

MetaJade: a DSL for Jade

Riccardo Battistini

riccardo.battistini2@studio.unibo.it

Marzo 2023

Il progetto ha lo scopo di realizzare il design e l'implementazione di un DSL per la programmazione di agenti, denominato MetaJade. Si tratta un linguaggio per la programmazione di agenti basato sulla piattaforma Jade, funzionale, staticamente tipizzato e con una sintassi ispirata allo pseudocodice. Catturando le principali astrazioni emerse dall'uso delle piattaforme ad agenti ed equipaggiandole con un linguaggio dedicato, si propone l'obiettivo di ridurre la complessità inerente alla programmazione di sistemi multi-agente. Come modello di riferimento per il suo sviluppo è stato impiegato Jadescript, un DSL realizzato con Xtext avente obiettivi simili. Lo strumento impiegato per la modellazione e l'implementazione del linguaggio è il *language workbench* MPS. Il suo utilizzo ha permesso di realizzare un linguaggio facilmente estendibile e integrabile con MPS stesso, senza la necessità di definire una grammatica formale per il parsing. Come base per l'implementazione è stato impiegato il linguaggio funzionale KernelF, ideato per fornire un insieme di primitive comuni.

1. Obiettivi e risultati

L'obiettivo del progetto è realizzare un linguaggio per la programmazione di agenti che consenta di sviluppare sistemi multi-agente basati sulla piattaforma Jade. Uno dei risultati sono degli esempi, in modo da confrontare le applicazioni in Java e in MetaJade. Gli esempi scelti sono il Ping Pong e uno scenario semplificato di gestione di un account bancario.

Inoltre si crea un plugin MPS per la distribuzione del linguaggio che impacchetti il DSL. Il plugin favorisce l'integrazione futura con altri linguaggi e la realizzazione di applicazioni più complesse, in un'ottica di programmazione orientata ai linguaggi.

2. Analisi dei requisiti

In questa sezione si discutono i requisiti funzionali e non funzionali del progetto, i paradigmi e le tecnologie utilizzati per l'implementazione e i requisiti impliciti per il suo svolgimento.

2.1. Requisiti funzionali

Di seguito si descrivono nel dettaglio i requisiti funzionali di MetaJade. I requisiti sono organizzati in base alle principali astrazioni supportate dal linguaggio e sono illustrati nelle tabelle 1, 2, 3, 4 e 5. Complessivamente rappresentano il sottoinsieme delle funzionalità offerte da Jade che sono supportate in MetaJade.

Id	Descrizione
P1	Avviare programmaticamente una piattaforma Jade.
P2	Configurare i parametri di avvio di una piattaforma Jade.

Tabella 1: Tabella dei requisiti funzionali relativi alla piattaforma.

Id	Descrizione
A1	Inizializzare un agente.
A2	Accedere all'AID di un agente, in particolare a proprietà come il nome completo dell'agente o il suo nome locale.
A3	Definire proprietà che costituiscono lo stato interno dell'agente, in modo che siano inaccessibili da altri agenti.
A4	Registrare le ontologie utilizzate da un agente per comunicare.
A5	Registrare il linguaggio di default SL.
A6	Pubblicare o ritirare i servizi offerti dall'agente sulle pagine gialle.
A7	Reperire servizi offerti da altri agenti sulle pagine gialle.
A8	Avviare programmaticamente un agente.
A9	Inizializzare un agente con parametri forniti da linea di comando.

Tabella 2: Tabella dei requisiti funzionali relativi agli agenti.

Id	Descrizione
B1	Inizializzare un comportamento.
B2	Attivare o disattivare un comportamento.
B3	Definire parametri di avvio di un comportamento.
B4	Definire comportamenti che eseguono una tantum o ciclicamente.
B5	Eseguire più comportamenti in serie.

Tabella 3: Tabella dei requisiti funzionali relativi ai comportamenti.

Id	Descrizione
O1	Definire un'ontologia.
O2	Definire concetti, azioni e predicati di un'ontologia.
O3	Definire concetti, azioni e predicati di un'ontologia composti da tipi di dato primitivi, ovvero booleani, stringhe, interi e decimali.
O4	Definire concetti, azioni e predicati di un'ontologia composti da altri concetti, azioni e predicati.
O5	Definire concetti, azioni e predicati di un'ontologia composti da liste o insiemi di tipi di dato primitivi o di altri concetti, azioni o predicati.

Tabella 4: Tabella dei requisiti funzionali relativi alle ontologie.

Id	Descrizione
M1	Inviare un messaggio o la risposta a un messaggio.
M2	Definire mittente, destinatari, performative, contenuto e ontologia di un messaggio.
M3	Accedere a mittente, destinatari, performative e contenuto di un messaggio.

M4	Consentire la ricezione bloccante di messaggi in fase d’inizializzazione di un agente.
M5	Definire handler di messaggi per consentire la ricezione non bloccante di messaggi in un comportamento.
M6	Innescare gli handler solo con messaggi aventi un determinato contenuto, performative, mittente o destinatario.

Tabella 5: Tabella dei requisiti funzionali relativi ai messaggi ACL.

Nella tabella 6 si descrivono ulteriori requisiti relativi alla struttura del linguaggio. Si tratta di requisiti impliciti che si affiancano ai requisiti del progetto definiti in precedenza.

Id	Descrizione
L1	Definire dei moduli, ovvero delle raccolte di dichiarazioni di funzioni, strutture dati e costanti riutilizzabili in più comportamenti e agenti.
L2	Definire un sistema d’importazione per rendere visibili gli elementi di un chunk in altri chunk. Un chunk è definito come un modulo, un’ontologia, un agente o un comportamento.
L3	Definire le azioni compiute in un chunk come una serie di espressioni con effetti collaterali in un linguaggio funzionale.
L4	Definire dei tipi per rappresentare entità specifiche al dominio del progetto, quali AID e Performative .

Tabella 6: Tabella dei requisiti funzionali relativi al linguaggio.

Nota

Rispetto alla definizione iniziale dei requisiti di base sono stati tralasciati gli hook **onStart**, **onEnd** e **done** nei comportamenti e **tearDown** negli agenti. Sono stati utilizzati solo i metodi **setup** e **action** di agenti e comportamenti.

Nota

Rispetto alla definizione iniziale dei requisiti di base è stato tralasciato l'utilizzo dell'interface API di Jade a favore di un approccio basato sull'utilizzo di task Gradle. I task sono utilizzati per il lancio e la configurazione della piattaforma e degli agenti Jade in modo programmatico.

2.2. Requisiti non funzionali

Di seguito si elencano i requisiti non funzionali identificati per il progetto.

- MetaJade deve poter essere diviso in moduli. Ciascuno di essi deve essere il meno interdipendente dagli altri possibile. Con modulo si intende il significato assegnato al termine nell'appendice A.
- Ciascuno dei linguaggi che compone MetaJade deve poter essere esteso facilmente, implementando determinati nodi interfaccia o estendendo specifici nodi astratti.
- I programmi in MetaJade dovrebbero essere più leggibili, concisi ed espressivi degli equivalenti in Java.

2.3. Paradigmi e Tecnologie

2.3.1. Programmazione orientata agli agenti

La programmazione orientata agli agenti (AOP) è stata introdotta in [1] nel 1990, come specializzazione della programmazione ad oggetti.

L'AOP è un paradigma di programmazione che impiega il concetto di Agente, ovvero un'entità computazionale autonoma, come strumento per la risoluzione di problemi relativi ai sistemi distribuiti con controllo autonomo e decentralizzato.

A partire dall'autonomia di un agente, si distinguono numerose altre caratteristiche:

- gli agenti autonomi sono interattivi, sociali, proattivi e situati;
- possono avere obiettivi e compiti o essere reattivi, intelligenti e mobili;
- possono operare all'interno di un MAS e possono interagire con altri agenti tramite atti comunicativi o nell'ambiente tramite azioni.

L'AOP riguarda sia le astrazioni specifiche agli agenti che la sintassi e la semantica dei linguaggi di programmazione.

Negli ultimi decenni si è ritenuto che le astrazioni identificate tramite l'AOP potessero essere sfruttate efficacemente nell'ambito di linguaggi di programmazione *general-purpose*. Questa idea ha portato alla diffusione di piattaforme ad agenti come Jade, strettamente legate alle tecnologie emergenti del tempo, come Java. Per l'implementazione di sistemi multiagente si è utilizzata prevalentemente la programmazione orientata

agli oggetti e un middleware per la comunicazione e la gestione del ciclo di vita degli agenti.

Nel contesto del cambio di paradigma di cui Jade è un esempio, negli ultimi anni sono stati proposti dei linguaggi specifici al dominio della programmazione ad agenti. Il progetto svolto nell'ambito di questo corso può essere considerato come un esempio di linguaggi di questo tipo.

2.3.2. Programmazione orientata ai linguaggi

Il paradigma della programmazione orientata ai linguaggi è stato introdotto nel 1994 in [2]. Sviluppare programmi con questo paradigma permette di lavorare in termini di concetti e nozioni che caratterizzano il problema che si vuole risolvere, senza che sia necessario tradurre idee in nozioni che possono essere comprese da un GPL, come classi o metodi.

La programmazione orientata ai linguaggi spinge gli sviluppatori a creare un proprio DSL o a estendere linguaggi esistenti con concetti specifici a un dominio, come parte del loro approccio alla risoluzione di problemi.

2.3.3. Jade

Il *Java Agent DEvelopment Framework* (Jade) [3] è un framework software, scritto in Java, utilizzato per implementare sistemi multiagente che rispettano la specifica della *Foundation for Intelligent Physical Agents* (FIPA) [4]. Fornisce un middleware che include:

- una piattaforma per agenti distribuita e interoperabile;
- servizi di pagine gialle e bianche per il reperimento degli agenti;
- servizi di scambio di messaggi e naming distribuiti;
- un'API per sviluppare agenti;
- una raccolta di strumenti grafici per l'amministrazione e il monitoraggio degli agenti.

Per maggiori informazioni si rimanda alla sezione 3.1, alla documentazione disponibile nel sito ufficiale <https://Jade.tilab.com/> e a [5].

2.3.4. MPS

Il *Meta Programming System* (MPS) è stato introdotto nel 2004 da Sergey Dmitriev allo scopo di fornire un IDE per lo sviluppo di programmi basati sul paradigma della programmazione orientata ai linguaggi. In MPS è possibile utilizzare diversi linguaggi, ciascuno per una parte specifica di un programma, in modo che lavorino coerentemente insieme. Il language workbench permette di utilizzare l'autocompletamento, la navigazione del codice, il refactoring e fornisce ulteriori strumenti di produttività, offrendo

un'esperienza simile a quella di programmare nei linguaggi general-purpose maggiormente diffusi. Per maggiori informazioni si rimanda alla documentazione ufficiale [6]. Nell'appendice A sono definiti concetti specifici al dominio di MPS utilizzati nel corso di questa relazione.

2.3.5. KernelF

KernelF è una raccolta di linguaggi che funge da punto di partenza per lo sviluppo di DSL. In particolare KernelF definisce tipi di dato primitivi, operatori di confronto aritmetico e logico, strutture dati generiche come liste e dizionari a partire dai quali creare funzionalità più avanzate e specifiche a un dominio, indipendentemente dal linguaggio in cui il DSL genera. Per maggiori informazioni su KernelF si rimanda al repository [iets3.opensource](https://github.com/iets3/opensource) e a [7].

2.4. Processo di design e implementazione adottato

Di seguito si descrive brevemente il processo di sviluppo che si è scelto di adottare per il design e l'implementazione del DSL. Il processo a cui si fa riferimento deriva dai suggerimenti forniti negli articoli di Markus Voelter [8] [9], esperto di sviluppo di DSL in MPS e ideatore di KernelF, e dal processo di sviluppo adottato dagli ideatori di Jadescript, così come descritto nella serie di articoli da loro pubblicati [10], [11], [12], [13], [14], [15].

Inizialmente si crea un *glossario* dei termini appartenenti al dominio in analisi. In seguito si costruisce un *modello del dominio*, ovvero si rappresentano le relazioni tra i termini del glossario in una notazione informale simile a UML. In particolare si indicano relazioni di tipo *is-a*. In altri termini si ridefinisce il glossario in modo che possa essere elaborato da una macchina. Nel contesto della costruzione di un DSL, questo modello del dominio funge da *metamodello*. In MPS il metamodello è definito tramite l'aspetto struttura di un linguaggio.

Per completare il DSL si definiscono un *sistema dei tipi* e una *sintassi specifica al metamodello*. In MPS si impiegano l'aspetto dell'editor e l'aspetto del sistema dei tipi.

Perché il DSL possa essere effettivamente eseguito è necessario stabilire una *semantica*, ovvero una definizione formale di come il codice scritto nel DSL sia generato in codice in uno specifico linguaggio. Dato che si è scelto d'impiegare le API di Jade in Java, sono stati implementati dei generatori per generare codice Java.

3. Design

Di seguito si descrivono i diversi aspetti del design di MetaJade in accordo con il processo di sviluppo indicato in sezione 2.4.

3.1. Glossario

Di seguito si descrivono le principali astrazioni legate al dominio degli agenti Jade.

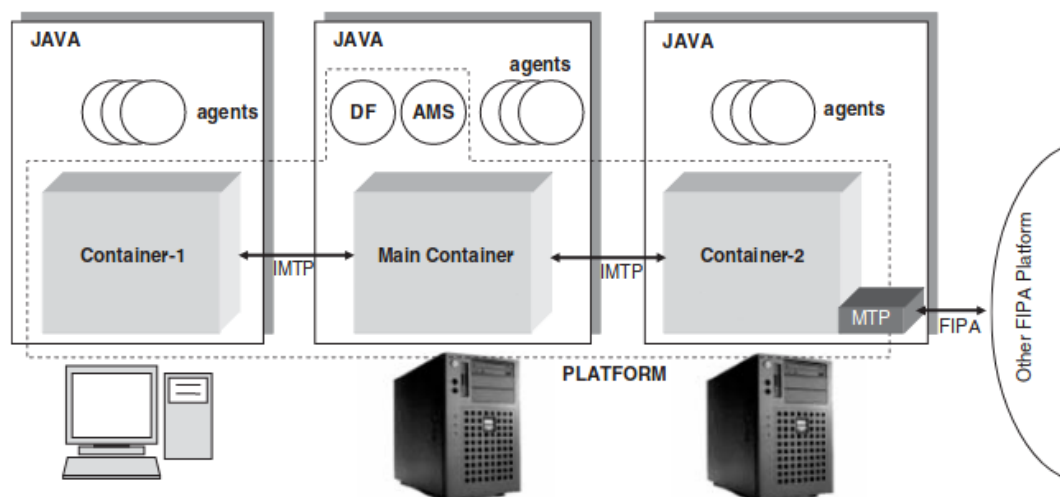


Figura 1: Rappresentazione dei principali elementi architetturali di Jade.

Modifica della figura 3.1 tratta da [5]

3.1.1. Piattaforma e Container

In figura 1 si osservano i principali elementi architetturali di Jade. Si distinguono:

- Piattaforma;
- Main Container e container periferici;
- Servizio di pagine gialle e bianche, forniti dagli agenti DF e AMS.

Una piattaforma può essere distribuita su più host. Ciascun container è eseguito da un host sulla rete. Ciascuna piattaforma ha un main container e altri container periferici che si registrano ad essa per creare la piattaforma.

I container ospitano uno o più agenti e consentono l'esecuzione e l'interazione tra agenti. Gli agenti possono scambiarsi messaggi usando i loro AID, indipendentemente dalla piattaforma e dai container che ospitano mittenti e destinatari. La piattaforma Jade si occupa di instradare in modo trasparente ed automatico i messaggi, astruendo dai canali di comunicazione impiegati e da dettagli come indirizzi, host e connessioni.

Il servizio di pagine gialle è fornito dall'agente DF ed è mostrato in figura 2. Il DF tiene traccia di tutti i servizi pubblicati dagli agenti appartenenti alla stessa piattaforma. Gli agenti interagiscono con il servizio tramite il paradigma publish/subscribe.

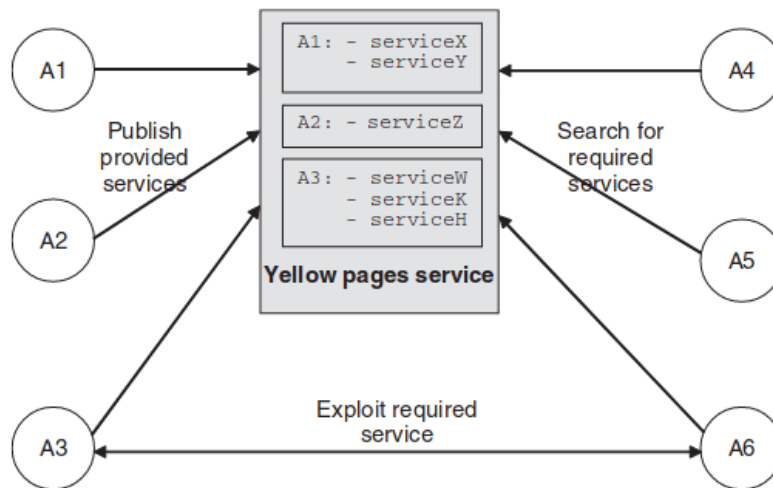


Figura 2: Rappresentazione del servizio di pagine gialle di Jade.

Figura 4.5 tratta da [5]

3.1.2. Agente

Ad ogni agente è permanentemente associato un *Agent Identifier* (AID), che individua univocamente l'agente. Un AID è composto da due parti, il nome dell'agente, utilizzato per identificarlo univocamente nella piattaforma, e un *Home Agent Platform*, che identifica univocamente la piattaforma in cui esegue l'agente.

Ciascun agente esegue in un thread ed attraversa un ciclo di vita composto da diversi stati. Ciascuna transizione può essere associata ad una procedura da eseguire prima che la transizione avvenga. In figura 3 è mostrato il ciclo di vita di un agente Jade, in accordo con la specifica FIPA.

- Quando un agente è avviato entra nello stato inizializzato, durante il quale non è ancora registrato all'AMS. Mentre è in questo stato l'agente non ha ancora un AID, perciò non può comunicare con altri agenti.
- Nello stato attivo un agente può eseguire comportamenti ed è registrato nell'AMS.
- Nello stato di attesa un agente è bloccato, in genere è in attesa di ricevere un messaggio ACL.
- Nello stato sospeso un agente non esegue alcun comportamento ed è fermo.
- Nello stato di transito un agente sta eseguendo una migrazione.
- Nello stato sconosciuto l'agente è stato terminato e rimosso dall'AMS.

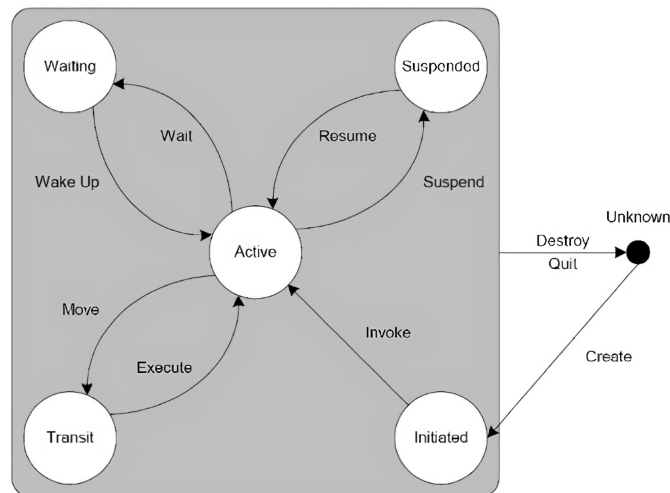


Figura 3: Rappresentazione del ciclo di vita di un agente FIPA.

Figura tratta dalle slide del corso

3.1.3. Comportamento

Un comportamento è una procedura che un agente esegue per portare a termine un compito. Sono il mezzo con cui gli agenti agiscono nell'ambiente in cui si trovano nel corso del loro ciclo di vita. Consentono ad un agente di operare in un unico thread e di svolgere comunque compiti complessi, che richiedono di eseguire task concorrentemente.

I comportamenti possono essere:

- una tantum
- ciclici
- periodici

I comportamenti una tantum sono per compiti da eseguire un'unica volta. Un comportamento ciclico è eseguito ripetutamente fino a quando non è disattivato esplicitamente. Un comportamento periodico è eseguito ad ogni tick di un timer avente un determinato periodo fino a quando non è disattivato esplicitamente.

Ciascun behaviour è definito da un'azione e da uno o più handler di eventi. L'azione determina la procedura eseguita dal comportamento quando è schedato. L'handler è una procedura parametrizzata eseguita all'occorrenza di un evento.

Un agente può attivare più comportamenti e ogni agente gestisce una coda privata di comportamenti, che tiene traccia dei comportamenti attivi e li ordina in una coda con priorità per effettuare lo scheduling.

Lo scheduling dei comportamenti è cooperativo piuttosto che preemptive. Perciò l'esecuzione di un behaviour non è mai interrotta per eseguire un altro comportamento. Quando l'esecuzione di un'azione o di un handler è avviata, prosegue fino alla terminazione della relativa procedura.

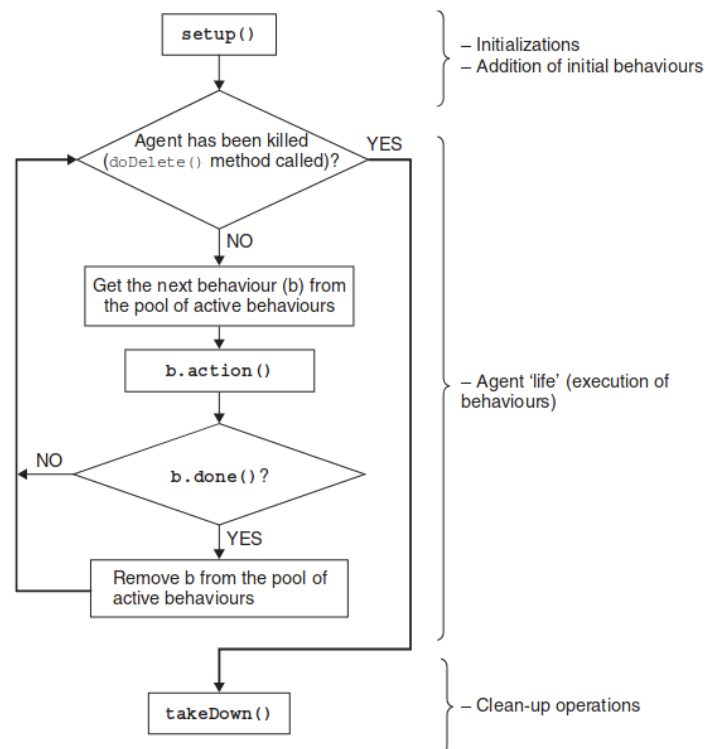


Figura 4: Diagramma di flusso del thread di un agente.

Figura 4.3 tratta da [5]

In figura 4 è riassunta la politica di scheduling cooperativo adottata in Jade.

3.1.4. Messaggio ACL

I messaggi che gli agenti si scambiano nella piattaforma rispettano le specifiche IEEE FIPA ACL. I messaggi ACL possono essere visti come strutture dati che contengono l'informazione che il mittente del messaggio vuole condividere con i destinatari. Alcuni dei dati inviati sono:

- AID del mittente del messaggio;
- AID dei destinatari del messaggio;
- Performative, ovvero l'atto comunicativo associato all'invio del messaggio;
- Contenuto del messaggio, espresso in un linguaggio per la rappresentazione della conoscenza adeguato;
- L'ontologia utilizzata per descrivere gli elementi del contenuto del messaggio specifici all'applicazione.

Lo scambio di messaggi avviene in modo asincrono, così come mostrato in figura 5. Ciascun agente ha una coda dei messaggi, che funge da *mailbox*, dove sono recapitati i messaggi inviati dagli altri agenti. Quando si aggiunge un nuovo elemento alla coda si notifica l'agente.

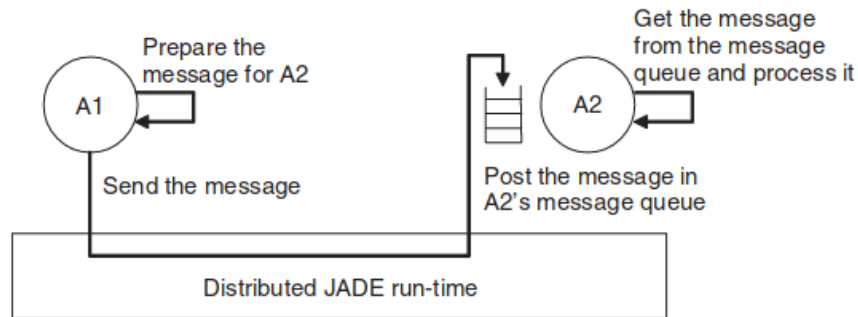


Figura 5: Rappresentazione del paradigma di scambio di messaggi asincrono utilizzato in Jade.

Figura 4.4 tratta da [5]

Il linguaggio dei contenuti adottato per la comunicazione e che stabilisce struttura e semantica dei messaggi è descritto nella FIPA SL Content Language Specification [16].

3.2. Metamodello

MetaJade è un linguaggio modulare. È costituito da diversi linguaggi MPS che raggruppano ciascuna funzionalità relative ad una specifica astrazione del dominio. Ci sono cinque moduli:

- **aclmessage**, definisce come inviare, ricevere, rispondere a messaggi e come creare dei template per la loro ricezione;
- **agent**, definisce come implementare agenti e fornire argomenti in input;
- **behaviour**, definisce come implementare comportamenti, le diverse tipologie e le istruzioni per gestirli;
- **core**, contiene i nodi astratti che gli altri moduli estendono e il sistema per gestire l'import di chunk in altri chunk;
- **ontology**, definisce come implementare ontologie e le sue componenti, azioni, predicati e concetti.

In figura 6 sono mostrati i punti di estensione di KernelF che sono utilizzati per realizzare i punti di estensione di MetaJade nel modulo **core**. Di seguito si descrivono, per ciascuno dei nodi astratti di KernelF indicati, come è stato sviluppato il corrispondente metamodello in MetaJade.

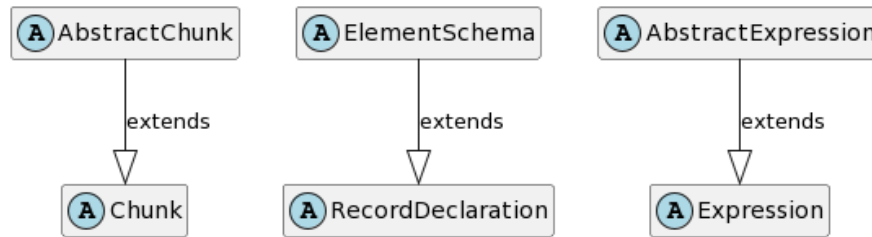


Figura 6: Rappresentazione dei punti di estensione di KernelF utilizzati in MetaJade.

- **Chunk** rappresenta un modulo che può essere importato in altri moduli e consente di definire gli elementi che possono essere visibili in altri moduli in cui è importato. I chunk sono anche i nodi radice del linguaggio. **AbstractChunk** è esteso dai nodi struttura che rappresentano i chunk di MetaJade, così come mostrato in figura 7.
- **RecordDeclaration** è utilizzato per definire i possibili elementi di un'ontologia, in modo che siano definiti come i record di KernelF. I record rappresentano la base per l'implementazione delle strutture dati di MetaJade. Un record è definito da un nome e da uno o più membri, ciascuno avente un proprio nome e un tipo. In figura 8 sono mostrati i tipi di record che possono essere dichiarati nelle ontologie.

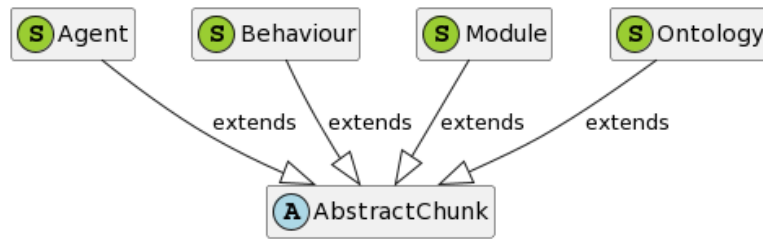


Figura 7: Rappresentazione dei chunk del sistema di moduli utilizzato in MetaJade.

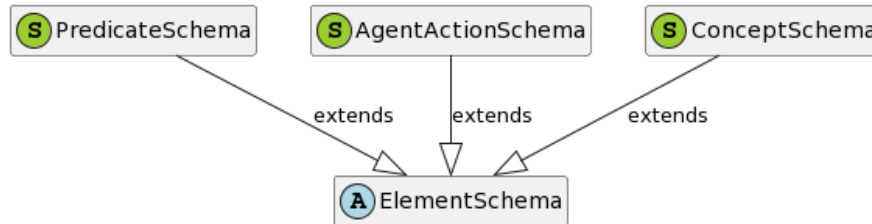


Figura 8: Rappresentazione del metamodello relativo elementi che possono comporre un'ontologia.

- **Expression** è utilizzato per definire nuove espressioni del linguaggio. Le espressioni relative al dominio di Jade utilizzano i punti di estensione mostrati in figura 9.

Ciascun modulo estende **AbstractExpression** con un proprio nodo astratto specifico:

- In **behaviour** si utilizza **AbstractBehaviourExpression**, esteso come mostrato in figura 10. Si utilizza il nodo intermedio aggiuntivo **AbstractBehaviourUnaryExpression** in modo da distinguere tra istruzioni che richiedono il riferimento ad uno o più comportamenti come argomento; In
- **agent** si utilizza **AbstractDFExpression**, esteso come mostrato in figura 11; In
- **aclmessage** si utilizza **AbstractMessageExpression**, esteso come mostrato in figura 12.

Strutturare il linguaggio in questo modo permette di estenderlo facilmente. Ad esempio si possono inserire nuovi chunk e nuove espressioni a partire dai nodi astratti indicati. Inoltre si riduce il riutilizzo di codice dato che si raggruppano gli elementi comuni a tutti i chunk, a tutte le istruzioni del linguaggio e quelli relativi ai singoli moduli.

L'implementazione del sistema dei tipi avviene a partire dai nodi struttura mostrati in figura 13. Di seguito si indicano le ragioni per cui si è scelto di inserire ciascuno dei tipi indicati nel linguaggio.

- Il tipo **performative** garantisce che nelle istruzioni per l'invio di messaggi ci sia un riferimento ad uno dei possibili performative.

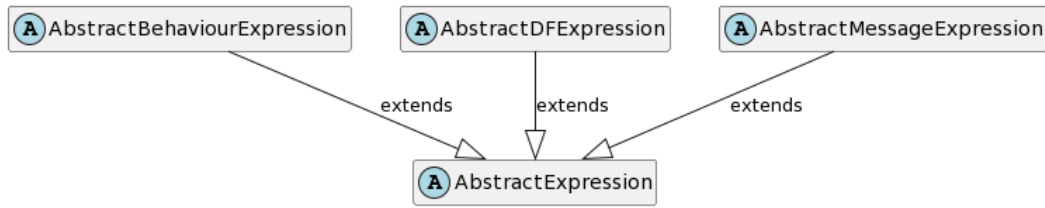


Figura 9: Rappresentazione dei punti di estensione per l'implementazione di espressioni specifiche al dominio di MetaJade.

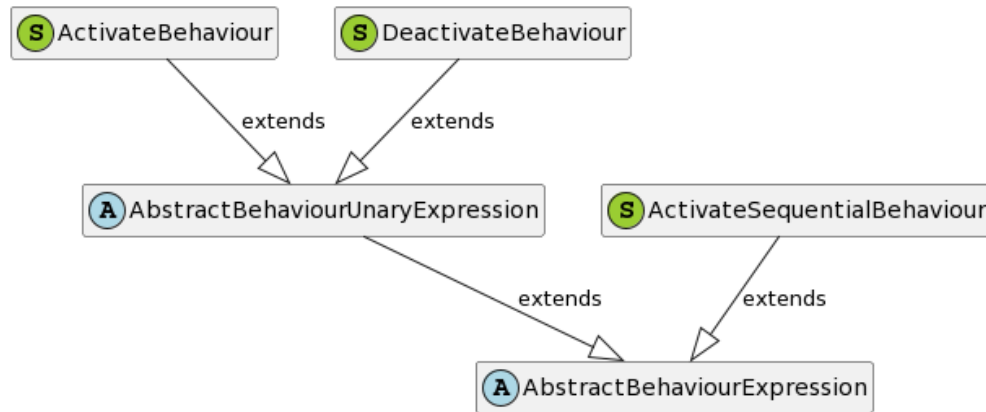


Figura 10: Rappresentazione del metamodello relativo alle espressioni per la gestione dei comportamenti.

- Il tipo **AID** garantisce che nelle istruzioni per l'invio di messaggi ci sia un riferimento ad un **AID**.
- Il tipo **Unit** rappresenta un valore di ritorno significativo, utilizzato per le espressioni con effetti collaterali introdotte da MetaJade.
- **BehaviourType** è utilizzato per assicurarsi che le istruzioni per l'attivazione o disattivazione di comportamenti contengano effettivamente i riferimenti a comportamenti.

In figura 14 sono mostrati i due tipi di comportamenti che possono essere creati. Si nota che **BehaviourKind** non estende **Type**. Questo perché non si utilizzano esplicitamente i tipi per rappresentare se il comportamento esegue una volta sola o ciclicamente. Analogamente non si utilizzano esplicitamente dei tipi per rappresentare l'appartenenza di un record ad un'ontologia. In questo caso si è ritenuto sufficiente imporre il vincolo strutturale per cui gli elementi di un'ontologia possono essere creati solo al suo interno.

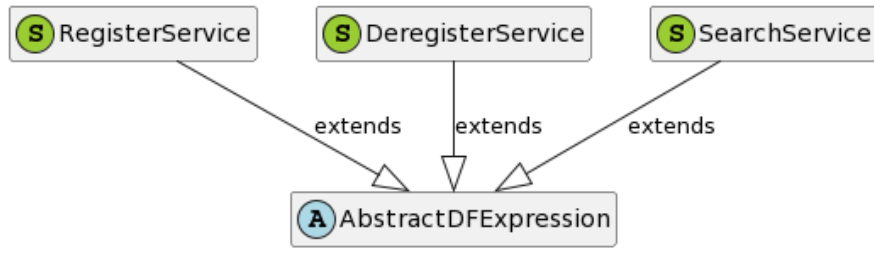


Figura 11: Rappresentazione del metamodello relativo alle espressioni per la gestione dei servizi di pagine gialle della piattaforma Jade.

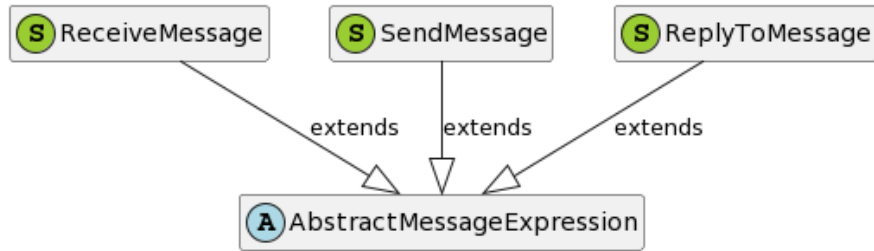


Figura 12: Rappresentazione del metamodello relativo alle espressioni per la gestione dei messaggi.

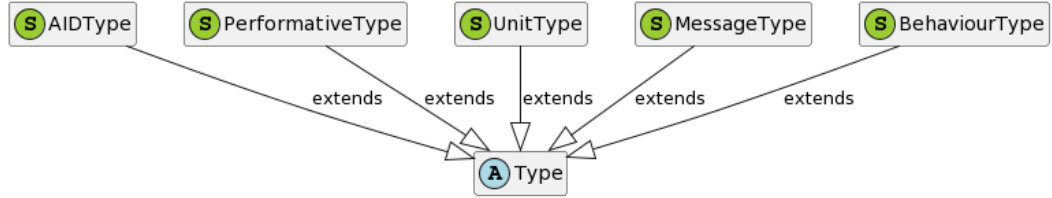


Figura 13: Rappresentazione del metamodello relativo ai tipi introdotti da MetaJade.

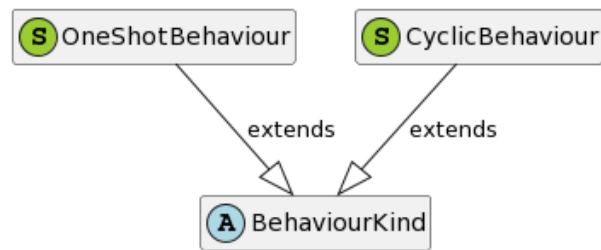


Figura 14: Rappresentazione del metamodello relativo ai possibili tipi di comportamento.

3.3. Sistema dei tipi

In questa sezione si discute informalmente il sistema dei tipi di MetaJade. Ad eccezione dei nodi introdotti specificamente per il dominio degli agenti Jade, gli altri tipi sono forniti da KernelF. I tipi forniti da KernelF non sono discussi nel dettaglio, per maggiori approfondimenti si rimanda all'appendice A in [7].

3.3.1. Tipi di base

I tipi di base rappresentano i tipi fondamentali e più semplici e non sono parametrici.

Il linguaggio supporta l'utilizzo di tipi numerici, booleani e stringhe. Un booleano può assumere i valori `true` e `false`.

I tipi numerici si dividono in `number`, `int` e `real`. Il tipo `number` indica un numero che ha un intervallo di valori ammissibili e una precisione. Si può specificare il tipo di un numero nei seguenti modi:

```
// tipo intero, intervallo illimitato
number => number [-inf|inf]{0}

// intero positivo
number [0|inf] => number [0|inf]{0}

// intero, intervallo specificato
number [10|20] => number [10|20]{0}

// numero reale con due cifre decimali, intervallo illimitato
number {2} => number [-inf|inf]{2}

// precisione data dai decimali nell'intervallo specificato
number [3.3|4.5] => number [3.3|4.5]{1}
```

Come tipi di base specifici al dominio degli agenti Jade sono stati introdotti `performative`, `AID` e `behaviour`.

3.3.2. Tipi delle collezioni

Esistono tre tipi di collezioni: `list`, `set` e `map`. Sono tutte sottotipo di `collection`, sono parametrizzate con il tipo di elementi che le compongono e sono covarianti e immutabili.

3.3.3. Tipi parametrici

In MetaJade si utilizzano i tipi parametrici `box`, `opt` e `message`. `box` permette di creare variabili e collezioni mutabili. `opt` permette di definire opzionali e `message` rappresenta i messaggi ACL e il loro contenuto.

3.3.4. Inferenza dei tipi

KernelF supporta l'inferenza dei tipi sia per i tipi primitivi che i tipi delle collezioni e i tipi introdotti da MetaJade. Come parte del processo di inferenza KernelF esegue dei calcoli aritmetici per verificare che i valori siano effettivamente negli intervalli indicati tramite il tipo `number`.

Dall'inferenza sono esclusi gli argomenti e i membri di record, così come il tipo di ritorno delle funzioni ricorsive. In entrambi i casi è perciò necessario indicare esplicitamente i tipi.

3.4. Sintassi

MetaJade estende KernelF. Per questa ragione molti elementi del linguaggio hanno una sintassi molto simile, se non identica, a quella degli elementi di KernelF. Per una panoramica dei costrutti forniti da KernelF, si rimanda all'appendice A in [7].

Di seguito si riportano i principali costrutti introdotti da MetaJade e un sottoinsieme dei costrutti di KernelF effettivamente utilizzati durante lo sviluppo del linguaggio.

La sintassi concreta di MetaJade e le modifiche alla sintassi concreta di KernelF sono definite nell'aspetto degli editor dei linguaggi del progetto.

Nota

Alcuni esempi di sintassi utilizzano placeholder, denotati da parentesi angolate. I possibili valori dei placeholder sono: `name`, `chunk`, `type`, `expression`, `top-level-expression`, `condition`, `performative`, `kind` e `matcher`. `name` indica una stringa, `condition` un'espressione booleana e `kind` un nodo che estende `BehaviourKind`. Gli altri placeholder corrispondono ad omonime entità in MetaJade. Si osserva che l'utilizzo dei tre punti di sospensione indica cardinalità multipla dei placeholder.

3.4.1. Sintassi nodi radice

La struttura di un nodo radice di MetaJade, ovvero di un chunk, è formata da alcune componenti fisse. Tutti i chunk, ad eccezione delle ontologie, hanno una sezione in cui si dichiarano i chunk importati. Agenti, comportamenti e moduli hanno una sezione per inserire espressioni di primo livello, come dichiarazioni di funzioni, record o enumerazioni. Le ontologie hanno una sezione in cui inserire come espressioni di primo livello dichiarazioni di azioni, predicati o concetti. Di seguito si mostra la sintassi di ciascuno dei tipi di chunk.

```
agent <name>

import <chunk>...

input <name> of type <type> with default value <expression>
...
```

```
action as <expression>

<top-level-expression>
...
```

```
<kind> behaviour <name>

import <chunk>...

input <name> of type <type>
    ...

setup as <expression>

<top-level-expression>
...
```

```
module <name>

import <chunk>...

<top-level-expression>
...
```

```
ontology <name>

<top-level-expression>
...
```

3.4.2. Sintassi delle espressioni di primo livello

La dichiarazione di una funzione:

```
fun <name>(<name>: <type>,...): <type> as <expression>
```

La dichiarazione di una costante:

```
val <name> of type <type> as <expression>
```

La dichiarazione di un'enumerazione:

```
// Non qualificata
enum <name> {
    <expression>
    ...
}

// Qualificata
enum <name><<type>> {
```

```
<name> -> <expression>
...
}
```

La dichiarazione di un record:

```
record <name> {
  <name>: <type>
  ...
}
```

Le dichiarazioni di un elemento di un'ontologia:

```
action <name> {
  <name>: <type>
  ...
}

concept <name> {
  <name>: <type>
  ...
}

predicate <name> {
  <name>: <type>
  ...
}
```

3.4.3. Sintassi delle espressioni KernelF

Operatori aritmetici, si applicano ai tipi numerici:

```
1 + 2
5 * 5
2 - 4
10 / 2
22 mod 3
```

Operatori logici, richiedono un'espressione che sia valutata in una condizione booleana:

```
// and
true && false
// or
true || false
// not
!true
```

Valori locali:

```
val <name> of type <type> as <expression>
```

Invocazione a funzione:

```
fun isEven(num: Int) {  
    num mod 2 == 0  
}  
isEven(2)
```

Funzioni di ordine superiore e lambda:

```
fun checkParity(num: Int, strategy: (Int => Boolean)) {  
    criteria.exec(num)  
}  
  
checkParity(2, {n: Int => {  
    if n mod 2 == 0 then true else false  
}})
```

Costruzione di una stringa e concatenazione con operatore infisso +:

```
"this is " + 1 + " dummy string"
```

Costruzione di un record:

```
record pingMsg {  
    msg: String  
}  
  
#pingMsg{"ping"}
```

Dichiarazione di un'istruzione if-else:

```
if <condition> then <expression> else <expression>
```

Dichiarazione di un'istruzione per gestire più alternative:

```
alt | <condition> => <expression> |  
    | ... |  
    | otherwise => <expression> |
```

Valori opzionali, dichiarati con il tipo `opt`. Possono essere estratti esplicitamente tramite l'operazione su collezioni `unpackOptions` o l'espressione `isSome` in un'istruzione if-else:

```
val dummyOpt: opt<Int> = 2  
  
if isSome(dummyOpt) as dummy then dummy else 0
```

Collezioni immutabili:

```
// Dichiarazione di collezioni vuote  
val dummyList as list<Int>()  
val dummySet as set<String>()  
val dummyMap as map<String, Int>()
```

```
// Dichiarazione con valori iniziali
val dummyList of type list of int as list(0,1,2)
val dummySet of type set of string as set("a","b","c")
val dummyMap of type map of string, int as map("a"->1,"b"->2)
```

Si osserva che quando si creano collezioni vuote è necessario indicare esplicitamente il tipo nel letterale, dato che non ci sono espressioni a partire da cui effettuare l'inferenza.

Variabili e collezioni mutabili:

```
val dummyVariable of type box<int> as box(0)

val dummyList of type box<list of int> as box(list<int>())
val dummySet of type box<set of string> as box(set<string>())
val dummyMap of type box<map of string, int> as box(map<string, int>())
```

Per una panoramica delle operazioni sulle collezioni si rimanda al menù di autocompletamento disponibile nell'editor in MPS.

3.4.4. Sintassi delle espressioni MetaJade

Di seguito si illustrano le istruzioni e le espressioni specifiche al dominio del progetto.

Stampa di valori su standard output:

```
log "dummyInt has value: " + 2
```

Gestione dei comportamenti:

```
activate <expression>

deactivate <expression>

stop

activate in sequence {
    <expression>
    ...
}
```

Gestione dei servizi:

```
register <expression>

deregister <expression>

val result of type opt<AID> = search <expression>
```

Si osserva che **search** è un'espressione che restituisce un opzionale contenente l'AID del primo agente che fornisce il servizio cercato.

Dato un AID sono disponibili due operazioni, per reperire il nome locale dell'agente e il nome completo, come mostrato di seguito:

```

fun dummy(aid: AID) {
    val agentLocalName: string = aid.localname
    val agentName: string = aid.name
}

```

Invio e ricezione di messaggi:

```

send message<<type>> {
    performative: <performative>
    receiver: <expression>
    other receivers: <expression>
    content: <expression>
}

reply with message<<type>> to <name> {
    performative: <performative>
    content: <expression>
}

on receive message<<type>> as <name>
    where <matcher> {
        <expression>
    }
}

```

Sono disponibili delle operazioni sui messaggi per estrarre il contenuto, il mittente e i destinatari. Di seguito si mostra un loro esempio di utilizzo:

```

on receive message<<type>> as msg {
    val content: <type> = msg.content
    val sender: AID = msg.sender
    val receivers: list<AID> = msg.receivers
}

```

3.4.5. Tracciamento degli effetti

KernelF è un linguaggio funzionale in cui le espressioni non hanno effetti collaterali. Perché possa far parte di linguaggi che hanno espressioni con effetti collaterali fornisce un meccanismo di tracciamento degli effetti. MetaJade estende KernelF con espressioni che hanno effetti collaterali. Una funzione ha un effetto quando include espressioni che usano effetti collaterali. Quando si richiede d'indicare che una funzione ha un effetto, si utilizza uno dei tre indicatori a disposizione, /R, /M o /RM, a seconda che l'effetto che si voglia tracciare sia rispettivamente in lettura, in scrittura o in lettura e scrittura.

Di seguito si riporta un esempio della sintassi per tracciare gli effetti in una funzione:

```

fun sendProblem/M(msg: message, error: int) {
    reply with message to msg {
        performative: Inform
        content: error
    }
}

```

```
}  
}
```

L'indicatore per il tracciamento degli effetti si può applicare anche al tipo funzione, come nel seguente esempio:

```
fun setResult/M(id: int, f: (int =>/M unit) {  
    f.exec(id)  
}  
  
setResult(2, |id: int => log "send id" + id |*)
```

4. Note implementative

Di seguito si riportano alcune scelte non banali fatte durante l'implementazione di MetaJade.

- Si è modificato significativamente il processo di generazione in Java delle funzioni definite in KernelF. Questo perché le funzioni erano originariamente generate con la keyword `static`. Ciò non permetteva d'invocare le API di Jade. Per questa ragione è stato creato uno script di preprocessing per modificare il processo di generazione. Lo script permette di sostituire i nodi relativi alle funzioni in KernelF con riferimenti a una versione personalizzata del nodo che rappresenta una funzione che genera in Java senza la keyword `static`. A questo scopo sono state utilizzate nel generatore delle etichette, in modo che fosse possibile effettuare l'invocazione di funzioni all'interno di una classe mantenendo il riferimento alla corrispondente istanza della classe in cui la funzione è dichiarata. Ciò ha richiesto di distinguere tra i casi in cui la funzione è dichiarata esternamente alla classe in cui è invocata o meno.
- È stato implementato un wrapper delle API di Jade per potersi interfacciare con KernelF. Si tratta di una serie di metodi che, restituendo il tipo `Unit` piuttosto che `Void`, permettono di generare in Java espressioni con effetti collaterali che si inseriscono nella catena d'invocazioni a funzioni di KernelF.
- La generazione d'istruzioni per gestire la ricezione di messaggi utilizza API di Jade diverse a seconda che l'invocazione dell'istruzione avvenga a partire da un agente o da un comportamento. Nel primo caso si utilizza il metodo `blockingReceive`, nel secondo la combinazione di `receive` e `block`. Nell'implementazione corrente se un handler per la ricezione è dichiarato in una funzione che si trova in un modulo ma è poi utilizzato tramite un agente, il processo di generazione nella fase di compilazione del codice generato fallisce. Ciò accade perché sia gli handler dichiarati in moduli che in comportamenti utilizzano `receive` e `block`.
- La generazione di una espressione per l'estrazione del contenuto di un messaggio varia a seconda che l'invocazione avvenga o meno all'interno di un handler per la

ricezione dei messaggi. Nel primo caso si utilizza il riferimento al contenuto, già estratto per determinare se attivare o meno l'handler in primo luogo. Nel secondo si effettua l'estrazione vera e propria.

- Si è scelto di codificare la comunicazione tra mittente e destinatario, utilizzando parte dello standard FIPA, tramite le classi **Action** e **Result**. Le azioni indicano richieste a un agente di effettuare un compito o di valutare un predicato. Il risultato incapsula le risposte, che contengono l'esito dello svolgimento di un compito o la valutazione di un predicato e il riferimento alla richiesta di svolgerlo. L'utilizzo di **Action** e **Result** per incapsulare richieste e risposte avviene in modo implicito in MetaJade. Nel codice Java generato avviene in base al tipo di elemento di un'ontologia utilizzato per inviare un messaggio o una risposta. In particolare è consentito l'invio di messaggi utilizzando l'espressione **send** solo se il contenuto è un'azione o il predicato di un'ontologia. Per l'espressione **reply** invece il contenuto del messaggio di risposta deve essere il concetto di un'ontologia. Questi controlli sono stati imposti sia lato MetaJade utilizzando delle checking rule che lato Java utilizzando delle condizioni per i generatori relativi alle espressioni **send** e **reply**.
- Al programmatore non è richiesto d'indicare esplicitamente le ontologie utilizzate da un messaggio o di registrare il loro utilizzo negli agenti. È sufficiente importare nel chunk l'ontologia relativa alla struttura dati a cui si vuole fare riferimento. A quel punto il corretto insieme di ontologie da registrare in un dato agente è ottenuto percorrendo ricorsivamente la catena aciclica di chunk importati alla ricerca di tutti i riferimenti a ontologie. Nell'implementazione corrente questo significa che se si importa un'ontologia e non la si utilizza l'agente che importa in modo diretto o indiretto il chunk in cui è presente il riferimento registrerà un'ontologia che non utilizzerà effettivamente per comunicare.
- Ciascun handler per la ricezione di messaggi effettua prima un'invocazione al metodo **receive** ed estrae il contenuto del messaggio per verificare se corrisponde a quello dichiarato dal programmatore. Se non c'è corrispondenza il messaggio è reinserito nella mailbox dell'agente utilizzando il metodo **putBack**. In questo modo è possibile definire più handler per gestire i diversi tipi di messaggi attesi.
- Per definire il tipo di messaggi ammissibile si è scelto di consentire l'utilizzo di un limitato insieme di espressioni booleane, dette "matcher" che permettono d'introdurre condizioni, in *and*, *or* o *not*, sul mittente, i destinatari e il performative del messaggio. Si tratta di un approccio che permette di evitare l'utilizzo di espressioni booleane arbitrarie per definire le condizioni di ricezione dei messaggi, simile a quello impiegato in Jadescript [15].
- Le ontologie generate in Java utilizzano la classe **CFReflectiveIntrospector**. L'utilizzo di questa classe consente di mappare **BaseOntology.SEQUENCE** a **java.util.List** e **BaseOntology.SET** a **java.util.Set**. Ciò è dovuto all'utilizzo da parte di KernelF della libreria **PCollections** per generare le collezioni in Java. Queste collezioni hanno un elevato grado di interoperabilità con le collezioni

standard di Java. Perciò `org.pcollection.PVector` e `org.pcollection.PSet` possono essere mappate a liste e insiemi in `java.util` senza incompatibilità. Si osserva che l'utilizzo di `CFReflectiveIntrospector` riduce l'interoperabilità degli agenti definiti in MetaJade rispetto ad altri agenti che possono essere definiti nell'ambito di Jade o più in generale della specifica FIPA.

- KernelF genera tipi di dato interi e decimali nel corrispettivo in Java di `BigInteger`. Jade riconosce solo i tipi `int`, `long` e `float`, `double`. Perciò sono stati implementati wrapper dei getter e setter nelle strutture dati delle ontologie in modo da garantire l'interoperabilità tra la piattaforma e il linguaggio.
- È stato limitato significativamente il numero di elementi di KernelF che è possibile utilizzare in MetaJade in modo da ridurre la complessità del linguaggio sia dal punto di vista dello sviluppo che dell'utilizzo. Ciò si è ottenuto sia evitando di importare i moduli dei linguaggi di KernelF non utilizzati che tramite il vincolo `can be ancestor` e gli stili di completamento.
- Ciascun tipo di chunk definisce quali altre tipologie di chunk può importare dichiarando dei vincoli in un apposito nodo che implementa l'interfaccia `IChunkDep`. Per poter definire questo genere di vincoli è stato modificato il sistema dei moduli offerto da `mbeddr`, estendendo il nodo `Chunk` e implementando l'interfaccia `IChunkDependency`. `mbeddr` è una dipendenza di KernelF.
- MPS non supporta del tutto la definizione di più generatori per uno stesso linguaggio. Per questa ragione si è scelto di utilizzare un `pattern` per la definizione di più generatori. In questo modo ciascun generatore è definito in maniera indipendente dagli altri ed è più facile estendere il linguaggio con nuovi generatori. Al momento è definito e utilizzato solo un generatore per Java.

5. Autovalutazione

L'autovalutazione consiste in:

- verifica dell'implementazione dei requisiti funzionali;
- descrizione dei test del linguaggio;
- descrizione del workflow adottato.

5.1. Verifica dell'implementazione dei requisiti funzionali

Rispetto ai requisiti funzionali indicati nel capitolo 2, si riportano di seguito le tabelle 7, 8, 9, 10 e 11 in cui sono forniti i nomi dei nodi o dei file che forniscono le funzionalità richieste da ciascun requisito.

Id	Descrizione	Implementazione
P1	Avviare programmaticamente una piattaforma Jade tramite azioni fornite dal workbench.	Tramite task definiti in <code>build.gradle.kts</code> .
P2	Configurare i parametri di avvio di una piattaforma Jade.	Tramite task definiti in <code>build.gradle.kts</code> .

Tabella 7: Implementazione dei requisiti funzionali relativi alle piattaforme.

Id	Descrizione	Implementazione
A1	Inizializzare un agente.	<code>Agent</code> <code>map_Agent</code>
A2	Accedere all'AID di un agente, in particolare a proprietà come il nome completo dell'agente o il suo nome locale.	<code>GetLocalName</code> <code>GetName</code> <code>switch_AIDOperation</code> <code>map_AID</code>
A3	Definire proprietà che costituiscono lo stato interno dell'agente, in modo che siano inaccessibili da altri agenti.	<code>Agent</code>
A4	Registrare le ontologie utilizzate da un agente per comunicare.	<code>map_Agent</code>
A5	Registrare il linguaggio di default SL.	<code>map_Agent</code>
A6	Pubblicare o ritirare i servizi offerti dall'agente sulle pagine gialle.	<code>DeregisterService</code> <code>RegisterService</code> <code>switch_ServiceExpression</code>
A7	Reperire servizi offerti da altri agenti sulle pagine gialle.	<code>SearchService</code> <code>switch_ServiceExpression</code>
A8	Avviare programmaticamente un agente tramite azioni fornite dal workbench.	Tramite task definiti in <code>build.gradle.kts</code> .

A9	Inizializzare un agente con parametri forniti da linea di comando.	AgentArgument AgentArgRef Agent
----	--	---------------------------------------

Tabella 8: Implementazione dei requisiti funzionali relativi agli agenti.

Id	Descrizione	Implementazione
B1	Avviare programmaticamente una piattaforma Jade tramite azioni fornite dal workbench.	Behaviour map_Behaviour
B2	Attivare o disattivare un comportamento.	BehaviourReference switch_BehaviourExpression ActivateBehaviour DeactivateBehaviour
B3	Definire parametri di avvio di un comportamento.	BehaviourArgument BehaviourArgRef Behaviour switch_BehaviourExpression
B4	Definire comportamenti che eseguono una tantum o ciclicamente.	OneShotBehaviour CyclicBehaviour BehaviourKind switch_BehaviourKindClass
B5	Eseguire più comportamenti in serie.	ActivateSequentialBehaviour switch_BehaviourExpression

Tabella 9: Implementazione dei requisiti funzionali relativi ai comportamenti.

Id	Descrizione	Implementazione
O1	Definire un'ontologia.	Ontology map_Ontology

O2	Definire concetti, azioni, predicati e proposizioni di un'ontologia.	ElementSchema switch_OntologySchemaClassConstructor switch_OntologySchemaClassType switch_ElementSchemaRecord switch_ElementSchemaType
O3	Definire concetti, azioni, predicati e proposizioni di un'ontologia composti da tipi di dato primitivi, ovvero booleani, stringhe, interi e decimali.	switch_OntologyRecordMember switch_OntologyRecordCardinality switch_OntologyRecordType
O4	Definire concetti, azioni, predicati e proposizioni di un'ontologia composti da altri concetti, azioni, predicati o proposizioni.	switch_OntologyRecordMember switch_OntologyRecordCardinality switch_OntologyRecordType
O5	Definire concetti, azioni, predicati e proposizioni di un'ontologia composti liste o insiemi di tipi di dato primitivi o di altri concetti, azioni, predicati o proposizioni.	switch_OntologyRecordMember switch_OntologyRecordCardinality switch_OntologyRecordType

Tabella 10: Implementazione dei requisiti funzionali relativi alle ontologie.

Id	Descrizione	Implementazione
M1	Inviare un messaggio o la risposta a un messaggio.	SendMessage ReplyToMessage
M2	Definire mittente, destinatari, performative, contenuto e ontologia di un messaggio.	SendMessage ReplyToMessage switch_PerformativeEnum PerformativeType PerformativeDeclaration

M3	Accedere a mittente, destinatari, performative e contenuto di un messaggio	GetContentOp GetReceiversOp GetSenderOp
M4	Consentire la ricezione bloccante di messaggi in fase d'inizializzazione di un agente.	ReceiveMessage case_BlockingReceiveMessageAsAction case_BlockingReceiveMessageAsResult
M5	Definire handler di messaggi per consentire la ricezione non bloccante di messaggi in un comportamento.	case_ReceiveMessageAsAction case_ReceiveMessageAsResult ReceiveMessage
M6	Innescare gli handler solo con messaggi aventi un determinato contenuto, performative, mittente o destinatario.	PerformativeMatch ReceiverMatch SenderMatch map_MessageMatcher

Tabella 11: Implementazione dei requisiti funzionali relativi ai messaggi ACL.

Id	Descrizione	Implementazione
L1	Definire dei Moduli, ovvero delle raccolte di dichiarazioni di funzioni, strutture dati e costanti riutilizzabili in più comportamenti e agenti.	Module map_Module
L2	Definire un sistema d'importazione per rendere visibili gli elementi di un chunk in altri chunk. Un chunk è definito come un modulo, un'ontologia, un agente o un comportamento.	AbstractChunk IChunkDep
L3	Definire le operazioni da compiere in un chunk come una serie di espressioni con effetti collaterali all'interno di un linguaggio funzionale.	AbstractExpression

L4	Definire dei tipi per rappresentare entità specifiche al dominio del progetto, quali <code>Performative</code> , <code>AID</code> , <code>ACLMessage</code> e <code>Behaviour</code> .	<code>PerformativeType</code> <code>AIDType</code> <code>MessageType</code> <code>BehaviourType</code> <code>switch_MetaJadeTypes</code>
----	--	--

Tabella 12: Implementazione dei requisiti funzionali relativi al linguaggio.

5.2. Test

Sono stati realizzati dei test per sperimentare quali siano gli strumenti messi a disposizione da MPS.

5.2.1. Test del linguaggio

Esistono diverse tipologie di test per verificare il funzionamento dei linguaggi in MPS. Per una panoramica delle possibilità si rimanda alla [documentazione ufficiale](#). Ci si è concentrati su due tipologie in particolare, i test dell'editor e i test dei nodi.

In `BehaviourActivation` si testa l'attivazione di un comportamento digitando una precisa sequenza di caratteri e scorciatoie da tastiera a partire da una posizione del cursore prestabilita.

In `MessageStatements` si verifica che siano riportate correttamente una serie di possibili errori relativi alle istruzioni per l'invio, la ricezione e la risposta a messaggi.

In particolare si verifica che:

- il tipo di contenuto di un messaggio dichiarato per effettuare un'operazione d'invio o ricezione corrisponda effettivamente;
- il tipo dei valori di ritorno delle operazioni per accedere a mittente, destinatari e contenuto di un messaggio sia corretto.

Questi test sono stati realizzati più per sperimentare alcune delle possibilità offerte da MPS che per verificare in modo esaustivo la correttezza del linguaggio. In quest'ultimo caso sarebbe stato necessario più tempo di quello a disposizione.

5.3. Workflow adottato

Nel corso del progetto sono stati realizzati i seguenti repository git:

- [metajade-examples](#),
- [jade-examples](#),

- [MetaJade](#),
- [metajade-examples-documentation](#),
- [metajade-antora-ui](#),
- [metajade-template](#).

Il repository [metajade-examples-documentation](#) fornisce un esempio di sistema multi-agente end-to-end, analizzando sia il codice sorgente del DSL che il codice Java generato. Rende disponibile un sito web statico realizzato tramite [Antora](#). Utilizza l'interfaccia utente definita in [metajade-antora-ui](#) e la documentazione disponibile in formato [AsciiDoc](#) in [metajade-examples](#).

Il repository [metajade-template](#) fornisce le basi per implementare un sistema multi-agente in MetaJade, sia dal punto di vista delle dipendenze necessarie che fornendo alcune indicazioni su come creare le singole entità e su come utilizzare l'editor di MPS.

Mentre i repository relativi agli esempi di applicazioni in Java e MetaJade e alla documentazione non adottano particolari convenzioni, il repository principale impiega la convenzione dei [conventional commit](#) e una variante del [GitLab Flow](#). In particolare non si utilizzano branch per le feature e non si creano issue relativi a questi branch. Un branch di produzione è utilizzato per pubblicare le nuove versioni del progetto. Si esegue un rilascio ogni volta che si effettua il merge del branch production nel branch main. Il rilascio consiste nella pubblicazione di un pacchetto maven nel [registro dei pacchetti](#) tramite una pipeline GitLab.

6. Istruzioni per l'utilizzo

Per utilizzare MetaJade, sia come sviluppatore di linguaggi che come utente, è possibile scaricare il codice sorgente o un pacchetto Maven dal registro di pacchetti.

In entrambi i casi un prerequisito è un'installazione di Java 11, la versione utilizzata da MPS 2021.2.x.

Se un'installazione di MPS 2021.2.x è già presente si può fornire il percorso a essa tramite la proprietà Gradle `mpsHomeDir`. In alternativa è necessario scaricare e installare MPS 2021.2.x [qui](#) se si sceglie di effettuare il build del progetto da MPS oppure si può utilizzare la versione scaricata localmente da Gradle come parte del processo di build.

6.1. Installazione del pacchetto Maven

Il linguaggio è distribuito tramite un pacchetto maven disponibile [qui](#). Per un esempio di installazione del pacchetto tramite Gradle si rimanda al repository [metajade-examples](#).

6.2. Build del progetto con Gradle

Per prima cosa si scarica il repository del progetto o lo si clona lanciando i seguenti comandi:

```
git clone https://gitlab.com/pika-lab/courses/ds/projects/ds-project-battistini-ay2122
cd ./ds-project-battistini-ay2122
```

Per scaricare le dipendenze ed effettuare il build si lancia il seguente comando:

```
./gradlew build
# or
# to get the JBR
./gradlew resolveArtifacts
# to build with the JBR
./gradlew build -Pjava11_home="build/jbrDownload/jbr"
```

Ora il progetto può essere aperto in MPS. Se Gradle è stato lanciato da MPS allora si dovrebbe riavviare il workbench.

6.3. Build del progetto da MPS

Per prima cosa si scarica il repository del progetto o lo si clona lanciando i seguenti comandi:

```
git clone https://gitlab.com/pika-lab/courses/ds/projects/ds-project-battistini-ay2122
cd ./ds-project-battistini-ay2122
```

Per scaricare le dipendenze:

```
./gradlew resolveLanguageLibs
```

Per effettuare il build del progetto in MPS si consiglia di partire prima da linguaggi e soluzioni nel *virtual package base*, per poi passare a quelli in `java`. La prima volta che si effettua il build potrebbe essere necessario [ignorare gli errori](#).

6.4. Esecuzione dei test

6.4.1. Test del linguaggio

Per lanciare i test dall'interno di MPS è necessario creare una nuova *path variable* di nome `project_home` e con valore il percorso al repository del progetto. Per creare una path variable: **Settings>Appearance & Behavior>Path Variables**. Successivamente basta eseguire i test, che si trovano nella soluzione `MetaJade.tests` nel *virtual package base*.

6.5. Lancio della piattaforma e degli agenti Jade

Per prima cosa si scarica il repository degli esempi o lo si clona lanciando i seguenti comandi:

```
git clone https://gitlab.com/metajade/metajade-examples
cd ./metajade-examples
```

Si può installare una versione di MPS tramite lo script Gradle tramite il seguente comando:

```
./gradlew resolveMps
```

In alternativa è necessario indicare il percorso ad un'installazione esistente, specificando la proprietà `mpsHomeDir` perché siano inserite nel classpath le librerie di MPS necessarie a lanciare con Gradle le classi generate.

Per lanciare la piattaforma Jade:

```
./gradlew startPlatform
```

Per lanciare l'esempio del PingPong:

```
./gradlew startContainer -Pagents=Ponger:MetaJade.examples.PingPong.Ponger
./gradlew startContainer -Pagents=Pinger:MetaJade.examples.PingPong.Pinger
```

Per lanciare l'esempio del BankAccount:

```
./gradlew startContainer -Pagents=BankServer:MetaJade.examples.BankAccount.BankServer

# Create Account
./gradlew startContainer -Pagents="BankClient:MetaJade.examples.BankAccount.BankClient(0, stanley, 43, 1)"
# Deposit
./gradlew startContainer -Pagents="BankClient:MetaJade.examples.BankAccount.BankClient(0, stanley, 43, 2)"
# Get the Balance
./gradlew startContainer -Pagents="BankClient:MetaJade.examples.BankAccount.BankClient(0, stanley, 43, 3)"
# Withdraw
./gradlew startContainer -Pagents="BankClient:MetaJade.examples.BankAccount.BankClient(0, stanley, 43, 4)"
# Get the list of Operations
./gradlew startContainer -Pagents="BankClient:MetaJade.examples.BankAccount.BankClient(0, stanley, 43, 5)"
```

7. Esempi di utilizzo

Nel repository [metajade-examples](https://gitlab.com/metajade/metajade-examples) sono disponibili due esempi di utilizzo di MetaJade.

Il primo, denominato “PingPong”, rappresenta un esempio di scambio di messaggi tra agenti e consente di dimostrare le seguenti feature del linguaggio:

- istruzioni **send** e **receive** per gestire l'interazione tra gli agenti;
- ontologia formata da elementi composti da tipi primitivi;
- passaggio di parametri a riga di comando;
- passaggio di parametri a un comportamento;
- mantenimento dello stato dell'agente **Pinger** tramite una variabile che tiene conto del numero di scambi di messaggi effettuati.

Il secondo, denominato “BankAccount”, rappresenta uno scenario semplificato di gestione di un account bancario. L'esempio è stato realizzato a partire dal tutorial sull'utilizzo delle ontologie in Jade disponibile [qui](#). In questo esempio sono presenti due agenti, **BankClient** e **BankServer**. Il **BankClient** può eseguire una di queste azioni:

- richiesta di creare un account;
- richiesta di accesso a informazioni relative alla lista di operazioni eseguite o al bilancio di un account;
- richiesta di effettuare un'operazione di deposito o prelievo.

Il **BankServer** ha il compito di processare le richieste dei client e di riportare eventuali errori dovuti a richieste di prelievo con fondi insufficienti, richieste di effettuare operazioni con ammontare negativo o richieste di effettuare azioni riguardanti account inesistenti.

Questo esempio consente di dimostrare la quasi totalità delle feature del linguaggio. Si tratta di un esempio giocattolo che vuole ricalcare il più possibile l'esempio originale in Java e mostrare aspetti del linguaggio che rientrano nei requisiti.

Dei due esempi realizzati si trova un corrispettivo in Java nel repository [jade-examples](#).

8. Conclusioni

Nel corso di questo progetto è stato possibile realizzare un linguaggio per la programmazione di agenti, basato sulla JVM, denominato MetaJade.

MetaJade contribuisce a isolare alcune delle astrazioni più rilevanti emerse dall'uso delle piattaforme ad agenti e le equipaggia con un linguaggio dedicato. Implementando un sottoinsieme delle funzionalità offerte da Jade, offre uno strumento per la risoluzione di problemi tramite il paradigma della programmazione ad agenti.

Il linguaggio è open source ed è disponibile [qui](#).

8.1. Sviluppi futuri

Gli sviluppi futuri includono aggiunte e migliorie al generatore e al linguaggio.

Per quanto riguarda il generatore l'attuale implementazione compila in Java. Nel caso si voglia mantenere la dipendenza su Jade si potrebbero scrivere dei nuovi generatori

in C# o Kotlin ad esempio. In alternativa si potrebbe non usare Jade e generare in uno degli altri linguaggi supportati da MPS. In ogni caso si possono aggiungere nuovi linguaggi a MPS impiegando `textgen` per convertire un template direttamente in testo.

Per quanto riguarda lo sviluppo del linguaggio esistono diverse opzioni. Tutte hanno in comune la necessità di far evolvere il metamodello, il sistema dei tipi, la sintassi e la semantica del linguaggio introducendo nuovi moduli. Questi moduli estendono il linguaggio in modo simile alle librerie ma senza alcune delle limitazioni, dato che permettono di arricchire direttamente il linguaggio e gli strumenti offerti dall'IDE.

Tra le principali estensioni che si potrebbero implementare si segnalano:

1. Esecuzione di agenti mobili, persistenti e replicati.
2. Gestione di eventi diversi dalla ricezione di messaggi.
3. Feature relative ai requisiti aggiuntivi od opzionali non realizzati.

Il primo punto può essere realizzato tramite un linguaggio che specifichi in modo dichiarativo come effettuare l'attivazione degli agenti e dei relativi container.

Il secondo punto richiede di realizzare un framework per consentire a un provider di eventi d'inviare notifiche agli agenti. In questo modo gli agenti potrebbero operare in ambienti complessi e dinamici tramite interazioni mediate dall'ambiente e non necessariamente tramite messaggi scambiati direttamente tra loro.

Lo svolgimento del terzo punto richiede di implementare le seguenti feature:

- impiego di protocolli d'interazione;
- definizione di un linguaggio per creare GUI Swing per interagire con gli agenti;
- sottoscrizione alle pagine gialle, per cui il DF notifica gli agenti quando un agente registra o ritira un determinato servizio dalla piattaforma;
- creazione di comportamenti di tipo `WakerBehaviour` o `TickerBehaviour`;
- composizione dei singoli comportamenti, definendo dei sotto comportamenti, tramite le classi `ParallelBehaviour` e `FSMBehaviour`;
- condivisione di dati tra sotto comportamenti utilizzando il `DataStore`.

8.2. Conoscenze apprese

Questo progetto mi ha permesso di acquisire conoscenze relative alla programmazione orientata agli agenti, in particolare ho avuto la possibilità di approfondire il funzionamento e le caratteristiche della piattaforma Jade e di realizzare semplici applicazioni multi agente.

Inoltre è stato possibile acquisire familiarità con MPS e il suo editor di proiezione, oltre che sperimentare la possibilità di comporre linguaggi modulari.

Infine ho acquisito maggiore confidenza nell'uso di strumenti per la CI/CD in GitLab, adottato i conventional commit per scrivere una storia dei commit più strutturata e acquisito esperienza nell'uso di Gradle per l'automazione dello sviluppo.

8.3. Commenti

MPS è strumento potente e con una curva di apprendimento ripida. Nel corso del progetto mi sono confrontato con diverse difficoltà, dovute sia all'inesperienza nell'utilizzo di MPS che nel design di DSL, oltre che alla carenza di documentazione ufficiale e alle ridotte dimensioni della comunità di utilizzatori del language workbench. Durante lo svolgimento del progetto sono state molto utili le seguenti risorse:

- i [corsi introduttivi](#) gratuiti offerti dagli sviluppatori di MPS,
- la raccolta di buone pratiche e indicazioni presso [MPS Platform Docs](#),
- i consigli reperibili al sito [specificlanguages.com](#),
- il corso realizzato da Voelter [mpsintroduction](#),
- [Physics](#), esempio di utilizzo di KernelF in un progetto di dimensioni significative disponibile pubblicamente,
- la raccolta di esempi scaricabile inclusa con l'installazione di MPS.

Riferimenti bibliografici

- [1] Y. Shoham, «AGENT0: A Simple Agent Language and Its Interpreter,» in *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 2*, T. L. Dean e K. R. McKeown, cur., AAAI Press / The MIT Press, 1991, pp. 704–709. indirizzo: <http://www.aaai.org/Library/AAAI/1991/aaai91-110.php> (cit. a p. 5).
- [2] M. P. Ward, «Language-Oriented Programming,» *Softw. Concepts Tools*, vol. 15, n. 4, pp. 147–161, 1994 (cit. a p. 6).
- [3] F. Belfemine, F. Bergenti, G. Caire e A. Poggi, «JADE - A Java Agent Development Framework,» in *Multi-Agent Programming: Languages, Platforms and Applications*, ser. Multiagent Systems, Artificial Societies, and Simulated Organizations, R. H. Bordini, M. Dastani, J. Dix e A. E. F. Seghrouchni, cur., vol. 15, Springer, 2005, pp. 125–147 (cit. a p. 6).
- [4] F. A. (2002), «FIPA Specification Repository,» *Foundation for Intelligent Physical Agents (FIPA)*, indirizzo: <http://www.fipa.org/repository/index.html> (visitato il 15/12/2022) (cit. a p. 6).
- [5] F. Belfemine, G. Caire e D. Greenwood, *Developing Multi-Agent Systems with JADE*. Wiley, 2007, ISBN: 978-0-47005747-6. DOI: [10.1002/9780470058411](https://doi.org/10.1002/9780470058411). indirizzo: <https://doi.org/10.1002/9780470058411> (cit. alle pp. 6, 8, 9, 11, 12).
- [6] «JetBrains MPS Documentation,» indirizzo: <https://www.jetbrains.com/mps/documentation> (cit. a p. 7).

- [7] M. Voelter, «The Design, Evolution, and Use of KernelF - An Extensible and Embeddable Functional Language,» in *Theory and Practice of Model Transformation - 11th International Conference, ICMT@STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings*, A. Rensink e J. S. Cuadrado, cur., ser. Lecture Notes in Computer Science, vol. 10888, Springer, 2018, pp. 3–55. DOI: [10.1007/978-3-319-93317-7_1](https://doi.org/10.1007/978-3-319-93317-7_1). indirizzo: <http://voelter.de/data/books/kernelf-designEvoUse.pdf> (cit. alle pp. 7, 17, 18).
- [8] M. Voelter, «When Is Something a (Domain- Specific) Language?» *Medium*, 18 giu. 2022. indirizzo: <https://markusvoelter.medium.com/when-is-something-a-domain-specific-language-83b7eff79ed4> (visitato il 15/12/2022) (cit. a p. 7).
- [9] M. Voelter, «High-Level Structure of DSLs: Three Patterns,» *Medium*, 21 set. 2017. indirizzo: <https://languageengineering.io/high-level-structure-of-dsls-three-patterns-7375c8baa2d3> (visitato il 15/12/2022) (cit. a p. 7).
- [10] F. Bergenti, S. Monica e G. Petrosino, «A scripting language for practical agent-oriented programming,» in *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*, J. D. Koster, F. Bergenti e J. Franco, cur., ACM, 2018, pp. 62–71. DOI: [10.1145/3281366.3281367](https://doi.org/10.1145/3281366.3281367). indirizzo: <https://doi.org/10.1145/3281366.3281367> (cit. a p. 7).
- [11] G. Petrosino e F. Bergenti, «An Introduction to the Major Features of a Scripting Language for JADE Agents,» in *AI*IA 2018 - Advances in Artificial Intelligence - XVIIIth International Conference of the Italian Association for Artificial Intelligence, Trento, Italy, November 20-23, 2018, Proceedings*, C. Ghidini, B. Magnini, A. Passerini e P. Traverso, cur., ser. Lecture Notes in Computer Science, vol. 11298, Springer, 2018, pp. 3–14. DOI: [10.1007/978-3-030-03840-3_1](https://doi.org/10.1007/978-3-030-03840-3_1). indirizzo: https://doi.org/10.1007/978-3-030-03840-3_1 (cit. a p. 7).
- [12] G. Petrosino, E. Iotti, S. Monica e F. Bergenti, «A Description of the Jadescript Type System,» in *Distributed Artificial Intelligence - Third International Conference, DAI 2021, Shanghai, China, December 17-18, 2021, Proceedings*, J. Chen, J. Lang, C. Amato e D. Zhao, cur., ser. Lecture Notes in Computer Science, vol. 13170, Springer, 2021, pp. 206–220. DOI: [10.1007/978-3-030-94662-3_13](https://doi.org/10.1007/978-3-030-94662-3_13). indirizzo: https://doi.org/10.1007/978-3-030-94662-3_13 (cit. a p. 7).
- [13] F. Bergenti e G. Petrosino, «Overview of a Scripting Language for JADE-Based Multi-Agent Systems,» in *Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, June 28-29, 2018*, M. Cossentino, L. Sabatucci e V. Seidita, cur., ser. CEUR Workshop Proceedings, vol. 2215, CEUR-WS.org, 2018, pp. 57–62. indirizzo: http://ceur-ws.org/Vol-2215/paper_10.pdf (cit. a p. 7).
- [14] G. Petrosino e F. Bergenti, «Extending Message Handlers with Pattern Matching in the Jadescript Programming Language,» in *Proceedings of the 20th Workshop "From Objects to Agents", Parma, Italy, June 26th-28th, 2019*, F. Bergenti e S. Monica, cur., ser. CEUR Workshop Proceedings, vol. 2404, CEUR-WS.org, 2019,

- pp. 113–118. indirizzo: <http://ceur-ws.org/Vol-2404/paper17.pdf> (cit. a p. 7).
- [15] G. Petrosino, E. Iotti, S. Monica e F. Bergenti, «Prototypes of Productivity Tools for the Jadescript Programming Language,» in *Proceedings of the 22nd Workshop "From Objects to Agents", Bologna, Italy, September 1-3, 2021*, R. Calegari, G. Ciatto, E. Denti, A. Omicini e G. Sartor, cur., ser. CEUR Workshop Proceedings, vol. 2963, CEUR-WS.org, 2021, pp. 14–28. indirizzo: <http://ceur-ws.org/Vol-2963/paper4.pdf> (cit. alle pp. 7, 25).
- [16] F. A. (2002), «FIPA SL Content Language Specification,» *Foundation for Intelligent Physical Agents (FIPA)*, indirizzo: <http://www.fipa.org/specs/fipa00008/SC00008I.html> (visitato il 18/02/2023) (cit. a p. 12).

A. Terminologia di MPS

In questa appendice si introduce la terminologia principale di MPS.

A.1. Progetto e Modulo

Un progetto MPS è composto da uno o più *moduli*. Ci sono quattro tipi di moduli:

- **Linguaggi:** definizione di diversi aspetti dei linguaggi che possono essere usati per scrivere dei programmi nelle soluzioni;
- **Generatori:** definiscono delle regole di conversione per trasformare del codice in codice di un altro linguaggio;
- **Soluzioni:** contengono il codice, programmi o librerie.
- **Devkit:** permettono di raggruppare più linguaggi e soluzioni insieme, in modo che gli utenti possano esprimere più dipendenze tramite un'unica dichiarazione.

Le soluzioni possono essere di tre tipi:

- **Sandbox:** contiene i programmi realizzati dagli utenti finali;
- **Soluzioni runtime:** contengono librerie di codice riutilizzabile, solitamente caricato da fonti esterne, come archivi jar;
- **Soluzioni plugin:** estendono le funzionalità dell'IDE, con nuovi elementi nei menu, nuove azioni o strumenti.

Un progetto è una raccolta di moduli. Un modulo è una raccolta di modelli. Un programma si trova in un modello.

A.2. Programma

Un programma rappresenta il codice scritto dagli utenti finali di un linguaggio ed è rappresentato sia dalla sua sintassi astratta che da quella concreta.

La sintassi concreta è utilizzata dagli utenti finali per creare e modificare i programmi. La sintassi astratta è una struttura dati che contiene i dati espressi con la sintassi concreta, ma senza i dettagli relativi alla notazione.

In un sistema basato sui parser, si mappa la sintassi concreta su quella astratta. Gli utenti manipolano un flusso di caratteri e il parser deriva l'albero della sintassi astratta (AST) utilizzando una grammatica.

In MPS si utilizzano *editor di proiezione*. Gli editor di proiezione mappano la sintassi astratta su quella concreta. In altri termini l'utente manipola direttamente l'AST. La sintassi concreta è una proiezione che dà un'esperienza di utilizzo simile a quella con cui si manipola il testo e che è definita dal linguaggio con cui è scritto il programma.

Un programma MPS è composto da uno o più *nodi radice*. Ciascun nodo radice può essere visto come un'astrazione equivalente ai file che contengono il codice sorgente negli altri linguaggi di programmazione.

A.3. Linguaggio

Un linguaggio è definito da diversi *aspetti*. Ciascuno di questi aspetti è implementato utilizzando uno specifico DSL e riutilizza il *BaseLanguage*, ovvero una variante di Java implementata in MPS con diverse estensioni.

I principali aspetti di un linguaggio sono:

- **Struttura:** definisce la sintassi astratta di un concetto.
- **Editor:** definisce la sintassi concreta di un concetto. In altri termini, definisce la notazione con cui è rappresentata l'istanza di un concetto. Inoltre personalizza il modo in cui l'utente interagisce con l'istanza di un concetto quando si modifica un programma tramite l'uso di azioni.
- **Vincoli:** impediscono all'utente di inserire codice non valido. Inoltre possono essere usate per definire la visibilità delle istanze di un concetto, ovvero lo *scope*. Uno scope permette di stabilire quali nodi sono visibili nel riferimento ad un concetto. L'inserimento del riferimento avviene tramite menu di completamento o digitando il nome del concetto.
- **Sistema dei Tipi:** codifica la semantica statica di un linguaggio. Garantisce la consistenza dei tipi nel programma attraverso delle regole dei tipi. Ciascuna di queste regole rappresenta un'equazione che compone un sistema risolto dal solver dei tipi di MPS. Questo aspetto permette inoltre di stabilire delle regole arbitrarie aggiuntive oltre a quelle strutturali, dette *checking rules*.
- **Generatore:** definisce una trasformazione che mappa un AST in input ad un AST in output. Ciascun generatore è composto da diversi tipi di regole di trasformazione, formate a loro volta da *template*. I template sono frammenti del linguaggio in cui si vuole generare che determinano come avviene la trasformazione tra gli AST. I generatori solitamente si affidano al *framework di un dominio*, ovvero ad un insieme di librerie invocato dal codice generato.

Nota

La generazione in MPS è svolta in fasi. L'output di un generatore può diventare l'input di un altro in una pipeline. Tramite i *piani di generazione* è possibile definire sequenze di generatori da utilizzare in una pipeline.

Per una descrizione più approfondita degli aspetti dei linguaggi, si rimanda alla [documentazione ufficiale](#).