



UNIVERSITY  
OF TRENTO

---

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

---

38050 Povo – Trento (Italy), Via Sommarive 14  
<http://www.dit.unitn.it>

THE TROPOS MODELING LANGUAGE. A USER GUIDE

Fabrizio Sannicolo', Anna Perini, and Fausto Giunchiglia

February 2002

Technical Report # DIT-02-0061



# Premessa

*Tropos è una nuova metodologia basata sul paradigma dei sistemi multi-agente che supporta il progettista in tutto il processo di sviluppo del software, dall'analisi dei requisiti all'implementazione del sistema. Essa vuole offrire un approccio strutturato allo sviluppo del software, basato sulla costruzione di modelli concettuali definiti secondo un linguaggio di modellazione visuale, i cui elementi base sono concetti quali agente (attore), credenze, obiettivi, piani e intenzioni. Tropos si caratterizza per tre idee chiave: (i) le nozioni di agente, goal, piani e altre nozioni mentalistiche sono usate lungo tutte le fasi di sviluppo del software; (ii) l'adozione di un approccio allo sviluppo del software guidato dai requisiti anziché dai vincoli dettati dalla piattaforma di implementazione scelta; (iii) la costruzione di modelli concettuali seguendo un approccio trasformatore di tipo incrementale. Questo lavoro si colloca all'interno di un progetto che coinvolge diverse università e istituti di ricerca nel mondo, tra le quali l'Università degli Studi di Trento e l'ITC - IRST. Obiettivo di questo documento è quello di fornire una guida all'uso della metodologia Tropos lungo tutte le fasi del processo di sviluppo del software con particolare enfasi al linguaggio di modellazione visuale. Il linguaggio utilizzato in Tropos è un linguaggio di specifica semiformale caratterizzato da un'ontologia, un meta-modello, una notazione grafica e un insieme di regole. L'ontologia è rappresentata da un insieme di concetti per la modellazione (attori, goal, piani) e di relazioni tra questi (dipendenze). Il meta-modello (descritto tramite diagrammi delle classi UML) è necessario per la specifica dei modelli Tropos. Ciascun concetto definito all'interno del meta-modello dispone della propria rappresentazione grafica che lo identifica lungo tutte le fasi del processo. Sono disponibili vari diagrammi che catturano aspetti statici e dinamici dei modelli da più punti di vista. Ogni diagramma è costruito seguendo un insieme di regole precise che guidano all'uso dei concetti durante le diverse fasi del processo di sviluppo del software.*

## **Parole chiave**

Ingegneria del software, sistemi multi-agente.



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>I</b>	<b>La Metodologia Tropos e il Linguaggio di Modellazione</b>	<b>5</b>
<b>2</b>	<b>La metodologia</b>	<b>7</b>
2.1	Fasi del processo di sviluppo . . . . .	8
2.2	I concetti chiave . . . . .	10
2.3	Attività di modellazione . . . . .	12
2.4	Tropos a confronto con altre metodologie . . . . .	14
<b>3</b>	<b>Il linguaggio di modellazione</b>	<b>15</b>
3.1	Il meta-modello . . . . .	15
3.1.1	Il livello meta-metamodello . . . . .	17
3.1.2	Il livello meta-modello . . . . .	18
3.1.3	Il livello del dominio . . . . .	29
3.1.4	Il livello oggetto . . . . .	29
3.2	Concetti chiave . . . . .	30
3.3	Notazione grafica e diagrammi . . . . .	31
3.3.1	Il diagramma degli attori . . . . .	32
3.3.2	Il diagramma dei goal . . . . .	36
3.3.3	Il diagramma delle abilità . . . . .	39
3.3.4	Il diagramma dei piani . . . . .	40
3.3.5	Il diagramma delle interazioni tra agenti . . . . .	42
<b>II</b>	<b>Un esempio. Specifica di un sistema di supporto all'organizzazione di meeting</b>	<b>45</b>
<b>4</b>	<b>Il sistema per l'organizzazione di meeting</b>	<b>47</b>
4.1	Definizione del problema . . . . .	47

4.2	Analisi e specifica dei requisiti . . . . .	48
4.2.1	Analisi dell'ambiente . . . . .	49
4.2.2	Analisi del sistema futuro . . . . .	55
4.3	Progettazione del sistema . . . . .	62
4.3.1	La fase architettonica globale . . . . .	62
4.3.2	La fase di dettaglio architettonica . . . . .	67
4.4	Implementazione del sistema . . . . .	69
	<b>Bibliografia</b>	<b>71</b>
	<b>A Glossario e Trasformazioni</b>	<b>77</b>
A.1	Glossario . . . . .	77
A.2	Trasformazioni . . . . .	79
	<b>B Grammatica di Formal Tropos</b>	<b>85</b>

# Elenco delle tabelle

2.1	Fasi del processo di sviluppo di <i>Tropos</i> . La metodologia è basata su cinque passi (4 + 1). Accanto al nome della fase c'è una breve descrizione schematica. . . . .	8
2.2	Mapping Tropos - JACK. . . . .	11
3.1	Architettura su quattro livelli. Accanto alla descrizione di ogni singolo livello sono riportati alcuni esempi di nozioni modellabili (colonna di destra). . . . .	16
4.1	Passo 2: processo di <i>identificazione</i> delle capability. . . . .	66
4.2	Passo 3: processo di <i>agentificazione</i> . Si identificano gli agenti e poi si assegnano le capability individuate al passo 2. . . . .	67
A.1	Actor Transformations tratte da [6]. . . . .	79
A.2	Goal Transformations tratte da [6]. . . . .	80
A.3	SoftGoal Transformations tratte da [6]. . . . .	81
A.4	Trasformazioni primitive per i goal tratte da [6]. . . . .	82
A.5	Trasformazioni primitive per i softgoal tratte da [6]. . . . .	83
A.6	Actor Primitive Transformations tratte da [6]. . . . .	83
B.1	Grammatica di <i>Formal Tropos</i> . . . . .	86





# Elenco delle figure

2.1	<i>Actor modeling</i> attraverso le cinque fasi del processo di sviluppo. . . . .	12
2.2	Attività di modellazione attraverso le cinque fasi del processo di sviluppo. . . . .	12
2.3	Tropos a confronto con altre metodologie AO per lo sviluppo di sistemi software. . . . .	14
3.1	Diagramma delle classi UML che specifica interamente il <i>meta-metamodello</i> di Tropos. . . . .	17
3.2	Diagramma delle classi UML che visualizza le relazioni che si possono istanziare a livello di dominio. . . . .	18
3.3	Diagramma delle classi UML che mostra la relazione esistente fra alcuni elementi del <i>meta-metamodello</i> e del <i>meta-modello</i> di Tropos. L'asterisco sull'arco etichettato <i>fulfillment</i> impone una restrizione al concetto Goal: solo l'Hardgoal può avere delle <i>T-L Formula</i> . . . . .	19
3.4	Diagramma delle classi UML che specifica il meta-modello di Tropos riferito al concetto ontologico di <i>Actor</i> . . . . .	21
3.5	Diagramma delle classi UML che specifica la porzione del meta-modello di Tropos riferito al concetto ontologico di <i>Goal</i> . . . . .	25
3.6	Diagramma delle classi UML che specifica il meta-modello di Tropos riferito al concetto ontologico di <i>Plan</i> . . . . .	28
3.7	Diagramma delle classi UML che specifica il <i>livello del dominio</i> di Tropos. L'attore Citizen <i>vuole</i> l'hardgoal get cultural information. . . . .	29
3.8	Diagramma delle classi UML che specifica il <i>livello del dominio</i> di Tropos riferito al dominio applicativo dell' <i>eCulture System</i> . . . . .	30
3.9	Diagramma delle classi UML che specifica il <i>livello oggetto</i> di Tropos riferito al dominio applicativo dell' <i>eCulture System</i> . . . . .	30
3.10	Diagramma degli attori estratto dal case study <i>eCulture System</i> . . . . .	32
3.11	Relazione fra l'attore generico Citizen e agent, position, role. . . . .	33
3.12	Estensione del diagramma degli attori che analizza le dipendenze tra attori del sistema e attori sociali per l'attribuzione delle <i>capability</i> . Il rettangolo tratteggiato racchiude i <i>subactor</i> ai quali Info Broker delega alcune competenze. . . . .	34
3.13	Un esempio tipico di diagramma degli attori estratto dal case study. . . . .	36

3.14	Rappresentazione grafica dei tipi di <i>analisi</i> ammesse nel diagramma dei goal di Tropos e condotte all'interno del balloon. . . . .	38
3.15	Diagramma delle abilità riferito alla capability query result. . . . .	39
3.16	Diagramma dei piani riferito al piano evaluate query results. . . . .	41
3.17	Esempio di diagramma delle interazioni tra agenti riferito alle interazioni possibili tra gli agenti. I messaggi sono asincroni. . . . .	42
4.1	Diagramma degli attori che introduce i due attori sociali dell'ambiente (MI e PP) con i rispettivi goal. . . . .	49
4.2	Diagramma dei goal costruito dal punto di vista dell'attore Potential Participant sul suo hardgoal attend a meeting. . . . .	50
4.3	Diagramma dei goal costruito dal punto di vista dell'attore Meeting Initiator parte prima. . . . .	52
4.4	Diagramma dei goal costruito dal punto di vista dell'attore Meeting Initiator parte seconda. . . . .	53
4.5	Diagramma degli attori riassuntivo tra Meeting Scheduler e Potential Participant. . . . .	54
4.6	Diagramma degli attori che visualizza le dipendenze fra l'attore di sistema Meeting Schedule System e gli attori sociali MI e PP. . . . .	56
4.7	Diagramma dei goal costruito dal punto di vista del Meeting Schedule System parte prima: sono analizzati l'hardgoal generate data meeting (cd,tp,l) e il softgoal effective organization. . . . .	58
4.8	Diagramma dei goal costruito dal punto di vista del Meeting Schedule System parte seconda: sono analizzati tre dei cinque goal che partecipano al soddisfacimento del goal <i>root</i> . . . . .	60
4.9	Diagramma dei goal costruito dal punto di vista del Meeting Schedule System parte terza: sono analizzati i due goal rimanenti, handle several meeting request in parallel e available info easy. . . . .	61
4.10	Diagramma degli attori riassuntivo uscente dagli <i>early</i> e <i>late requirements</i> : primo passo dell' <i>architectural design</i> . . . . .	63
4.11	Chiusura del primo passo dell' <i>architectural design</i> : sono introdotti nuovi attori di sistema con relative dipendenze. . . . .	64
4.12	Secondo passo dell' <i>architectural design</i> : identificazione delle capability attraverso l'analisi delle dipendenze di ciascun <i>subactor</i> . . . . .	65
4.13	Diagramma delle abilità di get query result:information (numero 1.a). . . . .	68
4.14	Diagramma dei piani riferito a evaluate query result. . . . .	69
4.15	Diagramma delle interazioni tra agenti fra alcuni agenti del sistema. . . . .	70

# Capitolo 1

## Introduzione

Con il diffondersi di applicazioni software nei settori dell'eBusiness e dei servizi (telecomunicazioni, pubblica amministrazione) si assiste ad una sempre più pressante richiesta di architetture aperte, che possano evolvere ed includere nuovi componenti software. Nel corso degli ultimi anni, il processo di progettazione di tali sistemi software è diventato sempre più complesso. La definizione di appropriati meccanismi di comunicazione, di negoziazione e di coordinamento tra i componenti software e gli attori umani motiva lo studio di tecniche e metodi che supportino il progettista attraverso tutto il processo di sviluppo del software, dall'analisi dei requisiti fino all'implementazione.

Una risposta a questo problema è venuta dalla scienza dell'ingegneria del software, che ha fornito nel corso degli ultimi anni molte tecniche, metodi e metodologie che hanno permesso di trattare sistemi software sempre più complessi. L'approccio sistematico all'analisi, alla progettazione, alla programmazione e manutenzione di sistemi software ha obbligato molte comunità scientifiche a darsi delle regole e linguaggi comuni per risolvere problemi di grossa complessità. Ecco allora la necessità di definire dei linguaggi capaci di conservare il rigore del formalismo matematico e, allo stesso tempo, conservare un certo grado di intuitività nel descrivere i domini applicativi ed eventuali soluzioni al problema che si vuole affrontare.

Accanto agli strumenti offerti dall'ingegneria del software servono dei paradigmi che permettano di modellare le entità coinvolte e gli obiettivi perseguiti da tali entità. A tal fine può essere utile adottare concetti definiti in comunità scientifiche diverse quali il concetto di agente, utilizzato in Intelligenza Artificiale e nell'ingegneria dei requisiti, seppur con significati leggermente diversi. L'agente possiede degli obiettivi da soddisfare, è autonomo nelle sue decisioni e nei suoi comportamenti, percepisce i cambiamenti provenienti dall'ambiente in cui opera e coopera con altri agenti al raggiungimento dei suoi obiettivi e di quelli comuni. Un gruppo di agenti che comunicano tra loro si definisce sistema multi-agente. I sistemi multi-agente offrono una soluzione a sistemi aperti, distribuiti e complessi.

L'approccio nato dall'incontro dell'ingegneria del software e dei sistemi multi-agente è de-

nominato *Agent-Oriented Software Engineering (AOSE)*. In questo contesto si colloca la nuova metodologia *Tropos*<sup>1</sup> per la costruzione orientata agli agenti di sistemi software. Tropos copre cinque fasi del processo di sviluppo del software: *early requirements* consente di analizzare e modellare i requisiti del contesto laddove il sistema software verrà calato, *late requirements* si rivolge invece allo studio dei requisiti del sistema software da realizzare, *architectural design* e *detailed design* con l'obiettivo di progettare l'architettura del sistema software e, infine, *implementation* dove si conclude con l'implementazione del codice.

Questo documento ha come scopo quello di fornire una guida all'uso della metodologia Tropos attraverso le cinque fasi del processo di sviluppo del software, fornendo in particolare la struttura del linguaggio di specifica. Tale linguaggio è un linguaggio di modellazione visuale semiformale composto da:

- **un'ontologia** – un insieme di concetti per la modellazione. L'ontologia definisce informalmente il significato di tali concetti, cioè non fornisce uno strumento formale che permetta di controllare la semantica dei modelli costruiti. Gli elementi che fanno parte dell'ontologia Tropos sono i seguenti: *attori, goal, piani, risorse, dipendenze, abilità e credenze*.
- **meta-modello** – definisce la sintassi del linguaggio. Il meta-modello fornisce uno strumento per controllare che i modelli siano sintatticamente corretti. È stato specificato con un insieme di diagrammi di classe UML.
- **notazione grafica e diagrammi** – stabilisce come raffigurare graficamente gli elementi dei modelli.
- **insieme di regole d'uso** – guidano la costruzione di un modello concettuale nelle varie fasi del processo.

Il meta-modello permette di costruire modelli del dominio applicativo sulla base delle informazioni raccolte nella primissima fase di analisi dei requisiti per poi raffinarli incrementalmente fino ad ottenere delle specifiche facilmente implementabili con qualche linguaggio di programmazione. Il meta-modello di Tropos è descritto tramite diagrammi delle classi UML. La base di partenza è stato il lavoro di Eric Yu dove si usano i concetti di attore, goal e dipendenze tra attori. L'attore è visto come un'entità strategica e sorgente d'intenzionalità. Dall'analisi di tale entità e delle sue dipendenze con altri attori mediante tecniche quali *means-ends analysis*, *contribution* e *AND-OR decomposition*, è possibile individuare gli agenti e le rispettive *capability* che andranno a comporre il sistema software che risolve il problema. Gran parte dei concetti definiti nel meta-modello vengono usati lungo tutte le fasi che compongono la metodologia Tropos. All'interno del modello definiamo più viste sullo stesso attraverso la costruzione di diagrammi specifici i quali consentono di approfondire gli aspetti statici e dinamici del modello. Per ciascun diagramma

---

<sup>1</sup>Il nome *Tropos* proviene dal greco *tropé* che significa "facilmente modificabile", "facilmente adattabile".

proponiamo la rappresentazione grafica, le linee guida che ne consentono l'uso attraverso tutte le fasi del processo e definiamo esattamente l'ingresso e l'uscita per ognuno. Per verificare il meta-modello di Tropos lo abbiamo applicato al caso di studio di un sistema per l'organizzazione di un meeting (d'ora in avanti indicato come *Meeting Scheduler System*).

Il documento che stiamo per presentare è ricavato da un lavoro di tesi [40] sviluppato all'IRST nell'ambito di un progetto che coinvolge numerose università ed istituti di ricerca nel mondo: IRST, Università degli Studi di Trento, Università di Toronto, Università di York, Università Federale di Pernambuco e Università di RWTH Aachen.

Questa guida all'uso è così strutturata. Il capitolo 2 descrive la metodologia Tropos, capitolo 3 il linguaggio di modellazione visuale con i relativi diagrammi, infine il capitolo 4 applica quanto visto al caso di studio *Meeting Scheduler System*.



## **Parte I**

# **La Metodologia Tropos e il Linguaggio di Modellazione**





## Capitolo 2

# La metodologia

Nel corso degli ultimi anni molti fattori hanno causato un aumento esponenziale della complessità dei sistemi software. Esempi di applicazioni dove è accaduto sono e-commerce, e-business, e-services e mobil computing. Il software deve essere basato su architetture aperte ed evolvere nel tempo per integrare nuovi componenti hardware e rispondere alla necessità di nuovi requisiti; è determinante il grado di autonomia e robustezza di un sistema software, sia esso residente su poche e limitate macchine, sia esso distribuito. Oltretutto deve servire gli utenti da più piattaforme senza necessitare di essere ricompilato ed operare con poche conoscenze circa l'ambiente di esercizio e circa i suoi utenti.

L'aumento della complessità richiede lo sviluppo di nuove tecniche e tool per l'analisi, progettazione, programmazione e manutenzione di sistemi software. A nostro avviso, questa pressante richiesta è soddisfatta analizzando non solo il *cosa* e il *come* di un sistema software, ma anche il *perché* noi lo usiamo. Questo può essere fatto mediante l'uso della metodologia *Tropos* che fruisce dell'esperienza e potenzialità del linguaggio di Eric Yu, \* [46, 48, 49].

Tropos è una metodologia AOSE che si basa su tre idee chiave [35, 34]:

1. Le nozioni di agente e di tutte le nozioni mentalistiche ad esso associate come *goal*, *plan*, *belief*, sono usate nella modellazione concettuale lungo tutte le fasi dello sviluppo software, dalla primissima fase di raccolta dei requisiti fino all'implementazione.
2. L'adozione di un approccio allo sviluppo del software detto "*requirement driven*" [29], dove si utilizzano concetti adatti alla modellazione dei requisiti (come quelli proposti in *goal oriented analysis* [9] e *NFR analysis* [10, 11, 9]) e li si portano anche in fasi successive quali quelle relative alla progettazione del sistema.
3. Costruzione di modelli concettuali seguendo un approccio trasformatore in cui si parte da un modello iniziale dell'ambiente e lo si raffina ed estende incrementalmente fino ad ottenere un artifact eseguibile.

La metodologia Tropos copre cinque fasi del processo di sviluppo: *early requirements*, *late requirements*, *architectural design*, *detailed design* e *implementation*. Tale metodologia è stata presentata in numerosi documenti con l'aiuto dei case study *eCulture System* [23, 24, 35] e *Meeting Scheduler System* [26, 40].

Il resto del capitolo è così strutturato: la sezione 2.1 illustra le fasi di cui si articola la metodologia, 2.2 i concetti chiave, 2.3 le attività di modellazione, infine in 2.4 confrontiamo Tropos con altre metodologie agent-oriented.

## 2.1 Fasi del processo di sviluppo

All'interno dell'ingegneria del software l'analisi dei requisiti rappresenta la fase iniziale di molte metodologie. In maniera del tutto analoga, l'obiettivo centrale dell'analisi dei requisiti in Tropos è quello di catturare un insieme di requisiti funzionali e non-funzionali per caratterizzare l'ambiente (*environment*) e il sistema futuro (*system-to-be*).

Fasi	Descrizione sintetica
<b>Early requirements</b>	Studio dell'ambiente o dell'organizzazione. L'output è un modello organizzativo che include gli attori rilevanti con associati i goal e le rispettive dipendenze.
<b>Late requirements</b>	Studio del system-to-be all'interno del suo ambiente operativo. Sono definiti i requisiti funzionali e non-funzionali.
<b>Architectural design</b>	Studio dell'architettura globale del sistema. Si compiono tre passi specifici: introduzione nuovi attori di sistema, identificazione capability e processo di agentificazione.
<b>Detailed design</b>	Studio di ciascun componente architetturale. Specifica di ogni singolo agente con relative capability, goal, belief e plan. L'architettura di implementazione è stata scelta (esempio, BDI).
<b>Implementation</b>	Implementazione del sistema elaborato nella fase precedente, utilizzando ad esempio delle piattaforme di sviluppo AOP (JACK).

**Tabella 2.1:** Fasi del processo di sviluppo di *Tropos*. La metodologia è basata su cinque passi (4 + 1). Accanto al nome della fase c'è una breve descrizione schematica.

Dalla tabella 2.1 è possibile notare come in Tropos questa fase viene suddivisa in due fasi distinte denominate rispettivamente **early requirements** e **late requirements** [23, 24, 6, 35, 34, 25].

Entrambe condividono gli stessi concetti e l'approccio metodologico, pertanto molte idee introdotte nella prima fase vengono riproposte anche nella seconda. Più precisamente, durante l'early requirements l'ingegnere identifica gli *stakeholder*<sup>1</sup> e li modella come attori sociali che dipendono fra loro per goal da soddisfare, piani da eseguire e risorse da fornire. Infatti, dall'identificazione di queste dipendenze è possibile capire perché alcune funzionalità sono preferibili ad altre ed avere un banco di prova per verificare che l'implementazione finale risulti allineata alle aspettative iniziali. Il risultato di questa fase è un particolare modello incentrato su attori sociali e loro dipendenze.

Nella seconda fase, **late requirements**, il modello concettuale prodotto precedentemente viene esteso per includere nuovi attori che rappresentano il sistema e dipendenze fra questi ultimi e quelli dell'ambiente. Tali dipendenze definiscono completamente i requisiti funzionali e non-funzionali.

Dopo aver catturato tutti i requisiti dell'ambiente e del sistema il passo successivo è rappresentato dalla fase di progettazione del sistema software. Anche in questo caso, si hanno due momenti con obiettivi distinti: **architectural design** e **detailed design**. Entrambe si focalizzano sulle specifiche del sistema in accordo con i requisiti prodotti dai due passi precedenti. **architectural design** definisce l'architettura globale del sistema in termini di *subsystem* interconnessi attraverso dati e flussi di controllo. I *subsystem* sono rappresentati nel modello con attori di sistema, mentre le interconnessioni con dipendenze. Questa fase si articola in tre passi:

1. Viene definita l'architettura del sistema software. Sono introdotti nel sistema dei nuovi attori risultanti dall'analisi eseguita nelle fasi precedenti:
  - i subgoal provenienti dall'analisi dei goal di sistema vengono delegati ai nuovi attori appena identificati;
  - introduzione di nuovi attori in funzione di particolari scelte architetture;
  - introduzione di nuovi attori che contribuiscono positivamente al soddisfacimento dei requisiti non-funzionali;
  - decomposizione degli attori in *subactor*. Il fine della decomposizione è espandere in dettaglio ciascun attore in riferimento ai propri goal e piani.

Il risultato finale di questo passo è l'estensione del modello, nel quale vengono aggiunti nuovi attori e le loro dipendenze.

2. Identificazione delle *capability*<sup>2</sup> necessarie agli attori per soddisfare i loro goal ed eseguire i piani. Le *capability* sono identificate analizzando il modello. In particolare ciascuna relazione di dipendenza può dar luogo a una o più *capability* attivate da eventi. Il risultato è

<sup>1</sup>Attori, o più in generale entità, che hanno interesse a partecipare in un sistema software si dicono anche *stakeholder*.

<sup>2</sup>La traduzione in italiano del termine *capability* è difficoltosa. Per questo motivo intendiamo usare indistintamente il termine *capability* oppure il suo sinonimo italiano "abilità".

una tabella in cui su un lato sono riportati gli attori e sull'altro le capability loro assegnate. Ancora non si stanno assegnando vincoli fisici ma solamente logici; non è sicuro che gli attori individuati e le rispettive capability siano quelle che poi verranno effettivamente implementate.

3. L'ultimo passo prevede l'assegnazione degli agenti e delle capability. Tale processo va sotto il nome di *agentificazione* poiché si definiscono definitivamente quali sono gli attori che verranno implementati nella fase successiva sotto forma di agenti run-time. In generale l'agentificazione non è unica e dipende dal progettista. Anche in questo caso si produce una tabella nella quale si elencano gli agenti e le capability assegnate attraverso la numerazione fatta al passo 2.

La fase successiva, **detailed design**, punta a specificare ciascun agente nel suo interno, le capability e le interazioni che lo coinvolgono. A questo punto, solitamente, la piattaforma di implementazione è stata scelta (ad esempio BDI) e ogni agente viene specificato tenendo conto della corrispondenza tra l'ontologia Tropos e i costrutti del linguaggio implementativo scelto.

L'ultima fase prevede l'**implementazione** di quanto ottenuto al passo precedente. Una piattaforma BDI classica è JACK [7, 14, 13]. La tabella 2.2 riassume la corrispondenza esistente tra i costrutti JACK e l'ontologia Tropos.

## 2.2 I concetti chiave

I modelli in Tropos sono acquisiti come istanze di un *meta-modello concettuale* (capitolo 3) composto di concetti e di relazione fra i concetti sotto elencati:

- **Actor** – modella un'entità che possiede obiettivi strategici e intenzionalità all'interno del sistema o dell'organizzazione. Un attore può rappresentare una persona fisica, un animale, una macchina, così come un componente software. Si distinguono tre specializzazioni dell'attore: *agent*, *role* e *position*. L'agente è caratterizzato da delle proprietà come *autonomia*, *reattività*, *proattività*, *abilità sociale* nel contesto in cui opera [44, 12, 28, 32, 1]. Ruolo è una caratterizzazione astratta del comportamento di un attore sociale all'interno di un contesto specifico del dominio applicativo, posizione rappresenta un insieme di ruoli tipicamente ricoperti da un singolo agente. Così un agente può occupare una posizione, mentre una posizione copre un ruolo. Notare che la nozione di attore in Tropos è una generalizzazione della definizione classica di agente software in *Artificial Intelligence (AI)*. Una discussione di questa questione si trova in [47].
- **Goal** – rappresenta l'interesse strategico dell'attore che partecipa ad un sistema o ad una organizzazione. Distinguiamo fra *hardgoal* e *softgoal* per il fatto che per i primi sappiamo dire con certezza e precisione quando sono soddisfatti, mentre per i secondi questo risulta

Costrutti	Descrizione
<b>Agent</b>	È usato per definire il comportamento di un agente intelligente. Questo include le capability che un agente possiede, i tipi di messaggi ed eventi ai quali risponde e i piani che usa per soddisfare un goal.
<b>Capability</b>	Include piani, eventi, belief e altre capability. Ad un agente possono essere assegnate più capability, una capability può essere assegnata a più agenti.
<b>Belief</b>	Attualmente, in Tropos, questo concetto è usato solo nella fase implementativa, ma stiamo considerando di spostarlo fino alla prima fase. Un database descrive un insieme di belief che un agente può avere.
<b>Event</b>	Gli eventi interni ed esterni specificati nel detailed design sono mappati negli eventi JACK. In JACK un evento descrive una condizione che lancia uno o più piani.
<b>Plan</b>	I piani contenuti all'interno della capability risultanti dal detailed design sono mappati nei piani JACK. Un piano in JACK è una sequenza di istruzioni che l'agente esegue per provare a soddisfare i goal e gestire gli eventi.

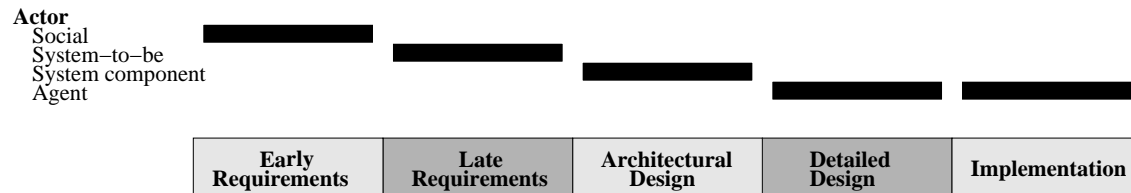
**Tabella 2.2:** Mapping Tropos - JACK.

più complicato. I softgoal sono utili per la modellazione delle qualità del software, come, ad esempio, la sicurezza, le prestazioni e l'affidabilità [11].

- **Plan** – rappresenta un opportuno insieme di azioni che permettono di soddisfare un goal.
- **Resource** – è un'entità fisica oppure un'informazione (insieme di dati).
- **Dependency** – una dipendenza fra due attori indica che un attore dipende dall'altro per soddisfare un goal, eseguire un piano oppure fornire una risorsa. L'attore che delega un proprio obiettivo fra quelli appena citati si chiama *depender*, mentre colui che riceve la dipendenza è il *dependee*. L'oggetto della dipendenza è detto *dependum*. Il dependee ha libertà di scegliere come risolvere il dependum; per questo fatto il depender diventa vulnerabile. Se il dependee fallisce nel consegnare il dependum il depender potrebbe risentirne negativamente e pregiudicare le sue abilità nel soddisfare i propri goal.
- **Capability** – rappresenta l'abilità di un attore nel definire, scegliere ed eseguire uno o più piani per soddisfare un goal. La capability si attiva in presenza di un evento.
- **Belief** – rappresenta la conoscenza di un attore circa lo stato del mondo.

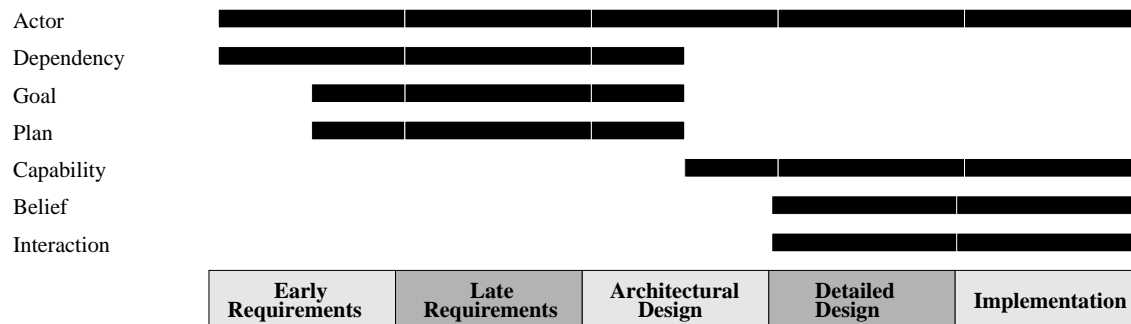
Notare come le nozioni di belief, goal (o desire) e plan (o intention) sono i concetti chiave anche nel paradigma BDI presentato in [38, 1].

### Focus su



**Figura 2.1:** Actor modeling attraverso le cinque fasi del processo di sviluppo.

### Attività di modellazione



**Figura 2.2:** Attività di modellazione attraverso le cinque fasi del processo di sviluppo.

## 2.3 Attività di modellazione

La costruzione di un modello Tropos dall'early requirements sino all'implementazione coinvolge molte attività che contribuiscono al processo di acquisizione, raffinamento ed evoluzione del modello stesso. Esse sono:

- *Actor modeling* – consiste nell'identificare e analizzare sia gli attori dell'ambiente sia quelli del sistema (compresi gli agenti di sistema). La figura 2.1 mostra quali tipologie di attori o agenti sono approfonditi in ciascuna fase della metodologia. A livello di early requirements l'attenzione viene posta sugli stakeholder del dominio, ovvero attori propri dell'ambiente che possiedono determinati goal da raggiungere. Durante i late requirements l'analisi è condotta su *system actor*, cioè su attori del sistema software che si dovrà realizzare. L'architectural design focalizza sulla struttura dell'attore di sistema specifico del system-to-be in termini di subactor. L'ultima fase di progettazione volge a definire le specifiche interne di ciascun agente con tutte le nozioni necessarie per passare poi all'implementazione finale del sistema software.

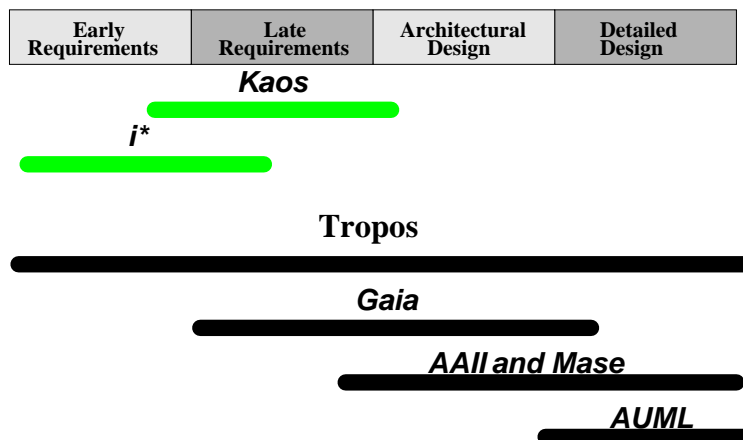
- *Dependency modeling* – consiste nell’identificazione degli attori che dipendono da altri per goal da soddisfare, piani da eseguire e risorse da fornire. Nella prima fase della metodologia, early requirements, si modellano le dipendenze fra gli attori sociali dell’organizzazione. Sono aggiunte nuove dipendenze in conseguenza dell’attività di *Goal modeling* descritta sotto. Operativamente, nei late requirements si compie lo stesso lavoro fatto negli early requirements tranne che ora si discutono gli attori del sistema futuro (*system-to-be*). Nella fase di architectural design i dati e flussi di controllo vengono modellati con delle dipendenze fornendo le basi per l’identificazione delle capability e la successiva agentificazione.
- *Goal modeling* – è l’attività di analisi dei goal dal punto di vista di un attore usando tre tecniche: means-ends analysis, contribution e AND-OR decomposition. In particolare, means-ends analysis punta a identificare goal, piani e risorse che forniscono i mezzi necessari per soddisfare il goal. Contribution permette di trovare goal che contribuiscono positivamente o negativamente al soddisfacimento del goal radice (*root*). AND-OR decomposition consente di costruire grafi AND-OR (tecnica tipica della comunità AI) di subgoal per capire come sia possibile soddisfare la radice. L’attività si esaurisce nell’architectural design quando si decompongono gli attori di sistema in subactor.
- *Plan modeling* – può essere considerata un’attività simile alla quella condotta sul goal. Comprende le medesime tecniche di analisi condotte però sul piano anziché sul goal.
- *Capability modeling* – inizia verso la fine dell’Architectural design quando i subactor di sistemi sono stati specificati in termini dei loro goal e dipendenze con gli altri attori. Ciascun subactor introdotto in questa fase potrebbe essere capace di definire, scegliere ed eseguire un piano per soddisfare i suoi goal. Durante il detailed design ciascun agente è ulteriormente specificato e codificato nell’ultima fase che prevede l’implementazione degli agenti e delle capability.
- *Belief modeling* – consiste nel rappresentare le credenze (*belief*) di ogni agente in riferimento allo stato del mondo che lo circonda. I belief guidano la scelta dei piani da eseguire e degli agenti stessi da implementare. È un’attività tipica della fase di detailed design quando ciascun agente viene specificato a micro livello.
- *Interaction modeling* – consiste nella specifica del protocollo comunicativo.

La figura 2.2 illustra qualitativamente il ruolo delle attività di modellazione all’interno delle fasi del processo di sviluppo, assieme ai concetti intenzionali. La modellazione della nozione di attore avviene lungo tutto il processo poiché si tratta di un concetto portante. Le altre attività sono specifiche all’interno di qualche fase; per esempio, le attività riguardanti le capability e le credenze sono rilevanti solo nella fase di detailed design e di implementazione.

## 2.4 Tropos a confronto con altre metodologie

L'obiettivo di questa sezione è confrontare Tropos con le altre metodologie e linguaggi di modellazione che seguono un'approccio agent-oriented (AO) come  $i^*$ , KAOS, GAIA, AAIL, MaSE e AUML. Una analisi più approfondita di ognuna di queste si trova in [40].

La figura 2.3 mette a confronto le seguenti metodologie:



**Figura 2.3:** Tropos a confronto con altre metodologie AO per lo sviluppo di sistemi software.

Come appare chiaro dalla figura 2.3, nessun'altra metodologia che abbiamo preso in esame copre tutte e quattro le aree di sviluppo del processo produttivo relative all'analisi e progettazione del software.

Per le fasi di early e late requirements Tropos si avvantaggia del lavoro fatto dalle comunità RE, e in particolare dal linguaggio  $i^*$  di Eric Yu, il quale dispone di una struttura versatile ed intuitiva per modellare le dipendenze strategiche fra attori distribuiti. È interessante notare che gran parte della metodologia Tropos può essere combinata con approcci che non sono agent-oriented (ad esempio, object-oriented o imperativo). Per esempio, si potrebbe usare Tropos per la fase degli early requirements e poi usare UML per le successive [25]. Particolare attenzione merita anche KAOS, una metodologia rivolta principalmente alla fase in cui si analizzano gli attori di sistema.

I modelli che si costruiscono con Gaia trattano la parte di analisi e raccolta dei requisiti (system-to-be) e la parte architetturale globale. AAIL è invece rivolta alla sola progettazione così come AUML che eredita l'approccio di UML ottimo per la definizione di diagrammi progettuali ma lontano da ciò che noi intendiamo quando parliamo di analisi e specifica dei requisiti di un sistema software. Tuttavia, per la fase di detailed design la nostra scelta è rivolta alle tecniche e strumenti di AUML che si sono riconosciuti essere assai validi.



## Capitolo 3

# Il linguaggio di modellazione

L'attività di modellazione concettuale tramite un linguaggio di specifica formale, semiformale o informale accompagna l'analista lungo tutte le fasi che conducono alla consegna di un software di qualità, dall'analisi dei requisiti alla specifica dei componenti software. In Tropos si inizia dagli *early requirements* ispirandosi a  $i^*$  e KAOS, per proseguire con i *late requirements*, *architectural design* e *detailed design* dove si sfruttano alcune tecniche tipiche di AUML. Infine, si passa all'implementazione, ad esempio utilizzando una piattaforma ad agenti, quale JACK.

Il linguaggio di modellazione visuale utilizzato in Tropos è un linguaggio di specifica semiformale per il quale sono stati definiti:

- **un'ontologia** – un insieme di concetti per la modellazione. L'ontologia definisce informalmente il significato di tali concetti, ma non fornisce uno strumento formale che permetta di controllare la semantica dei modelli costruiti. Gli elementi che fanno parte dell'ontologia Tropos sono i seguenti: *actor*, *goal*, *plan*, *resource*, *dependency*, *capability* e *belief*. Essi sono stati descritti in sezione 2.2,
- **meta-modello** – definisce la sintassi del linguaggio. Il meta-modello fornisce uno strumento per controllare che i modelli siano sintatticamente corretti. È stato specificato con un insieme di diagrammi di classe UML. La descrizione è data in dettaglio in sezione 3.1,
- **notazione grafica e diagrammi** – stabilisce come raffigurare graficamente gli elementi dei modelli. Notazione grafica e diagrammi sono descritti in dettaglio in sezione 3.3,
- **insieme di regole d'uso** – che guidano la costruzione di un modello concettuale nelle varie fasi del processo. Se ne dà un esempio nell'illustrare l'uso dei diagrammi nelle varie fasi del processo di sviluppo in sezione 3.3 e nel caso di studio finale nel capitolo 4.

### 3.1 Il meta-modello

Nella definizione del meta-modello si è tenuto conto dei seguenti requisiti:

1. avere una sintassi ben formata,
2. estendere in futuro il meta-modello senza doverlo stravolgere ogni qual volta sia necessario inserire nuovi elementi,
3. essere semplice ma espressivo,
4. non mescolare elementi pertinenti a livelli d'astrazione diversi. Viceversa si rischierebbe di perdere in precisione e in leggibilità del modello,
5. fornire un insieme di regole per la costruzione di modelli corretti; supportare strategie differenti per l'acquisizione del modello,
6. fornire tutti gli elementi concettuali necessari per specificare completamente i modelli degli early requirements così come quelli del detailed design,
7. uniformare il nostro linguaggio a molti altri che utilizzano un'architettura fondata su quattro livelli; in particolare UML (e dunque anche AUML), KAOS e, in parte,  $t^*$ ,
8. il meta-metamodello è strategico nel processo di integrazione fra il nostro lavoro, che punta più su aspetti informali e sulla definizione e correttezza della sintassi, e quello prodotto dal gruppo che sta lavorando su un sottoinsieme del linguaggio a semantica univoca, *Formal Tropos* [37, 22].

Il meta-modello del linguaggio di modellazione è definito da una architettura composta di quattro livelli: il *meta-metamodello*, il *meta-modello*, il *livello del dominio* e il *livello oggetto*. La tabella 3.1 riassume i quattro livelli fornendo una breve descrizione correlata da un semplice esempio. Tali livelli sono stati specificati usando i diagrammi delle classi UML versione 1.3.

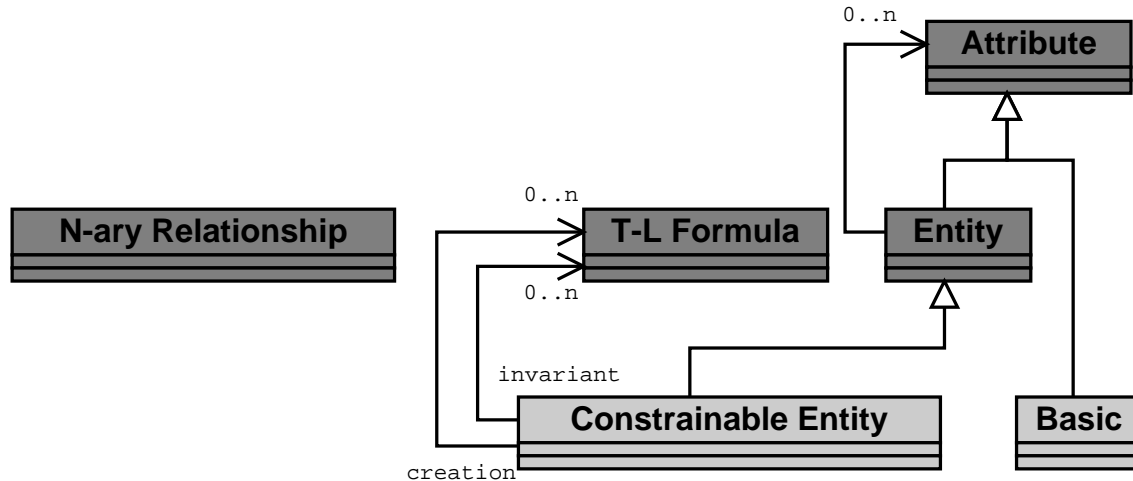
<b>Livelli</b>	<b>Descrizione</b>	<b>Esempi</b>
<b>meta-metamodello</b>	Definisce il linguaggio per la specifica del meta-modello.	Attribute, Entity.
<b>meta-modello</b>	Un'istanza del meta-metamodello. È il cuore del linguaggio.	Actor, Goal, Dependency.
<b>dominio</b>	Un'istanza del meta-modello. Definisce un modello del dominio applicativo.	PAT, Citizen: istanze di Actor.
<b>oggetto</b>	Un'istanza di un'entità definita al livello del dominio.	Mary: istanza Citizen.

**Tabella 3.1:** Architettura su quattro livelli. Accanto alla descrizione di ogni singolo livello sono riportati alcuni esempi di nozioni modellabili (colonna di destra).

### 3.1.1 Il livello meta-metamodello

*Il meta-metamodello di Tropos è la struttura più astratta dell'intero linguaggio; lo scopo primario è definire gli elementi minimi del linguaggio necessari per la specifica del meta-modello. Costituisce le fondamenta sulle quali costruire il meta-modello che ne è una particolare istanza.*

Con il grigio (scuro per le classi base, chiaro per quelle ausiliarie) sono mostrati in figura 3.1 gli elementi caratteristici del meta-metamodello.



**Figura 3.1:** Diagramma delle classi UML che specifica interamente il *meta-metamodello* di Tropos.

La classe *Entity*<sup>1</sup> è usata in Tropos per rappresentare gli elementi *intenzionali* e *non-intenzionali* che esistono nell'ambiente o nell'organizzazione (environment o organizational setting). Ogni istanza della classe *Entity*<sup>2</sup> è definita dal nome e da un insieme di attributi modellati con la classe *Attribute*. Quest'ultima definisce degli attributi che possono essere *Basic*, e dunque ad esempio interi, stringhe, booleani, oppure una struttura più complessa come un'entità. Un'istanza di *Entity* possiede 0..n istanze della classe *Attribute* che a sua volta può essere specializzata in *Basic* oppure in *Entity* stessa.

*Entity* generalizza (è la super classe di) *Constrainable Entity*: ogni sua istanza è definita da un insieme di attributi e proprietà (oltreché da un nome). I primi sono ereditati dalla super classe *Entity*, mentre le seconde sono modellate con la classe UML *T-L Formula* che sta per *Temporal Logic Formula*. *Constrainable Entity* può avere 0..n istanze della classe *T-L Formula*. Le proprietà fissano delle regole che l'istanza della classe *Constrainable Entity* deve soddisfare al momento della sua creazione e/o durante il suo ciclo di vita. Infatti, la classe *T-L Formula* ricopre due

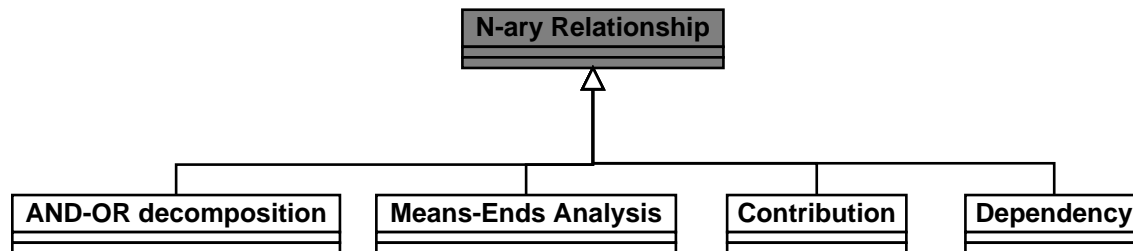
<sup>1</sup>Questo ed altri termini si possono trovare nel glossario in appendice A.1.

<sup>2</sup>Per evitare di appesantire la descrizione del linguaggio useremo "un'istanza della classe del meta-modello" anziché "un'istanza della meta-classe del meta-modello".

diversi ruoli, *creation* oppure *invariant*. La *creation*, come si può facilmente intuire, descrive una condizione che deve essere ottenuta all'atto della creazione dell'istanza della classe. *Invariant* esprime una condizione di invarianza dell'entità, un qualcosa che non muta lungo tutto il suo ciclo di vita.

T-L Formula e Attribute sono delle formule scritte in logica temporale di primo ordine che si usano di solito a livello oggetto per descrivere particolari caratteristiche delle entità del dominio applicativo che si sta studiando. Il linguaggio che specifica tali attributi e proprietà è detto *Formal Tropos*<sup>3</sup>.

L'ultimo elemento che appare all'interno del meta-metamodello è *N-ary Relationship*; raggruppa tutte le relazioni possibili che si possono definire sulle istanze di Entity a livello di dominio. La figura 3.2 mostra quali sono le relazioni che il linguaggio ammette (*N-ary Relationship* compare in 3.2 e anche nel diagramma di figura 3.1). Tali relazioni sono state colorate di bianco poiché esse non fanno parte del meta-metamodello bensì del meta-modello. Tuttavia, è utile introdurre già da ora perché permettono di capire in quale maniera si passa da un livello a quello successivo.



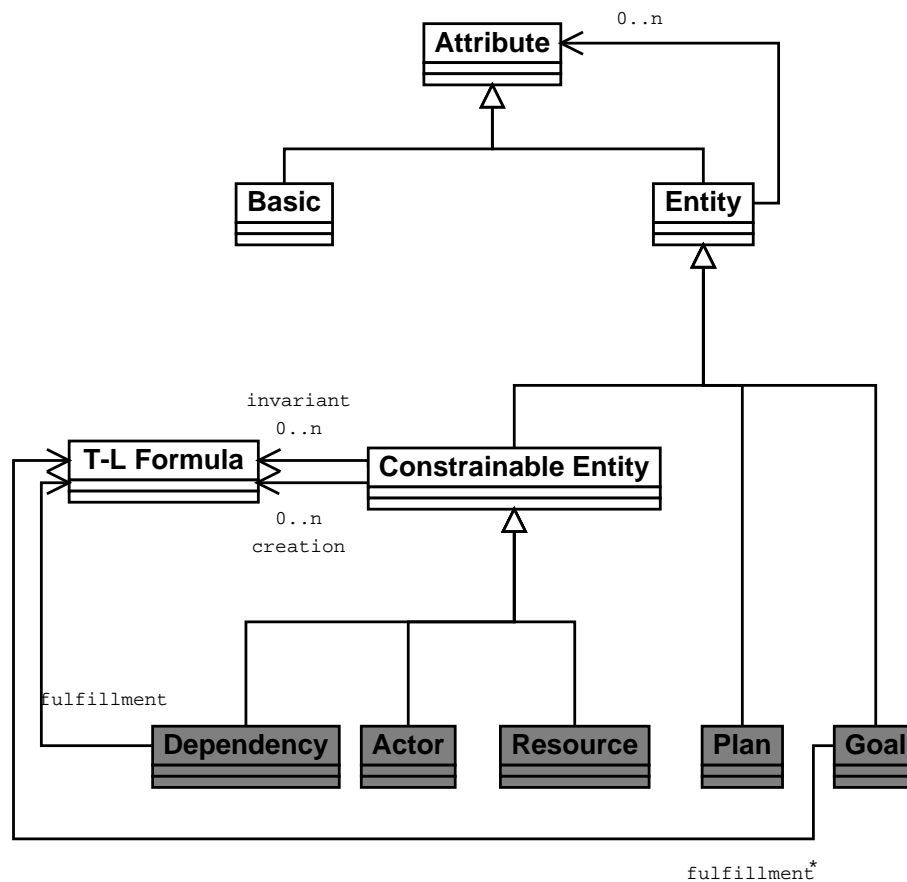
**Figura 3.2:** Diagramma delle classi UML che visualizza le relazioni che si possono istanziare a livello di dominio.

### 3.1.2 Il livello meta-modello

*Il meta-modello di Tropos è un livello logico (e non fisico o implementativo) che si focalizza sui concetti ontologici essenziali e indipendenti dal dominio applicativo.*

Il meta-modello è un'istanza del meta-metamodello e al suo interno troviamo le meta-classi che modellano gli elementi essenziali del linguaggio, ovvero *Actor*, *Goal Plan*, *Resource*, *Dependency*. La figura 3.3 mostra alcuni elementi del meta-modello (in grigio scuro) che estendono quelli del meta-metamodello (in bianco). Gli elementi intenzionali come *Actor* e *Dependency* assieme a *Resource* sono specializzazioni della classe *Constrainable Entity* che abbiamo incontrato al livello precedente; altri, come *Goal* e *Plan*, sono ottenuti specializzando la super classe *Entity*.

<sup>3</sup>La discussione del linguaggio *Formal Tropos* non è pertinente con gli obiettivi di questa tesi, dunque non entreremo nel merito. In appendice B.1 è riportata la grammatica pubblicata in [22]. Gli articoli di riferimento sono [37, 22].



**Figura 3.3:** Diagramma delle classi UML che mostra la relazione esistente fra alcuni elementi del *meta-metamodello* e del *meta-modello* di Tropos. L'asterisco sull'arco etichettato *fulfillment* impone una restrizione al concetto Goal: solo l'Hardgoal può avere delle *T-L Formula*.

Come già spiegato la presenza di *Constraintable Entity* è dovuta all'integrazione con il linguaggio *Formal Tropos* secondo cui ci sono entità (Goal e Plan) che possiedono solo attributi ed entità che invece possiedono anche proprietà di tipo invariant e creation (Dependency, Actor e Resource). In aggiunta a tutto ciò, sia la Dependency, sia Goal ricoprono il ruolo di *fulfillment* rispetto a T-L Formula (per la classe Goal questa associazione è valida solamente se si parla di Hardgoal e non di Softgoal) cosicché è possibile descrivere formalmente quando un goal è soddisfatto o meno. Per poter affermare che un goal o una dipendenza di tipo hardgoal è soddisfatta, abbiamo bisogno di esprimere delle condizioni tramite predicati precisi, corretti e ben formati.

Il meta-modello di Tropos estende la struttura offerta da \* quali attore (agent, role e position), goal e dipendenze tra attori come concetti primitivi per modellare un applicazione durante tutte le fasi del processo di sviluppo, non solo nell'early requirements. Il meta-modello è stato sviluppato intorno a questi concetti fondamentali e specificati attraverso diagrammi delle classi UML; sia i concetti, sia le relazioni fra gli stessi sono rappresentate attraverso delle classi.

Sono molte le definizioni che si trovano nella letteratura riguardanti gli elementi appena citati e diversi sono pure i significati attribuiti; noi abbiamo preso spunto dalla comunità dell'*Intelligenza Artificiale* dove si usano pesantemente concetti come agent, belief, desire, intention. Per chi desidera approfondire consigliamo le letture [5, 41, 38].

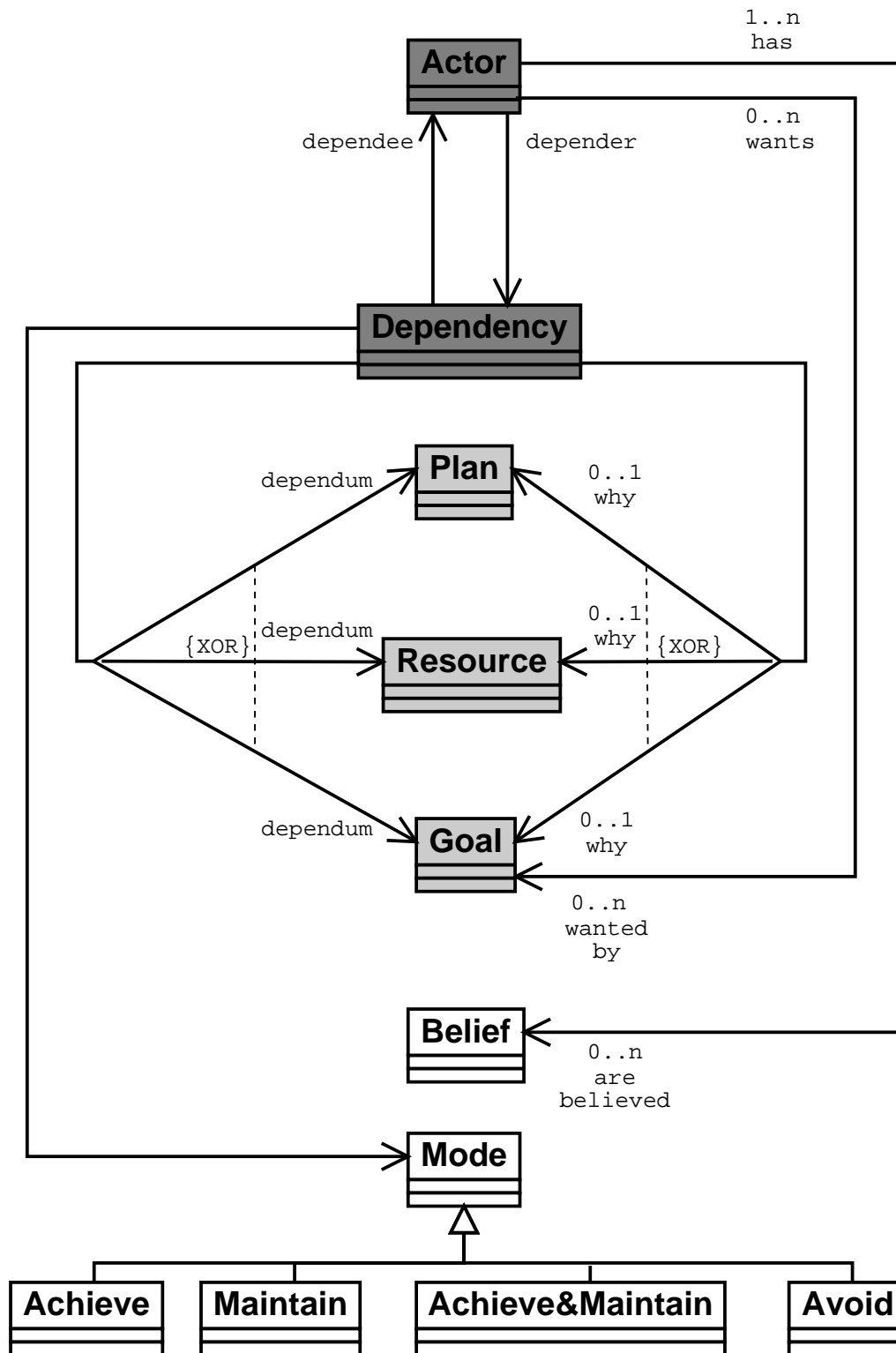
### **Il concetto di attore**

*L'attore è un'entità che possiede degli obiettivi strategici all'interno del sistema oppure dell'organizzazione che ospita il sistema software stesso; inoltre è sorgente di intenzionalità.*

L'attore possiede degli obiettivi strategici, cioè ha delle motivazioni precise che lo spingono a partecipare al sistema o all'ambiente. Nella figura 3.4, l'attore è rappresentato per mezzo di una classe UML chiamata *Actor*; le associazioni mostrano le relazioni significative che coinvolgono la classe suddetta.

L'istanza della classe *Actor* può avere 0..\* istanze della classe *Goal* la quale modella il concetto di obiettivo perseguito (*wanted by*) da un attore. Nel modello Tropos ogni attore che partecipa all'environment o al system-to-be possiede almeno un goal (anche sotto forma di dipendenza). Ciò risulta in perfetta sintonia con la filosofia secondo cui l'attore, in quanto sorgente di intenzionalità, autonomo e capace di decisioni, partecipa ad un sistema o ad una organizzazione poiché ha degli obiettivi che qualificano la sua presenza.

L'istanza della classe *Goal* è associata a 0..\* istanze della classe *Actor*. Tropos, e lo discuteremo meglio quando parleremo del diagramma degli attori, non ammette l'esistenza di un goal che non sia attribuito a qualche attore (implicitamente stiamo affermando che non ci possono essere *system goal*). Detto questo, al lettore potrebbe sorgere il dubbio legittimo che la cardinalità corretta sia 1..\* anziché 0..\*. Nell'ipotesi 1..\* non ci potrebbe essere nel modello Tropos un goal senza almeno un attore che se ne assuma la responsabilità; parimenti, almeno in via di principio, con 0..\* è possibile introdurre nel modello un goal senza che necessariamente sia attribuito ad un attore. Cerchiamo di spiegare il motivo di questa scelta attraverso un semplice esempio. Consideriamo la situazione particolare in cui il goal, nella sua accezione più generica, sia il frutto di una decomposizione in OR e che dunque non sia di proprietà di qualche attore ma rappresenti una possibile alternativa al soddisfacimento del goal radice (goal sul quale si compie la GA). Supponiamo che l'attore *Citizen* desideri decomporre il goal *get cultural information*. Usando i mezzi tipici della *Goal Analysis* [9, 15] il processo di raffinamento produce un grafo in OR di subgoal, rispettivamente *visit cultural institutions* e *visit cultural web system*. Potrebbe accadere che il primo venga assegnato all'attore stesso, mentre per il secondo non ci sia volontà di attribuzione. Essendo una decomposizione in OR, uno solo dei due sarà percorso mentre l'altro verrà presumibilmente abbandonato e pertanto non assegnato. Con l'ausilio di questo



**Figura 3.4:** Diagramma delle classi UML che specifica il meta-modello di Tropos riferito al concetto ontologico di *Actor*.

semplice e ricorrente esempio si spiega perché è necessario fissare la molteplicità 0..\* fra la classe Goal e quella Actor.

Un attore può avere 0..\* istanze della classe *Belief*; questa nozione, che è tipica del paradigma BDI (belief-desire-intention), potrà essere assai utile quando, in fase di *detailed design*, si sceglierà uno specifico paradigma per rappresentare gli stati mentali dell'attore. Fino a quel momento però, nulla è assunto per quanto riguarda l'architettura che modella gli stati interni. È interessante notare la molteplicità della relazione inversa: la classe *Belief* è associata ad 1..\* attori. Ciò significa che se esiste, il belief deve essere assegnato ad almeno un attore.

**Dependency** È una relazione a quattro campi tra: due istanze della classe *Actor*, una fra *Goal*, *Plan* e *Resource* e una opzionale ancora tra *Goal*, *Plan* e *Resource*.

L'attore che delega un qualche tipo di dipendenza ricopre il ruolo di *depender*, mentre quello che riceve l'incarico è il *dependee*. Per specificare il tipo di dipendenza (*dependum*) si sceglie uno fra Goal, Plan e Resource adottando una tipica notazione UML che permette di porre in OR esclusivo tutte le classi raccordate dalla linea verticale tratteggiata con a fianco l'etichetta {XOR} (si veda figura 3.4). L'ultima associazione (*why*), che sappiamo essere opzionale, è ottenuta ancora scegliendo tra Goal, Plan e Resource; la notazione è la stessa usata precedentemente, solo che questa volta la molteplicità è 0..1 sottolineando che si tratta di un campo non obbligatorio. Dunque, una dipendenza fra due attori risulta essere una quaterna  $\langle \textit{depender}, \textit{dependum}, \textit{dependee}, \textit{why}^* \rangle$  (l'asterisco indica che è un campo opzionale). Mettiamo in evidenza il fatto che la relazione è da ritenersi ben formata nel momento in cui si hanno due istanze della classe Actor e una tra Goal, Plan e Resource che ricopre il ruolo di dependum; istanziare la quarta associazione è del tutto arbitrario e a discrezione dell'analista. In quest'ultimo caso la conseguenza immediata è ridurre la relazione quaternaria a ternaria.

Mentre il significato attribuito ai ruoli di *depender*, *dependee* e *dependum* è abbastanza chiaro (ricordiamo che servono per modellare le dipendenze strategiche tra gli attori), la presenza del *why* può sembrare fuorviante o addirittura inutile. Così non è. Riprendiamo l'esempio dell'*eCulture System* tornando alla decomposizione del goal `get cultural information` da parte dell'attore *Citizen*. Nel compiere l'analisi l'attore ha prodotto, fra l'altro, il piano `visit eCulture System` (figura 3.13). La sua decomposizione ha individuato altri due piani, `access internet` e `use eCulture System`. Successivamente sono stabilite tre nuove dipendenze con la PAT, `internet infrastructure available` scaturita da `access internet`, `usable eCulture System` e `eCulture System available` da `use eCulture system`. Il ruolo *why* è ricoperto dai due piani che danno luogo a queste nuove dipendenze: dunque, il quarto campo della relazione di dipendenza serve a tenere traccia delle motivazioni che spingono l'attore a consegnare una sua responsabilità ad un altro. È sostanzialmente un documento storico che permette di risalire a posteriori al motivo che ha scaturito la dipendenza. Quindi, nel caso del plan `access internet`



che dà luogo al *dependum internet infrastructure available*, la struttura completa della dipendenza è  $\langle \textit{Citizen}, \textit{internet infrastructure available}, \textit{PAT}, \textit{access internet} \rangle$ .

Come appendice conclusiva, la figura 3.4 mostra l'associazione tra la classe *Dependency* e la classe *Mode*. In fase di analisi si dovrà procedere a specificare, mediante specializzazione di *Mode*, il modo di raggiungimento della dipendenza stabilita con un altro attore. Le possibilità a disposizione sono quattro:

- **achieve** – l'attore si prefigge di soddisfare il goal o la dipendenza una volta solamente, oppure,
- **maintain** – significa che l'obiettivo dovrebbe essere mantenuto nel tempo, oppure,
- **achieve&maintain** – è una combinazione dei precedenti, una volta raggiunto lo scopo, questo deve perdurare nel tempo, oppure,
- **avoid** – significa che l'obiettivo dovrebbe essere evitato.

Tropos dispone di tre specializzazioni della classe *Actor*:

- **agent** – nel senso classico inteso dalla comunità AI e in sintonia con quanto affermato nel capitolo 2: un attore con abilità nel contesto sociale in cui si trova ad operare, autonomo, reattivo e pro-attivo, oppure,
- **role** – è una caratterizzazione più fine del comportamento di un attore nell'ambito di un insieme di interazioni (dipendenze, eventi comunicativi, ecc.),
- **position** – rappresenta un insieme di ruoli, tipicamente ricoperti da un agente. Un agente può occupare una posizione, mentre una posizione ricopre un ruolo.

La letteratura corrente riporta parecchie sfumature a proposito del concetto di ruolo: ad esempio in [8, 33], il ruolo è inteso come un insieme di dipendenze e di protocolli comunicativi tra agenti, cioè rappresentano delle situazioni ricorrenti che si schematizzano in strutture concettuali per poterle in un futuro riusare. Tali strutture sono comunemente dette *patterns*<sup>4</sup>. Empiricamente si verifica che per certe classi di domini applicativi si possono adottare degli schemi simili per determinare una strategia di soluzione; il ruolo dei patterns è appunto quello di fornire delle soluzioni per alcune classi di problemi. Nel campo dell'ingegneria dei requisiti sono molto diffusi questi patterns poiché incarnano situazioni che un progettista si trova a dover affrontare continuamente: infatti, in un'approccio diffuso e consolidato da anni come l'object-oriented (OO) l'uso dei patterns è molto frequente. In AUML si parla di *agent-role* come un insieme di agenti che soddisfano alcune proprietà, comportamenti particolari e forniscono servizi comuni. In maniera

<sup>4</sup>All'URL <http://hillside.net/patterns/patterns.htm> è disponibile una guida completa dei patterns Object-Oriented.

analoga anche in [18] si introduce il ruolo come una rappresentazione astratta delle funzioni e servizi di un agente.

La relazione di dipendenza è stata introdotta da  $t^*$  in [46] e ripresa da Tropos; si dimostra utile per la rappresentazione delle dipendenze strategiche tra attori. Esistono, a riguardo, svariate teorie sulle dipendenze per i sistemi multi-agente. Una definizione classica tratta da [33], afferma che

*la teoria delle dipendenze è basata sull'idea che le interazioni fra gli attori provengano dalla incompatibilità tra gli obiettivi dei singoli attori e le sue risorse e capacità.*

Tipicamente, un attore non riesce a soddisfare tutti i suoi obiettivi e intenzioni; ci possono essere delle azioni, dei goal e dei comportamenti che in generale non sono raggiungibili senza il sostegno da parte di suoi simili, costringendolo ad interagire e stabilire appunto delle dipendenze di qualche genere. Tropos supporta questa teoria modellando le dipendenze strategiche fra attori mediante la relazione *Dependency*.

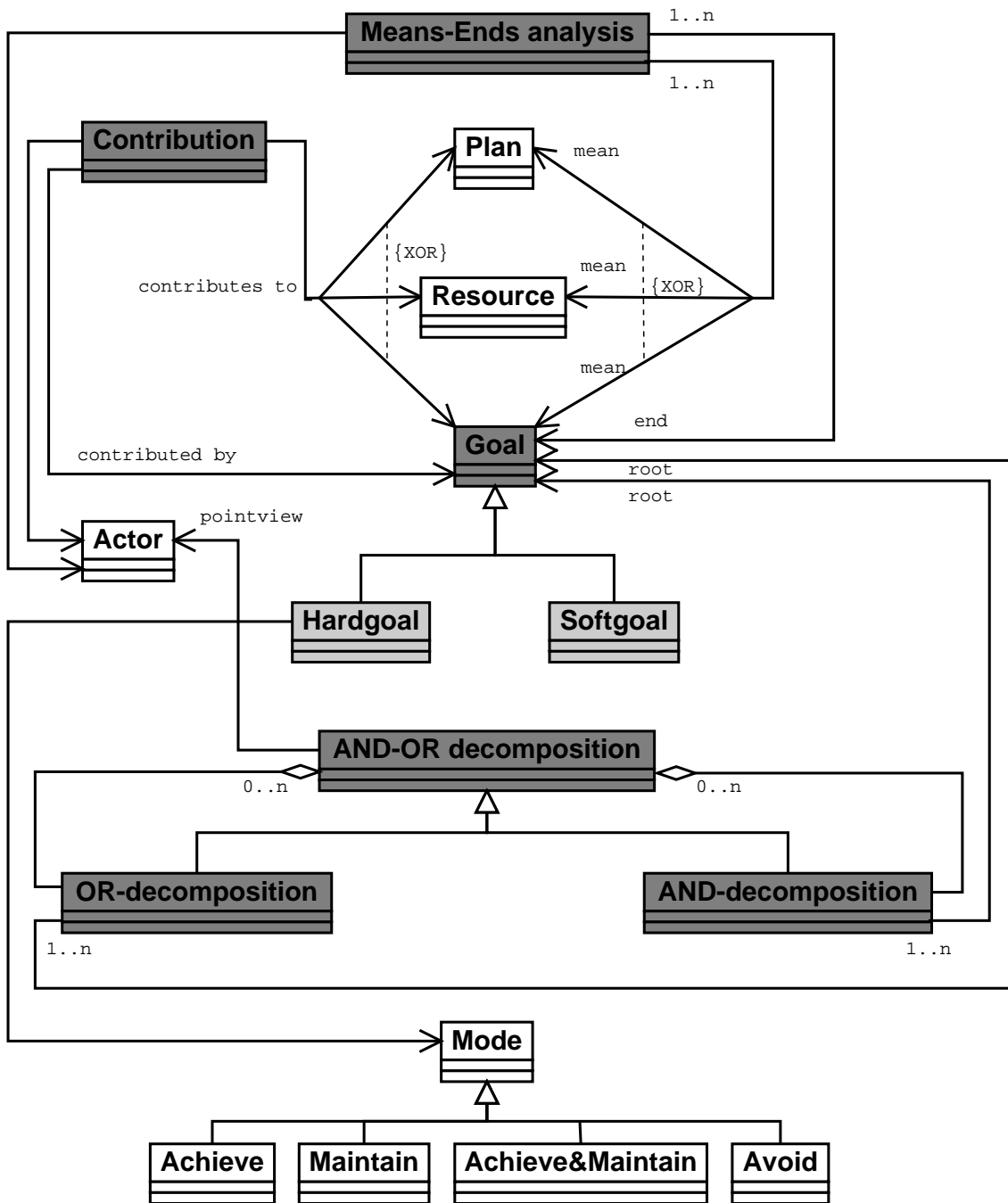
I tipi di dipendenza sono abbastanza intuitivi: può essere un goal generico, un piano da eseguire oppure una risorsa. Una *resource* rappresenta un'entità fisica o un contenitore di informazioni (ad esempio, il modello UNICO della dichiarazione dei redditi).

## **Il concetto di goal**

*Il goal modella gli obiettivi che un attore vuole raggiungere (il vero motivo per cui un attore partecipa ad un sistema).*

Il diagramma delle classi di figura 3.5 rappresenta il concetto di goal generico in Tropos.

La classe *Goal* può essere specializzata in *Hardgoal*, oppure in *Softgoal*. *Hardgoal* rappresenta un obiettivo che l'attore può potenzialmente e pienamente soddisfare eseguendo uno o più piani, sostanzialmente si può definire un criterio in base al quale stabilire se esso è stato raggiunto o meno (magari avvalendosi anche delle T-L Formula che esprimono le condizioni di *fulfillment* per un *hardgoal* attraverso predicati precisi e ben formati). Cattura quelli che in ingegneria dei requisiti si chiamano requisiti funzionali. Diversamente, un *softgoal* è un goal del quale non si riesce esattamente e con precisione ad affermare che è stato pienamente soddisfatto: analogamente a quanto detto per gli *hardgoal*, i *softgoal* modellano i requisiti non-funzionali in ingegneria del software. Tali requisiti, spesso chiamati di qualità, non possono essere definiti precisamente e in modo oggettivo (ad esempio *user friendly* riferito ad un'interfaccia grafica). Per determinare un criterio di soddisfacimento sono necessari sforzi aggiuntivi da parte dell'analista e del cliente. È auspicabile che vengano condotte ulteriori interviste con gli utenti del sistema software così da consentire un approfondimento delle caratteristiche di qualità che il prodotto finale dovrà avere. Questa pratica produrrà presumibilmente nuovi requisiti che verranno concretizzati in precise scelte progettuali e tecnologiche.



**Figura 3.5:** Diagramma delle classi UML che specifica la porzione del meta-modello di Tropos riferito al concetto ontologico di *Goal*.

I goal possono essere analizzati, dal punto di vista di un attore, mediante i processi di *Means-Ends analysis*, *Contribution* e *AND-OR decomposition*.

**Means-Ends analysis** È una relazione ternaria definita tra: un'istanza della classe *Actor*, il quale fornisce il suo punto di vista, un *Goal* che funge da obiettivo da raggiungere (*end*), motivo per cui si istanzia la *Means-Ends analysis*, e uno tra *Goal*, *Plan* e *Resource* che ricopre il ruolo di *mean*.

Se analizziamo la figura 3.5, la classe *Goal* (notare che essa ricopre il ruolo di “end”, cioè il fine da raggiungere) può avere 1..\* istanze della classe *Means-Ends analysis*. Questo permette all'analista di condurre più analisi sul medesimo goal per ottenere più visione dello stesso obiettivo. La *Means-Ends analysis* è condotta dal punto di vista di un solo attore e su un solo goal (molteplicità uguale a 1). L'associazione mancante per completare la relazione ternaria proviene dall'istanza di una delle entità poste in OR esclusivo. La cardinalità fra il ramo etichettato con {*XOR*} e la classe *Means-Ends analysis* è 1..\*; questo consente, ad esempio, allo stesso piano di essere coinvolto in altre analisi condotte su altri goal e quindi partecipare a più *means-ends analysis*.

*Means-Ends analysis* è un particolare tipo di analisi che permettere di capire *cosa* sia necessario fare per soddisfare un goal (*end*) in termini di piani, risorse, *hardgoal* e *softgoal* (*mean*).

**Contribution** La *Contribution* è una relazione ternaria tra: un'istanza della classe *Actor*, che fornisce il punto di vista di chi conduce l'analisi (*pointview*), un'istanza della classe *Goal* che riceve il contributo (*contributed by*) e una fra *Goal*, *Plan* e *Resource* (*contributes to*). L'idea è di trovare quei goal che contribuiscono positivamente o negativamente al raggiungimento del goal che si sta esplorando; da notare che essa è un caso particolare della *Means-Ends analysis*, dove il mezzo è solamente il goal.

La *Contribution* è adatta quando non si hanno informazioni sufficienti per poter individuare una strategie certa per soddisfare un goal; supponiamo il caso in cui la decomposizione del goal  $g_1$  porti ad individuare un goal  $g_2$ .  $g_2$  però non fornisce garanzie sul pieno e completo soddisfacimento di  $g_1$  e per questo motivo è probabilmente opportuno utilizzare lo strumento dei contributi.

Esiste la possibilità di misurare con un'opportuna metrica il peso dei contributi. La scala di misurazione è composta di cinque livelli, ++, +, -, --. In particolare, se il goal  $g_1$  contribuisce positivamente al goal  $g_2$  con la metrica ++, allora equivale a dire che  $g_2$  è automaticamente soddisfatto nell'istante in cui lo è  $g_1$ ; analogamente, se il piano  $p$  contribuisce positivamente al goal  $g$  con ++, allora  $p$  esegue  $g$ . Se il contributo fornito al goal  $g_2$  dal piano  $p$ , dal goal  $g_1$  o dalla risorsa  $r$  è + allora significa che questi contribuiscono solo parzialmente al soddisfacimento di  $g_2$ . Viceversa, se la metrica è -- oppure - abbiamo delle situazioni duali rispetto alle precedenti.

Il caso di contributi tra *softgoal* è leggermente diverso; in questo caso l'idea è valutare i re-

quisiti non-funzionali per tentare di capire quali situazioni sono favorite dalla presenza di contributi positivi, e quali invece vengono tralasciate poiché i requisiti di qualità forniscono metrica negativa. Una trattazione più ampia riguardo i requisiti funzionali e non-funzionali si trova in [9, 15, 2, 11, 10, 40].

**AND-OR decomposition** Nello spirito tipico dell'ingegneria del software che esorta alla decomposizione del problema in sotto problemi [27, 36], Tropos definisce la AND-OR decomposition come relazione ternaria che associa il punto di vista dell'attore, il *goal radice* e un insieme di istanze di *AND decomposition* oppure *OR decomposition*. Per goal radice intendiamo quel goal che vogliamo esplorare con le tecniche usuali della goal analysis.

La classe *Goal* ricopre il ruolo di *root* con entrambi le specializzazioni della *AND-OR decomposition* (figura 3.5). Un'istanza di *Goal* può avere 1..\* istanze della classe *AND-decomposition* (*OR-decomposition*). Analogamente a quanto fatto per le dipendenze, anche per la classe *Hard-goal* (e non per il softgoal) è previsto l'attributo *Mode* che sarà ulteriormente specializzato in **achieve, maintain, achieve&maintain e avoid**.

Le due classi *AND-decomposition* e *OR-decomposition* specializzano la *AND-OR decomposition* ereditando la relazione che fornisce il punto di vista dell'attore (*pointview*). La AND-OR decomposition, a sua volta, è un'aggregazione delle sue sotto classi; in particolare, un'istanza della classe AND-OR decomposition è composta di una sola istanza della AND-decomposition (*OR-decomposition*), mentre l'istanza di AND-decomposition (*OR-decomposition*) potrebbe possedere 0..\* di AND-OR decomposition.

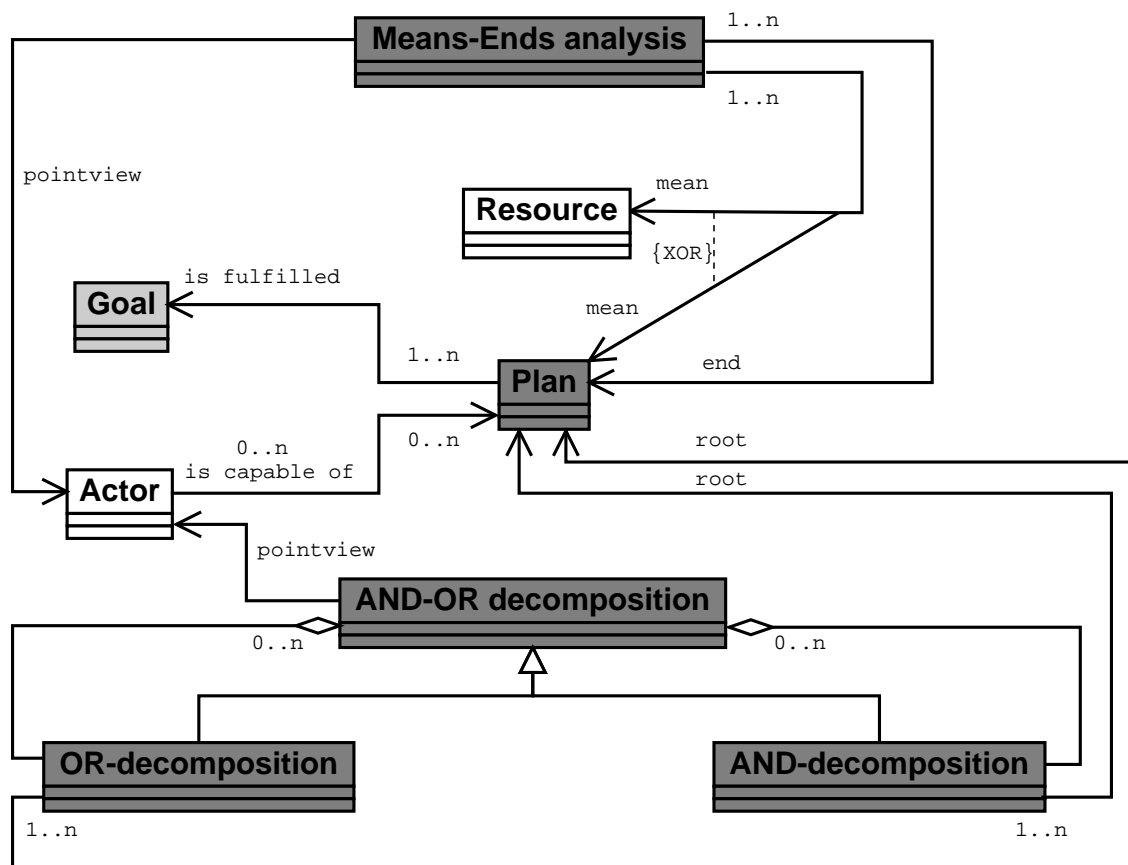
Applicare una AND-OR decomposition significa cercare dei goal che consentono di decomporre la root in tanti *subgoal* via via più semplici da risolvere. Se si tratta di una AND-decomposition allora per soddisfare il goal radice  $g$  è necessario che *tutti* i suoi subgoal siano soddisfatti, se invece questi sono posti in OR-decomposition il goal  $g$  è soddisfatto quando almeno *uno* dei suoi subgoal lo è. Un goal radice possiede almeno una decomposizione in AND oppure in OR; non solo, per avere una visione completa del problema, l'analista può, a suo piacimento, condurre più decomposizioni del medesimo goal (molteplicità 1..\* fra la root e AND oppure OR-decomposition). Così facendo, la scelta di quale strategia adottare per risolvere il goal radice ricade su un numero ampio di possibilità.

Una nota particolare: se il goal radice  $g_1$  risulta decomposto in un goal singolo  $g_2$ , allora il grafo così ottenuto è equivalente alla relazione ternaria *Contribution* con metrica ++ fra i due goal  $g_1$  e  $g_2$ .

### Il concetto di piano

*Il piano modella cosa è possibile fare per soddisfare un goal, una volta scelta l'opportuna strategia. Un attore è capace di scegliere, definire ed eseguire dei piani.*

Il diagramma delle classi UML di figura 3.6 rappresenta il concetto di plan in Tropos.



**Figura 3.6:** Diagramma delle classi UML che specifica il meta-modello di Tropos riferito al concetto ontologico di *Plan*.

L'attore è capace di definire, scegliere, eseguire  $0..*$  piani, mentre l'istanza di *Plan* può avere  $0..*$  attori (lo zero è incluso per lo stesso motivo discusso nel caso del *Goal*, può rappresentare un'alternativa non eseguita nel caso di una decomposizione in OR su un piano). Il piano esegue un solo *Goal*, mentre un goal può avere a disposizione da 1 a  $n$  piani che ne permettono il soddisfacimento. Esiste la possibilità che il piano contribuisca al soddisfacimento di un goal attraverso la relazione ternaria *Contribution* mediante opportune metriche (leggere sezione precedente).

Il meta-modello permette la costruzione di grafi AND-OR di piani come nel caso del goal. La *AND-OR decomposition* consente di decomporre il piano *root* in *subplan*; il processo può terminare solo quando i *subplan* sono direttamente eseguibili dall'attore che ha svolto l'analisi, oppure quando si rende conto lui stesso di non essere in grado di eseguirli. La *Means-Ends analysis*, rispetto al caso del *Goal*, restringe le entità che possono ricoprire il ruolo di *mean* alla *Resource* oppure al *Plan* mantenendo la stessa molteplicità  $1..*$ .

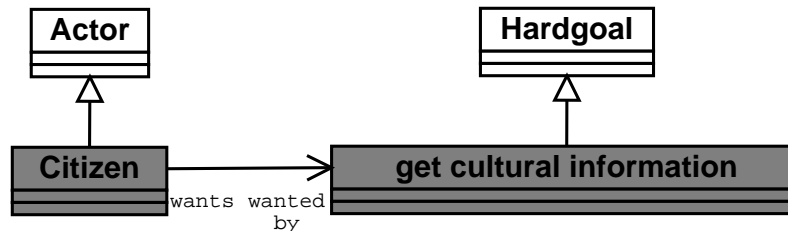
All'interno di varie comunità (AI, Agents, software engineering) il concetto di *plan* è spesso confuso con quello di *task*. Anche nel nostro caso la discussione si è protratta a lungo poiché ognuno portava ragioni fondate a favore di un termine piuttosto che per l'altro. Ad esempio il

plan nella comunità Agents è visto come una sequenza di azioni run-time che permettono il soddisfacimento di un goal. Diversamente il task è un compito da eseguire e generalmente non viene neppure specificato come viene eseguito. Per questo è più vicino ad un'intenzione (*intention* del paradigma BDI), cioè ad un obiettivo che un agente si impegna a raggiungere, piuttosto che ad una sequenza di azioni svolte a run-time.

Questa è uno dei tanti modi di intendere task e plan. La nostra decisione finale prevede di considerare, almeno nella fase di raccolta e modellazione dei requisiti, i due concetti equivalenti, lasciando libertà di scelta a chi usa la metodologia.

### 3.1.3 Il livello del dominio

*Il livello del dominio di Tropos (domain level) è un'istanza del meta-modello dipendente dal dominio applicativo. Tutti i suoi elementi sono istanze di classi del meta-modello (un'oggetto è un'istanza di una meta-classe, così come una relazione è un'istanza di una meta-classe).*



**Figura 3.7:** Diagramma delle classi UML che specifica il *livello del dominio* di Tropos. L'attore Citizen vuole l'hardgoal get cultural information.

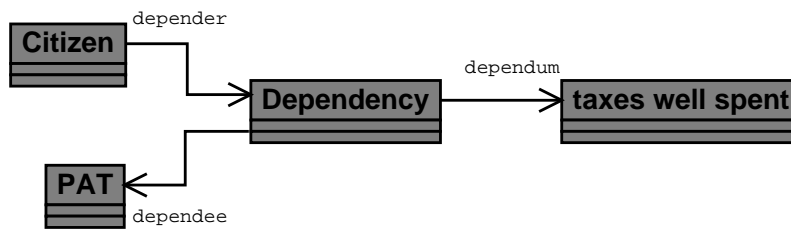
Come suggerisce il nome, il livello del dominio si riferisce ad uno specifico dominio applicativo; se i precedenti livelli erano indipendenti dal dominio e il loro scopo era di fissare i componenti e le loro relazioni, ora si vuole porre attenzione su situazioni particolari. La figura 3.7 riporta il domain level nel caso dell'hardgoal get cultural information assegnato all'attore Citizen.

L'oggetto Citizen è una specializzazione della classe Actor; esso eredita l'associazione unidirezionale e il ruolo (*wants*). L'altro oggetto del dominio è get cultural information istanza della classe Hardgoal. Notare che a questo livello parliamo di oggetti e non di classi perché siamo in un contesto che dipende dal dominio in esame.

La figura 3.8 mostra un diagramma UML dove i due attori Citizen (*dependor*) e PAT (*dependee*) sono coinvolti nella dipendenza con *dependum* taxes well spent.

### 3.1.4 Il livello oggetto

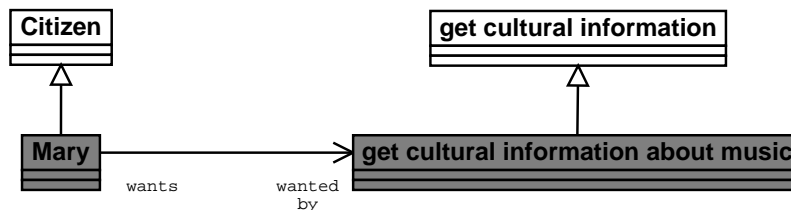
*Il livello oggetto di Tropos (instance level) è un'istanza del livello del dominio: lo scopo è scendere ancora nel livello di astrazione e nei dettagli per raffinare il modello.*



**Figura 3.8:** Diagramma delle classi UML che specifica il *livello del dominio* di Tropos riferito al dominio applicativo dell’*eCulture System*.

La figura 3.9 mostra il diagramma UML ottenuto istanziando il domain level di figura 3.7. L’attore Mary è un’istanza di Citizen, mentre get cultural information about music specializza get cultural information.

A questo livello è possibile specificare gli attributi e le proprietà di cui abbiamo discusso al paragrafo 3.1.1 parlando del meta-metamodello.



**Figura 3.9:** Diagramma delle classi UML che specifica il *livello oggetto* di Tropos riferito al dominio applicativo dell’*eCulture System*.

### 3.2 Concetti chiave

In questa sezione ricordiamo il significato in Tropos di alcuni concetti chiave nell’ingegneria del software quali *modello concettuale*, *diagramma* e *vista*.

I seguenti tre concetti, già stati usati nei precedenti capitoli, sono a volte considerati sinonimi nelle metodologie di modellazione. In Tropos,

**modello** è un grafo diretto etichettato dove i nodi sono istanze delle seguenti meta-classi del meta-modello: *Actor*, *Goal*, *Plan* e *Resource* e dove gli archi sono istanze delle meta-classi che modellano le relazioni ammesse. Le relazioni possibili sono: *Dependency*, *Means-Ends analysis*, *Contribution* e *AND-OR decomposition*. Il modello concettuale in Tropos è rappresentato dal livello del dominio e dal livello oggetto che sono dipendenti dal dominio applicativo.

Equivalentemente, adottando una terminologia matematica,

**modello** è un grafo diretto etichettato  $\{N, A\}$  dove:



- $\tau : N \rightarrow \{Actor, Goal, Plan, Resource\}$ ,
- $\tau : A \rightarrow \{Dependency, Means-Ends\ analysis, Contribution, AND-OR\ decomposition\}$

**diagramma** è una rappresentazione grafica di alcune proprietà del modello. I simboli grafici utilizzati sono definiti dalla “notazione grafica e diagrammi” che assegna un ben preciso simbolo grafico ad ogni concetto specificato nel meta-modello. Tali simboli grafici sono usati nella costruzione di specifici diagrammi che illustrano le proprietà del modello. Li descriveremo in dettaglio in sezione 3.3.

**vista** è un insieme di diagrammi che si focalizzano su parti specifiche del modello a seconda delle fasi del processo di sviluppo. A volte si usa anche il termine modello di early requirements, modello di late requirements e così via per riferirsi ad una porzione particolare del modello concettuale.

Nella prossima sezione andremo a descrivere la notazione grafica e i diagrammi del linguaggio; useremo diagrammi tratti dal case study *eCulture System* descritto all’interno degli articoli seguenti [23, 24, 6, 35, 34, 25]. Tali diagrammi sono stati rappresentati mediante il tool grafico Dia versione 0.88.1<sup>5</sup>.

### 3.3 Notazione grafica e diagrammi

L’obiettivo prossimo è descrivere la rappresentazione diagrammatica del modello nonché quella grafica. All’interno di questo percorso provvederemo a fornire anche un insieme di regole che guidano all’uso dei concetti nelle varie fasi della metodologia.

Tropos dispone di cinque diagrammi che catturano aspetti statici e dinamici del modello. Il diagramma degli attori e quello dei goal (rispettivamente, Actor diagram e Goal diagram) sono specifici per rappresentare il comportamento statico del modello, mentre il diagramma dei piani, delle abilità e delle interazioni fra agenti (rispettivamente, Plan diagram, Capability diagram e Agent Interaction diagram) per quello dinamico. In più, il diagramma degli attori accompagna l’analista dall’early requirements fino all’architectural design, mentre altri quali quello dei piani, delle abilità e delle interazione tra agenti sono tipicamente usati all’interno del detailed design.

Di tutti questi diagrammi vogliamo offrire un insieme di regole con l’obiettivo di fornire una guida all’uso dei concetti durante tutte le fasi del processo di sviluppo, e, per ognuno dei concetti introdotti, disporre di una rappresentazione grafica univoca che permette di associare intuitivamente la classe con l’icona che la rappresenta.

La rappresentazione diagrammatica e la notazione usano pesantemente stringhe, caratteri e nomi; il nostro suggerimento è di adottare una convenzione unica per regolamentare l’uso di questi strumenti indispensabili ai fini della lettura e della chiarezza dei diagrammi e del modello.

<sup>5</sup><http://www.lysator.liu.se/~alla/dia>.

### 3.3.1 Il diagramma degli attori

Il diagramma degli attori (*Actor diagram*) è una rappresentazione grafica del modello dove sono messi in evidenza gli attori, i loro goal e la rete di dipendenze fra gli attori; enfatizza gli aspetti statici del modello.

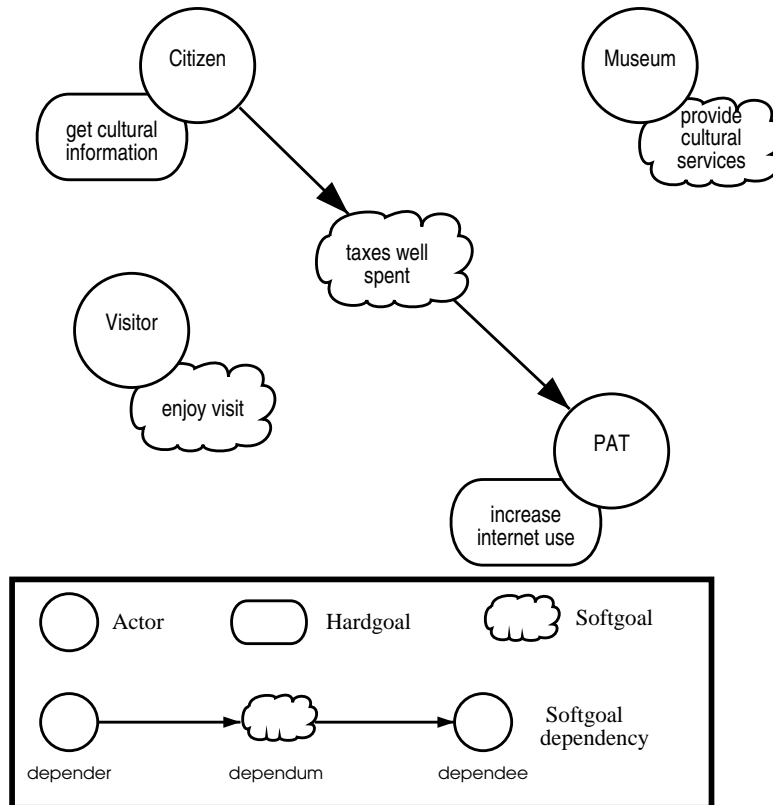


Figura 3.10: Diagramma degli attori estratto dal case study *eCulture System*.

Un esempio di diagramma degli attori estratto dal case study *eCulture System* è mostrato in figura 3.10. Il diagramma degli attori è l'analogo dello *Strategic Diagram* utilizzato in *t*.

Il diagramma mostra due attori Citizen e PAT che, oltre ad avere rispettivamente gli hardgoal *get cultural information* e *increase internet use*, sono impegnati in una dipendenza (istanza della classe *Dependency* del meta-model), dove il ruolo di *depender* lo svolge il Citizen mentre quello di *dependee* la PAT: il *dependum* è un softgoal, *taxes well spent*. Infine, sono presenti altri due attori, Visitor e Museum con i rispettivi softgoal, *enjoy visit* e *provide cultural services*.

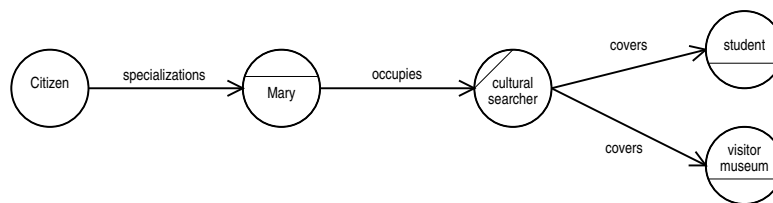
L'Actor diagram si presta ad accompagnare l'analista lungo tutte le fasi del processo di sviluppo; si inizia dalla prima fase, *early requirements*, dove si raccolgono il maggior numero di informazioni circa la struttura, la composizione dell'ambiente che circonda il sistema software

da progettare. Si tratta di capire quali sono gli attori rilevanti, quali dipendenze sussistono fra essi e i loro goal. Procedendo, da un lato il diagramma viene raffinato ulteriormente e magari ampliato di nuove dipendenze, attori e goal (tipicamente verso la fine degli early requirements e per tutti i late requirements), mentre dall'altra contribuisce all'identificazione delle capability nel processo che conduce all'"agentificazione" (architectural design).

### Termini e notazione

La figura 3.11 mostra la gerarchia fra i vari tipi di attore: da sinistra verso destra, *Citizen*, che rappresenta un comune cittadino, viene specializzato nell'agente *Mary*. Un agente può occupare diverse posizioni; nell'esempio assume quelle di ricercatore di informazioni culturali (*cultural searcher*) ricoprendo due diversi ruoli: *student* e *visitor museum*.

Dal punto di vista notazionale, l'attore è raffigurato come un cerchio con all'interno l'etichetta che identifica il nome (consigliamo di scrivere almeno la prima lettera del nome dell'attore maiuscola). Le sue specializzazioni (agent, position e role) differiscono fra loro dalla posizione del segmento interno.



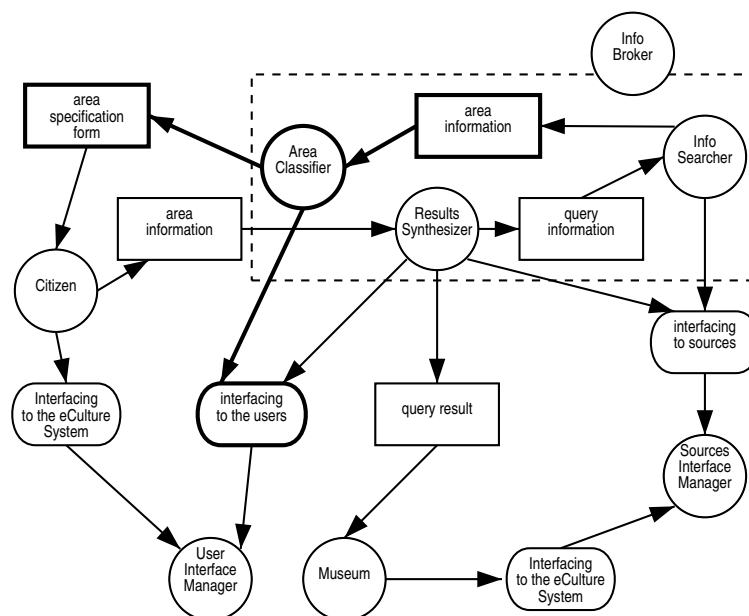
**Figura 3.11:** Relazione fra l'attore generico *Citizen* e agent, position, role.

Tropos riconosce due tipi di attore in base al livello dove questi vengono introdotti: attori sociali e di sistema (rispettivamente, *social* e *system actor*). Gli attori sociali sono presenti tipicamente nei diagrammi della prima fase, early requirements, e modellano gli attori dell'ambiente o dell'organizzazione che circonda il sistema software da progettare. I secondi invece sono propri del sistema software da modellare, quelli che poi parteciperanno attivamente alla soluzione del problema. Un esempio di attore sociale è la PAT; essa modella la Provincia Autonoma di Trento, l'ente con lo scopo di fornire un servizio culturale soddisfacente ai visitatori. Il suo goal, fra gli altri, è istituire e mantenere funzionante un servizio software che risponda ai desideri dei cittadini nonché dei visitatori. Per questo nei late requirements si introduce l'attore di sistema *e-Culture System* che eredita dalla PAT il goal. Quindi, l'*e-Culture System* modella il sistema software che andrà progettato e realizzato, mentre la PAT un attore dell'ambiente nel quale l'agente software verrà calato.

In fase di architectural design il diagramma degli attori viene esteso per identificare le *capabi-*

lity. Attraverso l'analisi delle dipendenze fra gli attori (di sistema e sociali) è possibile determinare quali abilità assegnare loro. Una capability rappresenta l'abilità di un attore di definire, scegliere ed eseguire un insieme di piani per soddisfare dei goal. Se pensiamo al tool JACK, una capability è composta di piani, risorse (pensiamo ad esempio ad un database) ed eventi. La figura 3.12 mostra l'estensione di un diagramma degli attori che consente di individuare le capability.

Il rettangolo tratteggiato racchiude i *subactor* di sistema, ovvero attori introdotti per aiutare Info Broker a portare a compimento i suoi goal. Ad esempio Area Classifier è responsabile per la classificazione delle informazioni fornite dall'utente; esso dipende da User Interface Manager per l'interfacciamento con gli utenti. Info Searcher dipende da Area Classifier per avere delle informazioni riguardo le aree tematiche alle quali l'utente è interessato (la dipendenza area information è modellata come una risorsa rappresentata graficamente mediante un rettangolo). Il diagramma permette di estrarre le capability per ciascun attore di sistema.



**Figura 3.12:** Estensione del diagramma degli attori che analizza le dipendenze tra attori del sistema e attori sociali per l'attribuzione delle *capability*. Il rettangolo tratteggiato racchiude i *subactor* ai quali Info Broker delega alcune competenze.

### Altre particolarità

L'intuizione di adottare un diagramma capace di cogliere gli aspetti significativi partendo dalla primissima fase dell'analisi dei requisiti è sviluppata da E. Yu in [46]. Il nostro Actor diagram propone molte analogie con lo *Strategic Diagram* (SD) di  $\tau$ . Tuttavia, a nostro giudizio, lo sforzo è stato superiore in quanto forniamo strumenti dei quali conosciamo le regole di collaborazione:

grazie al meta-modello, sappiamo i tipi di relazioni, le associazioni tra i vari elementi, cosicché possiamo dire con certezza quando il diagramma degli attori risulta essere *ben formato*. Oltretutto, nello SD di  $i^*$  manca il concetto essenziale di goal e piano assegnato ad un attore. Infatti, il linguaggio di Yu non consente ad un attore di possedere una delle due entità sopra citate, ma solamente di stabilire delle dipendenze.

In altri linguaggi ancora, pensiamo ad esempio a KAOS [15, 16, 43] per un'esplorazione più ampia), si usano i *system goal*: sono sostanzialmente obiettivi da raggiungere dal sistema nel suo complesso e non dal singolo attore. L'idea è di identificare gli obiettivi dell'intero sistema che, almeno in prima istanza, non necessariamente sono legati a degli attori; KAOS propone di iniziare con l'individuazione dei goal piuttosto che gli attori come invece suggerisce Tropos.

In conclusione, confrontando il nostro il diagramma degli attori con altri analoghi, esistono delle valide motivazioni per poter affermare che lo sforzo prodotto è qualificante sia sotto il profilo della leggibilità e della rappresentazione del diagramma, sia per quanto riguarda un certo livello, seppur minimo, di correttezza e formalismo che consente di produrre dei semilavorati soddisfacenti. D'altra parte, l'obiettivo di un linguaggio semiformale è quello di proporre una sintassi rigorosa mentre la semantica viene un po' trascurata sull'altare della leggibilità, della comprensibilità e, soprattutto, della facilità d'uso anche per persone non esperte [3, 21].

### Quando usare il diagramma degli attori

L'uso da farsi del diagramma degli attori è principalmente per avere una visione **statica** dell'ambiente e del sistema da sviluppare (early e late requirements, rispettivamente). Si mettono in evidenza concetti ad alto livello come attori, goal e dipendenze, ma non si deve pensare di usarlo per estrarre informazioni su comportamenti nel tempo e su come si muovono i flussi di dati. Come due attori si scambiano i messaggi, il tipo di protocollo usato, non sono certamente qualità che si scoprono dall'analisi del diagramma in questione: piuttosto serviranno altri diagrammi specifici.

Tuttavia, il diagramma degli attori diviene molto utile anche in fase di progettazione dell'architettura logica e fisica, laddove si individuano e assegnano le capability ai vari attori. Come si può facilmente notare, questo diagramma ha un'ampio spettro di azione, lo si trova nelle primissime fasi della metodologia, ma anche verso la fine quando ci si concentra su aspetti architettonici prima di passare all'atto finale, l'implementazione. Riassumendo il diagramma degli attori ha tre obiettivi strategici:

1. Raffigurare gli attori dell'ambiente e del sistema con relativi goal e dipendenze a seconda della fase in cui si lavora.
2. Mantenere traccia ed eventualmente integrare con nuovi attori, dipendenze e goal usciti da raffinamenti condotti nel diagramma dei goal che vedremo tra poco.
3. Aiutare il progettista ad identificare le capability per ciascun attore (*architectural design*).

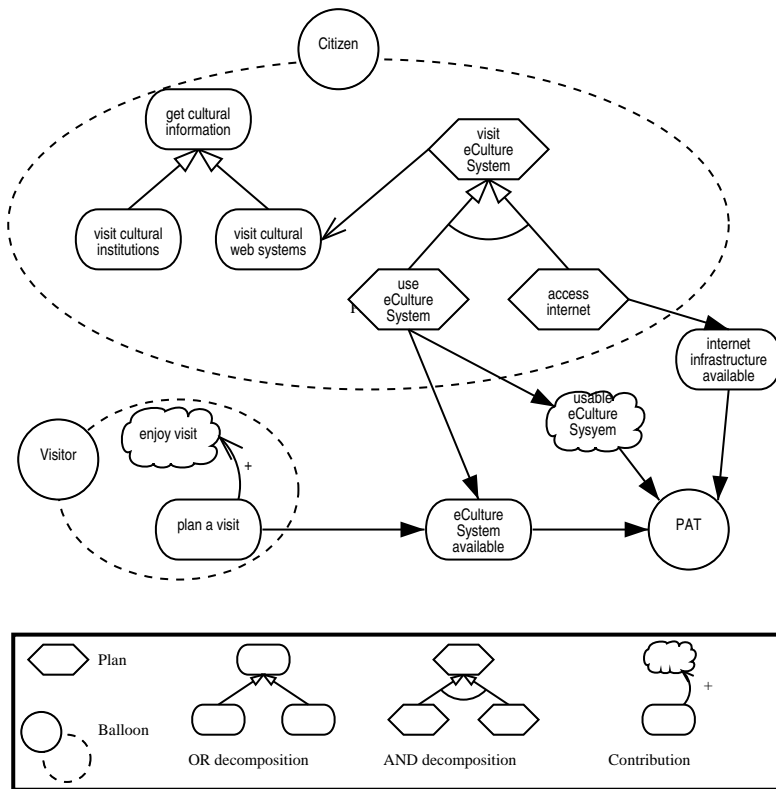
**Dove trovare altre informazioni**

Riferimenti per cogliere ulteriori informazioni sono [46, 48, 49, 6, 35, 25].

**3.3.2 Il diagramma dei goal**

*Il diagramma dei goal (Goal diagram) è il luogo dove si analizzano in profondità i goal e le dipendenze mediante le tecniche usuali di Means-Ends analysis, AND-OR decomposition e Contribution.*

La figura 3.13 mostra un esempio tipico diagramma dei goal estratto dal solito case study; † invece adotta un analogo che si chiama *Rational Diagram*.



**Figura 3.13:** Un esempio tipico di diagramma degli attori estratto dal case study.

In ingresso il diagramma dei goal riceve goal e dipendenze (in quest'ultimo caso l'attore deve ricoprire il ruolo di dependee) da quello degli attori: attraverso le relazioni definite nel meta-modello, è possibile decomporre i goal in subgoal, individuare soluzioni alternative che risolvono il goal iniziale per mezzo di altre entità oppure individuare quali contribuiscono positivamente o negativamente. In output fornisce tipicamente nuove dipendenze fra attori oppure piani e risorse che consentono di soddisfare il goal iniziale. Evidentemente, costruire un diagramma dei goal

in una fase piuttosto che in un'altra cambia il significato. Negli early requirements si compiono analisi sulle dipendenze esistenti fra i social actor introdotti nel diagramma degli attori. Un'analisi simile si produce anche su i system actor nei late requirements, una volta che il diagramma degli attori ha stabilito le relazioni fra gli attori del system-to-be e quelli dell'environment.

### Termini e notazione

La figura 3.13 mostra l'attore Citizen condurre l'analisi dell'hardgoal get cultural information. Esso viene decomposto tramite una OR-decomposition in due subgoal, visit cultural institutions e visit cultural web system. Poi, su quest'ultimo si applica una Means-Ends analysis determinando il piano (che ricopre il ruolo di mean) visit eCulture System che, a sua volta, si divide in un due subplan use eCulture System e access internet posti in AND. L'ellisse tratteggiata, chiamata *balloon*, demarca il confine tra lo spazio adibito all'uso delle tecniche di analisi e di investigazione, e lo spazio dove invece si esportano le nuove dipendenze con attori esterni.

All'interno del balloon di proprietà dell'attore Visitor, l'hardgoal plan a visit contribuisce al softgoal enjoy visit che proviene dall'Actor diagram di figura 3.10.

La conclusione di questo processo avviene quando un nodo contenuto in un albero non ha più figli all'interno del balloon (use eCulture System). A questo punto le possibilità sono due:

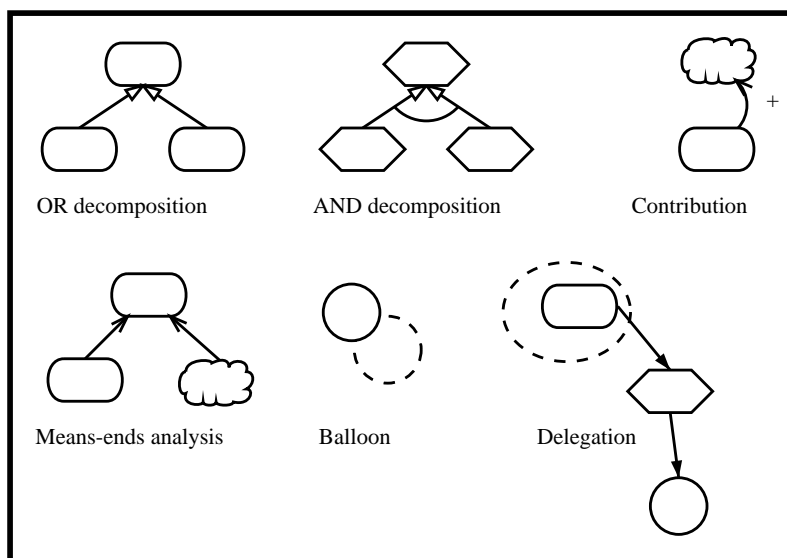
- l'attore valuta attentamente la strategia migliore (magari anche con l'ausilio dei requisiti di qualità) che è in grado di realizzare secondo le sue capacità, oppure,
- l'attore si rende conto di non essere in grado di risolvere un goal, eseguire un piano o altro e decide di delegarlo ad un altro attore (internet infrastructure available).

Nel secondo caso è importante memorizzare da qualche parte il motivo che ha spinto Citizen a delegare il goal internet infrastructure available alla PAT. Per fare questo, è sufficiente inserire nella *Dependency* il campo opzionale *why*. Così facendo, anche in futuro, conosceremo sempre l'origine che ha dato luogo alla nuova dipendenza.

Ogni tipo di analisi possiede la propria notazione (figura 3.14). La rappresentazione grafica della AND-decomposition ha in più rispetto alla OR-decomposition un archetto posto poco sotto le due frecce chiuse. Per indicare dei contributi verso un goal si usa una freccia leggermente storta con l'etichetta a fianco che fissa la metrica: i valori possibili sono --, -, +, ++. La Means-Ends analysis è visualizzata da una semplice freccia aperta, mentre il meccanismo di delega invece con una freccia piena.

### Altre Particolarità

Il diagramma dei goal consente di avere una visione microscopica del dominio applicativo. Il suo scopo è quello di determinare delle strategie opportune per risolvere i goal che appartengono



**Figura 3.14:** Rappresentazione grafica dei tipi di *analisi* ammesse nel diagramma dei goal di Tropos e condotte all'interno del balloon.

all'attore il quale fornisce il proprio punto di vista. Viceversa, il diagramma degli attori enfatizza su aspetti macroscopici, cioè su più attori che sono in relazione fra loro ma non ci si preoccupa della loro struttura interna.

Il Goal diagram è utile anche in fase di definizione dell'architettura globale del sistema quando si tratta di estendere l' Actor diagram proveniente dalle fasi precedenti per individuare le capability. In questo caso è probabile che il Goal diagram abbia introdotto nuovi attori per consentire una via migliore alla realizzazione del sistema software.

Per chi ha già conoscenze di  $i^*$  non potrà certo non notare alcune somiglianze con lo SR. Tuttavia il nostro diagramma è governato da regole ordinate e da passi incrementali (in  $i^*$  questo non avviene con il medesimo rigore, tutto è abbastanza informale). Le relazioni disponibili sono in numero maggiore e consentono di condurre tipi differenti di analisi.

### Quando usare il diagramma dei goal

Si consiglia l'uso per approfondire, investigare e proporre nuove soluzioni, dipendenze e goal. Lo strumento del balloon consente di localizzare e raggruppare tutto ciò che fa parte del mondo interiore del singolo attore. Riassumendo, il diagramma dei goal è utile nei seguenti casi:

- prodotto il diagramma degli attori, si catturano dipendenze e goal per poi analizzarli dal punto di vista dell'attore mediante le usuali tecniche di Means-Ends analysis, Contribution e AND-OR decomposition,
- nell'architectural design il diagramma dei goal è utile per arricchire quello degli attori di nuovi attori e dipendenze che partecipano all'individuazione delle capability.



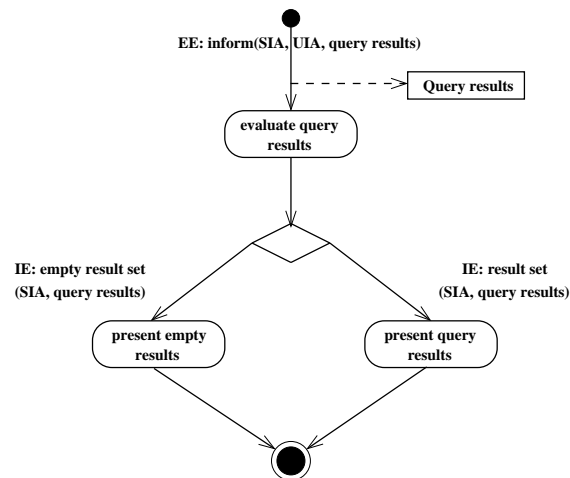
## Dove trovare altre informazioni

Altre informazioni sono reperibili in [40, 46, 48, 49, 6, 35, 25].

### 3.3.3 Il diagramma delle abilità

*Il diagramma delle abilità (Capability diagram) è un diagramma delle attività AUML che permette di modellare una capability (o un insieme di capability) dal punto di vista di uno specifico attore.*

Un esempio di diagramma delle abilità è mostrato in figura 3.15: quando sopraggiunge l'evento esterno `inform` viene attivata la capability `query results`. La transizione dallo stato iniziale a quello successivo prevede la lettura dell'informazione `Query results`. Il piano attivato è `evaluate query results`. Il nodo decisionale che compare serve a stabilire se il risultato della query ha prodotto informazioni ritenute valide. In tal caso viene attivato il piano `present query results` in risposta all'evento interno `result set`. Viceversa, l'evento lanciato è `empty results set` con il medesimo piano. In entrambi i casi dopo aver eseguito il piano la capability termina.



**Figura 3.15:** Diagramma delle abilità riferito alla capability `query result`.

Il diagramma delle abilità è previsto solo nella seconda fase di progettazione della metodologia (*detailed design*) quando è stata già scelta la piattaforma architeturale. Una volta che al passo precedente sono state individuate le capability queste devono essere implementate in termini di eventi esterni ed interni, piani che rispondono agli eventi, risorse sotto forma di database o strutture dati. Una capability è attivata dall'arrivo di un evento esterno. I nodi che in UML modellano le attività in Tropos sono piani, gli archi di transizione sono eventi e i belief oggetti.

<sup>6</sup>Il termine inglese corretto è *posted*.

## Termini e notazione

La capability è attivata da uno o più eventi. Distinguiamo due tipi di evento: *esterno* ed *interno*. Un evento si dice esterno se proviene da un agente diverso da quello che possiede la capability, interno se è lo stesso agente che lancia l'evento. Di solito l'evento interno è utile alla fine di un processo in cui risulta necessario compiere delle nuove azioni mediante qualche piano. L'evento esterno si usa per la spedizione di messaggi fra agenti. La sintassi dell'evento esterno è *EE:name-of-event(sender-agent, receiver-agent, name-of-capability)*, mentre per l'interno è *IE:name-of-event(name-of-agent, name-of-capability)*.

Il Capability diagram è assai utile quando si vuole capire a fondo gli aspetti dinamici della capability fatta di piani, risorse e eventi. Nell'istante stesso in cui un evento esterno viene impostato, si attivano diversi piani appartenenti alla stessa capability: il piano che risponde all'evento ha la capacità di lanciare un altro evento e quindi vi può essere un'ulteriore piano eseguito. In quasi tutte le capability sono necessarie delle informazioni circa lo stato del mondo e dell'agente stesso che possiede la capability (belief).

La notazione grafica adottata è la stessa del diagramma delle attività AUML: cambia il significato attribuito a ciascun elemento. L'ovale rappresenta il piano da eseguire, l'arco una transizione di stati, il rettangolo una credenza (belief) e il rombo un nodo decisionale.

## Quando usare il diagramma delle abilità

Il Capability diagram è assai utile quando si desidera approfondire gli aspetti dinamici legati ad una capability. Si possono rappresentare gli stati interni, come gli eventi vengono attivati, quali piani sono eseguiti e quali belief sono richiesti.

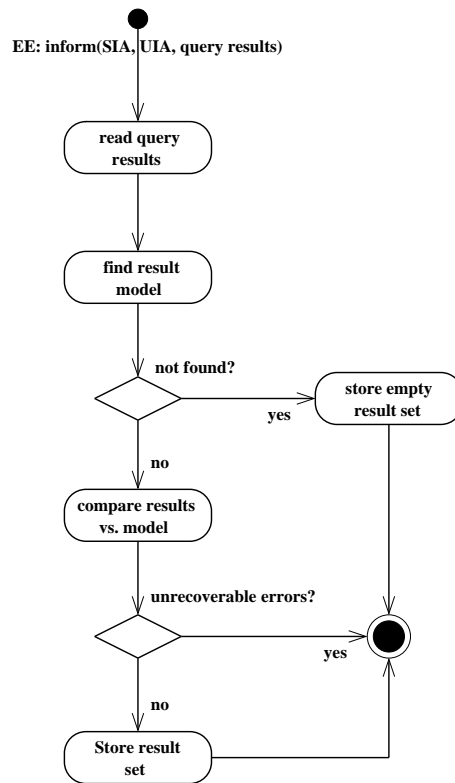
## Dove trovare altre informazioni

I riferimenti principali per l'activity diagram di UML e AUML sono in [40, 3, 4, 19, 20, 17, 31, 30]. Esempi di uso del Capability diagram negli articoli [23, 24, 6, 35, 34, 25, 40].

### 3.3.4 Il diagramma dei piani

*Il diagramma dei piani (Plan diagram) è un diagramma delle attività AUML che analizza gli stati interni di un piano contenuto in una capability.*

Il piano descrive cosa un agente deve fare quando occorre un evento interno o esterno; in particolare, un piano descrive una sequenza di azioni che lo conducono a soddisfare l'evento che lo ha lanciato. Ciascun nodo di un diagramma delle abilità può essere ulteriormente specificato usando un diagramma delle attività AUML. Per esempio la figura 3.16 mostra il diagramma dei piani di `evaluate query results`.



**Figura 3.16:** Diagramma dei piani riferito al piano evaluate query results.

Il piano viene attivato dall'arrivo dell'evento esterno `inform` presente all'interno della capability `query results` dell'agente `Synthesizer` e si conclude con il risultato della query. Gli stati interni sono sequenze di azioni elementari o atomiche che formano il piano attraverso l'analisi di condizioni logiche *true-false*. Per azione atomica intendiamo che ogni singolo stato corrisponde ad una istruzione oppure ad un insieme di istruzioni.

### Termini e notazione

La struttura del diagramma dei piani è molto simile a quella del diagramma delle abilità. Come suggerisce il nome si tratta di decomporre il piano in azioni elementari che consentono di raggiungere lo scopo per cui è stato invocato. Ogni ovale corrisponde ad una istruzione oppure ad insieme di istruzioni, ciascun arco ad una transizione da un'azione a quella successiva mentre il rettangolo ad una struttura dati. Sono presenti anche dei nodi decisionali (come nel caso del Capability diagram) che individuano i possibili rami *true-false*.

### Quando usare il diagramma dei piani

Una volta costruito il Capability diagram si devono specificare i piani che lo compongono. Si tratta di studiare a basso livello (siamo alla fase immediatamente prima dell'implementazione) quali

sono le azioni, le decisioni logiche e i costrutti necessari per realizzare un determinato scopo.

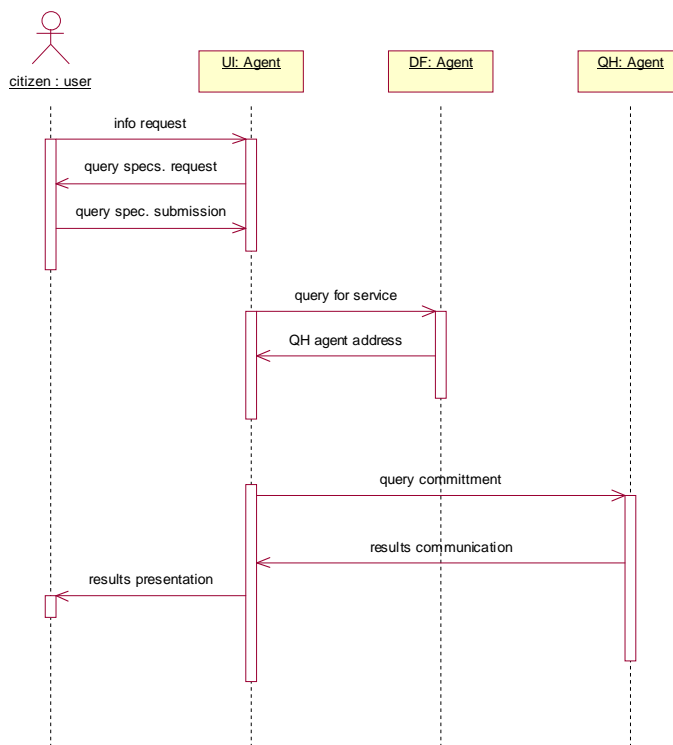
### Dove trovare altre informazioni

Si veda sezione “Dove trovare altre informazioni” per il diagramma delle abilità.

### 3.3.5 Il diagramma delle interazioni tra agenti

*Il diagramma delle interazioni tra agenti (Agent Interaction diagram) è un diagramma di sequenza AAML nel quale la linea di vita di ciascun agente è indipendente dalla specifica interazione che deve essere modellata. Le azioni comunicative fra i vari agenti corrispondono a messaggi asincroni.*

La figura 3.17 mostra un semplice diagramma delle interazioni tra agenti composto di alcuni agenti del dominio applicativo fin qui studiato.



**Figura 3.17:** Esempio di diagramma delle interazioni tra agenti riferito alle interazioni possibili tra gli agenti. I messaggi sono asincroni.

In particolare il diagramma modella le interazioni fra user (che può essere un cittadino), User Interface Manager (UIA), Directory Facilitator (DF) e Query Handler (QH). L’interazione inizia con la richiesta di informazioni da parte dell’utente verso lo UI e si conclude con la

presentazione dei risultati. In mezzo vi sono una serie di messaggi che i vari agenti si scambiano per poter presentare le informazioni all'utente.

#### **Termini e notazione**

La terminologia e la notazione è quella classica del diagramma di sequenza UML.

#### **Quando usare il diagramma delle interazioni tra agenti**

Il diagramma delle interazioni tra agenti è usato per specificare i messaggi che gli agenti si possono scambiare. È assai utile per determinare le loro interazioni e l'ordine di spedizione dei messaggi stessi.

#### **Dove trovare altre informazioni**

Le fonti principali sono i diagrammi di sequenza UML [3, 4, 19, 20, 17] e AUML [31, 30]. Applicazioni di *diagramma delle interazioni tra agenti* si trovano in [40, 23, 24, 6, 35, 34, 25].



## **Parte II**

**Un esempio. Specifica di un sistema di supporto all'organizzazione di meeting**





## Capitolo 4

# Il sistema per l'organizzazione di meeting

Quest'ultimo capitolo applica quanto visto ora al caso di studio di un sistema per l'organizzazione di meeting, d'ora in avanti indicato come *Meeting Scheduler System*. Tale esempio è utilizzato in numerose pubblicazioni [16, 46, 39, 40, 26] e per questo scelto a dimostrazione di Tropos. Il nostro obiettivo è quello di presentare la metodologia Tropos attraverso le fasi del processo di sviluppo descritte nei capitoli 2, 3 e con l'ausilio di un insieme di trasformazioni [6] che possono essere applicate al modello per raffinarlo incrementalmente dall'early requirements fino alla fase finale. Da un punto di vista pratico, l'ingegnere adotta le trasformazioni riportate nelle tabelle A.1, A.2 e A.3 a seconda che egli preferisca un approccio *top-down* piuttosto che *bottom-up*. A prescindere dall'approccio scelto ciascuna di queste trasformazioni sono frutto di combinazioni di altre, dette *primitive* (vedere tabelle A.4, A.5 e A.6), che non possono essere espresse come composizione di altre trasformazioni.

Le tabelle riportate in appendice sono quelle estratte da [6]. Tuttavia, vogliamo segnalare che rispetto a questo documento vi sono delle differenze nell'uso di termini e di notazione: (*i*) quelli che per noi sono *hardgoal* nell'articolo sono *goal*; (*ii*) la nostra relazione *Contribution* vale anche fra *plan* verso *goal* generici; (*iii*) nell'articolo non compaiono la *Means-Ends analysis* e la *Plan decomposition*; (*iv*) la notazione grafica non è la stessa poiché, al tempo, non era ancora stata definita.

### 4.1 Definizione del problema

In questa sezione ci preoccupiamo di riassumere brevemente la definizione del problema del *Meeting Scheduler System* come formulato in [42].

Organizzare degli eventi che coinvolgono un elevato numero di partecipanti non è semplice. Tipicamente, per promuovere un meeting deve essere presente la figura dell'organizzatore (*mee-*

*ting initiator*), cioè di una persona e/o sistema software che si presta a raccogliere le informazioni dai partecipanti (o potenziali) e, in base a queste, fissare data, luogo e ora.

L'organizzatore deve richiedere almeno le seguenti informazioni: (*t*) un insieme di date nelle quali i partecipanti sono già occupati (*exclusion set, es*); (*u*) un insieme di date preferite per lo svolgimento del meeting (*preference set, ps*); (*ui*) un range di ore nelle quali può avere luogo il meeting (*time period, tp*). Il *date range* è una struttura che contiene queste tre informazioni.

È utile distinguere fra partecipanti *attivi* e *importanti*. I primi sono coloro ai quali necessita un particolare tipo di equipaggiamento (workstation, connessione internet, proiettori, ecc.), ai secondi sono invece richieste delle preferenze circa il luogo di svolgimento del meeting.

Intersecando le date espresse dai partecipanti è assai probabile che scaturiscano dei conflitti. Un conflitto è “forte” (*strong*) quando nessuna data può essere trovata all'interno delle *ps*, del *tp* ed esternamente all'*es* (le *es* contengono le date indesiderate, quindi si devono cercare esternamente a tale insieme); è “debole” (*weak*) quando si trovano delle date possibili all'interno di *ps*, di *tp* ed esternamente a *es* di ciascun partecipante, ma intersecando tutte le *ps* il risultato è un insieme vuoto.

Per risolvere eventuali conflitti vengono proposte alcune strategie che sotto riportiamo:

- alcuni partecipanti si ritirano dal meeting;
- vengono aggiunte nuove *ps* o *tp*;
- vengono rimosse alcune *es* o *tp*;
- viene esteso il *date range* oppure *tp*.

La scelta della sala dove svolgere il meeting deve possedere alcune semplici caratteristiche: disponibilità per le date del meeting, essere equipaggiata secondo le richieste avanzate dai partecipanti e contenere un numero sufficiente di persone.

L'obiettivo del sistema software è quello di organizzare il meeting, cioè determinare le date, il luogo e le ore di svolgimento tenendo sempre in considerazione le proposte avanzate dai partecipanti. Il sistema si deve occupare inoltre di tenere aggiornati i partecipanti consentendo loro di modificare in ogni momento le informazioni personali (*es, ps, tp, ecc.*). Si auspica che le interazioni fra i vari utenti siano regolate e nel numero minimo possibile, così da evitare sprechi di tempo e risorse; le prestazioni devono essere elevate per garantire una rapida determinazione e comunicazione delle scelte del sistema software che organizza il meeting. Inoltre si deve essere in grado di assicurare che l'accesso al sistema sia permesso solo agli utenti autorizzati.

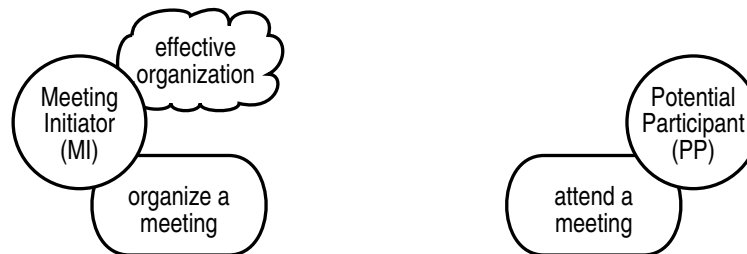
## 4.2 Analisi e specifica dei requisiti

Il livello di astrazione dal quale cominciare l'analisi del dominio applicativo è il primo quesito da risolvere. Evidentemente, quali attori, goal e dipendenze stabilire fin dal principio dipende,

in buona parte, dall'interpretazione soggettiva dell'analista. Tuttavia è importante ricordare che dobbiamo iniziare individuando gli *stakeholder* dell'environment, le loro dipendenze e così via, senza lasciarci trasportare dalla fretta di introdurre da un lato entità che andranno a comporre il sistema software finale e dall'altra vincoli restrittivi di progettazione. Se l'analista segue queste due considerazioni, la scelta del punto di partenza e del livello è a suo arbitrio: importante è riuscire a far comprendere al lettore le motivazioni che hanno condotto verso certe scelte piuttosto che altre.

### 4.2.1 Analisi dell'ambiente

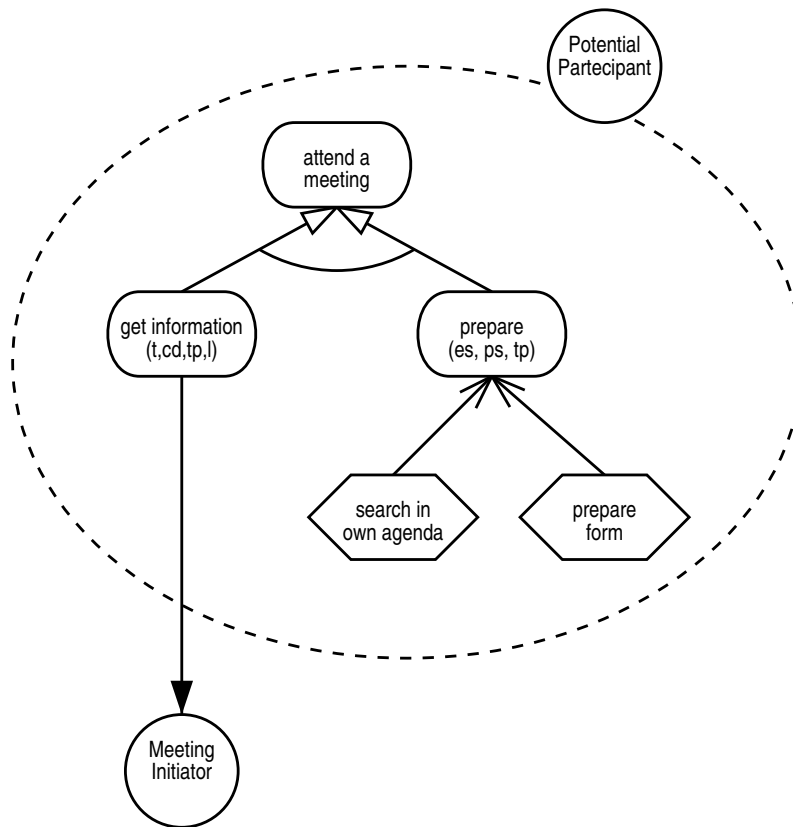
La descrizione del problema riportata in sezione 4.1 manifesta palesemente i social actor coinvolti nell'environment: Meeting Initiator (MI) e Potential Participant (PP). La trasformazione primitiva adottata per introdurre i due attori è *Actor Introduction (A-I)*. L'Actor diagram che dà il via all'analisi e modellazione dei requisiti è mostrato in figura 4.1.



**Figura 4.1:** Diagramma degli attori che introduce i due attori sociali dell'ambiente (MI e PP) con i rispettivi goal.

Al primo attore è assegnato l'hardgoal `organize a meeting` che rappresenta l'obiettivo principale che esso deve soddisfare, appunto organizzare un meeting. Esso viene introdotto grazie alla trasformazione primitiva *Goal Introduction (G-I)*. Accanto a questo, viene anche richiesto che l'organizzazione sia efficace; tale qualità viene modellata attraverso il softgoal `effective organization` (*SoftGoal Introduction (SG-I)*). Il secondo attore sociale modella tutti i potenziali partecipanti al meeting che MI deve organizzare. PP ha, per adesso, un unico goal, partecipare al meeting (`attend a meeting`).

Il passo successivo prevede l'analisi dei goal assegnati ai due attori costruendo vari Goal diagram. La figura 4.2 evidenzia il punto di vista del PP relativamente all'hardgoal `attend a meeting`; per decomporre il goal in subgoal abbiamo adottato la trasformazione *Goal AND-Decomposition* secondo un approccio top-down. Per partecipare ad un meeting PP deve ottenere informazioni quali: le aree tematiche trattate (`topic`, nella nostra notazione futura sarà indicato con `t`), la data (`calendar date`, `cd`), l'orario di svolgimento della manifestazione (`time`



**Figura 4.2:** Diagramma dei goal costruito dal punto di vista dell'attore Potential Participant sul suo hardgoal attend a meeting.

period,  $tp$ ) e il luogo della stessa (location,  $l$ ). Tutti questi dati possono pervenire a patto che l'organizzatore li metta a disposizione dei partecipanti o degli aspiranti partecipanti; per questo, il goal `get information (t,cd,tp,l)` viene delegato attraverso la *Goal Delegation* al MI il quale si prende carico di soddisfarlo come meglio crede. Oltre a ricevere queste informazioni, il PP deve preparare un insieme di date in cui non è disponibile (exclusion set,  $es$ ), un insieme di date preferite (preference set,  $ps$ ) e  $tp$ . Operativamente si compie una ricerca nella propria agenda per verificare quali sono le date disponibili (`search in own agenda`) e si compila una scheda dove vengono inseriti i dati (`prepare form`) che saranno poi consegnati al MI il quale provvederà ad organizzare l'evento.

In merito al goal `organize a meeting` (figura 4.3) abbiamo condotto una GA che ha permesso di introdurre in AND quattro subgoal, `collect information`, `generate date meeting (cd,tp,l)`, `size of meeting` e `provide (t,cd,tp,l)`. I primi tre vengono analizzati in 4.3, mentre il rimanente più la dipendenza appena consegnata dal PP al MI in 4.4.

`collect information` ha come obiettivo raccogliere quante più informazioni possibili a riguardo dei partecipanti; nome, cognome, recapito per comunicazioni urgenti, vincoli temporali,

organizzativi, ecc. Questo viene poi spezzato in AND in ulteriori due subgoal, *information about identity* e *collect constraints*. Per collezionare le informazioni generiche circa l'identità dei partecipanti si stabilisce una dipendenza fra MI e PP con *dependum information about identity*. Facciamo notare in questo caso che abbiamo istanziato il quarto campo opzionale della relazione *Dependency*: il ruolo di *dependor* è ricoperto dal MI, *dependee* da PP, *dependum* dal goal *information about identity* mentre il *why* dalla *resource information about identity*.

Il subgoal *collect constraints* ha lo scopo di collezionare dei vincoli che i partecipanti devono esprimere. Le “costrizioni” da noi identificate impongono che un generico partecipante (from all) manifesti delle *es*, *ps* e *tp* a prescindere dal ruolo occupato durante il meeting o nell'organizzazione dello stesso (l'organizzatore può essere anche un partecipante). Ciò che invece dipende dal ruolo del partecipante è l'equipaggiamento richiesto (proiettore, workstation, telefono, connessione network, ecc.) per poter, ad esempio, presentare un proprio lavoro alla conferenza. In tal caso il *dependum* è *equipment requirements (er)* mentre il *dependee* è *Active Participant (AP)*. Notare come AP sia un particolare attore che ricopre il ruolo di “partecipante attivo”. Infine vi sono altri vincoli che sono espressi stavolta dai partecipanti cosiddetti “importanti” (*Important Participant, IP*) mediante la risorsa *preference meeting (pm)*. Questa esprime le preferenze circa il luogo dove svolgere la manifestazione; non è esattamente un vincolo che l'organizzatore deve considerare completamente ma rappresenta solo un'indicazione.

Per soddisfare il subgoal *generate date meeting (cd,tp,l)* bisogna comporre la terna formata da: la data della conferenza (*cd*), l'orario durante i giorni di svolgimento (*tp*) e il luogo (*l*). È prematuro, a nostro avviso, addentrarci nell'esplorazione esaustiva del subgoal poiché non disponiamo ancora di elementi che permettano di capire esattamente come soddisfarlo (e dunque ne rimandiamo la discussione), però è possibile annotare che collezionare vincoli (*collect constraints*) e comunicare tempestivamente cambiamenti di qualunque tipo ai partecipanti (*rapid communication some change*) può contribuire positivamente al soddisfacimento del goal iniziale.

Il successivo subgoal da soddisfare è mostrato in figura 4.3 è *size of meeting*. L'idea è distinguere in base alla dimensione del meeting; appare evidente che organizzare una manifestazione interna ad un istituto di ricerca o ad un'organizzazione ove non sono ammessi partecipanti esterni rispetto ad una analoga che invece prevede persone che giungono appositamente per tale appuntamento da altre località o dall'estero, comporta una serie di complicazioni di carattere organizzativo-gestionale. Anche la scelta di quale sala adibire al meeting è fortemente condizionata da questo aspetto. Pertanto, abbiamo proposto una decomposizione in OR introducendo i due subgoal *close to visitors* e *open to visitors (Goal OR-Decomposition)*. Se l'intenzione è promuovere l'evento per soli dipendenti dell'azienda o dell'istituto più al massimo, qualche invitato, allora l'organizzazione si riduce a contattare ciascun PP direttamente (*contact each PP*). I modi per fare ciò sono vari: per telefono, per posta elettronica o andando fisicamente dall'interes-

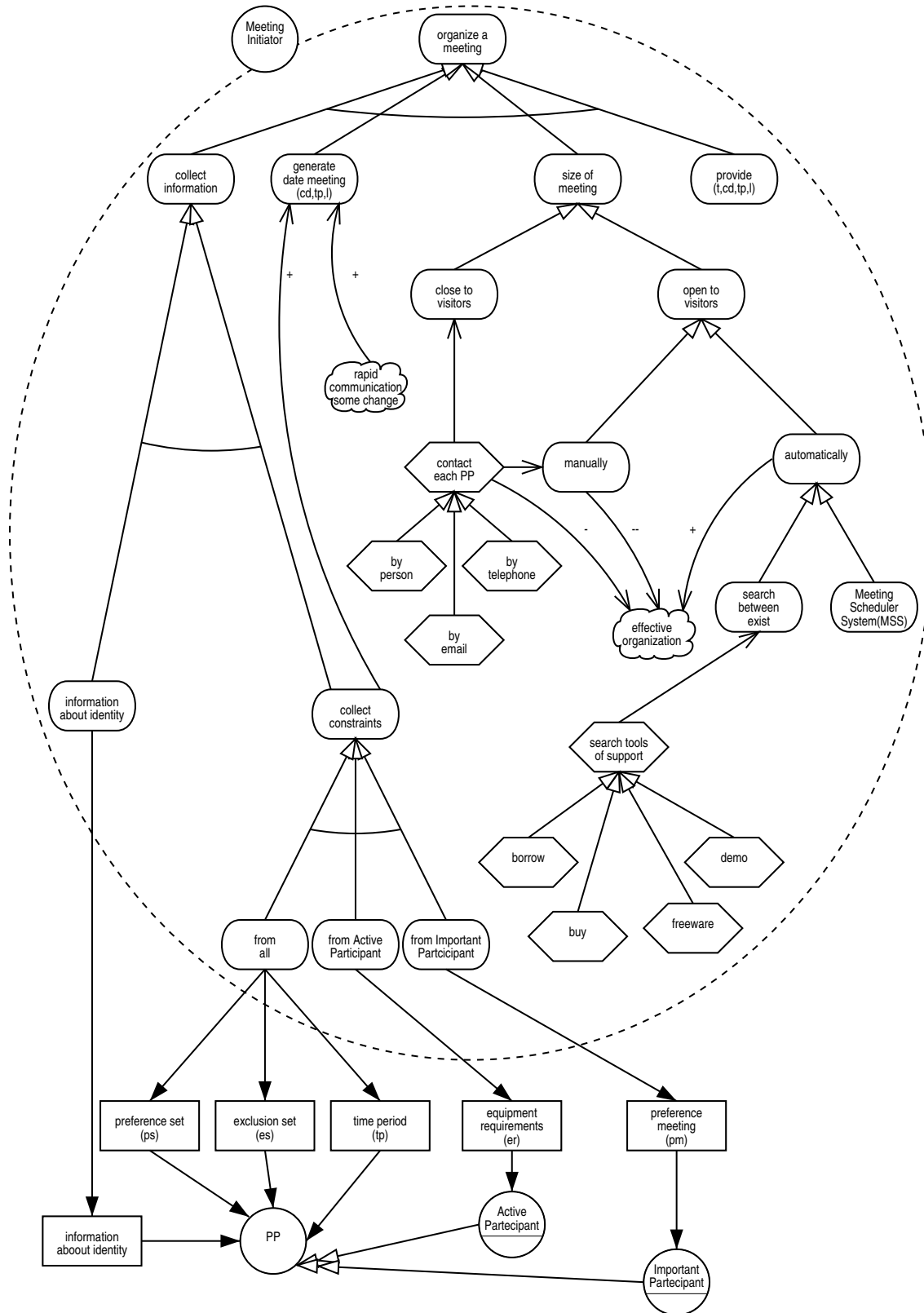
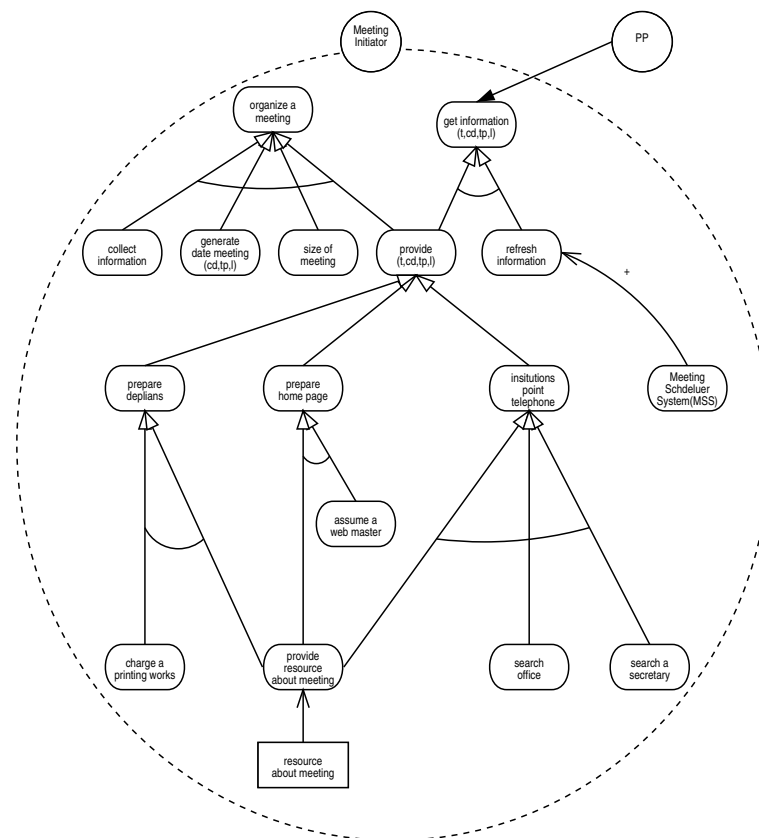


Figura 4.3: Diagramma dei goal costruito dal punto di vista dell'attore Meeting Initiator parte prima.

sato. Il plan da eseguire per poter organizzare un meeting “chiuso ai visitatori” contribuisce con metrica “-” al softgoal *effective organization* che è stato introdotto ancora precedentemente nel diagramma di figura 4.1.

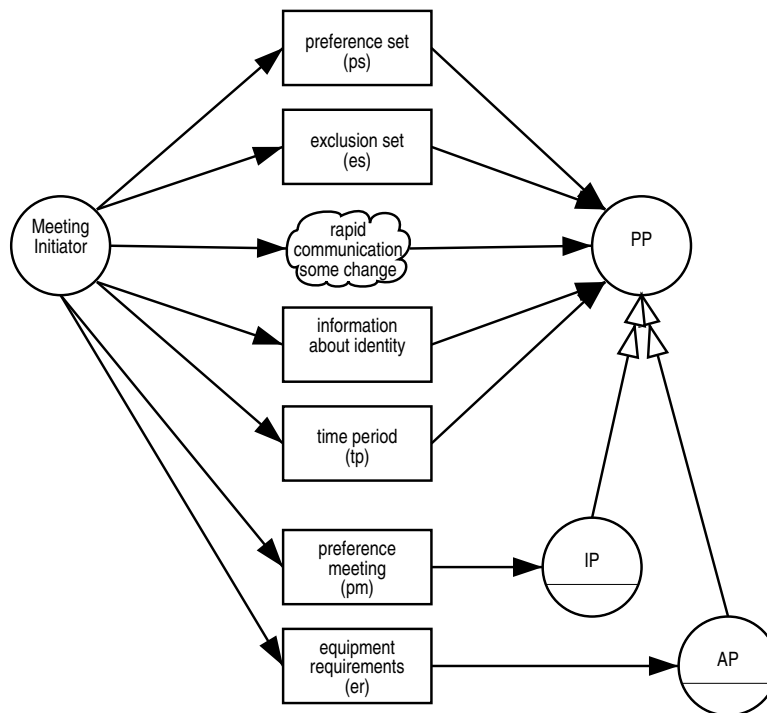
Se invece l’obiettivo è una manifestazione aperta alla partecipazione di molte persone provenienti da luoghi differenti, allora è necessario capire se è fattibile organizzarla manualmente (manually) oppure se sia necessario l’intervento di una qualche forma di agente esterno che automatizzi tale processo (automatically). Nel primo caso la complessità dell’operazione aumenta vistosamente perché contattare ciascun partecipante manualmente diventa faticoso oltreché dispendioso in termini di tempo, finanziario e risorse umane. Infatti, questa ipotesi di lavoro contribuisce (*Goal-SoftGoal Contribution*) assai negativamente al softgoal *effective organization*. Nel secondo caso invece esiste la possibilità di progettare ed implementare un sistema ex-novo, Meeting Scheduler System (MSS), che ha il compito di automatizzare tutte le procedure atte a promuovere il meeting o, in alternativa (ma non in OR esclusivo), cercare dei tools di supporto magari affittando, comperando, ricercando fra quelli freeware o delle demo.

L’ultimo subgoal che partecipa al soddisfacimento del goal radice è *provide (t,cd,tp,l)*. La sua analisi è visualizzata in figura 4.4. Per chiarezza sono riportati i tre subgoal che abbiamo



**Figura 4.4:** Diagramma dei goal costruito dal punto di vista dell’attore Meeting Initiator parte seconda.

già descritto nel frammento precedente del diagramma senza, ovviamente, le loro decomposizioni. Qui vogliamo porre l'attenzione sul subgoal rimanente, *provide (t,cd,tp,l)* e sulla dipendenza che PP ha delegato al MI con *dependum get information (t,cd,tp,l)* (vedere figura 4.2). La AND-decomposition compiuta su quest'ultimo ha evidenziato come il suo soddisfacimento è legato a quello del quarto subgoal, *provide (t,cd,tp,l)*, e al goal *refresh information* che modella la necessità che le informazioni comunicate ai partecipanti debbano essere continuamente aggiornate per consentire comunicazioni celeri; a questo contribuisce la progettazione di un MSS. La domanda da porsi nell'ottica del MI è la seguente: "Come posso fornire le informazioni riguardo il meeting a tutti i potenziali partecipanti?". Le strade da seguire possono essere tre: preparare dei deplians da distribuire, preparare una pagina web oppure istituire un punto telefonico al quale rivolgersi. *prepare deplians* implica attribuire l'incarico ad una stamperia (*charge a printing works*) e fornire le informazioni su cosa stampare (*provide resource about meeting*); il mezzo per far ciò è la risorsa *resource about meeting*. Nel caso in cui si proceda verso la costruzione di una o più pagine web è utile ingaggiare un professionista del web (*assume web master*) oltre a fornire le risorse. Infine, se si desidera avere un punto telefonico, le attività da fare diventano tre: ricerca di una persona che risponda al telefono, di un ufficio e disporre delle risorse precedentemente indicate.



**Figura 4.5:** Diagramma degli attori riassuntivo tra Meeting Scheduler e Potential Participant.

L'output di questa fase è un piccolo deliverables composto dall'Actor diagram di figura 4.5.



Sono riportati i social actor con le rispettive dipendenze individuate nel corso degli *early requirements*. Nel diagramma 4.3 ne individuamo quattro modellate come delle risorse per i potenziali partecipanti, più il softgoal che contribuisce positivamente alla determinazione della data del meeting (*rapid communication some change*). Come abbiamo già spiegato al paragrafo 3.3.1, se nel diagramma è presente un nodo senza figli, questo o viene delegato ad un altro attore oppure viene preso a proprio carico. È il caso del softgoal *rapid communication some change* che è assegnato al PP poiché, per promuovere una manifestazione, l'organizzatore sollecita tutti gli iscritti al meeting a comunicare celermente eventuali mutamenti nelle loro esigenze attraverso tale dipendenza. Infine, a seconda del ruolo, i PP che sono AP ereditano la dipendenza *er* mentre se sono IP ereditano *pm*.

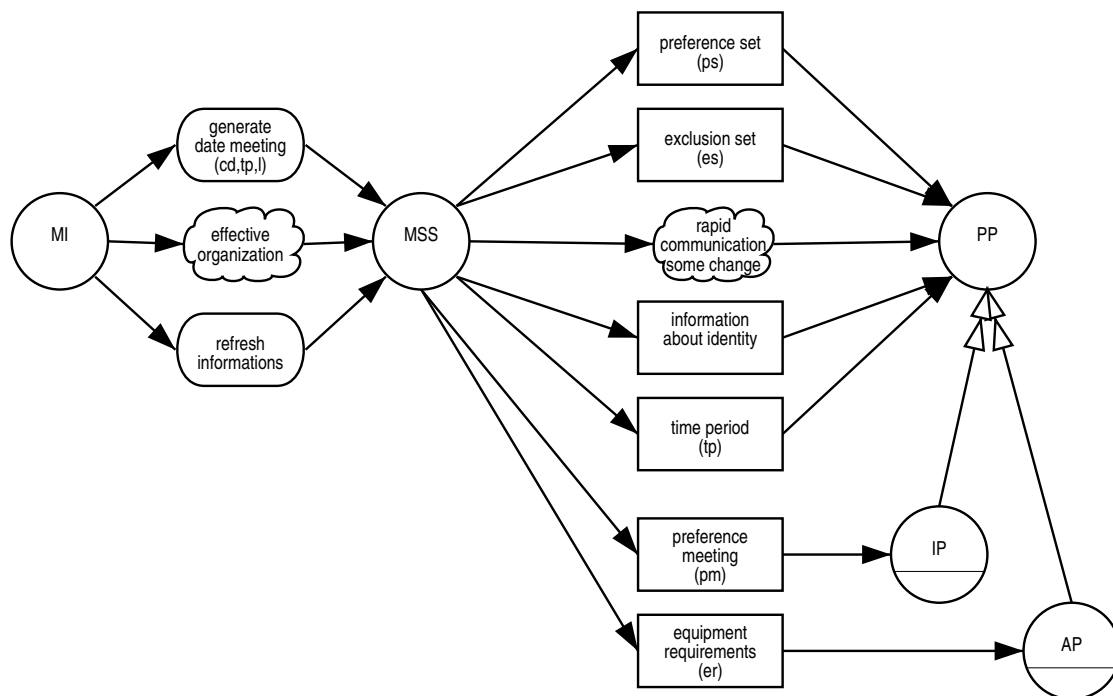
#### 4.2.2 Analisi del sistema futuro

Con la sezione 4.2.1 abbiamo concluso l'analisi dell'*environment* e degli stakeholder che vi partecipano. Adesso si tratta di definire quali sono, a nostro avviso, i *system actor*, cioè gli attori propri del sistema software che sarà costruito (*system-to-be*) e specificare in quale relazione stanno con i *social actor* introdotti nella fase precedente.

Supponiamo ora che all'organizzatore del meeting spetti il compito di promuovere un evento che richiama parecchi partecipanti e personalità importanti. In questo caso appare realistico delegare il compito ad un sistema software con determinati requisiti che, almeno in parte, sono stati già specificati. La nostra scelta ricade su questa opzione poiché, ad esempio, promuovere un meeting interno all'IRST probabilmente non giustifica lo sforzo di implementare un sistema software. Questa considerazione è figlia dell'analisi dei diagrammi precedenti. Infatti, la figura 4.3 fornisce indicazioni esplicite in tal senso: è sufficiente considerare che in caso di evento aperto ai visitatori il contributo fornito da un'organizzazione senza l'apporto di un sistema automatizzato viene da noi valutato molto negativamente (metrica “--”). Principalmente per questo motivo introduciamo l'attore di sistema Meeting Scheduler System (MSS) cui assegnamo tutte le dipendenze che prima gravavano su MI<sup>1</sup>.

Inoltre, ci sono delle nuove dipendenze che sempre MI delega al MSS. La loro provenienza è presto detta: riprendiamo la figura 4.3 laddove si discute del subgoal *generate date meeting* (*cd, tp, 1*). Abbiamo sottolineato che esso riceve due contributi da altrettanti goal, ma non abbiamo compiuto nessun tipo di decomposizione in AND-OR oppure means-ends analysis perché, a nostro avviso, non era ancora matura l'idea di come promuovere un meeting e, soprattutto, ancora non conoscevamo il carico di lavoro che implicava l'organizzazione di una manifestazione. Avendo compiuto la scelta di introdurre il sistema software, la conseguenza più immediata è quella di affidargli l'incarico. Anche il softgoal *effective organization* merita di essere approfondito dal punto di vista del MSS e non più dal MI poiché questo comporterà, presumibilmente,

<sup>1</sup>Sebbene in [6] le trasformazioni siano state applicate solo agli *early requirements*, qui le vogliamo sperimentare anche nei *late requirements*.



**Figura 4.6:** Diagramma degli attori che visualizza le dipendenze fra l'attore di sistema Meeting Schedule System e gli attori sociali MI e PP.

requisiti hardware e software specifici. L'ultimo goal delegato nasce dall'analisi dell'altro frammento di Goal diagram in figura 4.4. Tenendo conto anche dei requisiti posti nell'articolo [42] che descrive il dominio applicativo, risulta importante garantire continui aggiornamenti di informazioni che permettono ai partecipanti di avere costantemente informazioni sull'organizzazione del meeting e su eventuali cambiamenti. La porzione rimanente del grafo scaturita dall'analisi del subgoal provide  $(t, cd, tp, l)$  rimane di competenza del MI il quale dovrà pertanto provvedere a scegliere quale strada percorrere nel tentativo di soddisfare tale goal. La figura 4.6 riassume tutte le relazioni esistenti dopo queste considerazioni in un Actor diagram introducendo il *system actor* MSS.

Come per gli *early requirements*, il passo successivo prevede l'esplorazione dei goal degli attori (stavolta però di sistema). Il primo che abbiamo trattato è *generate date meeting*  $(cd, tp, l)$ . Per soddisfarlo pensiamo siano sufficienti ulteriori due goal, *identify*  $(cd, tp)$  e *identify location* posti in AND mediante la trasformazione *Goal AND-Decomposition*. Per identificare la data e le ore adatte a svolgere il meeting si richiede il soddisfacimento di quattro subgoal: *search strong conflict*, *search weak conflict*, *resolve conflict*, *decide date range*  $(cd)$  e *decide start and stop*  $(tp)$ . Nel descrivere il dominio teorico si accenna a due tipologie di conflitti: "forte" e "debole". Entrambi hanno una loro precisa connotazione e quindi è necessario individuarli per poi adottare la politica di risoluzione migliore. Un conflitto

è “forte” (*strong*) quando nessuna data può essere trovata all’interno delle ps, del tp ed esternamente all’es (le es contengono le date indesiderate, quindi si devono cercare esternamente a tale insieme); è “debole” (*weak*) quando si trovano delle date possibili all’interno di ps, di tp ed esternamente a es di ciascun partecipante, ma intersecando tutte le ps il risultato è un insieme vuoto. I due piani che fungono da “mezzi” servono proprio per la determinazione del tipo di conflitto.

Per risolvere eventuali conflitti vengono proposte alcune strategie che sotto riportiamo:

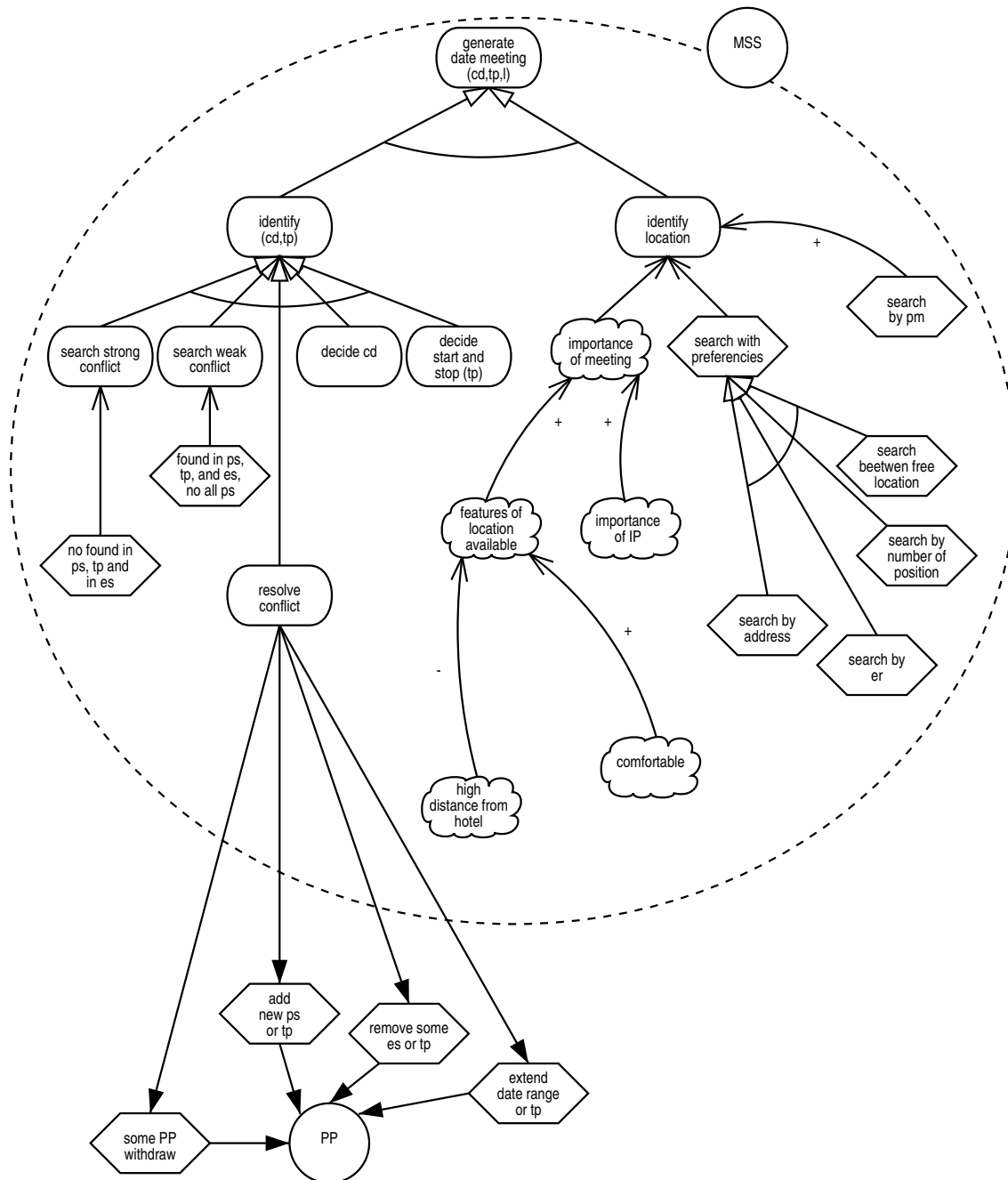
- alcuni PP si ritirano dal meeting (some PP withdraw);
- vengono aggiunte nuove ps o tp (add new ps or tp);
- vengono rimosse alcune es o tp (remove some es or tp);
- viene esteso il date range oppure tp (extend date range or tp).

Tutte queste possibilità sono modellate attraverso plan indirizzati verso i partecipanti i quali dovranno esprimere le loro decisioni in tal senso. Per completare l’analisi del goal identify (cd, tp) bisogna decidere la data e l’ora dell’evento, rispettivamente decide cd e decide start and stop (tp). Con questo, lo studio della sezione a sinistra del grafo è concluso.

L’altro subgoal che partecipa al soddisfacimento del goal iniziale è identify location. Per determinare il luogo adatto ad accogliere gli ospiti è utile valutare la portata del meeting (importance of meeting). Da un punto di vista prettamente operativo è necessario ricercare la sala una volta fissate le caratteristiche che il locale dovrà possedere (search by preferences). Si va dalla disposizione urbana della sala conferenze (search by address), alla disponibilità dell’equipaggiamento richiesto dai partecipanti attivi (search by er), al numero di persone che può ospitare (search by number of position). Il tutto, ovviamente, fra quelle libere e ancora affittabili (search by beetwen free location). Nel fare la scelta è auspicabile tenere presente anche le proposte avanzate dai partecipanti importanti (search by pm).

Il prossimo diagramma (figura 4.8) che andiamo a commentare mostra l’attore MSS che analizza il softgoal effective organization che ha ereditato dal MI. La means-ends analysis condotta su questo ha portato ad individuare dei mezzi che, almeno a nostro giudizio, possono consentire il suo soddisfacimento. Uno degli obiettivi è garantire che l’accesso alle informazioni sia consentito alle sole persone autorizzate. Per questo abbiamo introdotto il goal enforce privacy rules. Questo viene decomposto in avoid access db non-privileged il quale riceve due contributi entrambi con metrica “++” dai softgoal, security e robustness; se questi risultano soddisfatti, per quanto detto nel capitolo 3, anche il loro goal root rimane soddisfatto automaticamente. Il goal institution check point FireWall contribuisce positivamente alla sicurezza del sistema software così come il softgoal effective criptography.

Il goal successivo, manage interactions PP, ha lo scopo di gestire le interazioni fra i vari partecipanti. Lo spettro di interazioni è ampio: contattare i PP che non rispondono a sollecitazioni



**Figura 4.7:** Diagramma dei goal costruito dal punto di vista del Meeting Schedule System parte prima: sono analizzati l'hardgoal generate data meeting (cd, tp, l) e il softgoal effective organization.

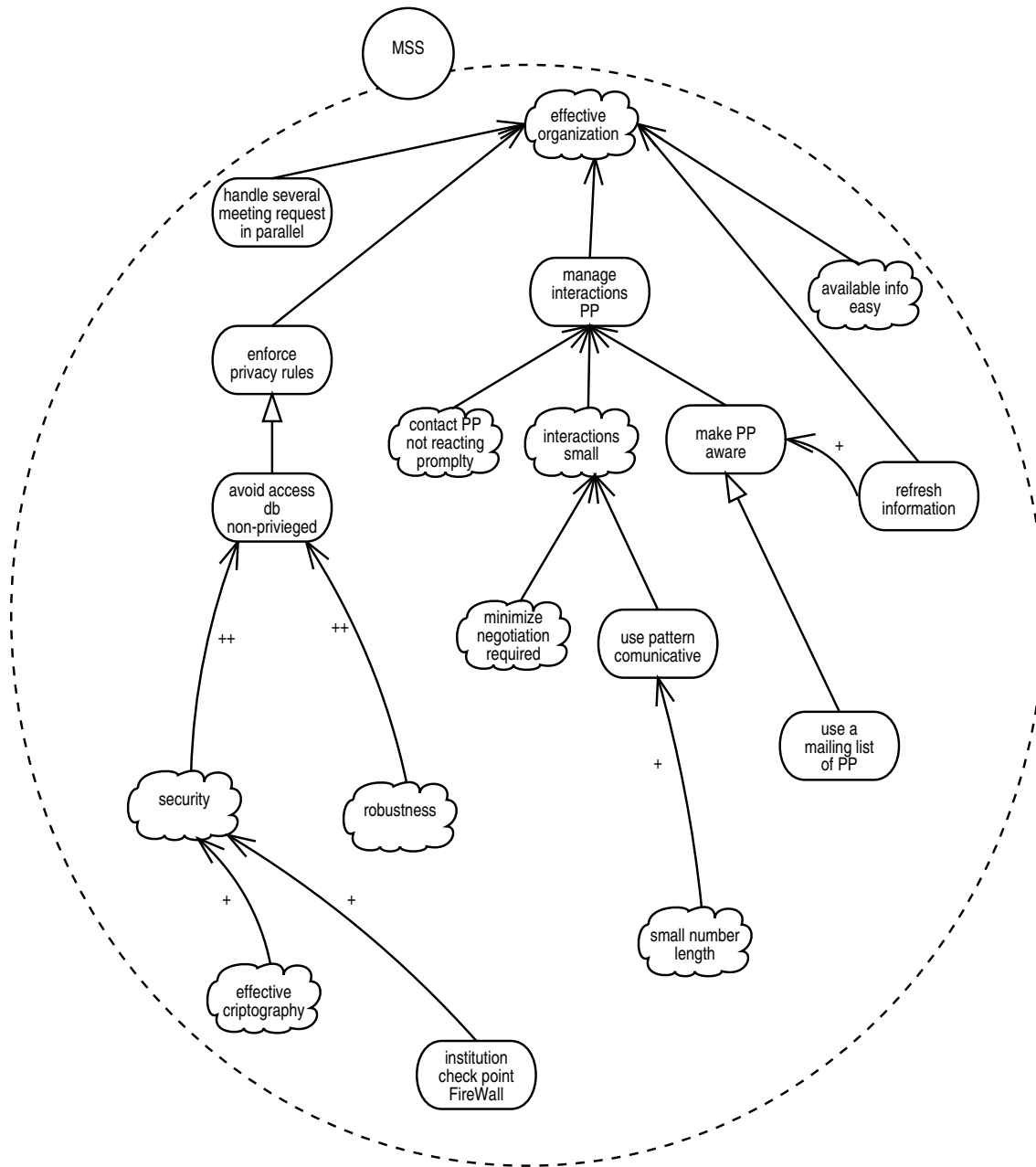
di qualunque genere (per esempio un richiesta di nuove informazioni o nuove date), tenere costantemente aggiornati i partecipanti, ecc. Certamente per facilitare la gestione dei messaggi è utile ridurre le interazioni (*interactions small*); questo può essere fatto ad esempio minimizzando le richieste di discussione fra i partecipanti e usando dei *patterns* comunicativi in modo da fissare la struttura dei messaggi e velocizzare anche le risposte. Per mantenere i partecipanti costantemente aggiornati sull'organizzazione del meeting (*make PP aware*) potrebbe essere utile l'uso di una mailing list. Un contributo positivo proviene dal goal *refresh information* che, fra l'altro, è uno dei subgoal che consente il soddisfacimento del softgoal *effective organization*.

I goal rimanenti *handle several meeting request in parallel* e *available info easy* sono analizzati, per motivi di spazio, in figura 4.9. Il primo viene decomposto in *manage concurrency*. Un partecipante non può essere contemporaneamente a due o più meetings così come una sala non può ospitare due o più riunioni. Queste ovvie considerazioni sono evidenziate dalla AND-decomposition tramite *avoid overlapping person* e *avoid overlapping room*. Entrambi ottengono contributi dal softgoal *elapsed time short* per indicare che:

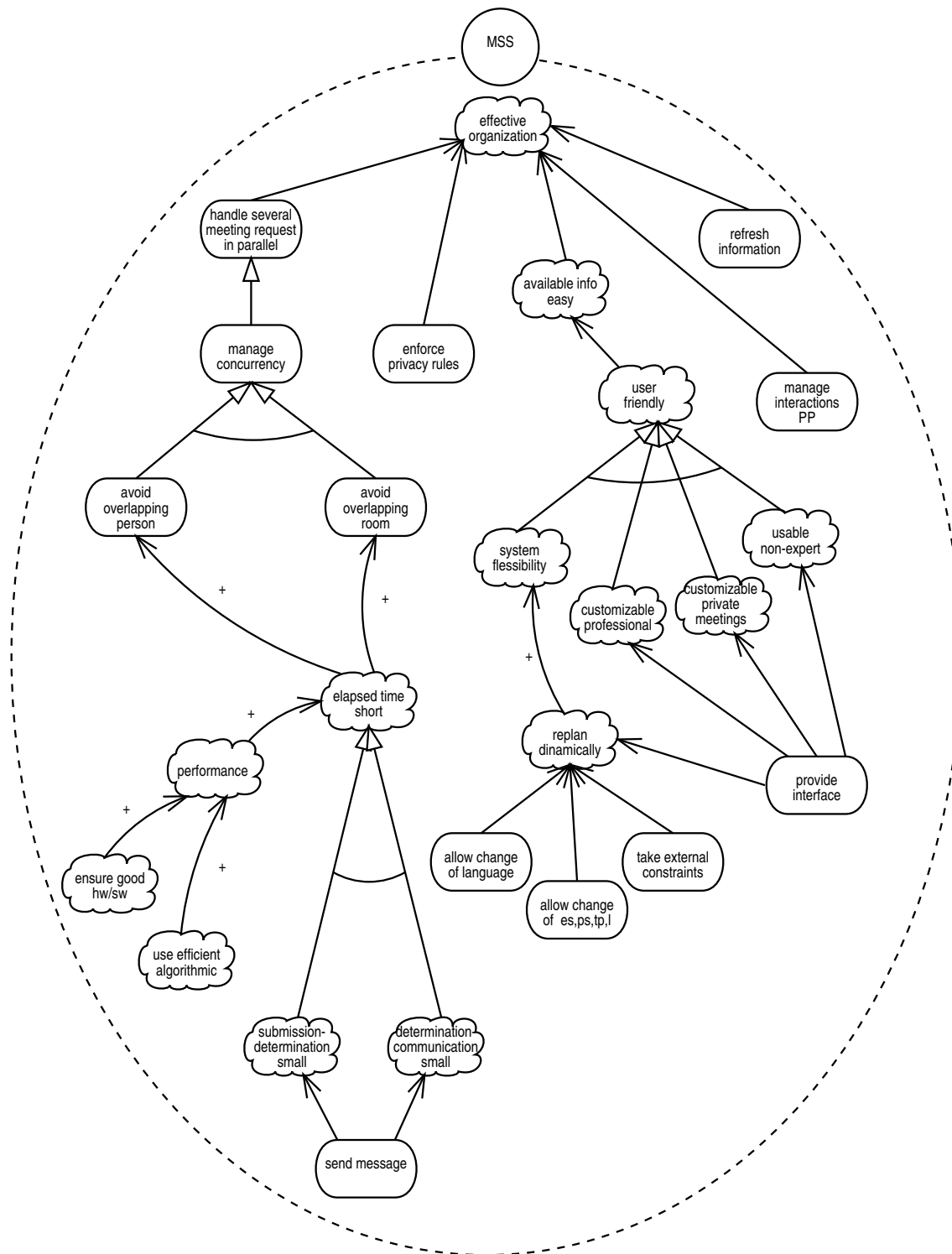
- il tempo che trascorre fra la richiesta di partecipazione ad un meeting e la determinazione del luogo e della data è breve (*submission-determination small*);
- il tempo che trascorre fra la determinazione del luogo e la data e la comunicazione a tutti i partecipanti è breve (*determination-communication small*).

Per entrambi i softgoal il *mezzo* di risoluzione è spedire rapidamente dei messaggi a tutti gli iscritti in modo che possano segnare sulla propria agenda l'impegno (*send message*). Concretamente, uno dei requisiti è garantire delle buone prestazioni (*performance*) in termini di hardware e software (*ensure good hw/sw*), l'utilizzo di algoritmi efficienti (*use efficient algorithmic*) ed altre qualità che non menzioniamo visto che non sono particolarmente rilevanti ai fini della discussione.

Il subgoal *available info easy* ha il compito di rendere facilmente accessibile il sistema software e, soprattutto, fornire informazioni circa il meeting in maniera semplice e disponibile a tutti, anche ai meno esperti. Per questo il mezzo è il softgoal *user friendly*. La sua decomposizione porta alla determinazione di quattro softgoal posti in AND: *system flessibility*, *customizable professional*, *customizable private meetings* e *usable non-expert*. Il sistema deve essere *flessibility*, possedere cioè un certo grado di flessibilità che gli consenta, ad esempio, di ripianificare dinamicamente (*replan dynamically*) in tempo reale ogni meeting che si trova a dover organizzare. Per fare questo il MSS deve permettere all'utente di cambiare la lingua a seconda delle sue preferenze, di modificare o integrare le sue preferenze in tema di *es*, *ps*, *tp* e *l* e di considerare anche dei vincoli esterni che potrebbero sopraggiungere una volta fissata la manifestazione. Il software deve essere configurabile a seconda che si tratti di eventi gestiti da privati cittadini (per esempio un appuntamento informale che si svolge dopo cena) oppure da professionisti. Infine, è richiesto che sia facilmente usabile anche dagli utenti non esperti. In tutti



**Figura 4.8:** Diagramma dei goal costruito dal punto di vista del Meeting Schedule System parte seconda: sono analizzati tre dei cinque goal che partecipano al soddisfacimento del goal *root*.



**Figura 4.9:** Diagramma dei goal costruito dal punto di vista del Meeting Schedule System parte terza: sono analizzati i due goal rimanenti, handle several meeting request in parallel e available info easy.

questi casi il mezzo opportuno è quello di fornire un'interfaccia che soddisfi i requisiti appena documentati; l'obiettivo è dunque provide interface.

Qui si chiude la macro fase di raccolta e modellazione dei requisiti. Il passo successivo ha lo scopo di individuare l'architettura logica che consente di arrivare ad implementare un sistema software basato sulla tecnologia MAS che realizzi il Meeting Scheduler System.

## 4.3 Progettazione del sistema

### 4.3.1 La fase architetturale globale

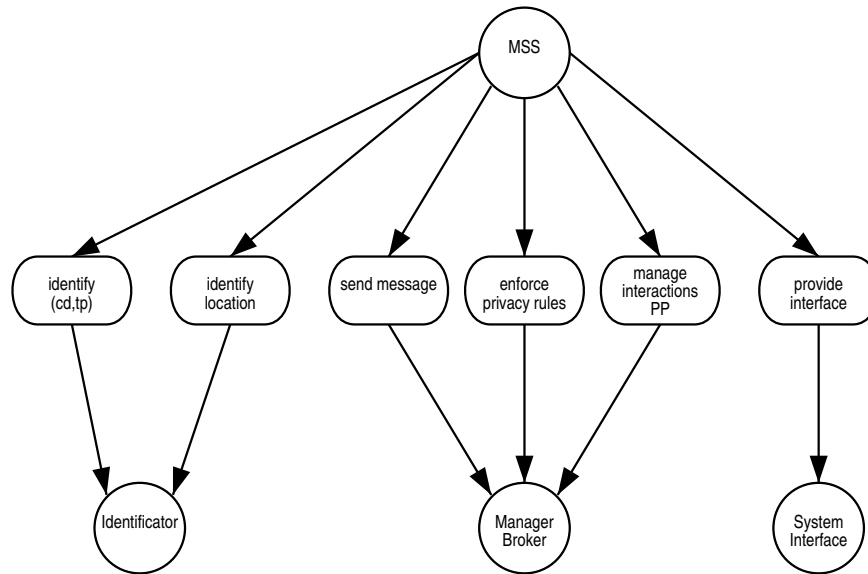
L'obiettivo di questo passo è quello di progettare l'architettura logica globale del sistema software mediante lo studio dei diagrammi che le due fasi precedenti hanno prodotto. Come discusso nel capitolo 2 l'*architectural design* è composto di tre passi: aggiunta di nuovi attori di sistema (compresi i subactor), identificazione e assegnazione delle capability agli attori e agentificazione degli stessi.

La prima fase dell'*architectural design* si apre con la figura 4.10 che riassume, a partire dal contributo fornito dagli ultimi diagrammi dei *late requirements*, gli attori e sottoattori di sistema e le loro dipendenze. L'attore Identifier ha ricevuto dal MSS i goal identify (cd,tp) e identify location ereditando le rispettive decomposizioni: può essere che egli desideri intergrarle in base al proprio punto di vista (entrambi i subgoal sono stati analizzati in figura 4.7). Manager Broker (MB) ha assunto l'onere di soddisfare tre goal (vedere figure 4.8 e 4.9): send message, enforce privacy rules e manage interactions PP. Il motivo che ha spinto a cedere tutti e tre gli obiettivi è presto spiegato: a nostro avviso, il futuro organizzatore di meeting dovrà avere solo compiti di coordinamento fra i vari agenti. Nella descrizione del dominio applicativo si dichiara esplicitamente che queste materie (sicurezza dei dati, interazioni fra i partecipanti e spedizione sempre a questi di eventuali messaggi e/o richieste di informazioni) sono di primaria importanza e pertanto vanno approfondite e curate attentamente. L'ultimo balloon dei late requirements identifica il goal provide interface come mezzo per soddisfare un insieme di softgoal; adesso questo viene dato in consegna all'attore System Interface.

L'attore di sistema Manager Broker ha collezionato tre goal per conto del MSS; in particolare, a nostro giudizio, per poter garantire un certo grado di sicurezza del sistema è utile che il goal enforce privacy rules sia delegato ad un subactor che introduciamo ora, Gate Manager (GM) (figura 4.11). L'altro subgoal, send message, è delegato a Sender Manager (SM) il quale si occuperà di tutte quelle situazioni in cui si dovranno contattare i partecipanti per qualunque motivo, dalla richiesta di informazioni aggiuntive alla comunicazione della data di svolgimento dell'appuntamento.

Infine, analizzando il diagramma 4.7 si può notare come uno dei subgoal che consentono il soddisfacimento di identify (cd,tp) è resolve conflict: dal punto di vista dell' "identifi-





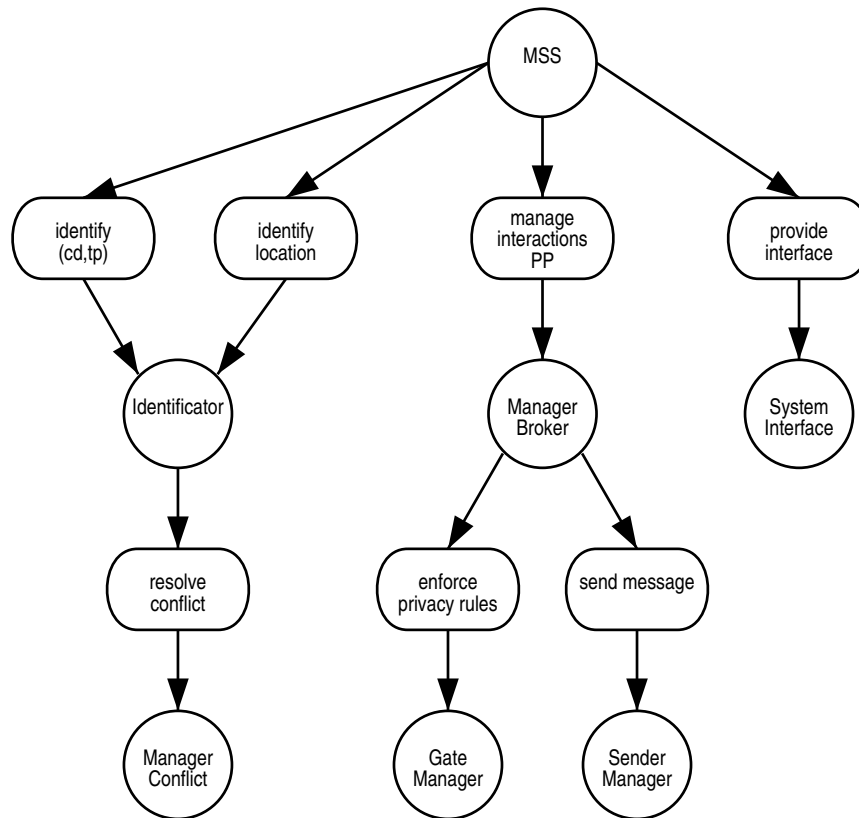
**Figura 4.10:** Diagramma degli attori riassuntivo uscente dagli *early* e *late requirements*: primo passo dell'*architectural design*.

catore” il carico di lavoro per risolvere quest’ultimo subgoal è eccessivo. Per questo viene delegato ad un nuovo subactor (Manager Conflict).

Il secondo passo dell'*architectural design* volge all’identificazione delle *capability*. Il diagramma 4.12 mostra un’estensione dell’Actor diagram con alcuni attori già introdotti precedentemente ed altri che spiegheremo fra breve. Il risultato di questo passo è legato principalmente alla sensibilità del progettista e alla sua visione del software. Per capire come sia stato possibile completare questo passo è utile affiancare al diagramma anche la tabella 4.1 in cui sono riportati gli attori e le rispettive *capability*.

Sender Manager deve essere in grado di chiedere e ricevere delle query dal Manager Information il quale gestisce tutte le informazioni riguardanti il meeting. Per questo necessita della *capability* *get query result* con la specifica del tipo di query che egli desidera. In questo caso si tratta di *information*, cioè le informazioni che poi dovranno essere spedite ai partecipanti (la sintassi della *capability* è: *name-of-capability: type-of-query*). Questi ultimi dipendono da SM per ricevere delle informazioni importanti; ad esempio, la richiesta di date aggiuntive per risolvere eventuali conflitti. Per questo Sender Manager deve possedere la capacità di spedire al PP dei messaggi specifici (*send important information*).

L’altro attore Gate Manager ha come scopo principale quello di filtrare i messaggi in entrata in modo da proibire l’accesso agli utenti non autorizzati. Prendiamo il caso dello User Manager (UM) il quale mediante login e password deve gestire l’accesso a solo certi utenti che deside-

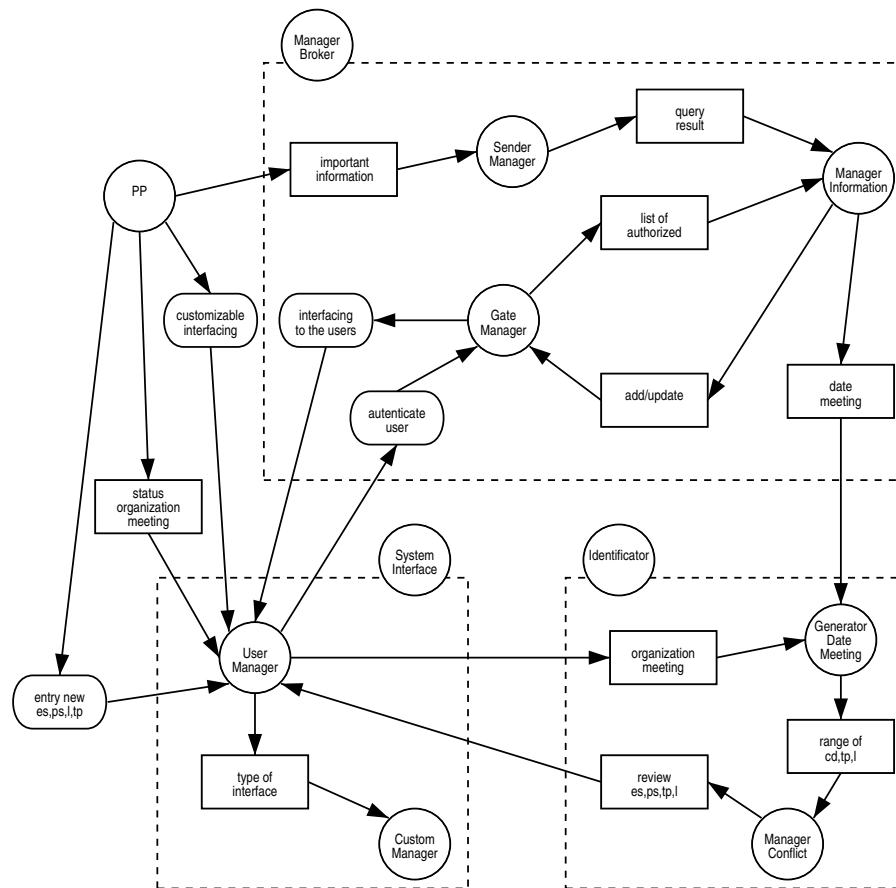


**Figura 4.11:** Chiusura del primo passo dell'*architectural design*: sono introdotti nuovi attori di sistema con relative dipendenze.

rano modificare e/o aggiornare i propri dati in maniera sicura. Per questo lo UM deve inoltrare login e password al GM il quale provvede a verificare se tale utente ha il diritto di accedere al sistema (`get query result:list of authorized`). Se la verifica ha dato esito positivo l'utente può modificare a suo piacimento le informazioni contenute nel database gestito da Manager Information (ad esempio può rivedere le date preferite, il luogo, la strumentazione, ecc). Il GM dispone di due capability: `get login/password` per ricevere login e password e `provide result` per comunicare all'interfaccia grafica l'esito della verifica.

Il Manager Information gestisce i database e, più in generale, le strutture dati che contengono le informazioni circa il meeting e i suoi partecipanti (`manage database`). Sono la fonte principale da cui può essere reperito qualunque tipo di materiale. Inoltre deve rispondere alle query che gli altri attori gli sottopongono (`respond query`).

L'Identificator possiede due subactor: Manager Conflict e Generator Date Meeting (GDM). Il primo ha lo scopo di risolvere eventuali conflitti proponendo al GDM un range di date, luoghi e orari che siano compatibili con le preferenze espresse dai PP (`propose range of cd,tp,l`). L'altra capability (`get review es,ps,tp,l`) ha l'obiettivo di ricevere dall'interfaccia



**Figura 4.12:** Secondo passo dell'architectural design: identificazione delle capability attraverso l'analisi delle dipendenze di ciascun *subactor*.

utente i parametri aggiornati o modificati nel caso che qualche partecipante abbia voluto o dovuto cambiare le sue preferenze. Il Generator Date Meeting deve essere in grado di richiedere  $cd, tp, l$  (`get cd, tp, l`) e fornire e/o aggiornare le proprie informazioni ai database e all'interfaccia una volta scelta la data del meeting (`add/update date meeting`).

System Interface attraverso User Manager visualizza lo status dell'organizzazione della manifestazione (`display status meeting`), richiede il meeting date e fornisce ai partecipanti i moduli per sottoscrivere (`provide form sub/unscripton`). Uno dei requisiti imposti nella fase precedente prevede che il sistema software sia facilmente personalizzabile a seconda che si tratti di un meeting organizzato da privati cittadini oppure da professionisti. Questa osservazione è recepita grazie all'attore Custom Manager il quale fornisce le interfacce adeguate allo User Manager.

L'ultimo passo dell'architectural design è detto *agentificazione*; si tratta di individuare gli agenti e le capability associate partendo dalla tabella costruita precedentemente (tabella 4.1). Gli agenti che identifichiamo sono quelli che poi saranno implementati nell'ultima fase della

<i>Nome-actor: subactor</i>	<i>N capability</i>	<i>Capability</i>
<b>Manager Broker:</b>		
Sender Manager	1.a	get query result:information
	2	send important information
Gate Manager	3	provide result
	4	get login/password
	1.b	get query result:list of authorized
	5	comunicate add/update
Manager Information	6	manage database
	7	respond query
<b>Identificator:</b>		
Manager Conflict	8	propose range of cd,tp,l
	9	get review es,ps,tp,l
Generator Date Meeting	10	add/update date meeting
	11	get cd,tp, l
<b>System interface:</b>		
User Manager	12	display status meeting
	13	get organization meeting
	14	provide form sub/unscription
Custom Manager	15	provide interface

**Tabella 4.1:** Passo 2: processo di *identificazione* delle capability.

metodologia Tropos (*implementation*).

La tabella 4.2 riepiloga quelli che a nostro avviso sono gli agenti e le loro capability.

All'agente Sender Agent (SA) vengono affidate le capability che già possedeva in qualità di attore (tabella 4.1), a cui si aggiunge la numero 13. All'agente Gate FireWall Agent (GFA) vengono affidate le capability numero 3, 4, 1.b e 5. Agent Resource Broker (ARB) modella il precedente attore Manager Information con, in più, la possibilità di richiedere quale siano le date, orario e luogo definitivo per svolgere l'evento (capability numero 13). Generator Date Agent (GDA) è il risultato dell'aggregazione degli attori Manager Conflict e Generator Date Meeting. In più, possiede anche la capability numero 5 che gli consente di comunicare all'Agent Resource Broker le date, ore e il luogo scelto per svolgere il meeting. Infine, abbiamo lo User Interface Agent (UIA) con tutte le proprietà che prima erano di competenza dei due attori User Manager e Custom Manager.

<i>Nome-agent</i>	<i>Capability assegnate</i>
<b>Sender Agent</b>	1.a, 2, 13
<b>Gate FireWall Agent</b>	3, 4, 1.b, 5
<b>Agent Resource Broker</b>	6, 7, 13
<b>Generator Date Agent</b>	10, 11, 8, 9, 5
<b>User Interface Agent</b>	12, 13, 14, 15, 1.a

**Tabella 4.2:** Passo 3: processo di *agentificazione*. Si identificano gli agenti e poi si assegnano le capability individuate al passo 2.

### 4.3.2 La fase di dettaglio architetturale

La fase di *Detailed design* ha lo scopo di specificare ciascun agente usando strumenti tipici di AUML quali activity diagrams e sequence diagrams. La scelta architetturale è rivolta all'approccio BDI.

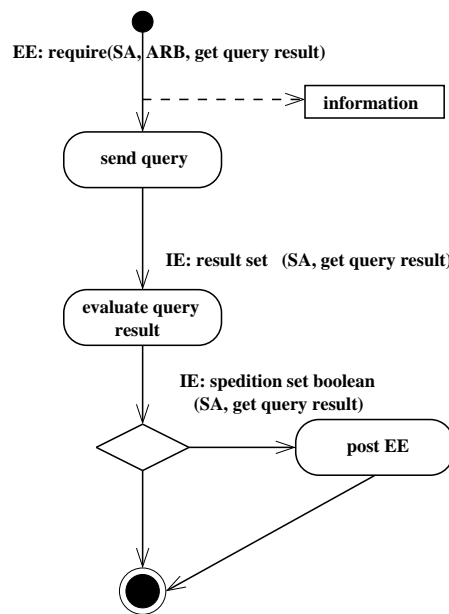
Abbiamo visto nel capitolo precedente che Tropos prevede per questa fase tre diagrammi (Capability diagram, Plan diagram e Agent Interaction diagram) i quali catturano, sotto varie forme, gli aspetti dinamici del sistema software che stiamo progettando.

Non è nostra intenzione esplorare tutte le capability che abbiamo individuato al passo 3 dell'architectural design ma solo alcune rappresentanti in maniera tale da far capire come i diagrammi vengono usati. Per questo motivo illustreremo la capability numero 1.a, il piano evaluate query result presente all'interno della capability suddetta e, infine, l'Agent Interaction diagram che mostra un tipico esempio di interazione fra vari agenti presenti nel sistema.

La capability 1.a ha il compito di fare una query all'agente ARB riguardo eventuali richieste di informazioni oppure comunicazioni urgenti. L'agente SA attiva il piano send query in presenza dell'evento esterno EE: `require(SA,ARB,get query result)` dopo aver letto il tipo di query da sottoporre a ARB. Nel caso specifico SA deve comunicare urgentemente con i PP per chiedere nuove informazioni.

Dopo aver spedito la query l'agente attende una risposta che, se arriva, permette di valutare la sua consistenza (`evaluate query result`). A questo punto l'agente può prepararsi a spedire le informazioni ai PP oppure terminare nel caso in cui l'esito della query sia stato fallimentare (ARB non risponde, le informazioni necessarie per contattare i PP sono insufficienti, nel frattempo MSS ha sospeso la procedura, ecc.).

La figura 4.14 mostra il Plan diagram riferito al piano evaluate query result. Il piano viene attivato dall'arrivo dell'evento interno IE: `result set(SA,get query result)`; in risposta a questo viene lanciata l'azione (o un insieme di azioni) che permette di leggere la risposta della query. A questo punto, il piano deve cercare le informazioni necessarie per poter inoltrare il messaggio ai partecipanti. Per questo verifica prima se la risposta contiene gli indirizzi di posta



**Figura 4.13:** Diagramma delle abilità di `get query result:information` (numero 1.a).

elettronica dei PP e, successivamente, il tipo di messaggio che deve spedire (aggiunta di nuove date, comunicazioni urgenti, ecc.). In entrambi i casi, se la risposta non contiene quanto detto sopra il piano setta una variabile booleana a *false*, altrimenti la setta a *true* intendendo che è pronto per spedire tutti i messaggi ai partecipanti interessati.

L'ultimo passo del *detailed design* consiste nello specificare i messaggi che i vari agenti (e attori dell'environment) si scambiano. A tal proposito presentiamo l'Agent Interaction diagram di figura 4.15.

L'attore MI inoltra la richiesta di organizzazione del meeting all'agente GDA mediante il messaggio `generator date meeting`. Come prima cosa, l'agente deve recuperare tutte le informazioni riguardanti i partecipanti. Per far ciò deve passare attraverso l'interfaccia grafica la quale, a sua volta, attende che i PP forniscano i propri *es*, *ps*, *tp* (qui presentiamo il caso semplice in cui non vi siano partecipanti attivi o cosiddetti importanti). I PP rispondono alla richiesta di UIA compilando la scheda relativa (`provide personal information (subscribe)`). La fase successiva prevede la risoluzione di eventuali conflitti ricorrendo ancora una volta alla comunicazione diretta con i PP se necessario (per brevità non riportiamo il caso in cui siano necessari più contatti con i partecipanti per risolvere i conflitti). A questo punto lo UIA comunica a ARB e a GDA le date immesse dai partecipanti. Una volta ottenute le informazioni e risolti i vari conflitti l'agente GDA può fissare la data di svolgimento del meeting e comunicarlo sia al ARB sia a UIA il quale provvederà a informare i partecipanti e l'organizzatore del meeting.

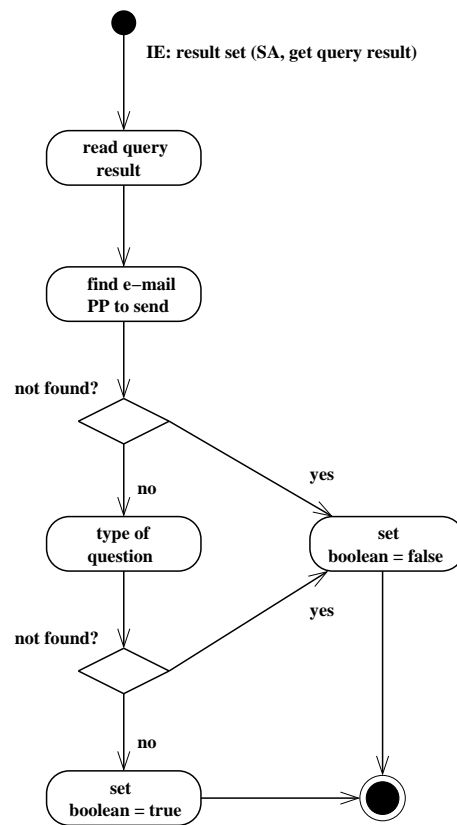
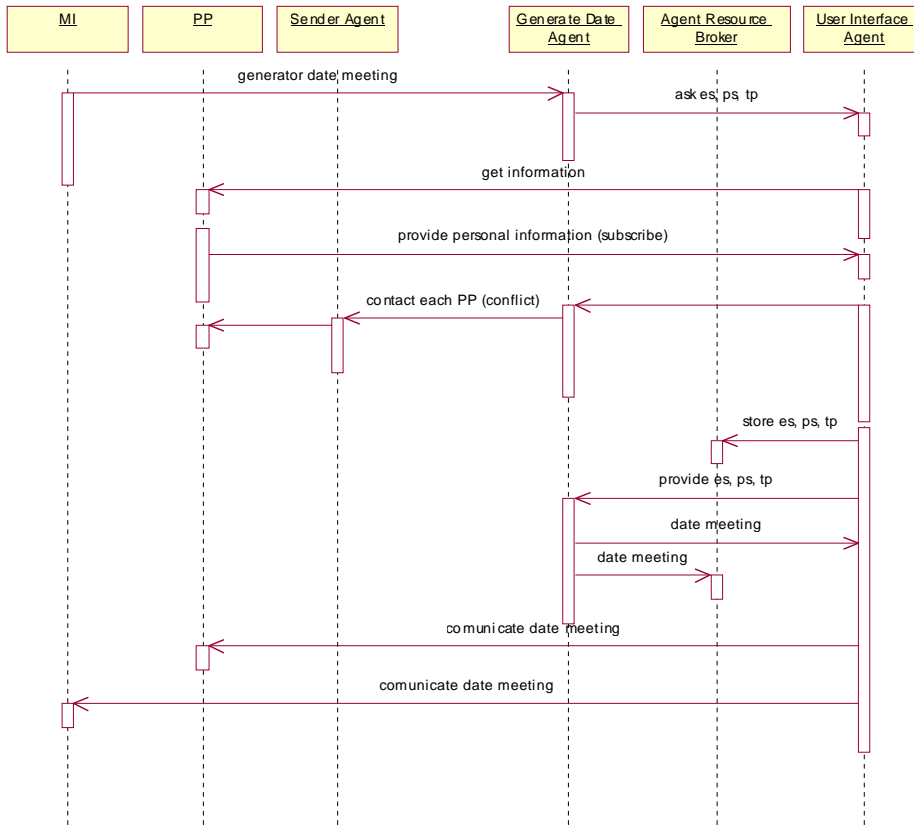


Figura 4.14: Diagramma dei piani riferito a evaluate query result.

## 4.4 Implementazione del sistema

L'ultima fase della metodologia non è stata approfondita per mancanza di spazio e, soprattutto, perché non era obiettivo di questo esercizio arrivare fino in fondo all'implementazione del case study *Meeting Scheduler System*.



**Figura 4.15:** Diagramma delle interazioni tra agenti fra alcuni agenti del sistema.



# Bibliografia

- [1] *Autonomous Agents: The Fifth International Conference on Autonomous Agents*, Montreal, Canada, may, 28 2001.
- [2] A. I. Antòn, J. H. Dempster, and D. F. Siege. Deriving Goals from a Use-Case Based Requirements Specification for an Electronic Commerce System.
- [3] G. Booch, J. Rambaugh, and J. Jacobson. *OMG Unified Modeling Language Specification*. Object Management Group, June 1999. Version 1.3.
- [4] G. Booch, J. Rambaugh, and J. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.
- [5] M. E. Bratman. Intentions in communications. Chapter 2.
- [6] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modelling early requirements in Tropos: a transformation based approach. In Wooldridge et al. [45], pages 151–168.
- [7] P. Busetta, R. Rönquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java. AOS Technical Report tr9901, jan 1999. <http://www.jackagents.com/pdf/tr9901.pdf>.
- [8] G. Caire, F. Leal, P. Chainhoand, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent oriented analysis using MESSAGE/UML. In *Proceedings of the Fifth International Conference on Autonomous Agents*, Montreal CA, May 2001.
- [9] L. Chung, S. Liao, W. Huaiqing, E. Yu, and J. Mylopoulos. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1), February 2001.
- [10] L. Chung and B. A. Nixon. Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach. In *Proceedings, 17th International Conference on Software Engineering (ICSE)*, Seattle, WA, pages 25–36, April, 24 - 28 1995.
- [11] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

- [12] P. Ciancarini and M. Wooldridge. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering – Proceedings of the First International Workshop (AOSE-2000)*, Limerick, Ireland, June 2000. Springer-Verlag Lecture Notes in Computer Science, Volume 1957.
- [13] M. Coburn. JACK Intelligent Agents User Guide. Technical report, Agent Oriented Software Pty. Ltd, 2000.
- [14] Michael Coburn. Jack intelligent agents user guide. AOS technical report, Agent Oriented Software Pty Ltd, July 2000. <http://www.jackagents.com/docs/jack/html/index.html>.
- [15] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.
- [16] R. Darimont, A. van Lamsweerde, and P. Massonet. Goal-Directed elaboration of requirements for a meeting scheduler: problems and lesson learnt. In *Proceedings RE'95 - 2nd IEEE Symposium on Requirements Engineering*, pages 194–203, York, March 1995.
- [17] H.-E. Eriksson and M. Penker. *UML Toolkit*. Wiley Computer, 1998.
- [18] J. Ferber and O. Gutknecht. Aalaadin: a meta-model for the analysis and design of organizations in multi-agent system. Technical report, Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, December 1997.
- [19] M. Fowler and K. Scott. *UML Distilled*. ADDISON - WESLEY, first edition, Settembre 2000. Aggiornata alla versione 1.3 OMG UML Standard.
- [20] A. Fuggetta. Webbook di Ingegneria del software, 2000. <http://www.cefriel.it/~alfonso/WebBook/index.htm>.
- [21] A. Fuggetta, C. Ghezzi, S. Morasca, A. Morzenti, and M. Pezzè. *Ingegneria del Software: Progettazione, Sviluppo e Verifica*. Mondadori Informatica, settembre 2000 edition, 1996.
- [22] A. Fuxman. *Formal analysis of early requirements specifications*. PhD thesis, University of Toronto, Toronto, Canada, 2001.
- [23] P. Giorgini, A. Perini, J. Mylopoulos, F. Giunchiglia, and P. Bresciani. Agent-oriented software development: A case study. Technical report, Centro per la Ricerca Scientifica e Tecnologica ITC-IRST, <http://sra.itc.it>, 38050 Povo (Trento), Italy, November 2000. This paper extends "Agent-Oriented Software Development: A Case Study", June 13 - 15.
- [24] P. Giorgini, A. Perini, J. Mylopoulos, F. Giunchiglia, and P. Bresciani. Agent-oriented software development: A case study. In S. Sen J.P. Müller, E. Andre and C. Frassen, editors, *Proceedings of the Thirteenth International Conference on Software Engineering - Knowledge Engineering (SEKE01)*, Buenos Aires - ARGENTINA, June 2001.

- 
- [25] F. Giunchiglia, J. Mylopoulos, and A. Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams. Technical Report No. 0111-20, ITC-irst, November 2001.
- [26] F. Giunchiglia, A. Perini, and F. Sannicolò. Knowledge level software engineering. In J.-J.C. Meyer and M. Tambe, editors, *Intelligent Agents VIII*, LNCS 2333, pages 6–20. Springer-Verlag, Seattle, WA, USA, Proceedings of the eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL) edition, August 2001. Also IRST Technical Report 0112-22, Istituto Trentino di Cultura, Trento, Italy.
- [27] N. R. Jennings and M. Wooldridge. Agent-Oriented Software Engineering. In *Handbook of Agent Technology*. (ed. J. Bradshaw) AAAI/MIT Press, 2000. See also <ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/publications/agt-handbook.pdf>.
- [28] J. Lind. Issues in Agent-Oriented Software Engineering. In *Agent-Oriented Software Engineering – Proceedings of the First International Workshop (AOSE-2000)*, pages 45–58, Limerick, Ireland, jun, 10 2000.
- [29] J. Mylopoulos and J. Castro. *Tropos: A Framework for Requirements-Driven Software Development*, pages 261–273. Lecture Notes in Computer Science. Springer-Verlag, 2000.
- [30] J. Odell, B. Bauer, and J. P. Müller. Agent UML: A formalism for specifying multiagent software system. In P. Cianciarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering – Proceedings of the First International Workshop (AOSE-2000)*, Limerick, Ireland, June, 10 2000.
- [31] J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for Agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proc. of Agent-Oriented Information System Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, Austin, TX, 2000.
- [32] Object Management Group (OMG). Agent Technology, Green Paper. OMG Document ec/2000-08-01, 1 August 2000. Version 1.0.
- [33] H. Van Dyke Parunak and J. Odell. Representing social structures in UML. In *Proceedings of the Fifth International Conference on Autonomous Agents*, Montreal CA, May 2001.
- [34] A. Perini, P. Bresciani, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Towards an Agent Oriented approach to Software Engineering. In *Proceedings of the Workshop, dagli oggetti agli agenti: tendenze evolutive dei sistemi software*, Modena, Italy, September 2001.
- [35] A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A Knowledge Level Software Engineering Methodology for Agent Oriented Programming. In *Proceedings of the Fifth International Conference on Autonomous Agents*, Montreal CA, 28 May - 1 June 2001.

- [36] C. Petrie. Agent-Based Software Engineering. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering – Proceedings of the First International Workshop (AOSE-2000)*, 2000.
- [37] M. Pistore, A. Fuxman, J. Mylopoulos, and P. Traverso. Model Checking Early Requirements Specifications in Tropos. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering (RE01)*, Toronto, Canada, August 2001.
- [38] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. Technical report, Australian Artificial Intelligence Institute, April 1995.
- [39] W. N. Robinson and S. Volkov. A Meta-Model for Restructuring Stakeholder Requirements. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, USA, May 17-24 1997. IEEE Computer Society Press.
- [40] F. Sannicolò. Tropos: una Metodologia ed un Linguaggio di Modellazione Visuale Semi-formale. Master's thesis, Università degli Studi di Trento - Facoltà di Scienze Matematiche Fisiche e Naturali, via Sommarive, Povo, Trento, December 2001. Also IRST Technical Report 0201-01, Istituto Trentino di Cultura, Trento, Italy.
- [41] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1), 1993.
- [42] A. v. Lamsweerde, R. Darimont, and P. Massonet. The Meeting Scheduler System - Problem Statement. Technical report, Université Catholique de Louvain - Département d'Ingénierie Informatique, B-1348 Louvain-la-Neuve (Belgium), October 1992.
- [43] A. van Lamsweerde and R. Darimont. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In *Proceedings 4th ACM Symposium on the Foundations of Software Engineering (FSE4)*, pages 179–190, San Francisco, October 1996.
- [44] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.
- [45] M.J. Wooldridge, G. Weiß, and P. Ciancarini, editors. *Agent-Oriented Software Engineering II*. LNCS 2222. Springer-Verlag, Montreal, Canada, Second International Workshop, AOSE2001 edition, May 2002.
- [46] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, University of Toronto, 1995.
- [47] E. Yu. Software Versus the World. In Wooldridge et al. [45], pages 206–225.
- [48] E. Yu and J. Mylopoulos. From E-R to “A-R” – modeling strategic actor relationships for business process reengineering. In P. Loucopoulos, editor, *Proceedings of 13th Int. Conf.*

*on the Entity-Relationship Approach (ER'94)*, number 881 in Lecture Notes in Computer Science, pages 548–565, Manchester, U.K., December 1994. Springer-Verlag.

- [49] E. Yu and J. Mylopoulos. Understanding ‘why’ in software process modeling, analysis and design. In *Proceedings Sixteenth International Conference on Software Engineering*, pages 159–168, Sorrento, Italy, May 1994.



# Appendice A

## Glossario e Trasformazioni

### A.1 Glossario

**Actor** è un'entità che possiede degli obiettivi strategici all'interno del sistema oppure dell'organizzazione che ospita il sistema software stesso; inoltre è sorgente di intenzionalità.

**Agent** è un attore con autonomia, abilità sociale, reattività, proattività.

**AND-OR decomposition** è un particolare tipo di analisi che permette di decomporre un goal detto *root* in subgoal posti in AND oppure in OR. AND-OR decomposition si specializza in AND-decomposition oppure OR-decomposition a seconda del tipo di analisi che si desidera condurre.

**Belief** rappresenta la conoscenza di un attore sullo stato del mondo.

**Capability** rappresenta l'abilità di un attore di definire, scegliere ed eseguire un insieme di piani per soddisfare dei goal.

**Contribution** è un particolare tipo di analisi che permette di identificare goal che possono contribuire positivamente o negativamente al soddisfacimento di un altro goal. La metrica utilizzata è --, -, +, ++.

**Dependency** è un'entità che rappresenta le relazioni esistenti fra i vari attori dell'ambiente o dell'organizzazione. Un attore dipende (depender) da un altro (dependee) per soddisfare un goal, eseguire un piano, disporre di una risorsa. L'oggetto del contendere è detto dependum.

**Entity** è un elemento che rappresenta concetti intenzionali e non-intenzionali che esistono nell'ambiente o nell'organizzazione.

**Goal** modella gli obiettivi che un attore vuole raggiungere (il vero motivo per cui un attore partecipa ad un sistema). Raggruppa sia il concetto di hardgoal sia quello di softgoal.

**Grafo diretto**  $G = \langle N, A \rangle$  è un insieme non vuoto  $N$  di nodi e una collezione  $A \subseteq N^2$  di coppie ordinate di nodi distinti di  $N$ . Ogni coppia di nodi è chiamata arco.

**Grafo diretto etichettato** è una coppia  $\langle G, \tau \rangle$  tale che:

- $G$  è un grafo diretto  $G = \langle N, A \rangle$
- $\tau : N \cup A \rightarrow \mathcal{L}$ , dove  $\mathcal{L}$  è un opportuno insieme di etichette.

**Hardgoal** è un goal che un attore può pienamente e potenzialmente soddisfare eseguendo uno o più piani, sostanzialmente si può definire un criterio in base al quale stabilire se esso è stato raggiunto o meno.

**Means-Ends analysis** è un particolare tipo di analisi che permettere di capire *cosa* sia necessario fare per soddisfare un goal (*end*) in termini di piani, risorse, hardgoal e softgoal (*mean*).

**Mode** è un attributo che specifica come soddisfare la dipendenza o il goal. Si specializza in *achieve, maintain, achieve&maintain, avoid*.

**N-ary Relationship** raggruppa tutte le possibili relazioni che si possono istanziare al domain level.

**Plan** modella cosa è possibile fare per soddisfare un goal, una volta scelta l'opportuna strategia. Un attore è *capace di scegliere, definire ed eseguire* dei piani.

**Position** rappresenta un insieme di ruoli, tipicamente ricoperti da un agente. Un agente può occupare una posizione, mentre una posizione ricopre un ruolo.

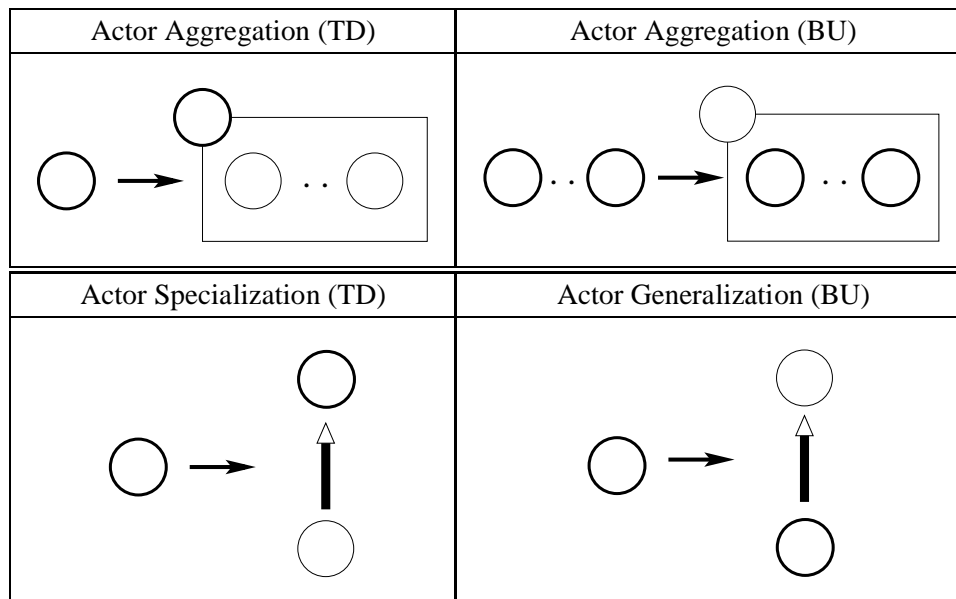
**Resource** rappresenta un'entità fisica o un contenitore di informazioni.

**Role** è una caratterizzazione più fine del comportamento di un attore nell'ambito di un insieme di interazioni (dipendenze, eventi comunicativi, ecc.).

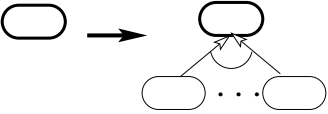
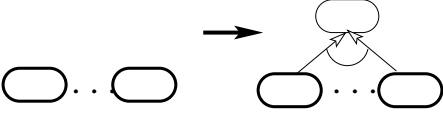
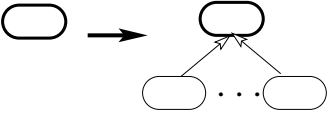
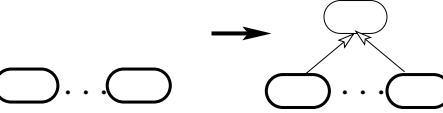
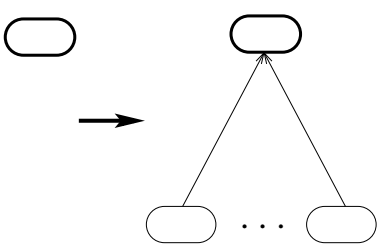
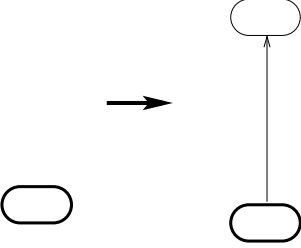
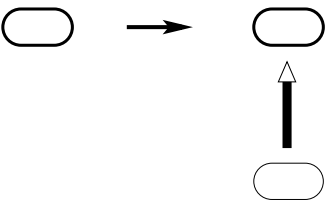
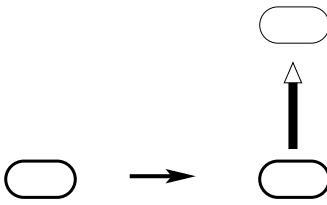
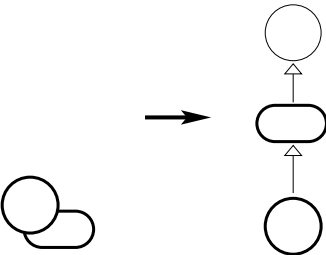
**Softgoal** è un goal del quale non è si riesce esattamente e con precisione ad affermare che è stato soddisfatto.



## A.2 Trasformazioni



**Tabella A.1:** Actor Transformations tratte da [6].

Goal AND-Decomposition (TD)	Goal AND-Composition (BU)
	
Goal OR-Decomposition (TD)	Goal OR-Composition (BU)
	
Precondition Goals (TD)	Precondition Goals (BU)
	
Goal Specialization (TD)	Goal Generalization (BU)
	
Goal Delegation	
	

**Tabella A.2:** Goal Transformations tratte da [6].

SoftGoal-Gol Contribution (TD)	Goal-SoftGoal Contribution (BU)
SoftGoal-SoftGoal Contribution (TD)	SoftGoal-SoftGoal Contribution (BU)
SoftGoal-AND Decomposition (TD)	SoftGoal-AND Composition (BU)
SoftGoal-OR Decomposition (TD)	SoftGoal-OR Composition (BU)
SoftGoal Specialization (TD)	SoftGoal Generalization (BU)
SoftGoal Delegation	

Tabella A.3: SoftGoal Transformations tratte da [6].

Goal Primitive Transformations	
Goal Introduction (G-I)	
Goal Precondition (G-P)	
Goal AND decomposition (G-DEC- <b>&amp;</b> )	Goal OR decomposition (G-DEC- <b>OR</b> )
Goal Delegation (G- <b>DEL</b> )	Goal Generalization (G- <b>G</b> )

**Tabella A.4:** Trasformazioni primitive per i goal tratte da [6].

SoftGoal Primitive Transformations	
Softgoal Introduction (SG-I)	
Goal-Softgoal Contribution (SG-C-G)	Softgoal-Softgoal Contribution (SG-C-SG)
Softgoal AND decomp. (SG-DEC- <b>&amp;</b> )	Softgoal OR decomp. (SG-DEC- <b>OR</b> )
Softgoal Delegation (SG- <b>DEL</b> )	Softgoal Generalization (SG- <b>G</b> )

**Tabella A.5:** Trasformazioni primitive per i softgoal tratte da [6].

Actor Primitive Transformations	
Actor Introduction (A-I)	
Actor Aggregation (A-A)	Actor Generalization (A-G)

**Tabella A.6:** Actor Primitive Transformations tratte da [6].



## Appendice B

# Grammatica di Formal Tropos

Per completezza includiamo la grammatica scritta in *Backus-Naur Form* (BNF) tratta da [22] dove si discutono tutti i concetti, caratteristiche e particolarità del linguaggio formale *Formal Tropos*. Qui ritroviamo concetti come Entity, Actor, Dependency, Goal, eccetera, che sono alla base anche del nostro lavoro di tesi. In più figurano aspetti che nella scrittura di linguaggi formali devono essere approfonditi per garantire correttezza e un certo grado di formalismo. Formal Tropos è stato applicato al case study Insurance Company [37, 22].

*declaration* := (*entity* | *actor* | *dependency* | *global-properties*)\*  
*entity* := **Entity** *name* [*attributes*] [*creation-properties*] [*invar-properties*]  
*actor* := **Actor** *name* [*attributes*] [*creation-properties*] [*invar-properties*] [*actor-goals*]  
*dependency* := **Dependency** *name type mode Depender name Dependee name* [*attributes*]  
[*creation-properties*] [*invar-properties*] [*fulfill-properties*]  
*global-properties* := **Global** [*name*] *global-property*<sup>+</sup>  
*actor-goals* := (**Goal** | **Softgoal**) *name mode* [*fulfill-properties*]  
*attributes* := **Attribute** *attribute*<sup>+</sup>  
*attribute* := *attribute-facets name : sort*  
*attribute-facets* := [**constant**] [**optional**] ...  
*sort* := *name* | **integer** | **string** | **boolean** | **date** | ...  
*type* := **type** (**goal** | **softgoal** | **task** | **resource**)  
*mode* := **mode** (**achieve** | **maintain** | **achieve&maintain** | **avoid**)  
*creation-properties* := **Creation** *creation-property*<sup>+</sup>  
*invar-properties* := **Invariant** *invar-property*<sup>+</sup>  
*fulfill-properties* := **Fulfillment** *fulfill-property*<sup>+</sup>  
*creation-property* := *property-category event-category property-origin*<sup>1</sup> *temporal-formula*  
*fulfill-property* := *property-category event-category property-origin*<sup>1</sup> *temporal-formula*  
*invar-property* := *property-category property-origin*<sup>1</sup> *temporal-formula*  
*global-property* := *property-category temporal-formula*  
*property-category* := [**constraint** | **assertion** | **possibility**]  
*event-category* := **trigger** | **condition** | **definition**  
*property-origin* := [**for depender** | **for dependee** | **domain**]  
*temporal-formula* := ...

**Tabella B.1:** Grammatica di *Formal Tropos*.



# Indice analitico

<b>A</b>		
Actor		
agent	23	achieve&maintain 23
dependee	22	Actor 20
depender	22	agent 23
position	23	Attribute 17
role	23	avoid 23
social	33	Basic 17
system	33	Belief 22
		Constrainable Entity 17
		Dependency 22
		Entity 17
		intenzionali 17
		maintain 23
		Mode 23
		N-ary Relationship 18
		non-intenzionali 17
		position 23
		Resource 24
		role 23
		T-L Formula 17
<b>B</b>		
balloon	37	
BNF	85	
<b>C</b>		
case study		
MSS	47	
<b>D</b>		
Dependency		
dependum	22	
Mode	23	
diagramma		
definizione	31	
degli attori	32	
dei goal	36	
dei piani	40	
delle abilità	39	
delle interazioni tra agenti	42	
<b>E</b>		
Elementi del Linguaggio		
achieve	23	
		<b>F</b>
		Formal Tropos 16
		<b>G</b>
		Goal
		Hardgoal 24
		Softgoal 24
		grafo
		diretto 78
		diretto etichettato 78
		<b>L</b>
		Linguaggio di modellazione 15

livello		architectural design .....	9
del dominio .....	29	Belief modeling .....	13
meta-metamodello .....	17	Capability modeling .....	13
meta-modello .....	18	Dependency modeling .....	13
oggetto .....	29	detailed design .....	10
<b>M</b>		early requirements .....	8
meta-modello .....	15	Goal modeling .....	13
definizione .....	18	Interaction modeling .....	13
Mode		late requirements .....	9
achieve .....	23	Plan modeling .....	13
achieve&maintain .....	23		
avoid .....	23	<b>V</b>	
maintain .....	23	vista	
modello		sul modello .....	31
in Tropos .....	30		
<b>P</b>			
patterns .....	23		
<b>R</b>			
Relazioni			
AND-OR decomposition .....	27		
Contribution .....	26		
Dependency .....	22		
Means-Ends analysis .....	26		
requirement driven .....	7		
requirements			
functional .....	24		
non-functional .....	24		
Ruoli			
end .....	26		
mean .....	26		
<b>S</b>			
stakeholder .....	9		
<b>T</b>			
Tropos .....	7		
Actor modeling .....	12		