin, Heidelberg, 2012. Springer-Verlag.