Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Tropos: Processo e frammenti

M. Cossentino, V. Seidita, N. Miciletto, R. Rubino

Rapporto Tecnico N.:
RT-ICAR-PA-05-06 **Luglio 2005**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sede di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sede di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Tropos: Processo e frammenti

M. Cossentino[1], V. Seidita[2], N. Miciletto[2], R. Rubino[2]

Rapporto Tecnico N.: **RT-ICAR-PA-05-06**

Data: **Luglio 2005**

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Palermo Viale delle Scienze edificio 11 90128 Palermo

[2] Università degli Studi di Palermo Dipartimento di Ingegneria Informatica Viale delle Scienze 90128 Palermo

Indice

1 Introduzione

Di seguito riportiamo una panoramica delle fasi di *Tropos*, delle activities svolte in ogni fase, e delle interazioni esistenti tra le varie fasi. Come si vede dalla doppia freccia tra l'*early requirements* e la *late requirements*, si ha un'iterazione tra queste fasi che permette di individuare requisiti sfuggiti in un primo step di analisi e dettagliarli prima di passare all'implementazione dell'architettura.

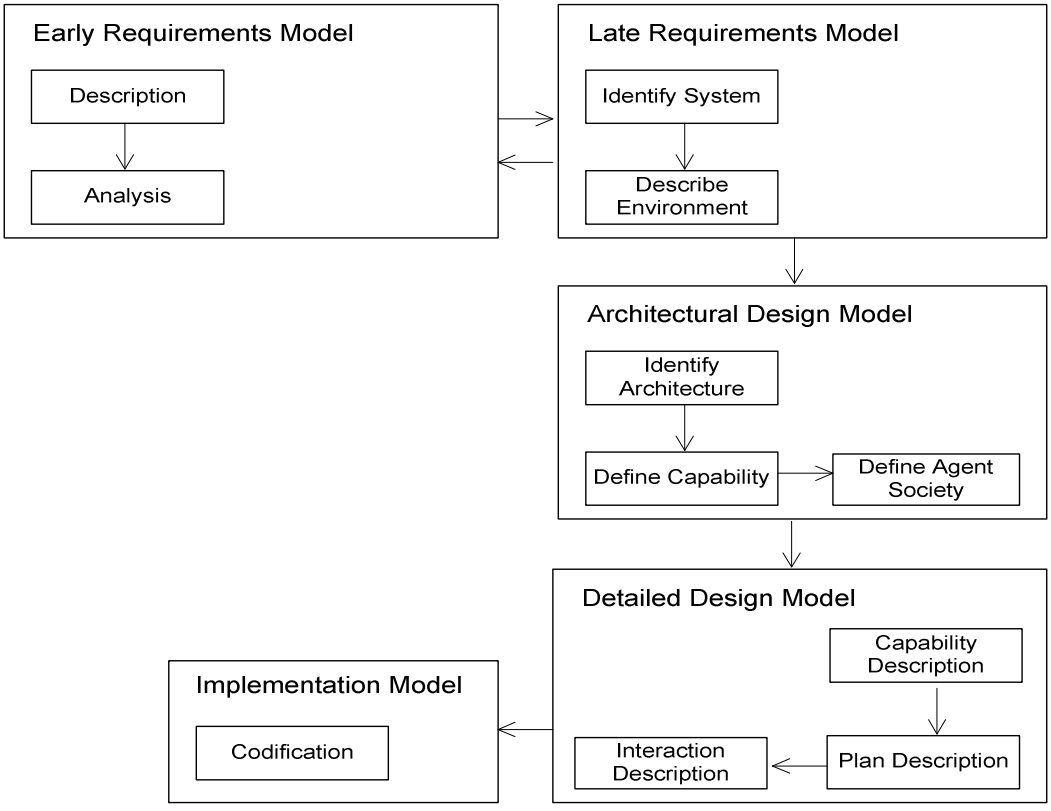


Figura 1: La metodologia *Tropos*

Il meta-modello di *Tropos* 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 linguaggi di programmazione ad hoc, ma su questo argomento torneremo al capitolo 6.

2 Il Processo Tropos

Tropos include cinque discipline:

- *Early Requirements*: sono individuati gli stakeholders e i loro goals principali; gli stakeholders vengono rappresentati come attori sociali che, per il raggiungimento dei goals, e le resources di cui necessitano, dipendono da altri attori sociali;
- *Late Requirements*: viene incluso un nuovo actor, il system-to-be, che rappresenta l'architettura software da implementare; vengono individuati i goals che deve raggiungere (e che corrispondono ai requisiti funzionali e non derivanti dal System Requirements), e le dependencies, con gli altri actors dell'ambiente, necessarie al loro soddisfacimento;
- *Architectural Design*: definisce l'architettura globale del system-to-be in termini di sub-actors, ognuno con goals e plans, derivanti da quelli del system-to-be; vengono individuate le dependencies tra essi, intese come flussi di dati e di controllo, e vengono quindi elencati gli agenti software che soddisfaranno le specifiche fin qui determinate;
- *Detailed Design*: si passa allo studio degli agenti, definendo le capabilities, le beliefs, e descrivendo i goals e i plans dei sub-actors individuati nella precedente fase, attraverso diagrammi che permettano, nella fase successiva, un mapping diretto con la piattaforma del MAS scelto;
- *Implementation*: le specifiche prodotte dalla precedente fase, sono trasformate in uno scheletro per l'implementazione, che viene realizzata effettuando un mapping dai costrutti di *Tropos* a quelli della piattaforma di programmazione ad agenti scelta; il codice è quindi, da questo, generato in automatico.

Qui sotto riportiamo la rappresentazione dell'intero processo come discipline SPEM.

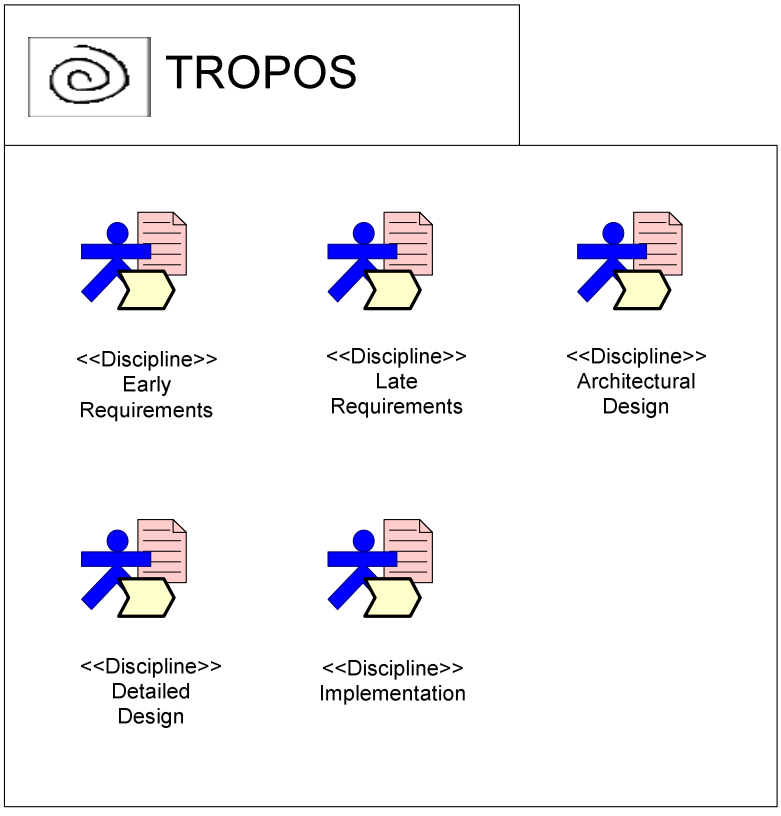


Figura 2. Le discipline del processo Tropos

Possiamo inoltre raffigurare la diretta corrispondenza tra le discipline e le fasi di *Tropos*.

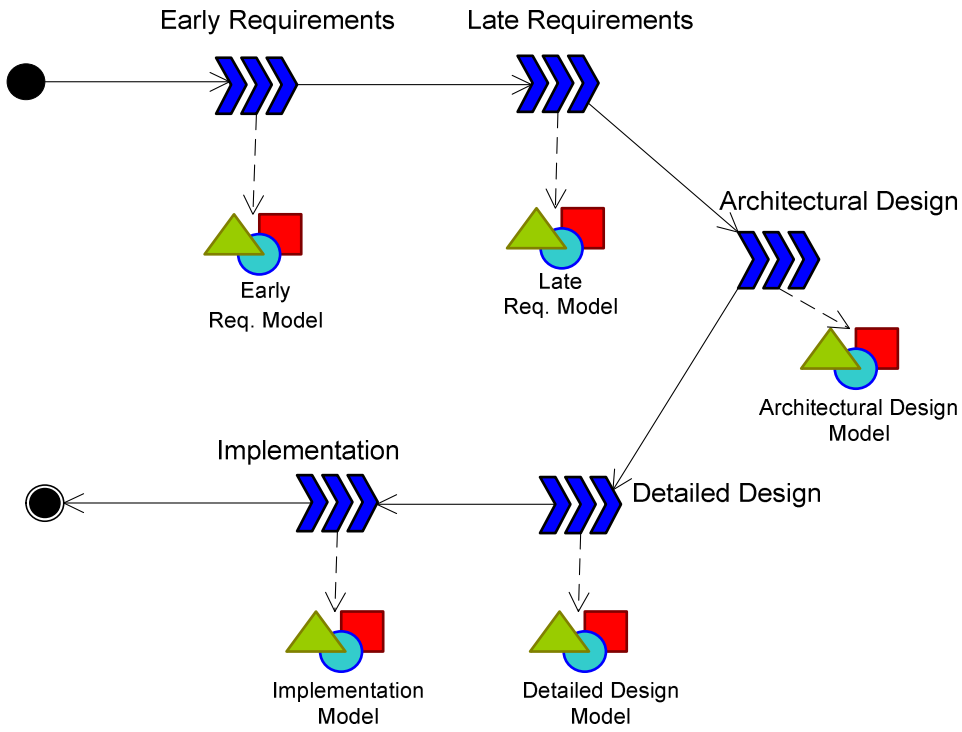


Figura 3. Fasi e modelli del processo Tropos

Il processo *Tropos* è composto da cinque differenti fasi: *Early Requirements*, *Late Requirements*, *Architectural Design*, *Detailed Design* e *Implementation* (lo stesso nome delle discipline).
Ogni fase produce una serie di work products, ed è composta da una o più work definitions (sotto-fasi), ognuna responsabile della creazione o modifica di uno o più artefatti interni al modello stesso.
I dettagli di ogni fase e delle loro relative work definitions saranno discussi successivamente.
Riportiamo di seguito le fasi con una breve descrizione:

Fasi	Descrizione sintetica
Early Requirements	Studio dell'ambiente o dell'organizzazione. L'output è un modello organizzativo che include gli attori rilevanti con associati i goals e le rispettive dependencies.
Late Requirements	Studio del system-to-be all'interno del suo ambiente operativo.
Architectural Design	Studio dell'architettura globale del system-to-be. Si compiono tre passi specifici: introduzione sub-actors del system-to-be, identificazione capabilities e processo di agentificazione.
Detailed Design	Studio di ciascun componente architetturale. Specifica di ogni singolo agente con relative capabilities, goals, beliefs e plans.
Implementation	Implementazione del system-to-be, utilizzando ad esempio delle piattaforme di sviluppo AOP come JACK.

Tabella 1. Fasi del processo di sviluppo *Tropos*.

Dalla tabella 1 è possibile notare come, in *Tropos*, il momento della raccolta dei requisiti venga suddiviso in due fasi distinte, denominate rispettivamente *Early Requirements* e *Late Requirements*, entrambe condividono gli stessi concetti e l'approccio metodologico, pertanto, molte idee introdotte nella prima fase, vengono utilizzate anche nella seconda. Più precisamente, durante l'*Early Requirements*, si identificano gli stakeholders, modellati come attori sociali che dipendono fra loro, per goals da soddisfare, plans da eseguire e resources di cui necessitano.
E' grazie all'identificazione di queste dipendenze, che è possibile capire perchè alcune funzionalità sono preferibili ad altre, ed avere un banco di prova per verificare se l'implementazione finale risulti allineata alle aspettative iniziali del committente.
Il risultato di questa fase è un particolare work product incentrato su attori sociali e loro dipendenze (Goal Diagram).

Nella seconda fase, *Late Requirements*, il work products precedentemente prodotto, viene esteso per includere il nuovo actor che rappresenta il system-to-be e le dependencies che ha con gli altri actors. Tali dependencies definiscono completamente i requisiti funzionali e non-funzionali.

Dopo aver individuato tutti i requisiti dell'ambiente e del system-to-be, il passo successivo è rappresentato dalla fase di progettazione del sistema software. Anche in questo caso, si hanno due momenti con obiettivi distinti individuati dall'*Architectural Design* e dalla *Detailed Design*, che si focalizzano sulle specifiche del system-to-be, in accordo con i requisiti prodotti nei due passi precedenti.

L'*Architectural Design* definisce l'architettura globale del system-to-be, in termini di sotto-sistemi (sub-actors) interconnessi attraverso flussi di dati e di controllo. I sotto-sistemi sono rappresentati con actors, mentre le interconnessioni con dependencies.
Questa fase si articola in tre passi:

1. Viene definita l'architettura del sistema software. In particolare il system-to-be viene decomposto in sub-actors in base a:
- stile di architettura scelto;
- requisiti del system-to-be;
- goals del system-to-be.
Il risultato finale di questo passo è l'estensione del work product, nel quale vengono aggiunti nuovi actors e loro dependencies.

2. Identificazione delle capabilities necessarie agli actors per soddisfare i loro goals ed eseguire i plans. Le capabilities sono identificate analizzando il Sub-Actors Diagram, in particolare ciascuna relazione di dependency può dar luogo ad una o più capabilities scatenate da eventi. Il risultato è una tabella (Actor's Capabilities) in cui, da un lato sono riportati gli actors e dall'altro le capabilities loro assegnate.

L'ultimo passo prevede l'individuazione degli agenti e l'assegnamento delle relative capabilities; l'output prodotto è l'Agent's Capabilities, una tabella simile, per impostazione, a quella realizzata al passo precedente.

La fase successiva, *Detailed Design*, punta a dettagliare gli agenti, le loro capabilities e le interazioni che questi hanno tra di essi. A questo punto, solitamente, la piattaforma di implementazione è stata scelta, 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, attraverso una piattaforma AOP, di quanto ottenuto al passo precedente. Una tipica piattaforma utilizzata è JACK; a questo proposito riportiamo la tabella 2, che riassume la corrispondenza esistente tra i costrutti JACK e gli elementi concettuali di *Tropos*.

Costrutti	Descrizione
Agent	E' usato per definire il comportamento di un agente intelligente. Questo include le capabilities che possiede, i tipi di messaggi ed eventi ai quali risponde, e i plans che usa per soddisfare un goal.
Capability	Include plans, eventi, beliefs e altre capabilities. Ad un agente possono essere assegnate più capabilities, e una capability può essere assegnata a più agenti.
Belief	Un database che descrive un insieme di conoscenze che un agente possiede.
Event	Gli eventi interni ed esterni specificati nella *Detailed Design* sono mappati negli events JACK. In JACK un evento descrive una condizione che lancia uno o più plans.
Plan	I plans contenuti all'interno di una capability sono mappati nei plans JACK. Un plan in JACK è una sequenza di istruzioni che l'agente esegue per soddisfare i goals e gestire gli events.

Tabella 2. Mapping *Tropos* - JACK

3 Le fasi del processo *Tropos*

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), questo viene fatto nelle prime due fasi del processo di sviluppo, cioè nell'*Early Requirements* e nella *Late Requirements*.

3.1 La fase Early Requirements

La fase *Early Requirements* coinvolge due differenti process roles, e produce due work products e due documenti di testo.

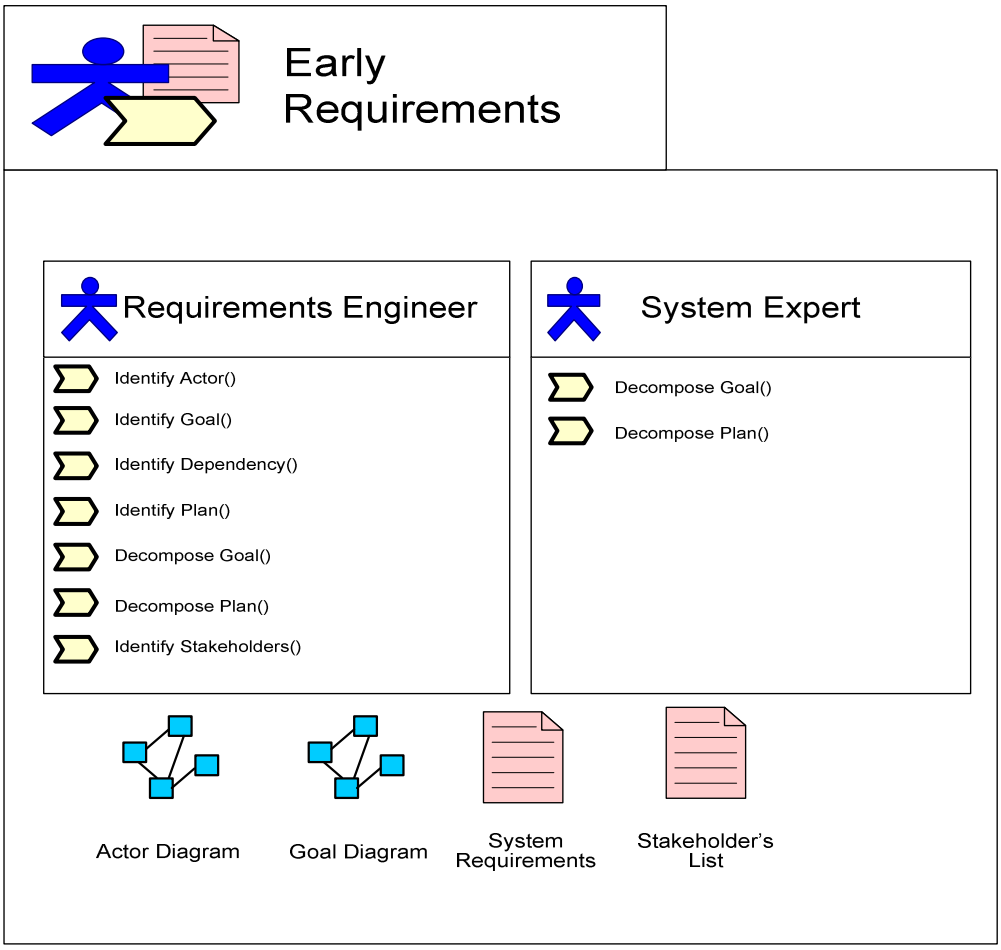
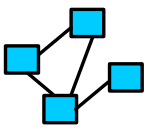


Figura 4. Descrizione della fase Early Requirements rappresentata come una disciplina SPEM

Il simbolo  indica il generico diagramma di *Tropos* e non un diagramma UML.

Il processo che deve essere eseguito in questa fase, è descritto nella seguente figura. Questa è composta da due work definitions (Description e Analysis) e i relativi work products.

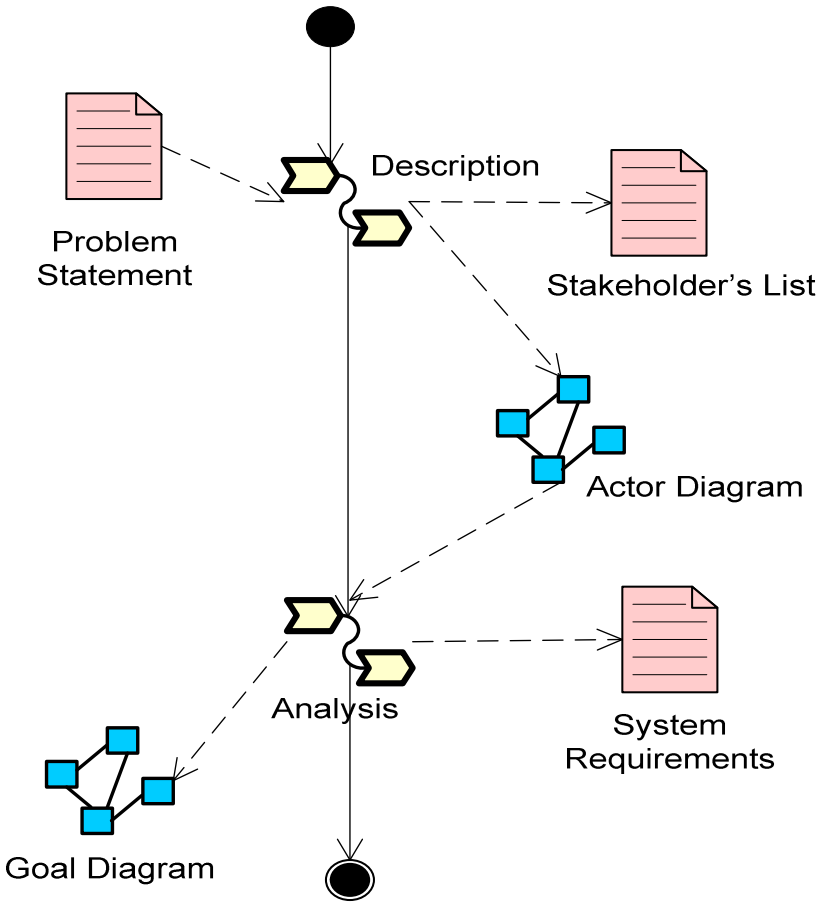


Figura 5. La fase *Early Requirements* descritta in termini di work definitions (sotto-fasi) e work products

Le due work definitions (sotto-fasi) sono composte da alcune activities come descritto nella figura 6. Questa figura descrive anche i process roles coinvolti in questa parte del processo. Come il processo si svolge, sarà descritto nelle seguenti sotto-sezioni.

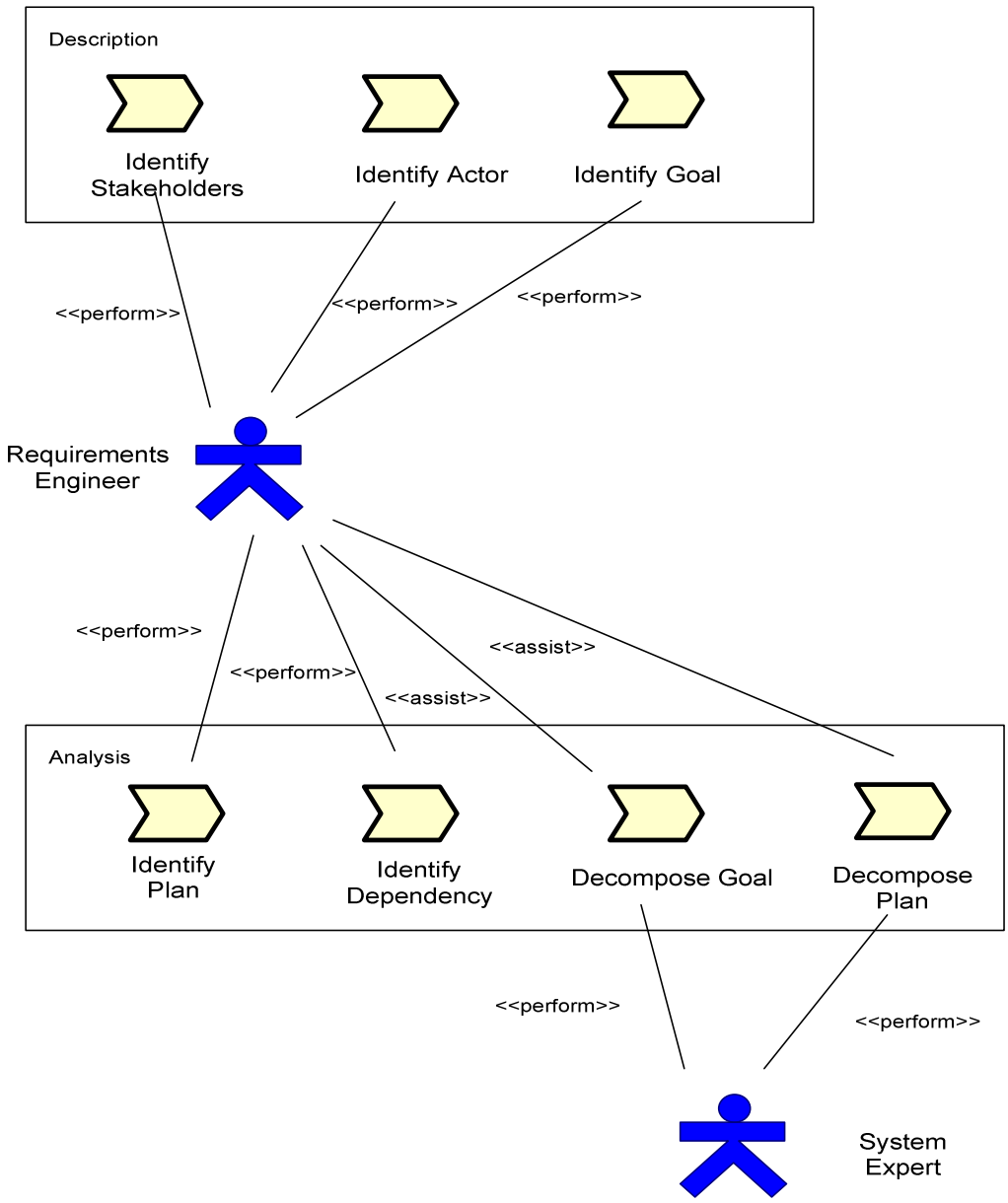


Figura 6. Le activities della Early Requirements divise in work definitions

Ecco qui di seguito un sommario delle activties di questa fase:

Work Definition	Activity	Descrizione dell'Activity	Process Roles Coinvolti
Description	Identify Stakeholders	Elencare gli stakeholders	Requirements Engineer (perform)
Description	Identify Actor	Elencare gli actors	Requirements

		dell'ambiente	Engineer (perform)
Description	Identify Goal	Elencare goals principali per ogni actor	Requirements Engineer (perform)
Analysis	Identify Dependency	Elencare relazioni tra actors per il raggiungimento dei goals	Requirements Engineer (perform)
Analysis	Identify Plan	Elencare i plans di ogni actor per raggiungere i suoi goals	Requirements Engineer (perform)
Analysis	Decompose Goal	Individuare sub-goals tramite tecniche d'analisi specifiche	System Expert (perform) Requirements Engineer (assist)
Analysis	Decompose Plan	Dividere plans a più alto livello in plans elementari	System Expert (perform) Requirements Engineer (assist)

3.1.1 Process Roles coinvolti

Due process roles sono coinvolti nella disciplina *Early Requirements*, descritti qui di seguito:

3.1.1.1 Requirements Engineer

E' responsabile di:

1. Elencare gli stakeholders dell'ambiente operativo;
2. Identificare gli "attori sociali" presenti nell'ambiente operativo
3. Identificare i goals che ogni actor persegue nel sistema da implementare;
4. Identificare le dependencies che intercorrono tra gli actors per l'esecuzione di plans, soddisfacimento di goals, ecc.
5. Identificare i plans che devono essere svolti dagli actors per il soddisfacimento dei loro goals;
6. Supporta il System Expert nella decomposizione dei goals;
7. Supporta il System Expert nella decomposizione dei plans.

3.1.1.2 System Expert

E' responsabile di:

1. Decomporre i goals principali, attraverso tecniche d'analisi specifiche, in sub-goals maggiormente dettagliati, con l'assistenza del Requirements Engineer;
2. Decomporre i plans principali, in plans elementari, più particolareggiati, che concorrono al raggiungimento di uno o più goals; in questo è assistito dal Requirements Engineer.

3.1.2 Frammenti della *Early Requirements*

In questa fase, sono individuati due frammenti: Description e Analysis, di seguito descritti.

3.1.2.1 Frammento "Description"

3.1.2.1.1 Descrizione

In questo frammento, per prima cosa si analizza il problem statement, redatto in collaborazione con il committente, dal quale si realizza una lista degli stakeholders. In seguito si individuano gli actors coinvolti nel progetto e i rispettivi goals, suddividendoli in hard e soft in base al grado di importanza nel contesto organizzativo. Si realizza, con queste informazioni, l'Actor Diagram, cioè un singolo diagramma che mostra tutti gli actors e i loro goals principali.

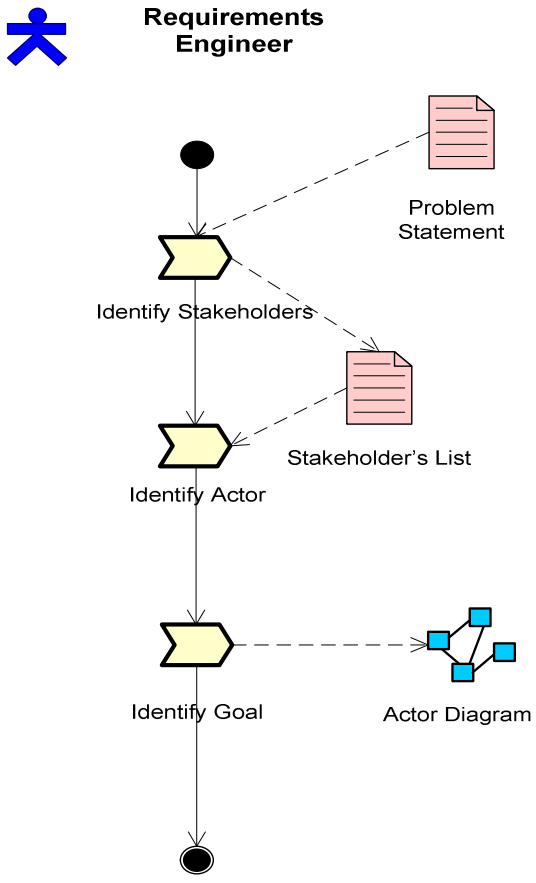


Figura 7. Frammento Description

3.1.2.1.2 Notazione

L'individuazione degli stakeholders, avviene attraverso la discussione col committente del problem statement, dal quale vengono altresì definiti i goals principali di ogni actor.
Individuati stakeholders e goals, vengono modellati nell'Actor Diagram, come descritto nella fig.1 dell'articolo [1]:

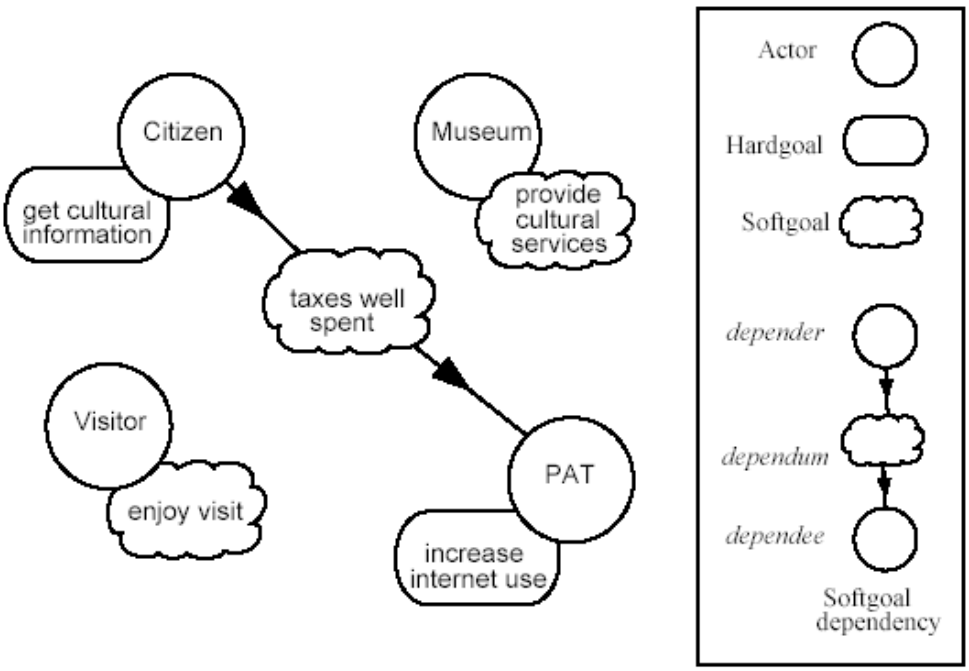
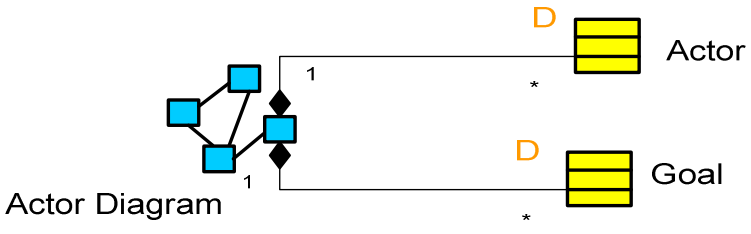


Figure 1. Actor diagram modeling the stakeholders of the eCultural project.

3.1.2.1.3 Relazioni con il MAS Meta-Model

La seguente figura illustra le relazioni tra il work product ottenuto e il MAS meta-model:



Il simbolo  rappresenta un elemento del MAS Meta-Model.

La lettera D (*defined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato introdotto per la prima volta nel processo.

3.1.2.1.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Problem Statement	Actors	Stakeholder's List
	Goals	Actor Diagram

3.1.2.1.5 Glossario

Il frammento Description usa i seguenti elementi:

Actors - rappresentano entità che hanno obiettivi strategici (goal) e volontà, nel sistema o nell'assetto organizzativo.

Goals - rappresentano gli interessi strategici degli actors. Si distinguono in hard e soft, in base alla necessità o meno di un loro soddisfacimento.

3.1.2.1.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato come input del frammento Analysis, per l'identificazione di plans e dependencies degli actors individuati dall'Actor Diagram.

3.1.2.2 Frammento “Analysis”

3.1.2.2.1 Descrizione

Questo frammento amplia l’Actor Diagram individuando, per prima cosa, i plans che ogni actor deve svolgere per il raggiungimento dei goals, poi, attraverso tecniche d’analisi specifiche, vengono individuati ulteriori goals e plans; questi vengono visualizzati nel Goal Diagram, che descrive ogni actor, con goals, plans e le dependencies che questi ha con gli altri.

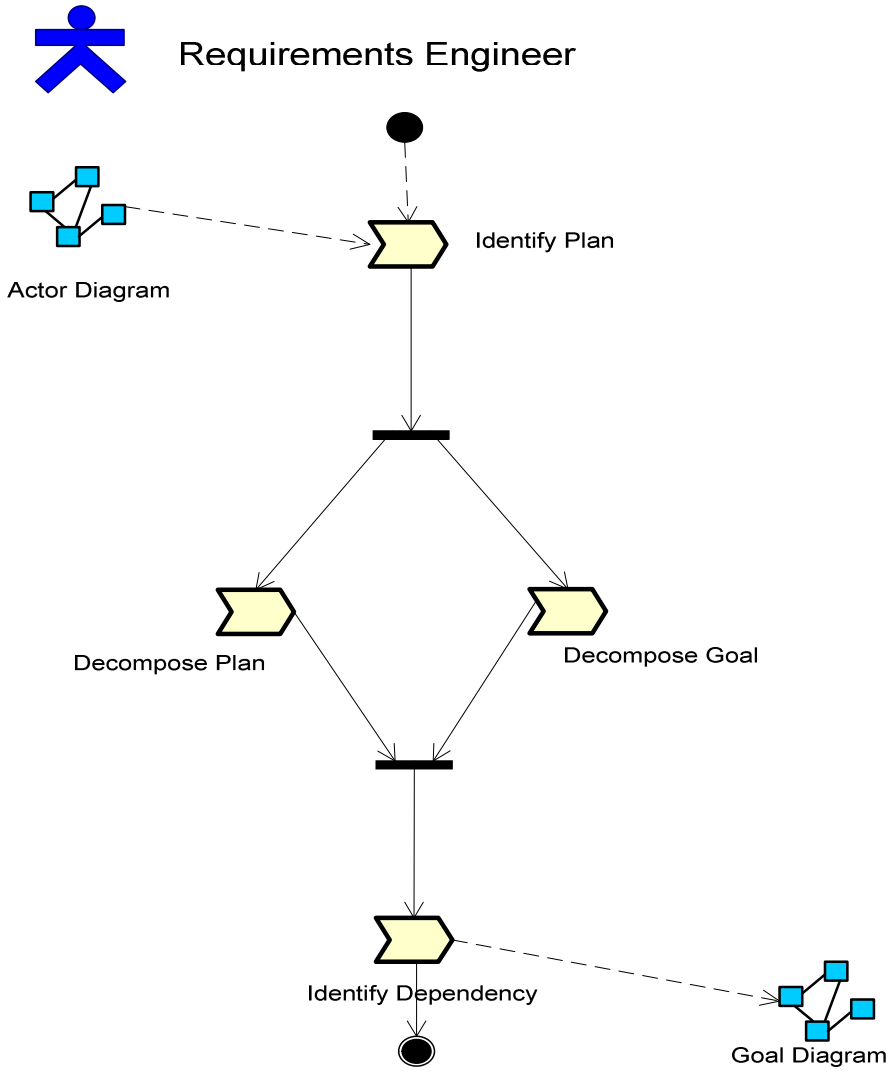


Figura 8. Frammento Analysis

3.1.2.2.2 Notazione

Per individuare i goals che possono contribuire positivamente o negativamente (p.es. fattori di sicurezza violata) al raggiungimento di quelli principali, viene eseguita una Contribution Analysis. Per individuare i sub-goals e sub-plans in cui possono essere scomposti goals radice o plans primari, viene eseguita una AND-OR Decomposition. Per individuare goals o plans che permettano il raggiungimento di un determinato goal, viene eseguita una means-end analysis.
Un esempio di quanto detto è visibile nelle seguenti figure che fanno riferimento all'articolo [1]

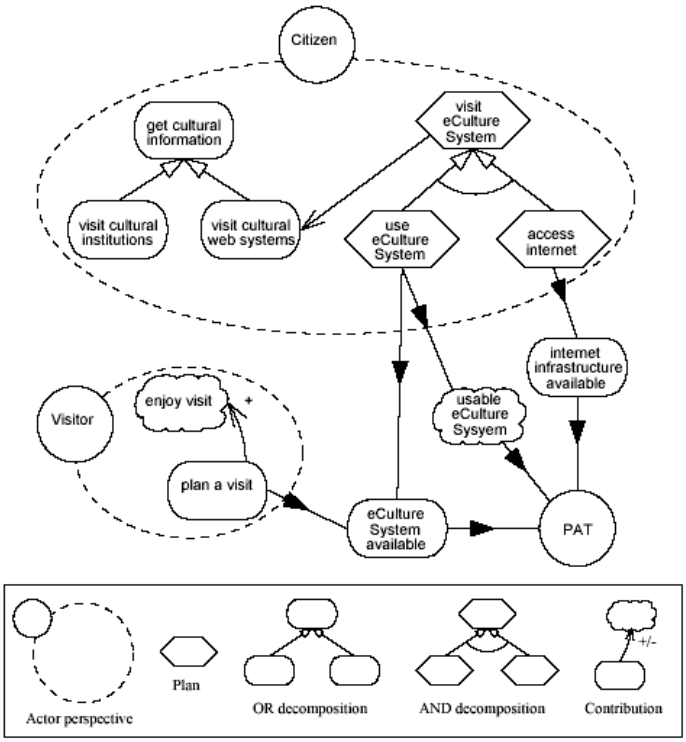


Figure 2. Goal diagrams for Citizen and Visitor. Notice the goal and plan decomposition, the means-end analysis and the (positive) softgoal contribution.

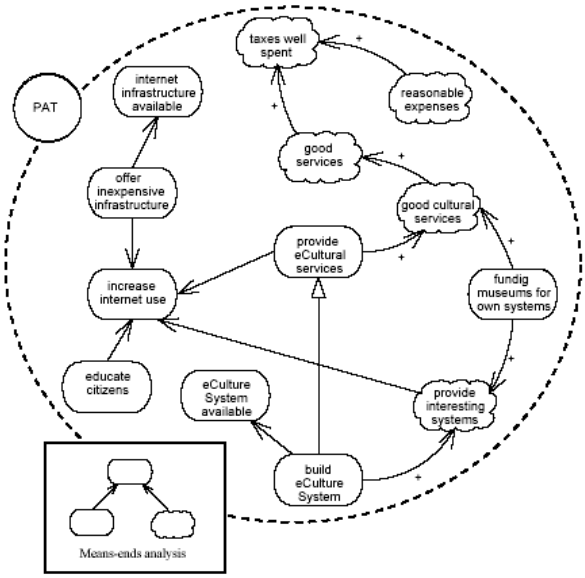
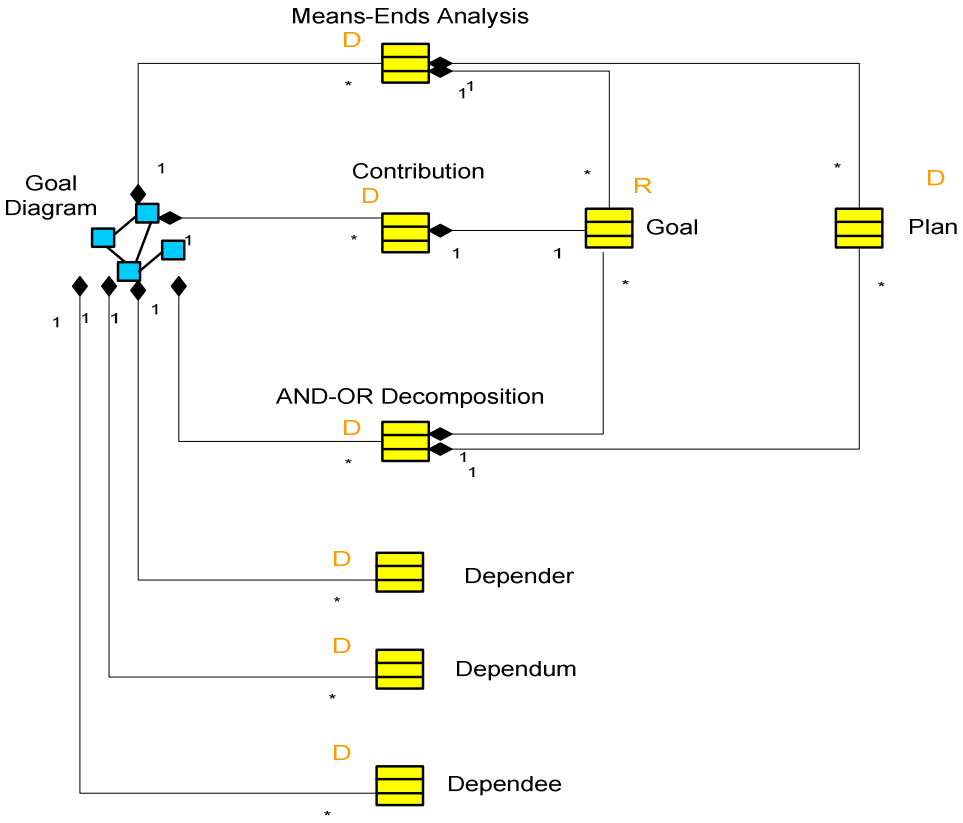


Figure 3. Goal diagram for PAT.

3.1.2.2.3 Relazioni con il MAS Meta-Model

La seguente figura illustra le relazioni tra il work product ottenuto e il MAS meta-model:



Il simbolo  rappresenta un elemento del MAS Meta-Model.

La lettera D (*defined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato introdotto per la prima volta nel processo.

La lettera R (*refined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato definito precedentemente ed ora è stato raffinato.

3.1.2.2.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Actor Diagram	Depender	Goal Diagram
	Dependum	
	Dependee	
	Goals	
	Plans	

3.1.2.2.5 Glossario

Il frammento Analysis usa i seguenti elementi:

Dependee – è un elemento della dependency, cioè l'actor che possiede "l'oggetto" richiestogli.

Dependum - è "l'oggetto" della dependency (come l'esecuzione di un plan, lo sfruttamento di una resource, ecc.).

Depender - è un elemento della dependency, cioè un actor che richiede un "oggetto" ad un altro actor.

Plans - rappresentano un insieme di procedure la cui esecuzione, solitamente, porta al soddisfacimento dei goals.

Goals - rappresentano gli interessi strategici degli actors. Si distinguono in hard e soft, in base alla necessità o meno di un loro soddisfacimento.

3.1.2.2.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato come input del frammento Identify System della fase successiva (*Late Requirements)*, come ausilio all'individuazione dei goals e dei plans che avrà il system-to-be, ed è fondamentale, nel secondo frammento (Describe Enviroment), per l'identificazione delle dependencies tra actors e system-to-be.

3.2 La fase Late Requirements

La fase *Late Requirements* coinvolge due differenti process roles e produce due work products.

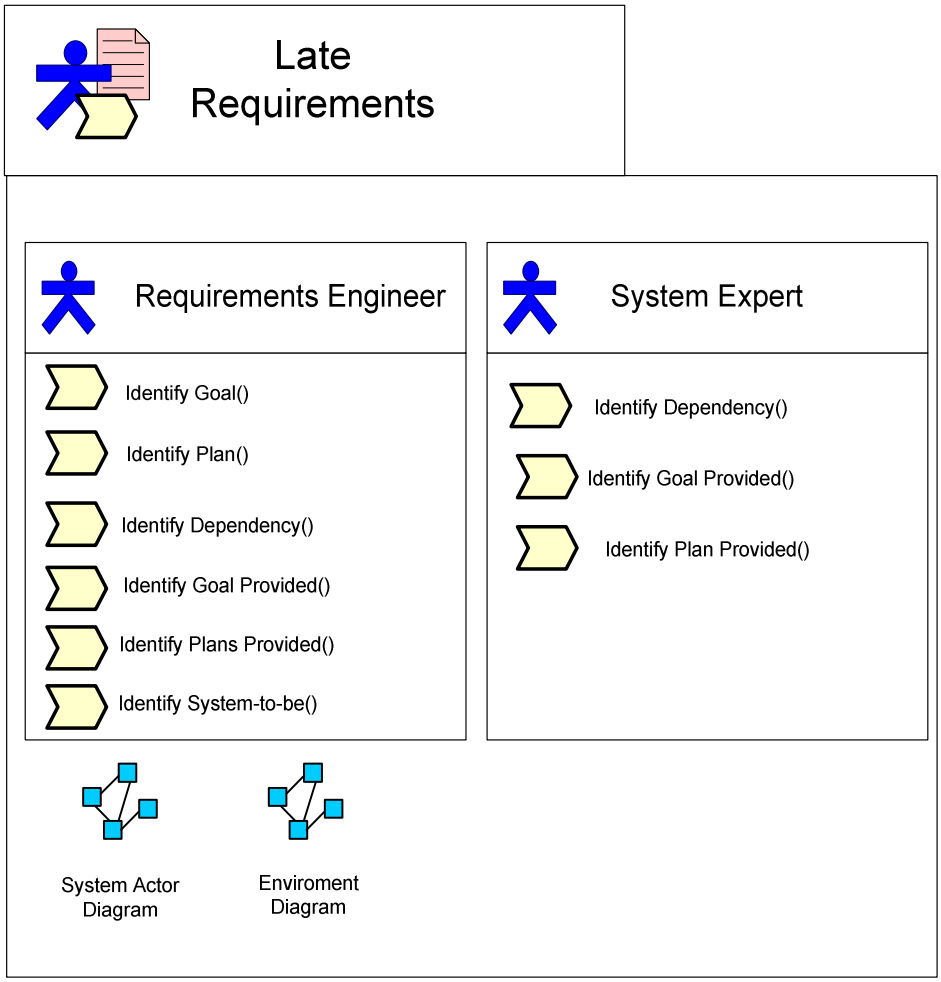


Figura 10. Descrizione della fase *Late Requirements* rappresentata come una disciplina SPEM

Il processo che deve essere eseguito in questa fase è descritto nella seguente figura. Questa è composta da due work definitions (Identify System e Describe Enviroment) e i relativi work products.

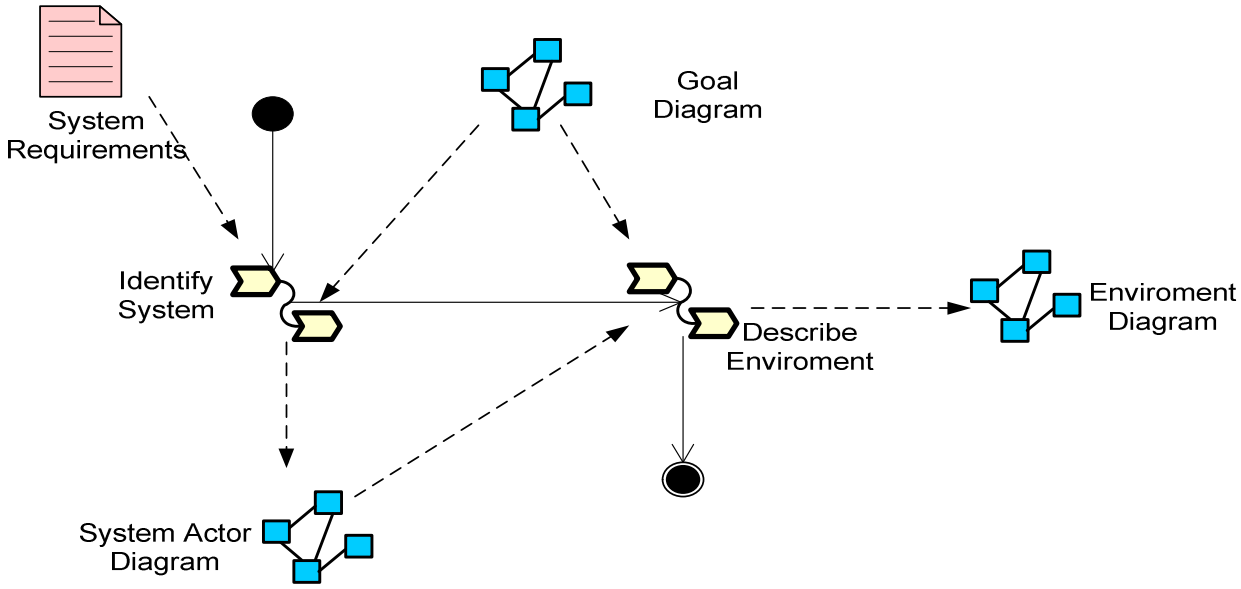


Figura 11. La fase *Late Requirements* descritta in termini di work definitions (sotto-fasi) e work products

Le due sotto-fasi sono composte da alcune activities come descritto nella figura 12. Questa figura descrive anche i process roles coinvolti in questa parte del processo.
Come il processo si svolge, sarà descritto nelle seguenti sotto-sezioni.

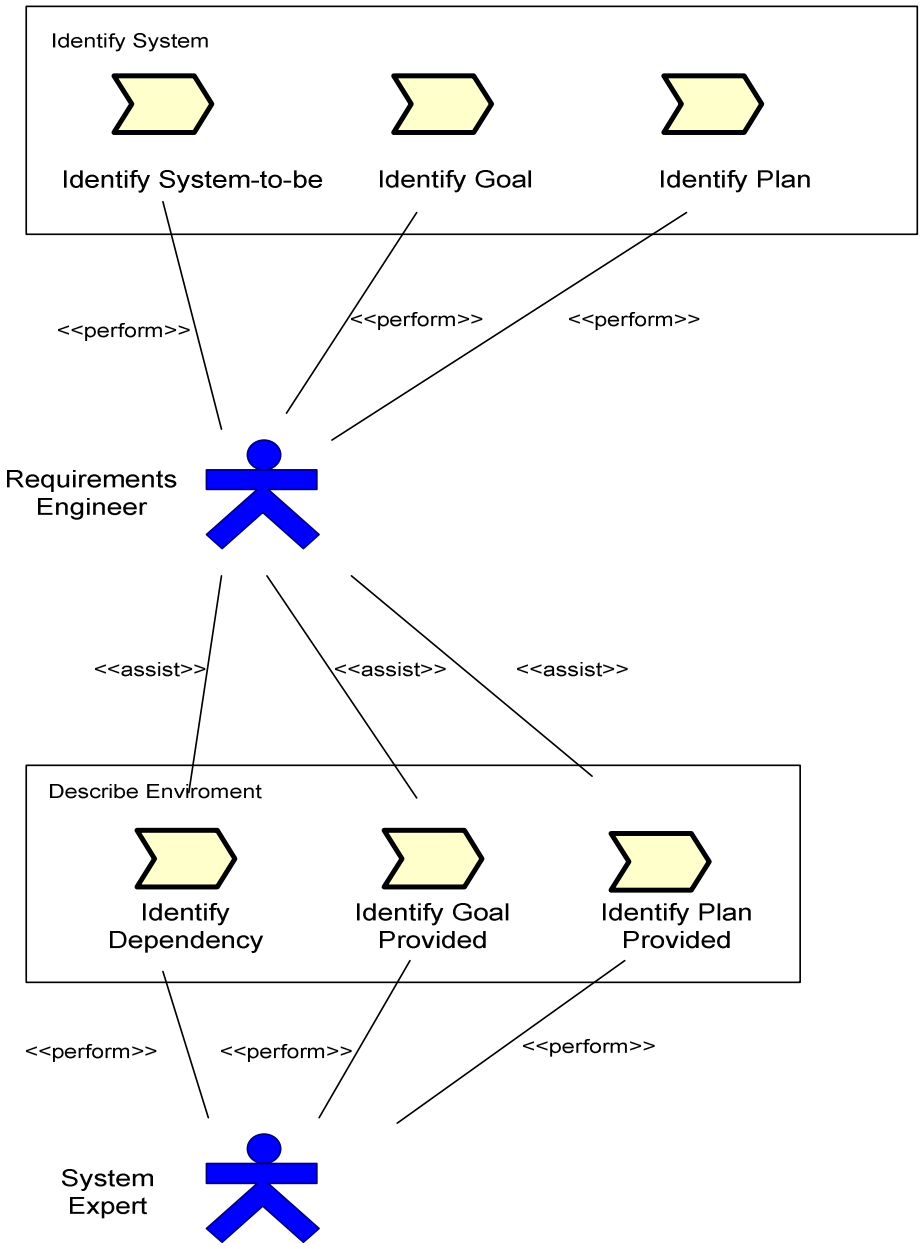


Figura 12. Le activities della *Late Requirements* divise in work definitions

Ecco qui di seguito un sommario delle activities di questa fase:

Work Definition	**Activity**	**Descrizione dell'Activity**	**Process Roles Coinvolti**
Identify System	Identify System-to-be	Identifica l'attore sistema	Requirements Engineer (perform)
Identify System	Identify Goal	Elencare i goals del system-to-be descritti nel System Requirements	Requirements Engineer (perform)
Identify System	Identify Plan	Elencare i plans del system-to-be per il	Requirements Engineer

		raggiungimento dei goals	(perform)
Describe Enviroment	Identify Dependency	Elencare le relazioni tra il system-to-be e gli actors	System Expert (perform) Requirements Engineer (assist)
Describe Enviroment	Identify Goal Provided	Elencare i goals degli actors che possono essere delegati al system-to-be	System Expert (perform) Requirements Engineer (assist)
Describe Enviroment	Identify Plan Provided	Elencare i plans delegati dagli actors al system-to-be	System Expert (perform) Requirements Engineer (assist)

3.2.1 Process Roles coinvolti

Due process roles sono coinvolti nella disciplina *Late Requirements*, descritti qui di seguito:

3.2.1.1 Requirements Engineer

E' responsabile di:

1. Introdurre l'actor system-to-be;
2. Identificare i goals che l'actor system-to-be vuole raggiungere;
3. Identificare i plans che l'actor system-to-be deve compiere per il raggiungimento dei suoi goals;
4. Supporta il System Expert nell'individuare le dependencies tra l'actor system-to-be e gli altri actors;
5. Supporta il System Expert nell'individuare i goals che possono essere delegati al system-to-be;
6. Supporta il System Expert nell'individuare i plans che possono essere delegati al system-to-be.

3.2.1.2 System Expert

E' responsabile di:

1. Individuare le dependencies tra il system-to-be e gli altri actors con l'assistenza del Requirements Engineer;
2. Individuare i goals che possono essere delegati al system-to-be; in questo è assistito dal Requirements Engineer ;
3. Individuare i plans che possono essere delegati al system-to-be; in questo è assistito dal Requirements Engineer.

3.2.2 Frammenti della *Late Requirements*

In questa fase, sono individuati due frammenti: Identify System e Describe Enviroment, di seguito descritti.

3.2.2.1 Frammento "Identify System"

3.2.2.1.1 Descizione

In questo frammento, analizzando il System Requirements e il Goal Diagram, viene definito per la prima volta l'actor system-to-be, con propri goals e plans, e vengono individuati ulteriori requisiti, funzionali e non, che vanno ad aggiungersi a quelli già elencati nel System Requirements, aggiornandolo.

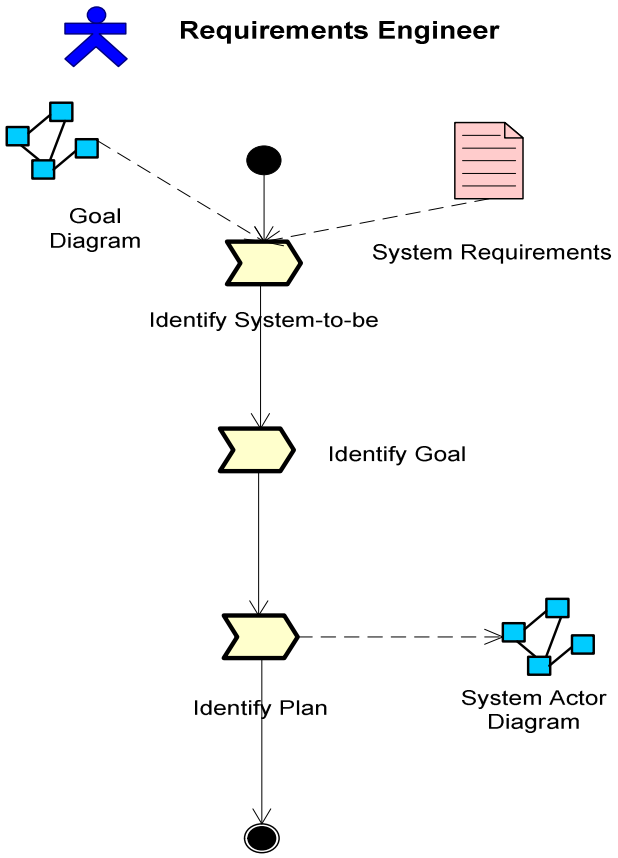


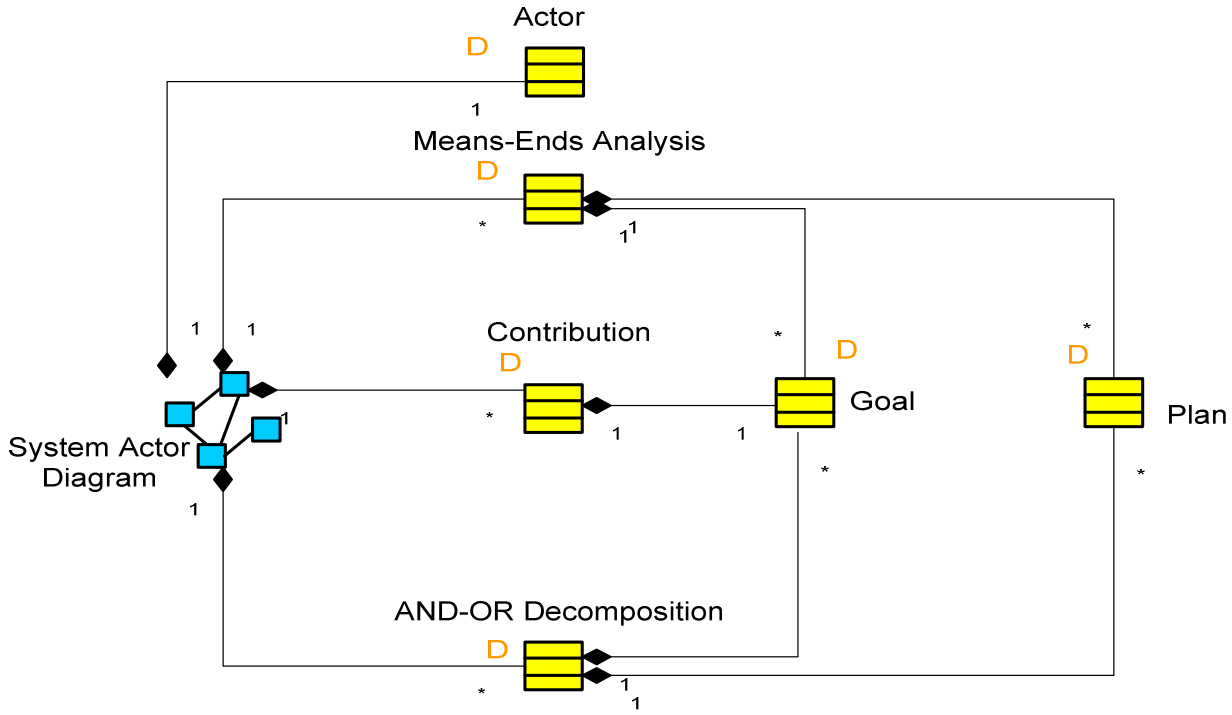
Figura 13. Frammento Identify System

3.2.2.1.2 Notazione

Per identificare i goals e i plans del system-to-be, vengono utilizzate tecniche di means-end analysis, contribution analysis e AND-OR Decomposition.

3.2.2.1.3 Relazioni con il MAS Meta-Model

La seguente figura illustra le relazioni tra il work product ottenuto e il MAS meta-model:



Il simbolo  rappresenta un elemento del MAS Meta-Model.

La lettera D (*defined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato introdotto per la prima volta nel processo.

3.2.2.1.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Goal Diagram	Actor	System Actor Diagram
System Requirements	Goals	
	Plans	

3.2.2.1.5 Glossario

Il frammento Identify System usa i seguenti elementi:

Actors - rappresentano entità che hanno obiettivi strategici (goal) e volontà, nel sistema o nell'assetto organizzativo.

Goals - rappresentano gli interessi strategici degli actors. Si distinguono in hard e soft, in base alla necessità o meno di un loro soddisfacimento.

Plans - rappresentano un insieme di procedure la cui esecuzione, solitamente, porta al soddisfacimento dei goals.

3.2.2.1.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato, insieme al Goal Diagram, come input del frammento Describe Enviroment per l'identificazione delle dependencies tra gli actors e il system-to-be; inoltre, il System Requirements qui modificato, sarà utilizzato nel frammento Identify Architecture, della fase *Architectural Design*, per la scomposizione del system-to-be in sub-actors, in base ai requisiti qui esplicitati.

3.2.2.2 Frammento “Describe Enviroment”

3.2.2.2.1 Descrizione

In questo frammento, analizzando il System Actor Diagram e il Goal Diagram, si individuano i goals e i plans degli actors dell’ambiente operativo, che possono essere delegati al system-to-be; fatto ciò si individuano le dependencies tra il nuovo actor inserito (il system-to-be) e gli altri actors.

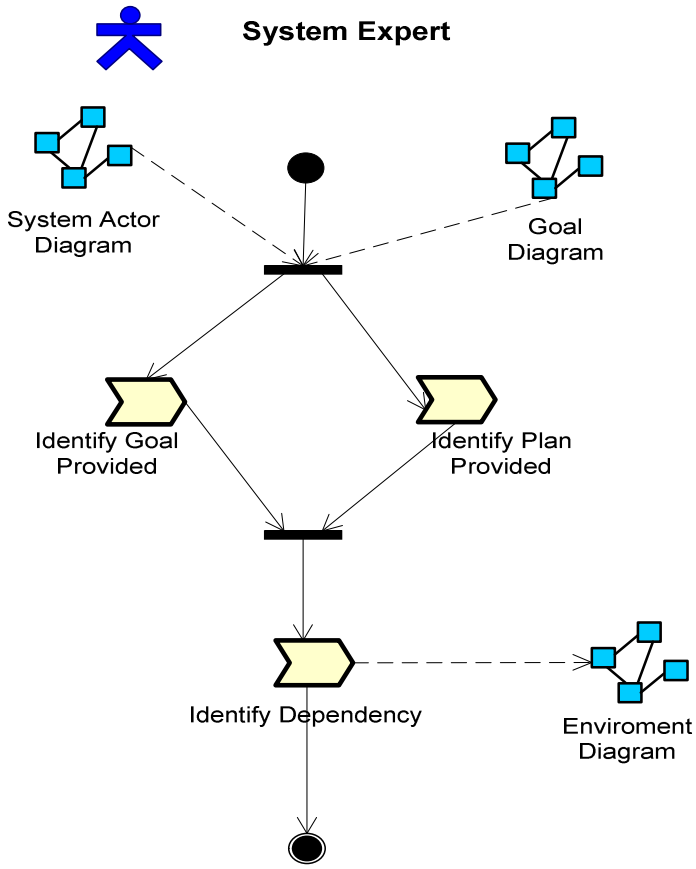


Figura 14. Describe Enviroment

3.2.2.2.2 Notazione

Si analizza il Goal Diagram, che contiene gli actors dell'ambiente operativo con i loro goals, e tramite le trasformazioni Goal e Softgoal Delegation (vedi paragrafo 7.2), si vede quali di

questi possano essere affidati al system-to-be per il loro soddisfacimento, individuando, pertanto, le relazioni di dipendenza.
Un esempio di quanto realizzato è riportato nella figura di seguito, che fa riferimento all'articolo [1]:

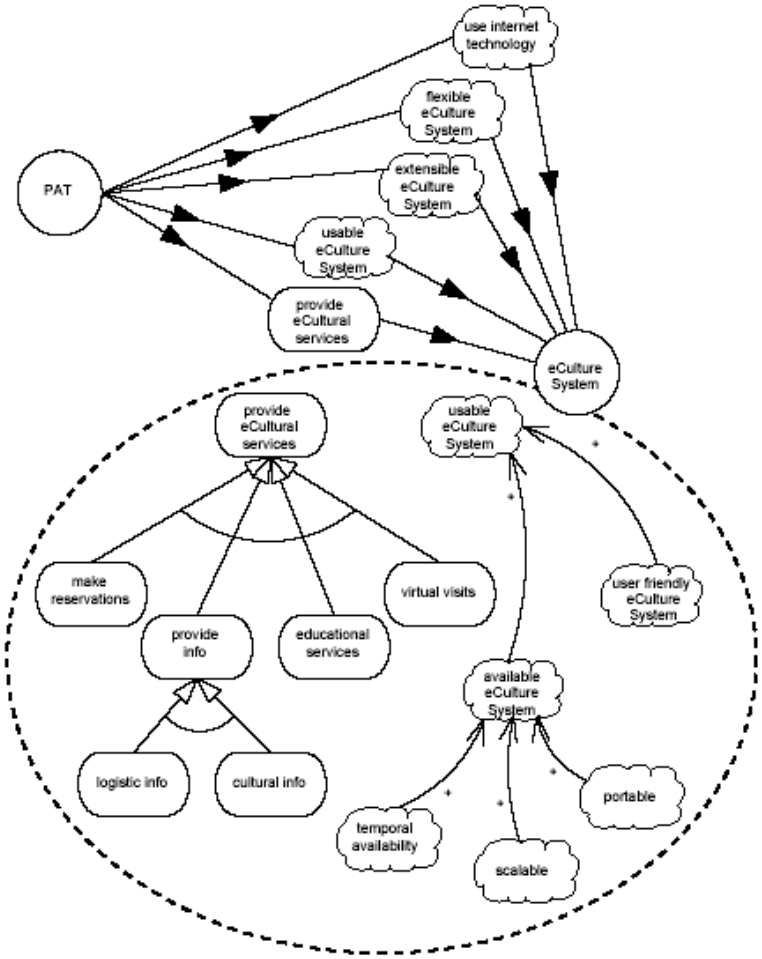
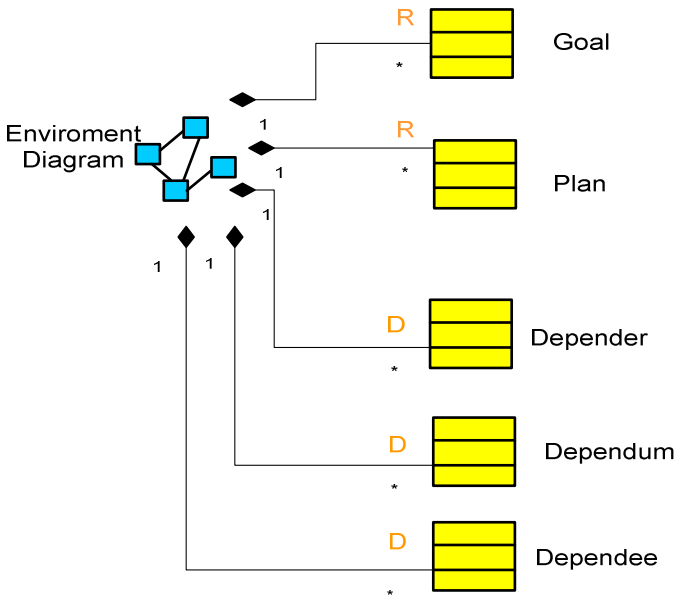


Figure 4. A portion of the actor diagram including PAT and eCulture System and goal diagram of the eCulture System.

3.2.2.2.3 Relazioni con il MAS Meta-Model

La seguente figura illustra le relazioni tra il work product ottenuto e il MAS meta-model:



Il simbolo  rappresenta un elemento del MAS Meta-Model.

La lettera D (*defined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato introdotto per la prima volta nel processo.

La lettera R (*refined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato definito precedentemente ed ora è stato raffinato.

3.2.2.2.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Goal Diagram	Dependee	Enviroment Diagram
System Actor Diagram	Dependum	
	Depender	
	Goals	
	Plans	

3.2.2.2.5 Glossario

Il frammento Describe Enviroment usa i seguenti elementi:

Dependee – è un elemento della dependency, cioè l'actor che possiede "l'oggetto" richiestogli.

Dependum - è "l'oggetto" della dependency (come l'esecuzione di un plan, lo sfruttamento di una resource, ecc.).

Depender - è un elemento della dependency, cioè un actor che richiede un "oggetto" ad un altro actor.

Goals - rappresentano gli interessi strategici degli actors. Si distinguono in hard e soft, in base alla necessità o meno di un loro soddisfacimento.

Plans - rappresentano un insieme di procedure la cui esecuzione, solitamente, porta al soddisfacimento dei goals.

3.2.2.2.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato come input del frammento Identify Architecture, della fase *Architectural Design*, per una scomposizione del system-to-be in sub-actors dettata, oltre che dalla scelta del tipo di architettura software e dalle specifiche del System Requirements, anche dall'ambiente operativo (definito per l'appunto dal Enviroment Diagram) nel quale dovrà operare.

3.3 La fase Architectural Design

La fase *Architectural Design* coinvolge due differenti process roles e produce due work products e due documenti di testo.

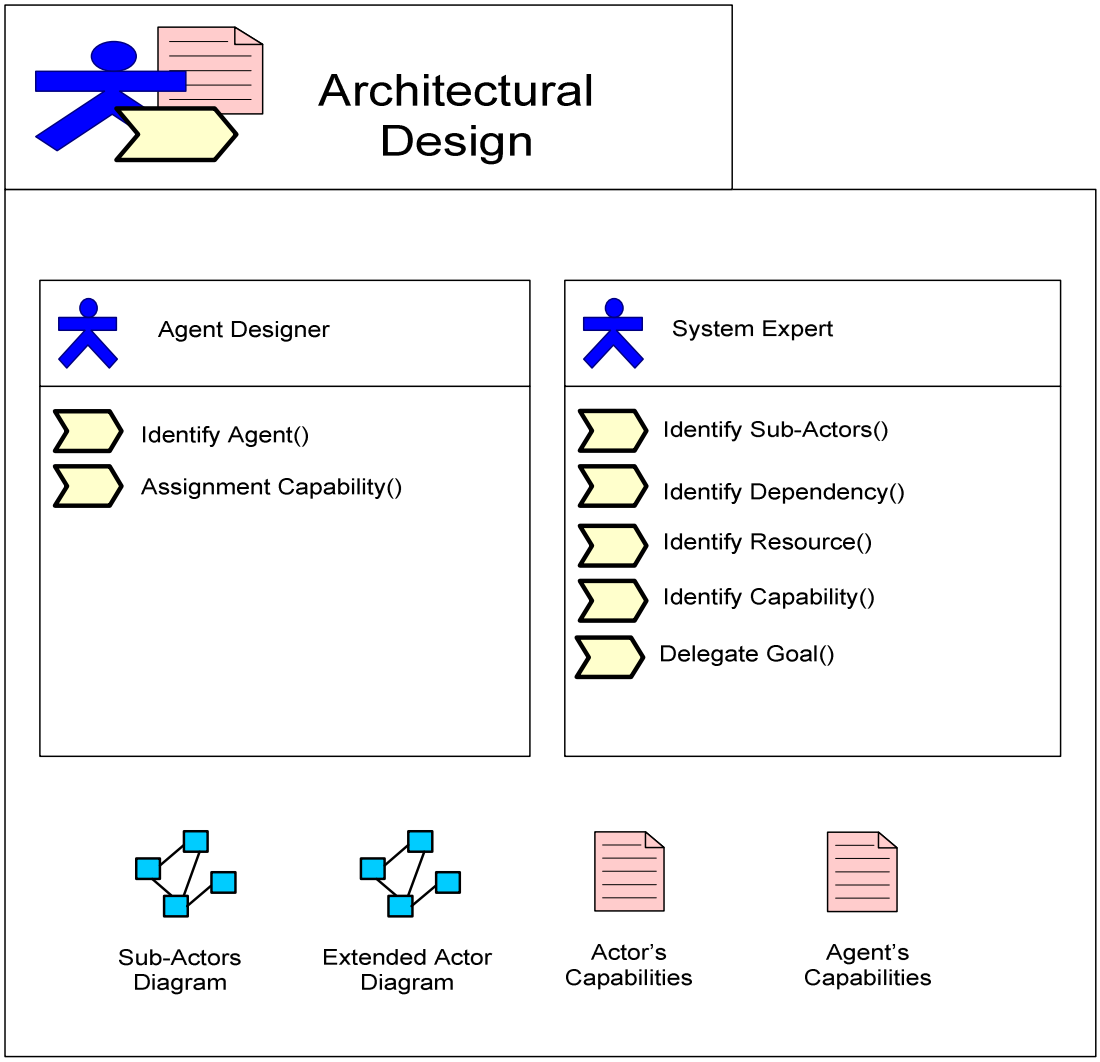


Figura 15. Descrizione della fase *Architectural Design* rappresentata come una disciplina SPEM

Il processo che deve essere eseguito in questa fase è descritto nella seguente figura. Questa è composta da tre work definitions (Identify Architecture, Define Capability e Define Agent Society) e i relativi work products.

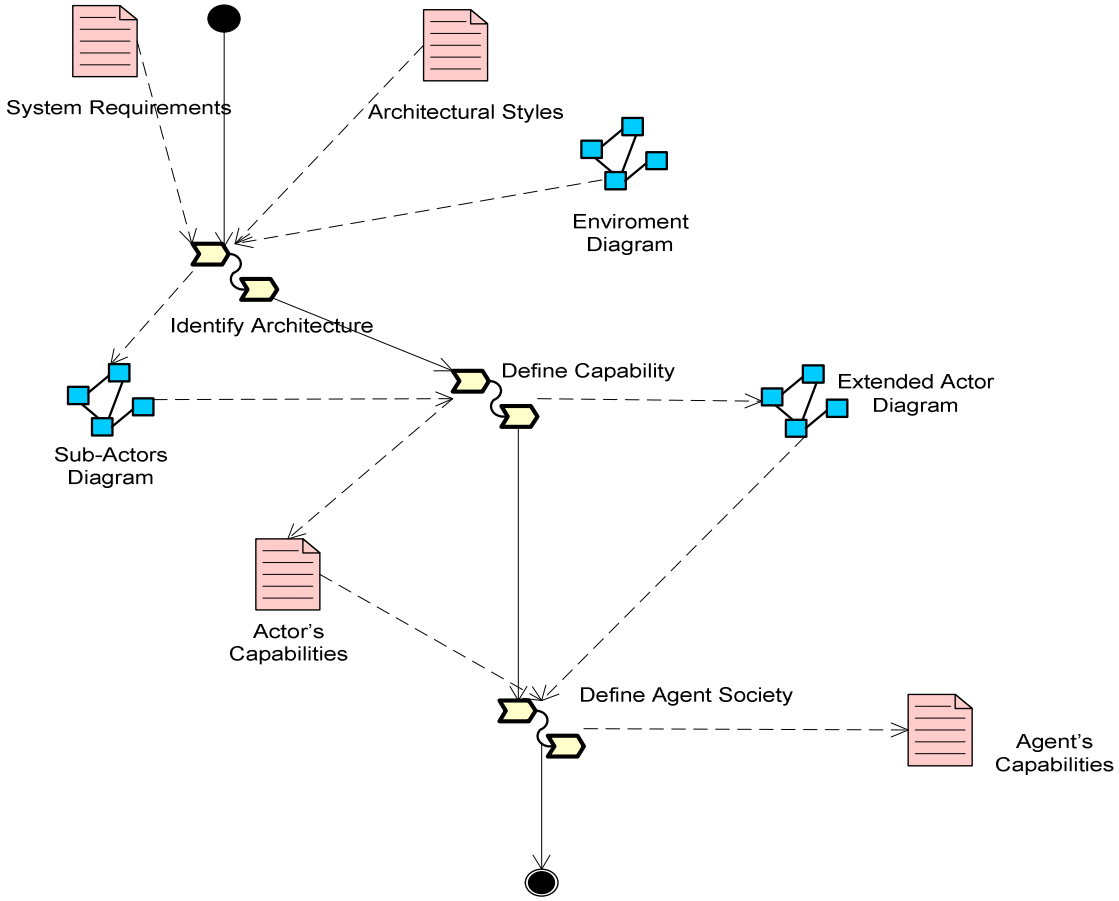


Figura 16. La fase *Architectural Design* descritta in termini di work definitions (sotto-fasi) e work products

Le tre sotto-fasi sono composte da alcune activities come descritto nella figura 17. Questa figura descrive anche i process roles coinvolti in questa parte del processo. Come il processo si svolge, sarà descritto nelle seguenti sotto-sezioni.

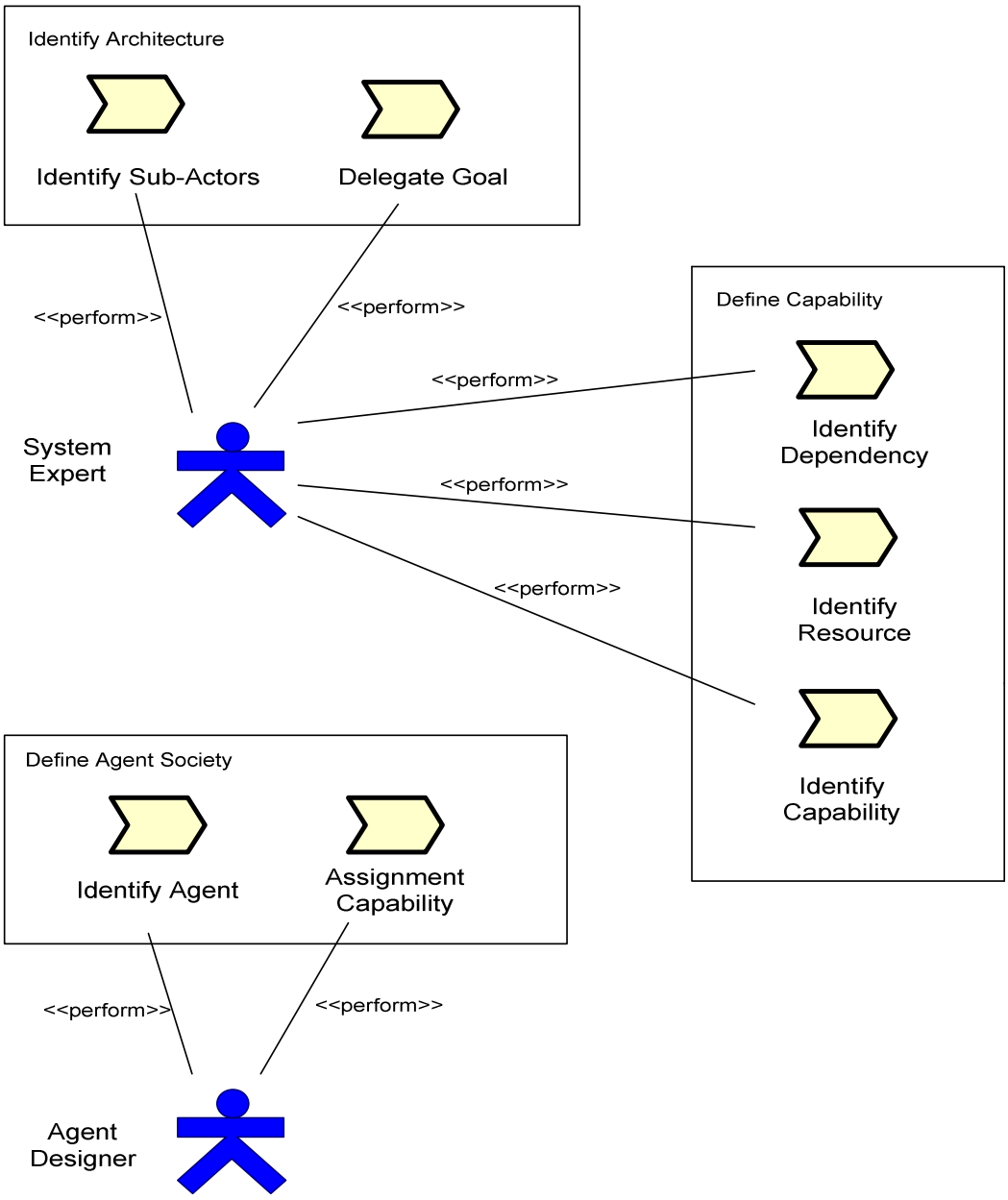


Figura 17. Le activities della *Architectural Design* divise in work definitions

Ecco qui di seguito un sommario delle activities di questa fase:

Work Definition	**Activity**	**Descrizione dell'Activity**	**Process Roles Coinvolti**
Identify Architecture	Identify Sub-Actors	Scomporre il system-to-be in sub-actors	System Expert (perform)
Identify Architecture	Delegate Goal	Delegare goals ai sub-actors	System Expert (perform)
Define Capability	Identify Dependency	Identificare le dependencies tra sub-actors e ambiente operativo	System Expert (perform)

Define Capability	Identify Resource	Identificare le resources necessarie	System Expert (perform)
Define Capability	Identify Capability	Identificare le capabilities degli actors	System Expert (perform)
Define Agent Society	Identify Agent	Identificare gli agenti	Agent Designer (perform)
Define Agent Society	Assignment Capability	Assegnare le capabilities agli agenti	Agent Designer (perform)

3.3.1 Process Roles coinvolti

Due process roles sono coinvolti nella disciplina *Architectural Design*, descritti qui di seguito:

3.3.1.1 System Expert

E' responsabile di:

1. Identificare i sub-actors nel quale il system-to-be può essere scomposto, attraverso un'analisi dei goals del system-to-be, in accordo allo stile architettonico scelto e attraverso un'analisi completa dei requisiti di sistema e dei goals che precedentemente erano richiesti da questo richiesti;
2. Delegare i goals del system-to-be ai sub-actors individuati;
3. Identificare le dependencies che si istaurano tra i nuovi actors inseriti e quelli dell'ambiente operativo, nonché, ovviamente, tra sub-actor e sub-actor;
4. Identificare le resources necessarie al soddisfacimento dei vari goals;
5. Identificare le capabilities necessarie per il raggiungimento dei goals e l'esecuzione dei plans.

3.3.1.2 Agent Designer

E' responsabile di:

1. Definire gli agenti durante la work definition Define Agent Society;
2. Assegnare le capabilities precedentemente individuate ai corrispettivi agenti.

3.3.2 Frammenti dell'*Architectural Design*

In questa fase, sono individuati tre frammenti: Identify Architecture, Define Capability e Define Agent Society, di seguito descritti.

3.3.2.1 Frammento “Identify Architecture”

3.3.2.1.1 Descrizione

In questo frammento, l’actor system-to-be, viene scomposto in sub-actors in base all’architettura di sistema scelta (Architectural Styles.doc), ai requisisti introdotti dalla precedente fase (System Requirements.doc), e da un’analisi dei goals del system-to-be (Enviroment Diagram).

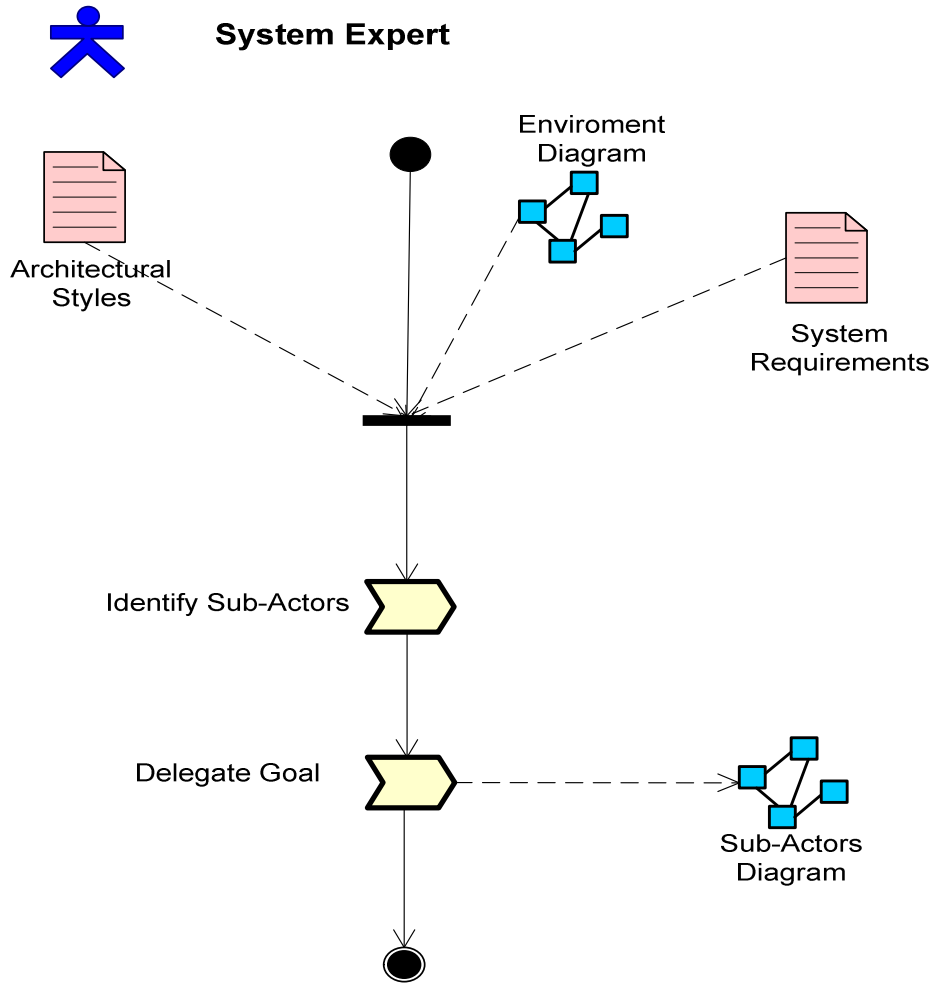


Figura 18. Frammento Identify Architecture

3.3.2.1.2 Notazione

Il sytem-to-be viene scomposto in sub-actors in base a un'analisi dei suoi goals, in accordo con lo stile architetturale scelto e in base a specifici requisiti funzionali e non. In seguito vengono delegati ai sub-actor individuati i goals del system-to-be.

Un esempio di quanto realizzato è riportato nella figura seguente, che fa riferimento all'articolo [1]:

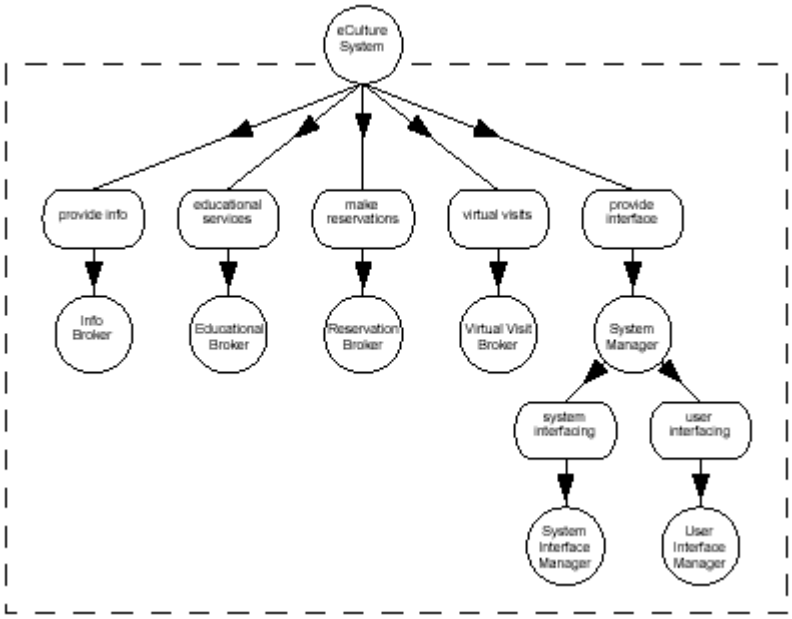
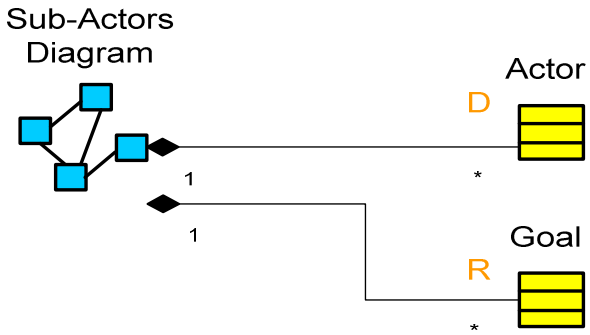


Figure 6. Actor diagram for the eCulture System architecture (step 1).

3.3.2.1.3 Relazioni con il MAS Meta-Model

La seguente figura illustra le relazioni tra il work product ottenuto e il MAS meta-model:



Il simbolo  rappresenta un elemento del MAS Meta-Model.

La lettera D (*defined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato introdotto per la prima volta nel processo.

3.3.2.1.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Architectural Style	Actors	Sub-Actors Diagram
Enviroment Diagram	Goals	
System Requirements		

3.3.2.1.5 Glossario

Il frammento Identify Architecture usa i seguenti elementi:

Actors - rappresentano entità che hanno obiettivi strategici (goal) e volontà, nel sistema o nell'assetto organizzativo.

Goals - rappresentano gli interessi strategici degli actors. Si distinguono in hard e soft, in base alla necessità o meno di un loro soddisfacimento.

3.3.2.1.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato come input del frammento Define Capability per l'identificazione delle capabilities degli actors, che successivamente saranno raccolte nel documento Actor's Capabilities, e per identificare le resources necessarie a ogni sub-actors per fornire "servizi" agli actors dell'ambiente operativo.

3.3.2.2 Frammento “Define Capability”

3.3.2.2.1 Descrizione

Questo frammento, partendo dal Sub-Actors Diagram, identifica le dependencies tra i sub-actors e gli actors dell’ambiente, identifica le resources necessarie per raggiungere i goals e per l’esecuzione dei plans e, infine, identifica le capabilities che deve possedere ogni actor.

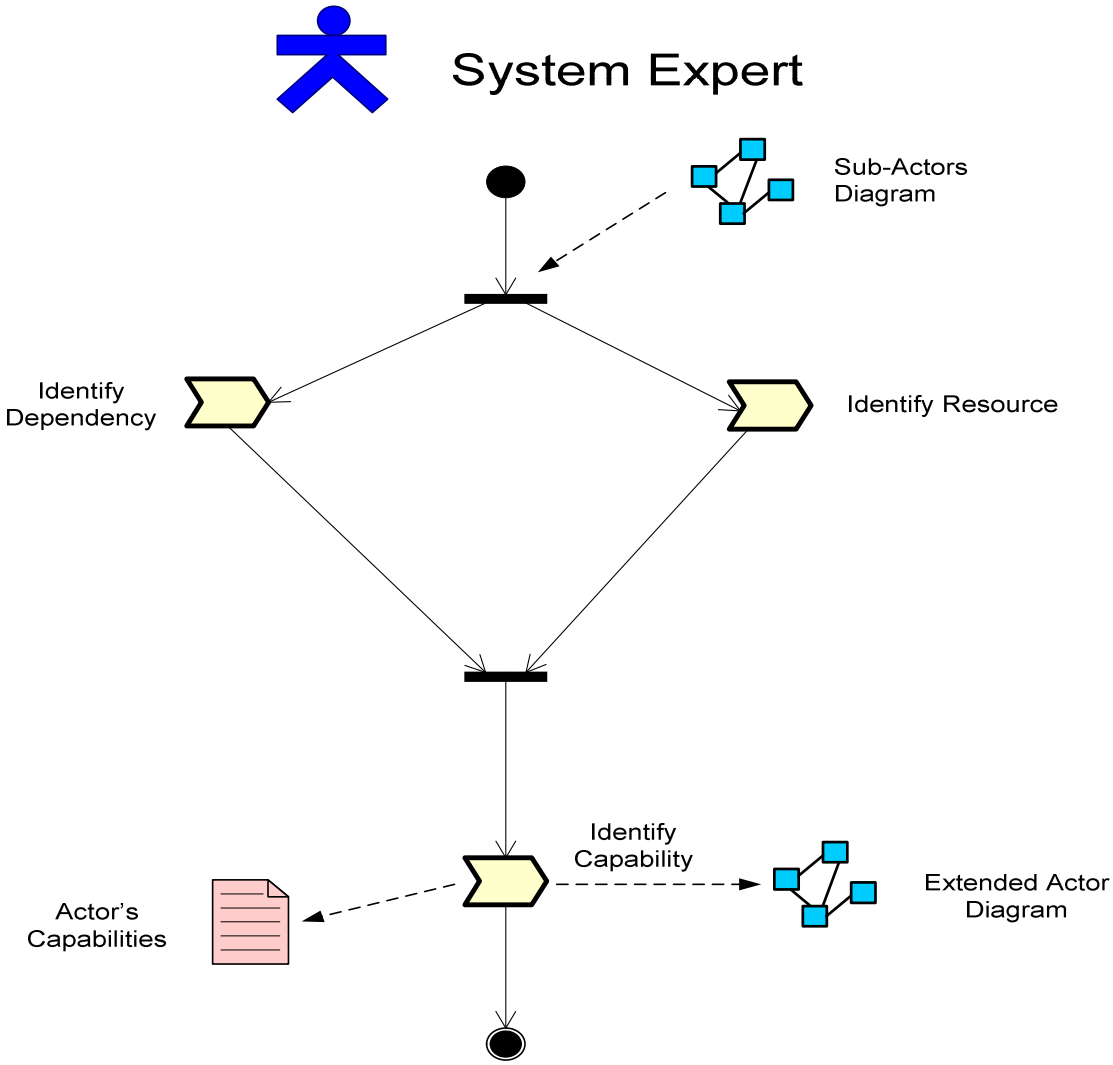


Figura 19. Frammento Define Capability

3.3.2.2.2 Notazione

Le capabilities possono essere facilmente identificate analizzando, per ogni actor, le dependencies in entrata e in uscita, come mostrato nella seguente figura dell'articolo [1]:

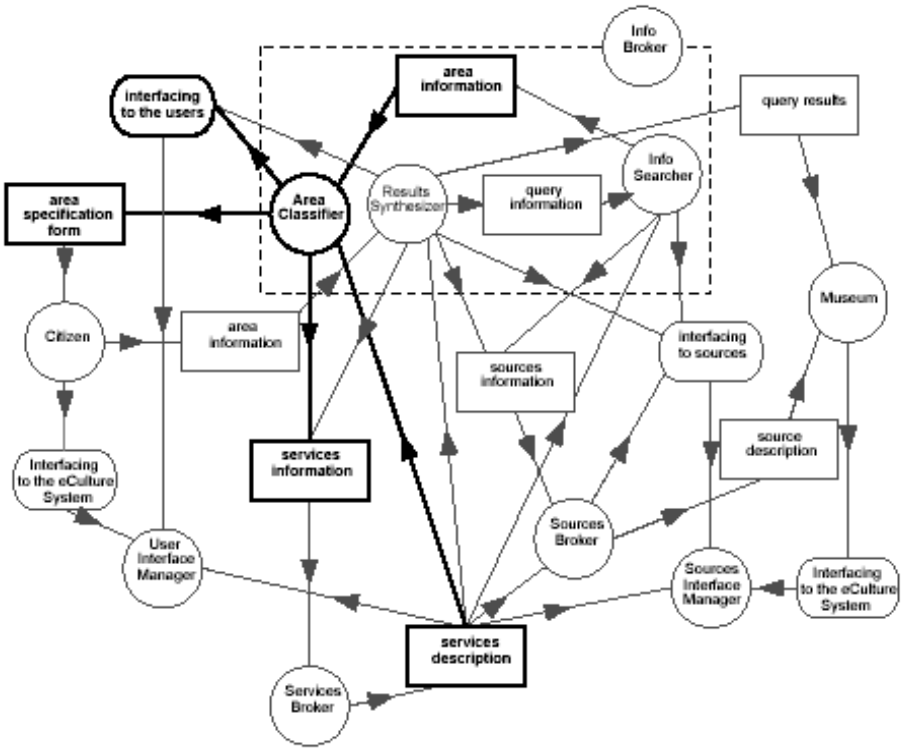


Figure 8. Identifying actor capabilities from actor dependencies w.r.t. the Area Classifier.

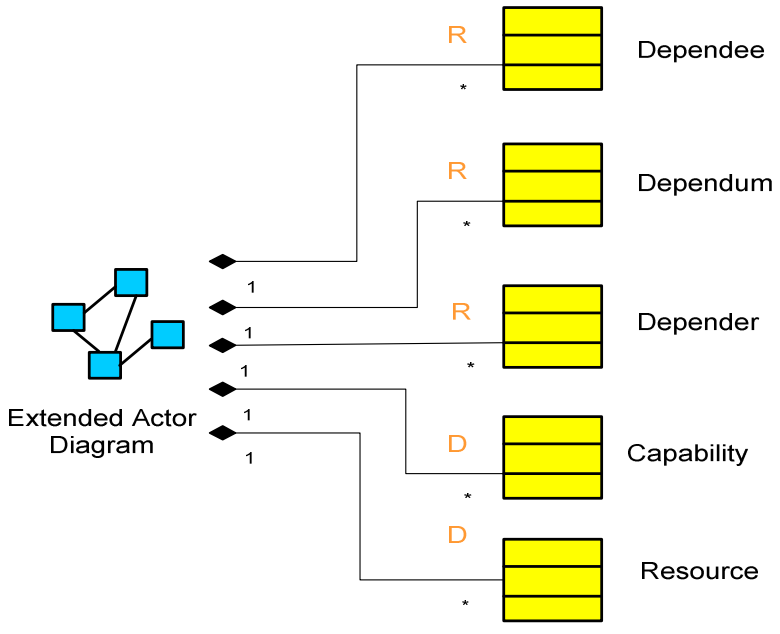
In output, oltre all'Extended Actor Diagram, si ottiene un documento di testo che elenca tutti gli actors con le rispettive capabilities assegnategli, come come mostrato nella seguente figura dell'articolo [1]:

Table 1. Actors' capabilities

Actor Name	*N*	Capability
Area Classifier	1	Get area specification form
	2	Classify area
	3	Provide area information
	4	Provide service description
Info Searcher	5	Get area information
	6	Find information source
	7	Compose query
	8	Query source
	9	Provide query information
		Provide service description
Results Synthesizer	10	Get query information
	11	Get query results
	12	Provide query results
	13	Synthesize area query results
		Provide service description
Sources Interface Manager	14	Wrap information source
		Provide service description
Sources Broker	15	Get source description
	16	Classify source
	17	Store source description
	18	Delete source description
	19	Provide sources information
		Provide service description
Services Broker	20	Get service description
	21	Classify service
	22	Store service description
	23	Delete service description
	24	Provide services information
User Interface Manager	25	Get user specification
	26	Provide user specification
	27	Get query results
	28	Present query results to the user
		Provide service description

3.3.2.2.3 Relazioni con il MAS Meta-Model

La seguente figura illustra le relazioni tra il work product ottenuto e il MAS meta-model:



Il simbolo  rappresenta un elemento del MAS Meta-Model.

La lettera D (*defined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato introdotto per la prima volta nel processo.

La lettera R (*refined*) indica che l'elemento del MAS Meta-Model a cui fa riferimento è stato definito precedentemente ed ora è stato raffinato.

3.3.2.2.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Sub-Actors Diagram	Dependee	Extended Actor Diagram
	Dependum	Actor's Capabilities
	Depender	
	Resources	
	Capabilities	

3.3.2.2.5 Glossario

Il frammento Define Capability usa i seguenti elementi:

Dependee – è un elemento della dependency, cioè l'actor che possiede "l'oggetto" richiestogli.

Dependum - è "l'oggetto" della dependency (come l'esecuzione di un plan, lo sfruttamento di una resource, ecc.).

Depender - è un elemento della dependency, cioè un actor che richiede un "oggetto" ad un altro actor.

Resources - rappresentano un'entità fisica o un insieme di informazioni di cui un actor ha necessità. Se l'actor non possiede la resource di cui necessita, viene richiesta, attraverso relazioni di dependency, dall'actor che la possiede.

Capabilities - rappresentano l'abilità di un actor nel definire, scegliere ed eseguire un plan per il raggiungimento di un goal, date alcune condizioni del mondo, e in presenza di un evento specifico.

3.3.2.2.6 Relazioni con gli altri frammenti del processo

Il work product realizzato in questo frammento è utilizzato come input del successivo frammento, Define Agent, nell'activity Identify Agent, per l'identificazione degli agenti, mentre è utilizzato come input del frammento Capability Description, della fase *Detailed Design*, congiuntamente all'Actor's Capabilities e all'Agent's Capabilities (prodotto nel successivo frammento) come

supporto all'individuazione delle conoscenze di base (belief) che ogni agente deve possedere per espletare al meglio il proprio ruolo.
L'Actor's Capabilities, oltre a quanto già detto, è utilizzato come input del successivo frammento, Define Agent, nell'activityAssignment Capability, per assegnare le capabilities necessarie ad ogni actor.

3.3.2.3 Frammento “Define Agent Society”

3.3.2.3.1 Descrizione

Partendo dall’Extended Actor Diagram, vengono individuati, dall’Agent Designer, gli agenti che verranno implementati nel sistema, assegnando a questi, sulla base dell’Actor’s Capabilities, le capability che ognuno di questi dovrà possedere per svolgere determinati plans al fine di raggiungere i relativi goals.

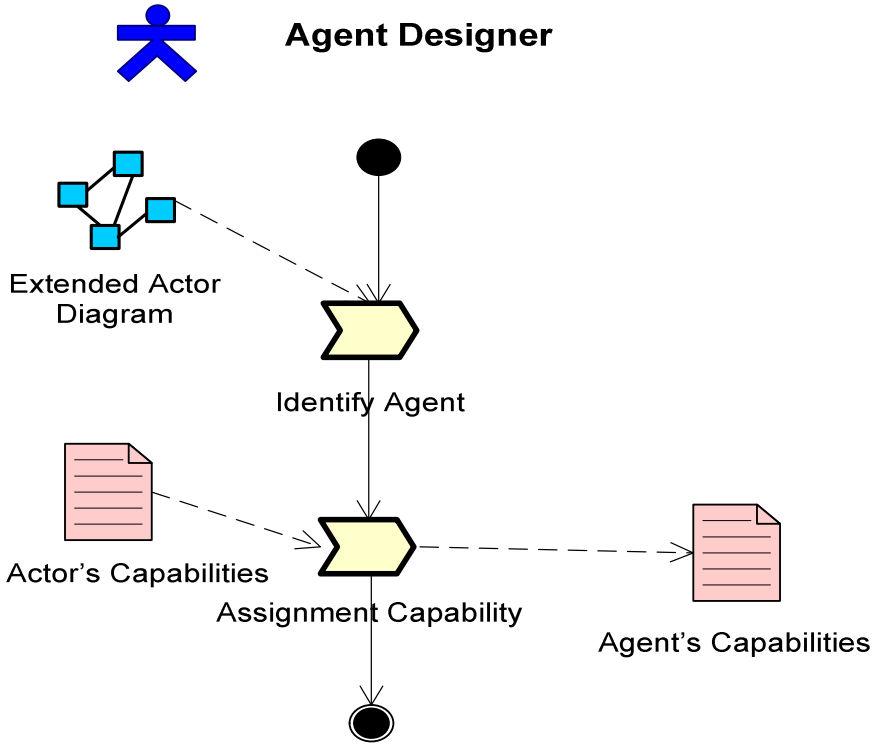


Figura 20. Frammento Define Agent Society

3.3.2.3.2 Notazione

Ogni agente deve avere innanzitutto un nome univoco e ad ognuno di essi può essere assegnata una o più capabilities tra quelle elencate nell’Actor’s Capabilities. In generale il procedimento di assegnamento di quest’ultime non è unico, ma dipende dalle scelte dell’Agent Designer. Tropos, per supportare l’Agent Designer, offre un insieme di patterns predefiniti ricorrenti nella letteratura multi-agente, come mostrato in [12].
In output si ottiene un documento di testo, come mostrato nella seguente figura, che fa riferimento all’articolo [1]:

Table 2. Agent types and their capabilities.

Agent	Capabilities
Query Handler	1, 3, 4, 5, 7, 8, 9, 10, 11, 12
Classifier	2, 4
Searcher	6, 4
Synthesizer	13, 4
Wrapper	14, 4
Agent Resource Broker	15, 16, 17, 18, 19, 4
Directory Facilitator	20, 21, 22, 23, 24, 4
User Interface Agent	25, 26, 27, 28, 4

3.3.2.3.3 Relazioni con il MAS Meta-Model

Non ci sono relazioni dirette con il MAS Meta-Model.

3.3.2.3.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Actor's Capabilities		Agent's Capabilities
Extended Actor Diagram		

3.3.2.3.5 Glossario

Il frammento Define Agent Society usa i seguenti elementi:

Agent - è un modulo software, derivante dall'actor, che possiede capabilities, goals, plans.

3.3.2.3.6 Relazioni con gli altri frammenti del processo

Il documento prodotto da questo frammento è utilizzato, insieme all'Actor's Capabilities e all'Extend Actor Diagram, come input del frammento Capability Description, nella fase *Detailed Design*, per la modellazione delle capabilities; è usato inoltre nel frammento della fase *Implementation* per la generazione del codice e della documentazione.

3.4 La fase Detailed Design

La fase *Detailed Design* coinvolge due differenti process roles e produce tre work products.

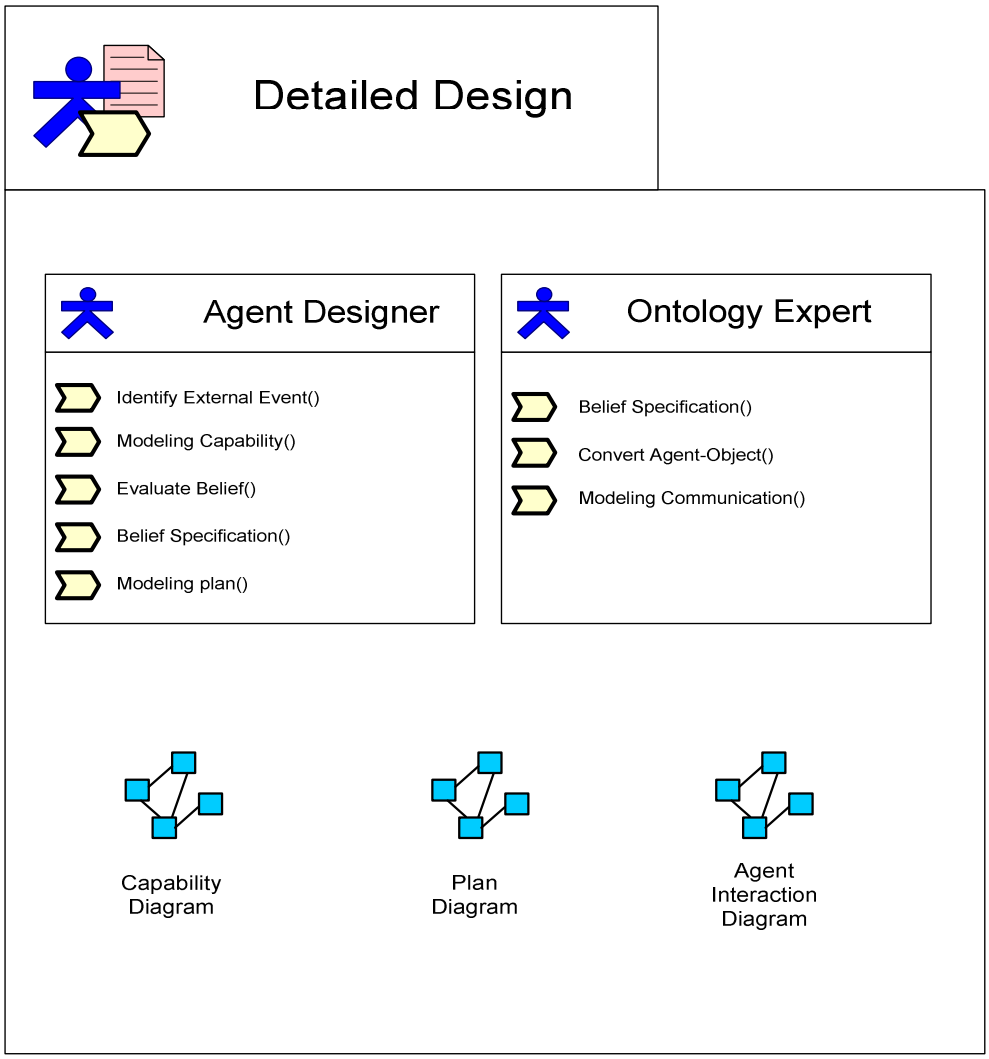


Figura 21. Descrizione della fase *Detailed Design* rappresentata come una disciplina SPEM

Il processo che deve essere eseguito in questa fase è descritto nella seguente figura. Questa è composta da tre work definitions (Capability Description, Plan Description e Interaction Description) e i relativi work products.

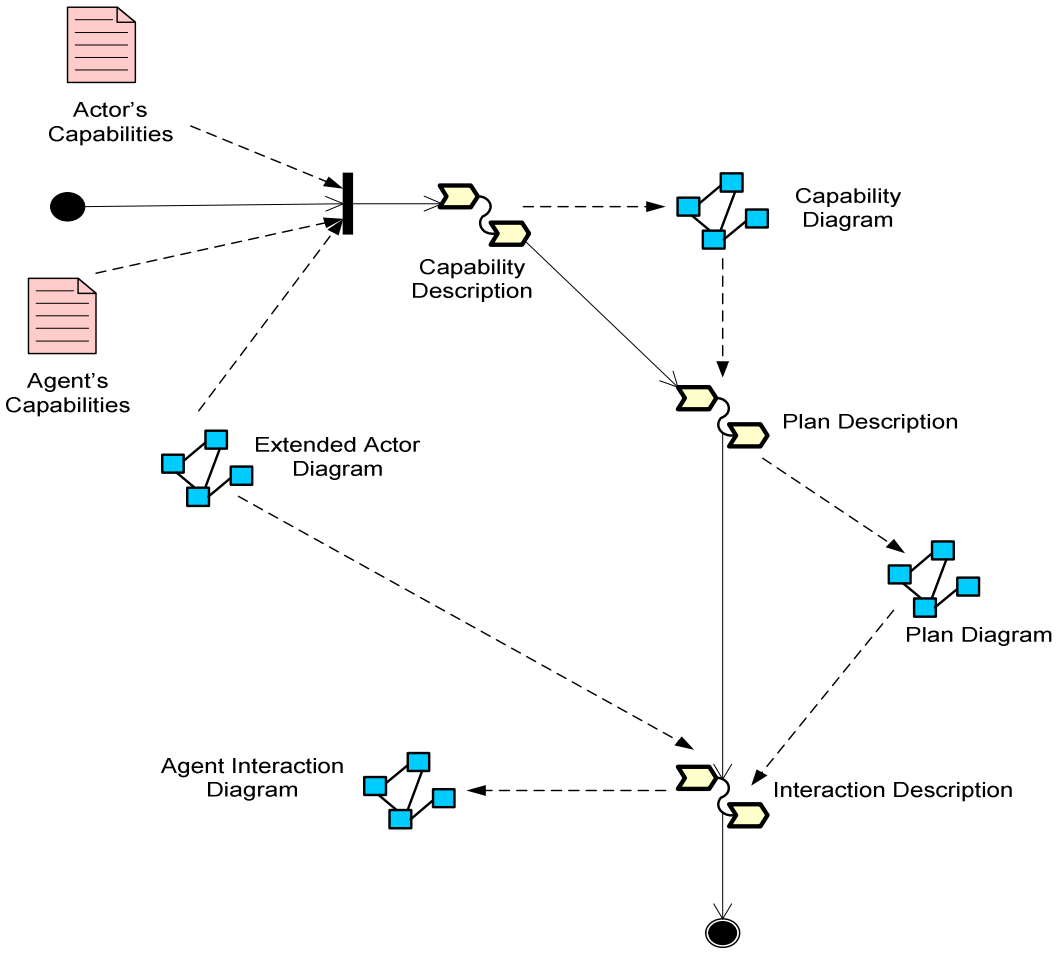


Figura 22. La fase *Detailed Design* descritta in termini di work definitions (sotto-fasi) e work products

Le tre sotto-fasi sono composte da alcune activities come descritto nella figura 23 . Questa figura descrive anche i process roles coinvolti in questa parte del processo. Come il processo si svolge, sarà descritto nelle seguenti sotto-sezioni.

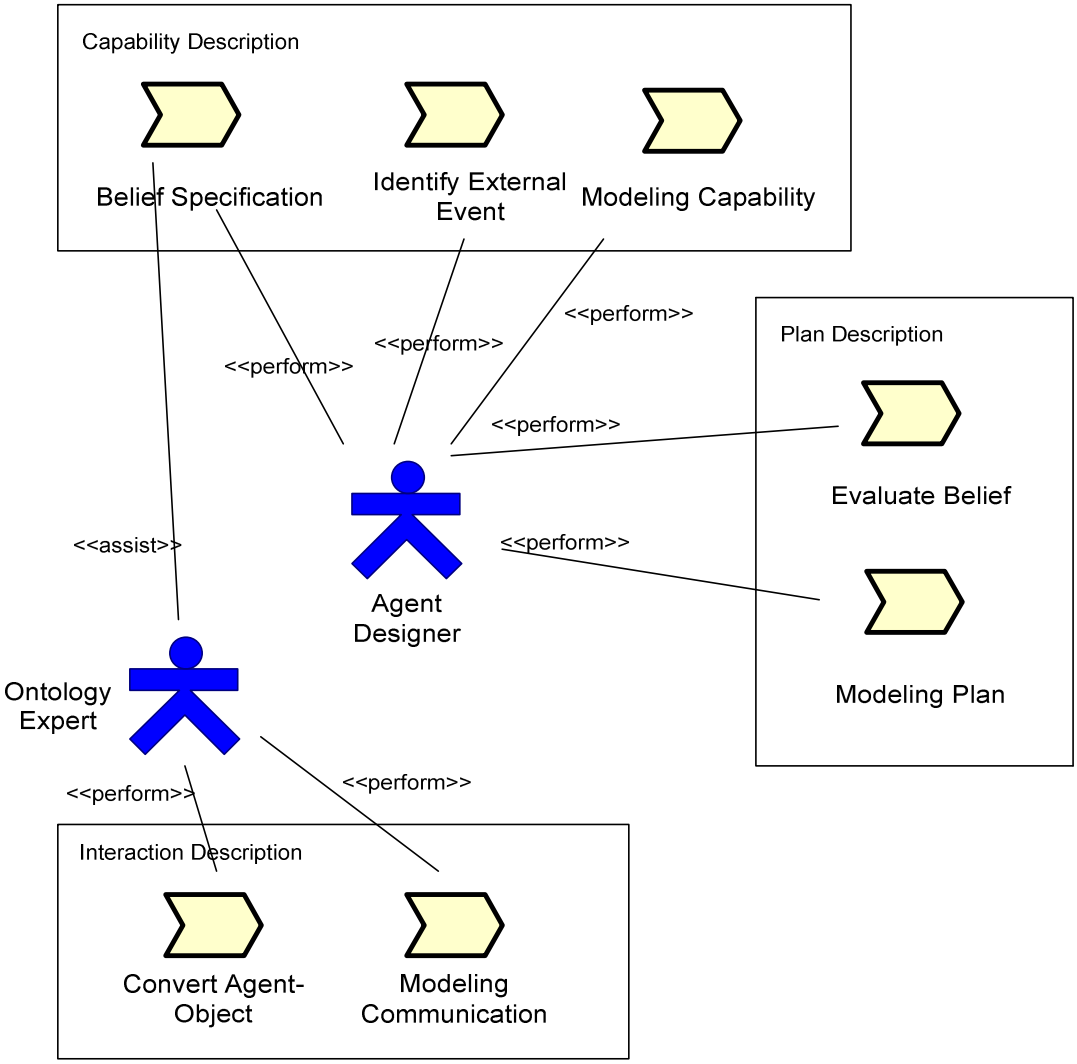


Figura 23. Le activities della Detailed Design divise in work definitions

Ecco qui di seguito un sommario delle activities di questa fase:

Work Definition	**Activity**	**Descrizione dell'Activity**	**Process Roles Coinvolti**
Capability Description	Belief Specification	Elencare le beliefs di ogni agente	Agent Designer (perform) Ontology Expert (assist)
Capability Description	Identify External Event	Elencare gli eventi scatenanti per ogni capability	Agent Designer (perform)
Capability Description	Modeling Capability	Modellare ogni capability come un diagramma di attività UML	Agent Designer (perform)
Plan Description	Evaluate Belief	Valutare quali beliefs, per ogni agente, siano	Agent Designer (perform)

		necessarie nell'esecuzione di un plan	
Plan Description	Modeling Plan	Modellare ogni plan come un diagramma di attività UML	Agent Designer (perform)
Interaction Description	Convert Agent-Object	Convertire gli agenti coinvolti in una comunicazione, in oggetti del diagramma di sequenza	Ontology Expert (perform)
Interaction Description	Modeling Communication	Modellare ogni comunicazione tra agenti in un diagramma di sequenza AUML	Ontology Expert (perform)

3.4.1 Process Roles coinvolti

Due process roles sono coinvolti nella disciplina *Detailed Design*, descritti qui di seguito:

3.4.1.1 Agent Designer

E' responsabile di:

1. Elencare le beliefs di ogni agente, in questo è assistito dall'Ontology Expert;
2. Elencare, per ogni capability, quali siano gli eventi esterni che scatenino l'attivazione della stessa;
3. Modellare ogni capability come un diagramma di attività UML: le azioni rappresentano i plans e le beliefs sono modellate come oggetti;
4. Valutare quali beliefs entrino in gioco nella definizione di ogni plan della singola capability;
5. Modellare ogni plan come un diagramma di attività UML.

3.4.1.2 Ontology Expert

E' responsabile di:

1. Assiste l'Agent Designer nel riconoscere le beliefs di ogni agente;
2. Effettuare il mapping degli agenti coinvolti in una comunicazione, in oggetti di un diagramma di sequenza UML;
3. Modellare le comunicazioni tra gli agenti come diagrammi di sequenza AUML, la life-line di ognuno di questi è indipendente dalla specifica interazione modellata;

3.4.2 Frammenti della *Detailed Design*

In questa fase, sono individuati tre frammenti: Capability Description, Plan Description e Interaction Description, di seguito descritti.

3.4.2.1 Frammento “Capability Description”

3.4.2.1.1 Descrizione

Partendo dall’Extended Actor Diagram, dall’Actor’s Capabilities e dall’Agent’s Capabilities, l’Agent Designer specifica le beliefs di ogni actor (cioè le basi di conoscenza del mondo che questi possiede) e modella le capabilities. Per far ciò, partendo dal punto di vista di ogni singolo agente, si definisce un Capability Diagram per ogni capability, e si identifica l’evento esterno che dà inizio al diagramma. Un Capability Diagram è un diagramma di attività UML, dove gli stati rappresentano i plans, gli archi di transizione definiscono la fine dello stato e le beliefs vengono modellate come oggetti.

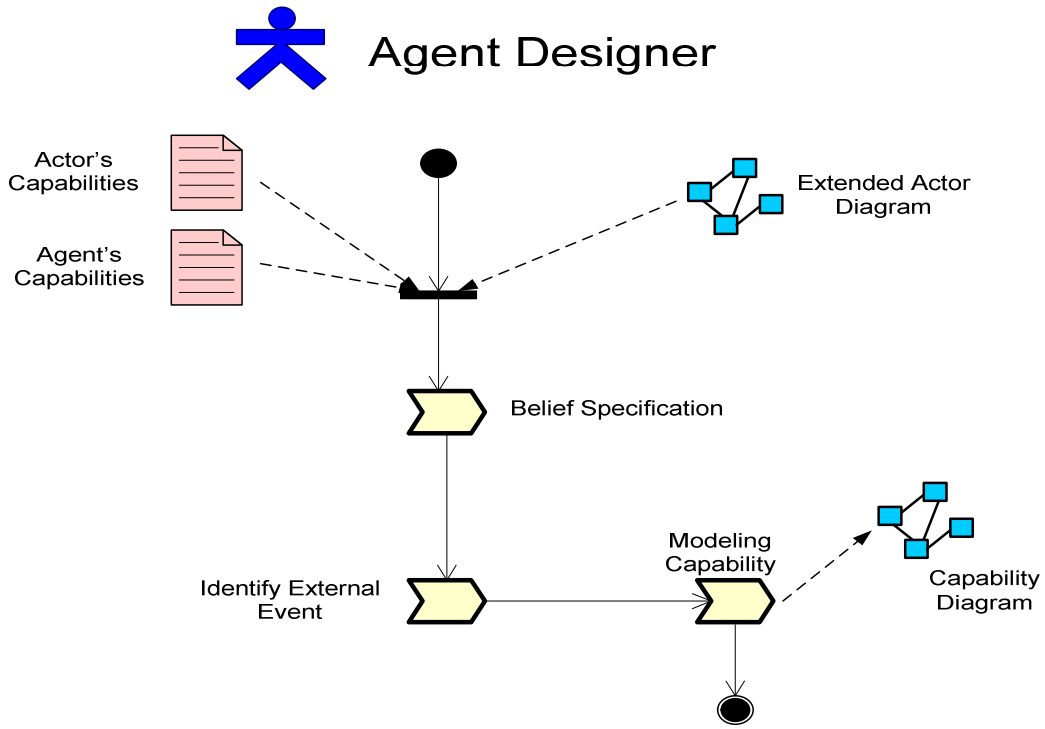


Figura 24. Frammento Capability Description

3.4.2.1.2 Notazione

Ogni capability, partendo dal punto di vista di ogni agente, viene modellata come diagramma di attività UML.
Un evento esterno dà inizio al capability diagram, gli stati modellano i piani, gli archi di transizione determinano la fine di uno stato, e le beliefs sono modellate come oggetti.
Un esempio di quanto detto è riportato nella seguente fig. che fa riferimento all’articolo [1]:

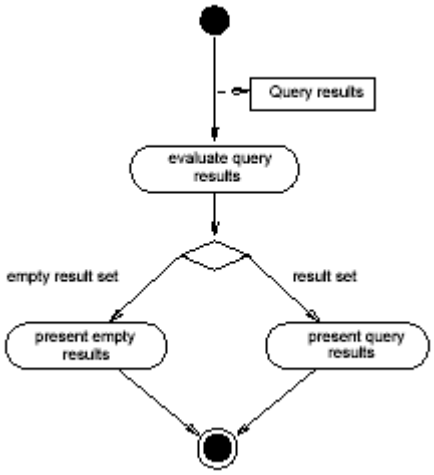


Figure 9. Capability diagram represented as an UML activity diagram.

3.4.2.1.3 Relazioni con il MAS Meta-Model

La seguente figura illustra le relazioni tra il work product ottenuto e il MAS meta-model:

Il simbolo  rappresenta un elemento del MAS Meta-Model.

La lettera D (*defined*) indica che l'elemento del MAS Meta-Model cui fa riferimento è stato introdotto per la prima volta nel processo.

3.4.2.1.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Extended Actor Diagram	Belief	Capability Diagram
Actor's Capabilities		

Agent's capabilities		

3.4.2.1.5 Glossario

Il frammento Capability Description usa il seguente elemento:

Belief - rappresenta la conoscenza del mondo di un actor.

3.4.2.1.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato come input del frammento Plan Description per la modellazione dei plans, è usato inoltre dal frammento della fase *Implementation* per la generazione del codice e della documentazione.

3.4.2.2 Frammento “Plan Description”

3.4.2.2.1 Descrizione

Partendo dai Capability Diagram, in questo frammento vengono valutate le basi di conoscenza possedute da ogni agente (beliefs), e vengono modellati i singoli plans di ogni Capability Diagram, realizzati con diagrammi di attività UML, che descrivono come il plan venga eseguito quando, nel caso di varie possibilità d’azione, ne venga scelta una piuttosto che un’altra.

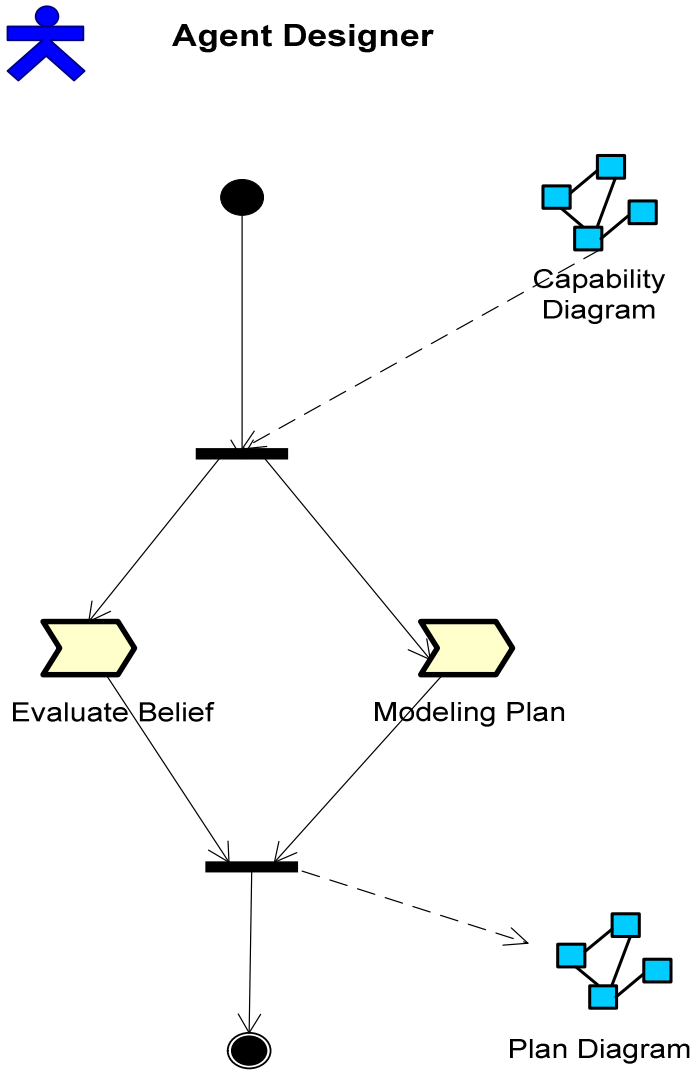


Figura 25. Frammento Plan Description

3.4.2.2.2 Notazione

Ogni nodo del Capability Diagram viene ulteriormente dettagliato con un diagramma di attività UML, come mostrato nella seguente fig. che fa riferimento all'articolo [1]:

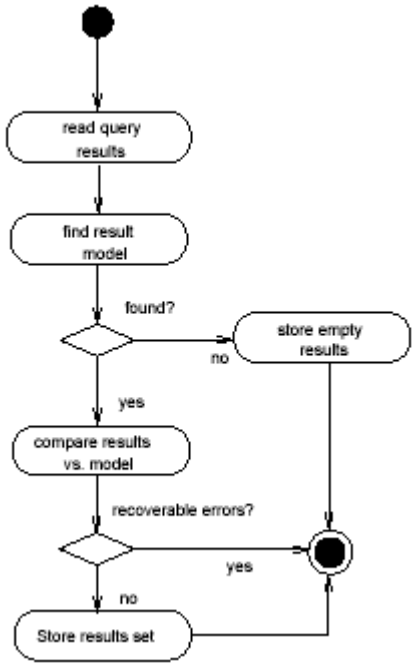


Figure 10. Plan diagram for the plan evaluate query. Ovals correspond to simple or complex actions, arcs to transitions from an action to the subsequent one, start and end states transitions to events.

3.4.2.2.3 Relazioni con il MAS Meta-Model

Non ci sono relazioni dirette con il MAS Meta-Model.

3.4.2.2.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Capability Diagram		Plan Diagram

3.4.2.2.5 Glossario

Non ci sono elementi da definire.

3.4.2.2.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato come input del frammento Interaction Description (di questa stessa fase) per la modellazione delle interazioni, è usato inoltre dal frammento della fase *Implementation* per la generazione del codice e della documentazione.

3.4.2.3 Frammento “Interaction Description”

3.4.2.3.1 Descrizione

L’Ontology Expert, partendo dal Plan Diagram e dall’Extended Actor Diagram, modella le interazioni tra gli agenti definendo un Agent Interaction Diagram che consiste in un diagramma di sequenza AUML; converte inoltre gli agenti in oggetti e definisce i messaggi che questi si scambiano.

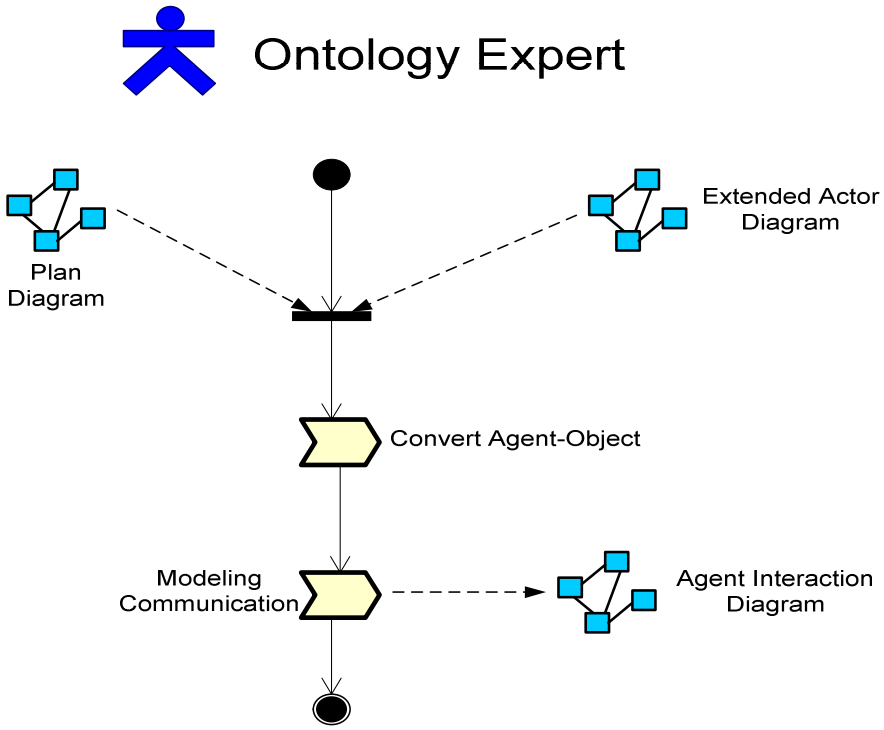


Figura 26. Frammento Interaction Description

3.4.2.3.2 Notazione

Ogni agente corrisponde ad un oggetto, e per modellare la comunicazione fra essi si realizzano dei diagrammi di sequenza AUML, dove gli archi corrispondono a messaggi asincroni che si scambiano gli agenti e la life-line è indipendente dalla specifica interazione che modella la comunicazione, come mostrato nella seguente figura dell’articolo [1]:

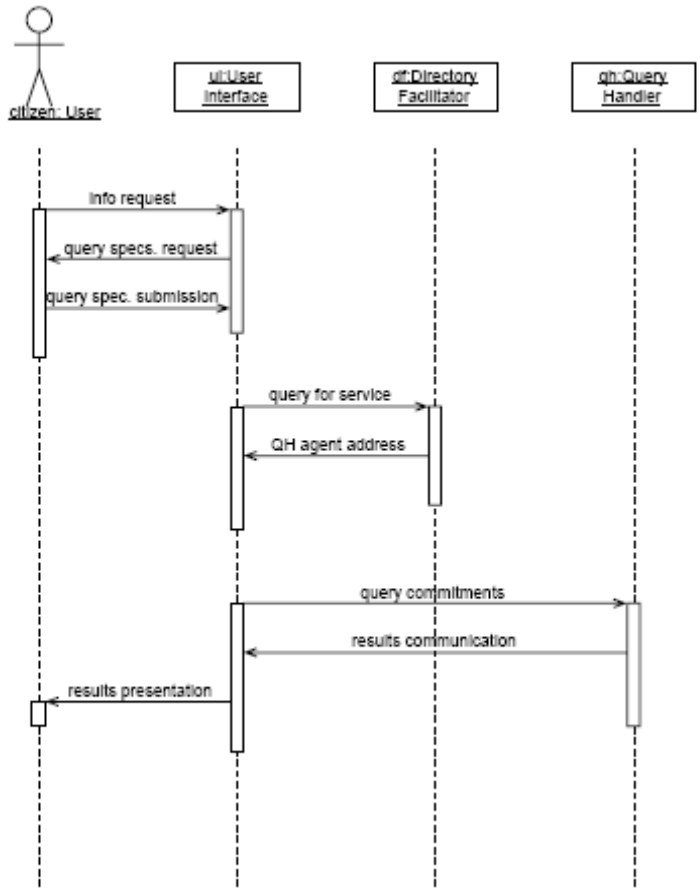


Figure 11. Agent interaction diagram. Boxes represent agents and arrows model communicative acts.

3.4.2.3.3 Relazioni con il MAS Meta-Model

Non ci sono relazioni dirette con il MAS Meta-Model.

3.4.2.3.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Plan Diagram		Agent Interaction Diagram
Extended Actor Diagram		

3.4.2.3.5 Glossario

Non ci sono elementi da definire.

3.4.2.3.6 Relazioni con gli altri frammenti del processo

Il work product prodotto da questo frammento è utilizzato come input del frammento della fase successiva (*Implementation*) per la generazione del codice e della documentazione.

3.5 La fase Implementation

La fase *Implementation* coinvolge un solo process role, produce due "documenti di testo", e implica la scelta di una piattaforma ad agenti, nel nostro caso, si suppone venga usata la piattaforma JACK poiché permette un facile mapping tra gli elementi di *Tropos* e i propri costrutti.

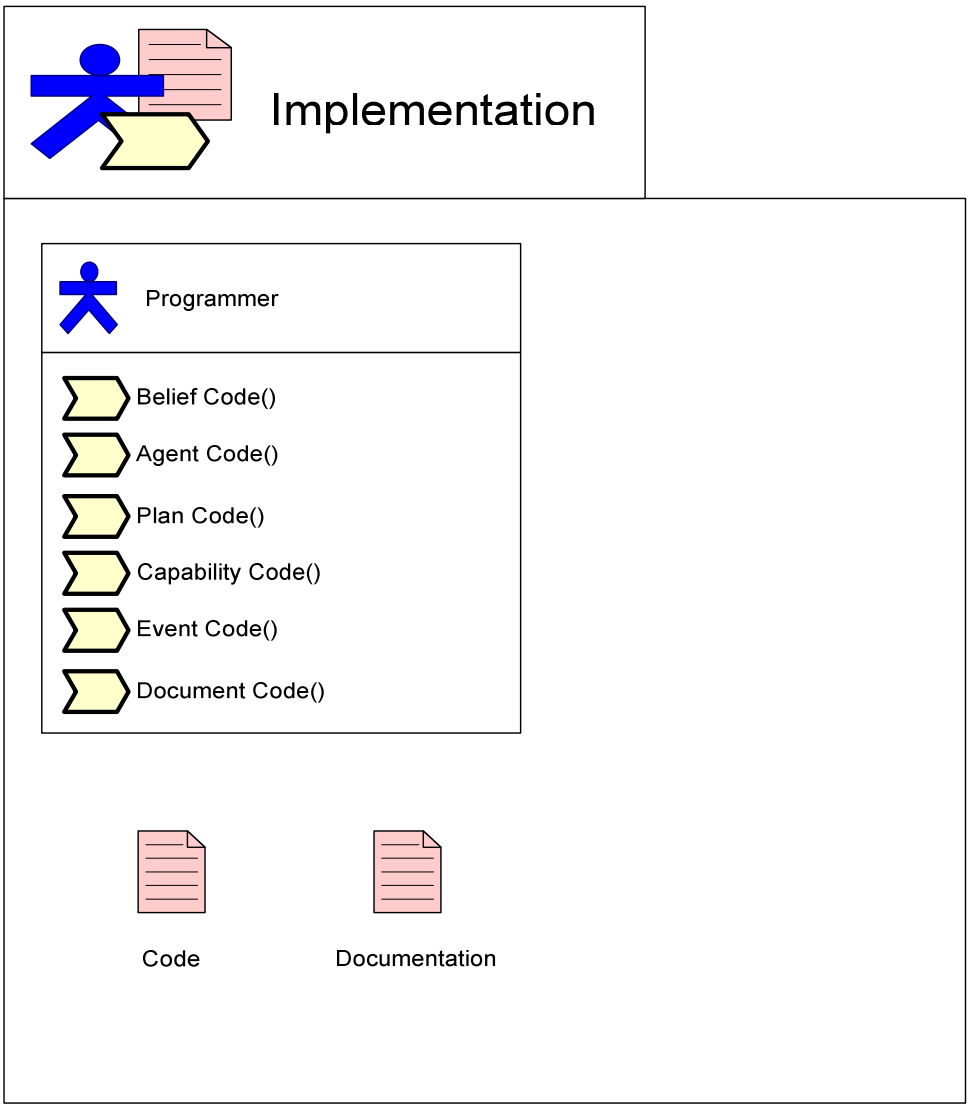


Figura 27. Descrizione della fase *Implementation* rappresentata come una disciplina SPEM

Il processo che deve essere eseguito in questa fase è descritto nella seguente figura. Questa è composta da una work definition (Codification) e i relativi work products.

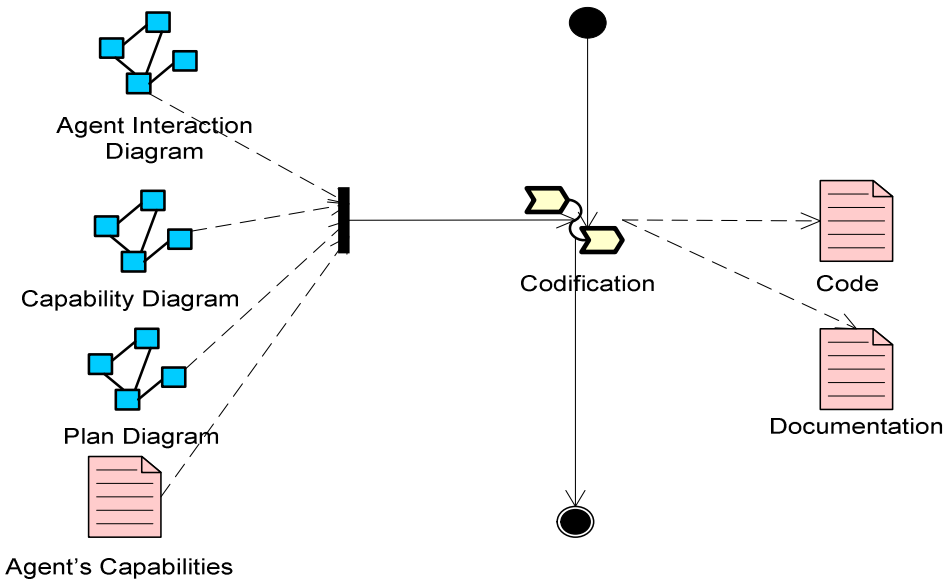


Figura 28. La fase *Implementation* descritta in termini di work definitions (sotto-fasi) e work products

La sotto-fase è composta da alcune activities come descritto nella figura 29 . Questa figura descrive anche il process role coinvolto in questa parte del processo. Come il processo si svolge, sarà descritto nelle seguenti sotto-sezioni.

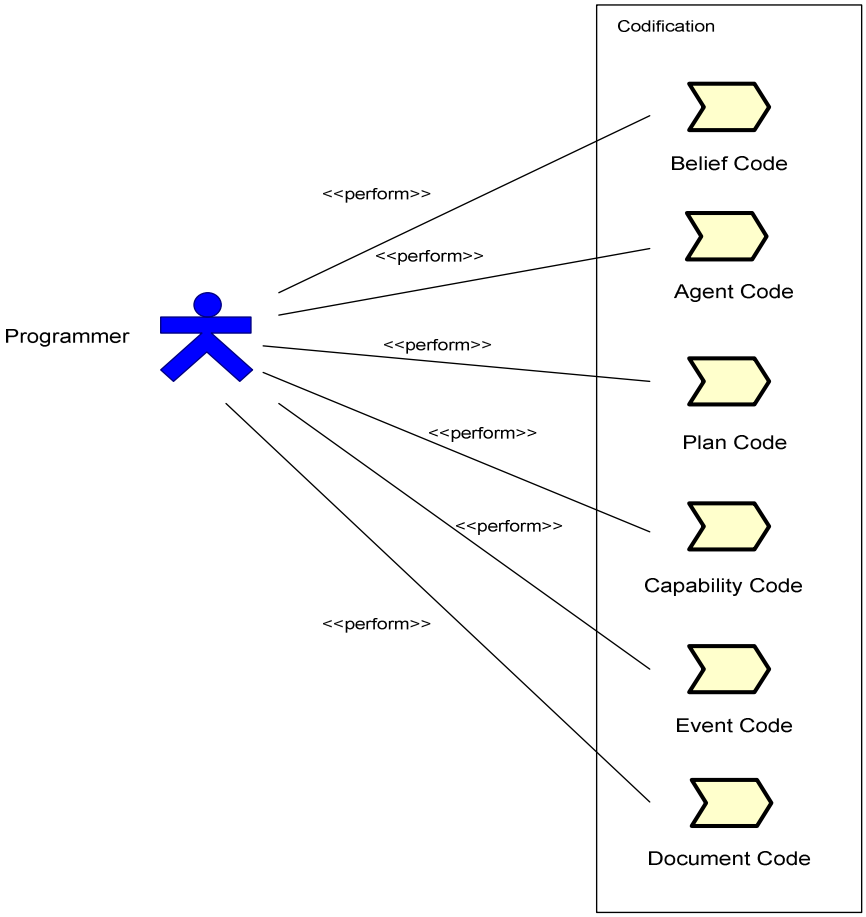


Figura 29. Le activities della *Implementation* divise in work definitions

Ecco qui di seguito un sommario delle activities di questa fase:

Work Definition	Activity	Descrizione dell'Activity	Process Roles Coinvolti
Codification	Agent code	Definisce il codice dell'agente	Programmer (perform)
Codification	Belief code	Definisce il database	Programmer (perform)
Codification	Plan code	Definisce il codice dei plans	Programmer (perform)
Codification	Capability code	Definisce il codice riguardante le capabilities	Programmer (perform)
Codification	Event code	Definisce gli eventi scatenanti le capabilities	Programmer (perform)
Codification	Document code	Crea la documentazione del codice	Programmer (perform)

3.5.1 Process Roles coinvolti

Un process role è coinvolto nella disciplina *Implementation*, descritto qui di seguito:

3.5.1.1 Programmer

E' responsabile di:

1. Definire il codice degli agenti. In particolare, un agente JACK è usato per definire un comportamento di un agente software intelligente;
2. Definisce il database relazionale che descrive le beliefs assegnate agli agenti;
3. Definisce il codice dei plans. In particolare, in JACK un plan è una sequenza di istruzioni che l'agente deve eseguire per raggiungere i goals;
4. Definisce il codice riguardante le capabilities;
5. Definisce il codice degli eventi scatenanti le capabilities;
6. Documenta il codice ottenuto.

3.5.2 Frammento dell'*Implementation*

In questa fase, è individuato un unico frammento: "Codification", descritto di seguito.

3.5.2.1 Frammento "Codification"

3.5.2.1.1 Descrizione

In questo frammento, partendo dalla piattaforma BDI scelta, e basandosi sulle specifiche ottenute dalla *Detailed Design*, si procede alla codifica dell'intero sistema multi-agente. In *Tropos*, si

segnala la piattaforma JACK, che permette un mapping immediato tra gli elementi ottenuti nella precedente fase (Agents, Beliefs, Events, Plans e Capabilities), e i suoi costrutti.

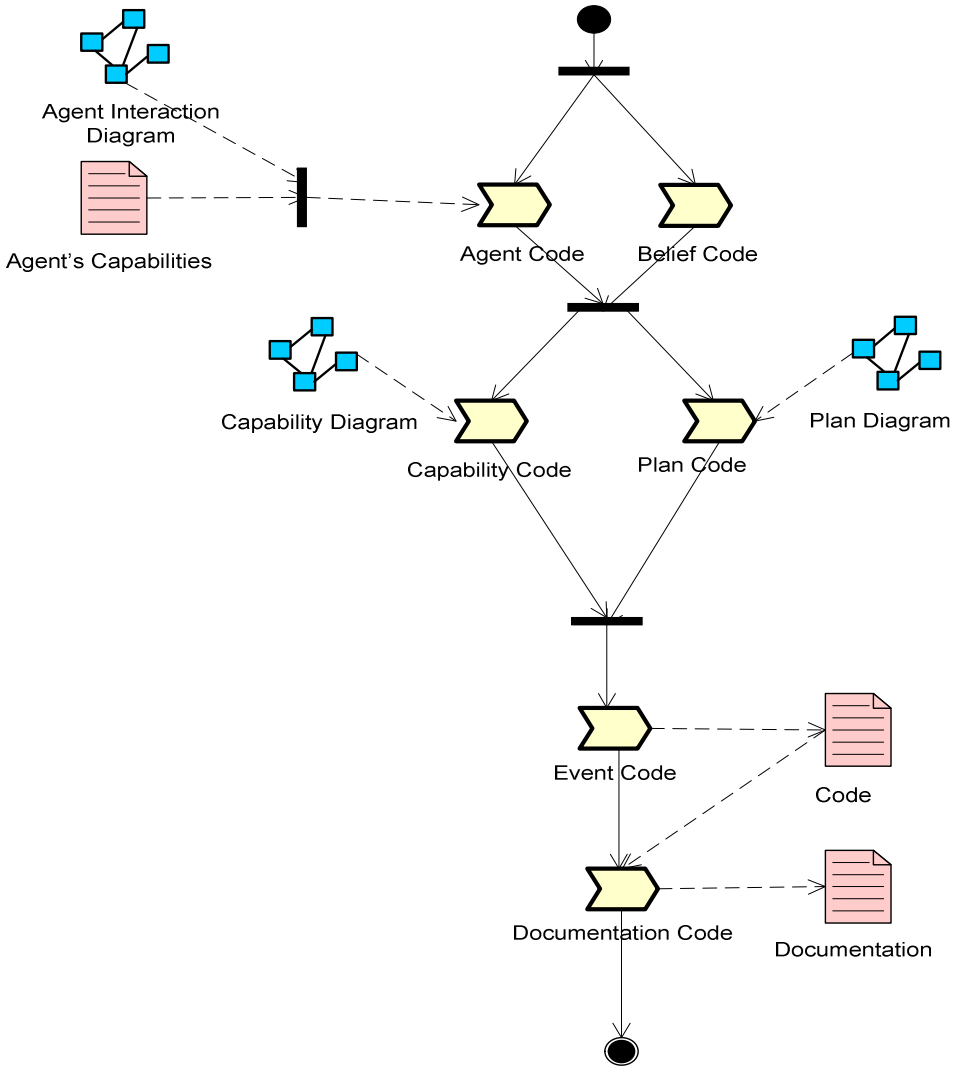


Figura 30. Frammento Codification

3.5.2.1.2 Notazione

Supponiamo che la piattaforma BDI d'implementazione sia JACK, in quanto esiste una diretta corrispondenza tra i suoi costrutti e gli elementi del processo *Tropos*. Infatti:
-Un agente JACK è usato per definire il comportamento di un agente software intelligente; includendo le capabilities che esso possiede e i plans che egli usa per raggiungere i suoi goals;
-Una JACK capability può includere plans, events, beliefs e altre capabilities; ad un'agente possono essere assegnate più capabilities;
-La belief, in JACK, è il database che raccoglie le conoscenze di un agente;
-Un event, in JACK, descrive una triggering condition per le azioni degli agenti;

-Un JACK plan è una sequenza di istruzioni che l'agente esegue per raggiungere i suoi goals; questo è contenuto in una capability.

La seguente figura dell'articolo [1] mostra l'ambiente di sviluppo JACK. La prima finestra si concentra sulla dichiarazione di tutti gli agenti; la seconda elenca tutte le capabilities associate agli agenti e la terza mostra i plans associati con una capability.

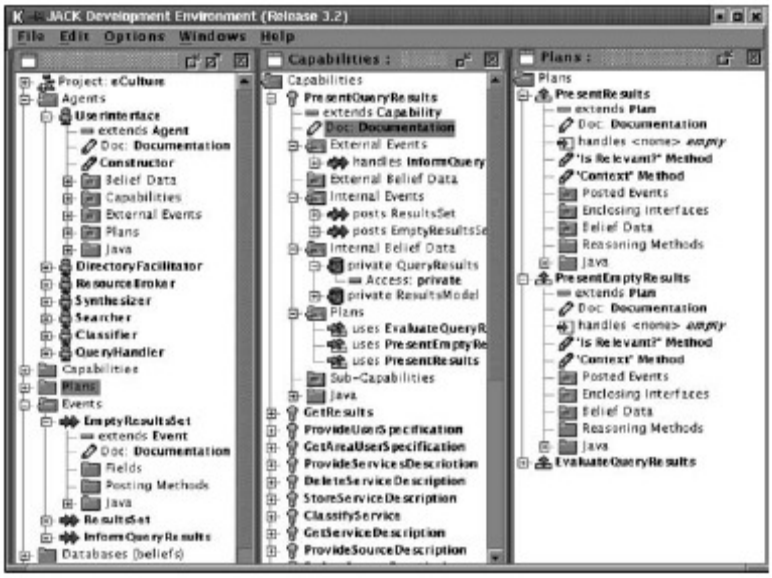


Figure 12. JACK Developing Environment for the eCulture project.

3.5.2.1.3 Relazioni con il MAS Meta-Model

Non ci sono relazioni dirette con il MAS Meta-Model.

3.5.2.1.4 Input/Output

Gli inputs, gli outputs, e gli elementi che devono essere progettati in questo frammento, sono descritti nella seguente tabella:

Input	Elementi da definire	Output
Agent Interaction Diagram		Code
Capability Diagram		Documentation
Plan Diagram		
Agent's Capability		

3.5.2.1.5 Glossario

Non ci sono elementi da definire.

3.5.2.1.6 Relazioni con gli altri frammenti del processo

Non ci sono relazioni con altri frammenti del processo.

4 Dipendenze tra i Work Products

Il seguente diagramma descrive le dipendenze tra i vari work products realizzati durante tutte le fasi del processo di sviluppo *Tropos*.
P. es. il Capabilty Diagram dipende dall'Actor's Capabilities, dall'Extended Actor Diagram e dall'Agent's Capabilities, e permette, all'Agent Designer, di ottenere le specifiche per la definizione delle beliefs di ogni actor e modellarne le relative capabilities.

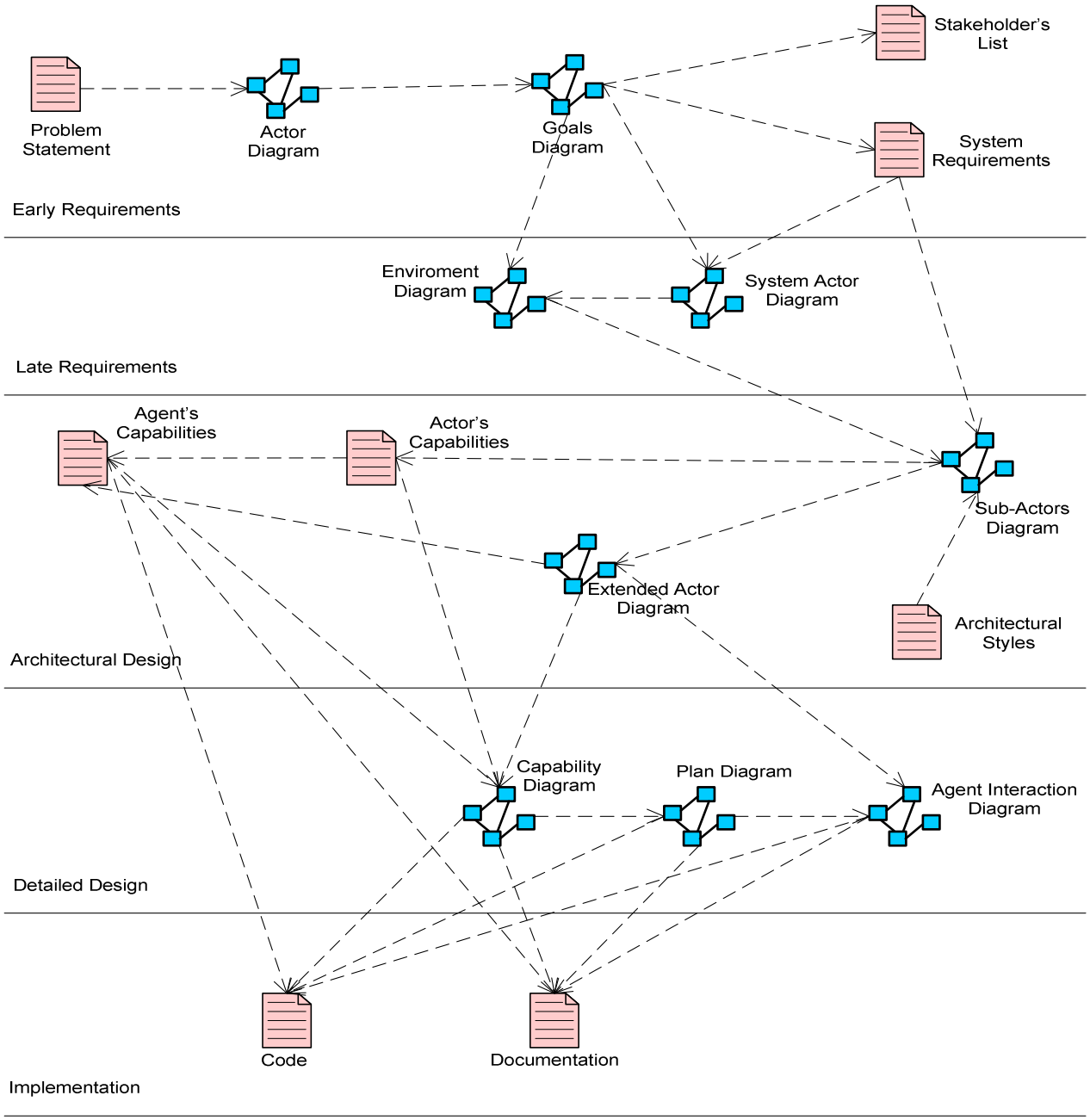


Figura 31. Dipendenze tra i work products

5 MAS Meta-Model

5.1 Generalità

Il linguaggio di modellazione, è il punto centrale attorno al quale ruota tutta la metodologia *Tropos*, la cui sintassi è definita in termini di meta-modello UML.
Seguendo gli approcci standard (per approfondimenti vedi [11]), il meta-modello è organizzato in quattro livelli, come mostrato nella tabella 3, questo rende *Tropos* un linguaggio estensibile, nel senso che si possono aggiungere facilmente nuovi costrutti.

Livello	Descrizione	Esempi
Meta-Meta-modello	Linguaggio di base per gli elementi strutturali	Attributi, Entità
Meta-Modello	Nozioni del livello di conoscenza	Actor, Goal, Dependency
Dominio	Entità del dominio applicativo	Istanze di Actor, Goal, Dependency: cittadino, ecc.
Istanze	Istanze del modello del dominio	Istanze di cittadino: Mario, ecc.

Tabella 3. Architettura a quattro livelli del meta-modello di *Tropos*

Il livello **Meta-Meta-modello** fornisce le basi per il livello sottostante, in particolare esso contiene primitive di linguaggio che permettono un facile inserimento di nuovi costrutti.
Il livello **Meta-modello** fornisce i costrutti per modellare le entità e i concetti del livello di conoscenza.
Il livello **Dominio** contiene la rappresentazione delle entità e dei concetti di un domino applicativo specifico, costruiti come istanze dei costrutti del livello superiore.
Il livello **Istanze** contiene le istanze del livello dominio.

5.2 Elementi

I modelli concettuali di *Tropos*, vengono sviluppati come istanze dei seguenti concetti del Meta-Model (vedi anche [1]):

Actor: rappresenta un'entità che ha obiettivi strategici e volontà, nel sistema o nell'assetto organizzativo. Rappresenta un agente fisico (p. es. una persona, una macchina, ecc.), o un agente software, che può avere ruoli o ricoprire posizioni. Per ruolo si intende il comportamento che assume l'agente software in un contesto specifico, mentre per posizione si intende, semplicemente, un insieme di ruoli. Alla fine, ciò che si intende per agente, non si discosta molto dal concetto che dello stesso si ha in IA.

Goals: rappresentano gli interessi strategici degli actors. Si distinguono in hard e soft, in base alla necessità o meno di un loro soddisfacimento; solitamente i softgoals, sono utili per modellare le

qualità del software, quali sicurezza, performance e maintainability. Hard e soft goals hanno rappresentazione grafica differente, e a ognuno di essi deve essere associato almeno un actor (il depender, cioè, deve essere sempre definito).

Plans: rappresentano un insieme di procedure la cui esecuzione, solitamente, porta al soddisfacimento dei goals.

Resources: rappresentano un'entità fisica o un insieme di informazioni di cui un actor ha necessità. Se l'actor non possiede la resource di cui necessita, viene richiesta, attraverso relazioni di dependency, all'actor che la possiede.

Dependency: è la relazione tra due actors, esprimente il fatto che uno di essi, il *depender*, necessita, per qualche motivo, di un "oggetto", chiamato *dependum* (come l'esecuzione di un plan, lo sfruttamento di una resource, ecc.) posseduto da un altro actor, il *dependee*. Le dependencies permettono, ad un actor, di raggiungere uno o più goals, che da soli non potrebbero soddisfare, o la cui realizzazione risulterebbe difficoltosa o dispendiosa. Allo stesso tempo, però, si può avere una "vulnerabilità" da parte del *depender* in quanto, se il *dependee* non riesce a fornire il servizio richiestogli, il *depender* può vedere compromessa la sua capacità di raggiungere i propri goals.

Capability: rappresenta l'abilità di un actor nel definire, scegliere ed eseguire un plan, per il raggiungimento di un goal, date alcune condizioni del mondo, e in presenza di un evento specifico.

Belief: rappresenta la conoscenza del mondo di un actor.

5.3 Metodi di modellazione

Ognuno degli elementi descritti nel paragrafo precedente (escludendo resources e beliefs) viene modellato attraverso vari metodi specifici che forniscono work products, che vengono via via raffinati nello sviluppo del processo (vedi anche [1]).
Tali metodi sono:

Actor modeling: consiste nell'identificazione e nell'analisi degli actors dell'ambiente, dell'actor system-to-be e degli agenti. In particolare, nella fase *early requirements*, questa attività si concentra nella modellazione del dominio d'applicazione delle parti interessate e dei goals che perseguono. Graficamente, sono rappresentati attraverso *actor diagrams*, in cui l'actor è rappresentato da un cerchio, i goals da ovali, e i softgoals da nuvolette. Durante la fase *late requirements*, l'*actor modeling* si concentra nella definizione dell'actor system-to-be, insieme alle sue dependencies con gli actors dell'ambiente operativo; mentre nella fase *architectural design* si concentra sulla struttura interna dell'actor system-to-be, specificandolo in termini di sub-actors, interconnessi attraverso flussi di dati e controllo modellati come dependencies. Nella fase *detailed design* gli agenti sono definiti specificando tutte le nozioni richieste dalla piattaforma d'implementazione scelta, e infine, durante la fase *implementation* si ha la codifica degli agenti.

Dependency modeling: consiste nell'individuare le relazioni di dipendenza tra un actor e l'altro, per soddisfare goals, eseguire plans, e sfruttare resources. La rete di dependencies tra actors è rappresentata da due frecce connesse da un simbolo grafico che varia in accordo al *dependuum*, in relazione al fatto che questo sia un goal, un plan o una resource. Nella fase *early requirements* si focalizza sulla modellazione delle dependencies tra gli actors dell'assetto organizzativo. Durante la *late requirements* si concentra nell'analizzare le dependencies dell'actor system-to-be con gli actors

individuati nella fase precedente. Nell'*architectural design* i flussi di dati e di controllo fra i sub-actors del system-to-be sono modellati in termini di dependencies fornendo le basi per il *capability modeling* che inizierà più tardi nell'*architectural design* insieme al mapping degli agenti.

Goal modeling: permette al progettista di analizzare i goals dal punto di vista di uno specifico actor, usando tre tecniche di analisi:

1) *Means-end analysis*: procede ridefinendo un goal in sub-goals per identificare quali siano le resources, i plans e i softgoals che forniscono mezzi (means) per raggiungere un goal (end); è una relazione ternaria definita tra un actor, un goal (end) e un plan, una resource o un goal(means);
2) *Contribution analysis*: permette al progettista di individuare i goals che possono contribuire positivamente o negativamente per soddisfare il goal che si sta analizzando; è una relazione ternaria tra un actor e due goals, può essere contrassegnata con una metrica qualitativa, attraverso i simboli +, ++, -, --, secondo l'importanza del contributo dato;
3) *AND-OR decomposition*: applica decomposizioni AND e OR ad un goal radice per ottenere sub-goals, che forniscono un modello più dettagliato; è anch'essa una relazione ternaria definita tra un goal radice e i suoi sub-goals.

La goal modeling è inizialmente sviluppata nella fase *early requirements*, per identificare gli actors e i loro goals. Questi sono rappresentati attraverso *goal diagrams* che rappresentano la prospettiva di uno specifico actor come una coppia di cerchi (actor perspective), un grafo che contiene nodi che sono goals (ovali) e/o plans (esagoni) e archi che rappresentano le relazioni tra i nodi (dependencies). Gioca un ruolo analogo (identificare e giustificare le dependencies tra actors) nella *late requirements* e nell'*architectural design*, durante la quale contribuisce alla realizzazione della decomposizione dell'actor system-to-be in sub-actors.

Plan modeling: può esser considerata come una tecnica d'analisi complementare alla *goal modeling* e utilizza i suoi stessi procedimenti in funzione però dei plans.

Capability modeling: comincia al termine dell'*architectural design*, quando i sub-actors del system-to-be sono stati specificati in termini dei loro goals e dependencies con gli altri actors. Ogni sub-actor del system-to-be deve essere fornito di capabilities specifiche per la definizione, la scelta e l'esecuzione di un plan per il raggiungimento dei propri goals. Inoltre possono essere fornite capabilities sociali per gestire le dependencies con gli altri actors. Nella *detailed design* ogni capabilities dell'agente è specificata in maggior dettaglio e quindi codificata durante l'*implementation*.

5.4 Descrizione grafica

Il Meta-Modello di *Tropos* è descritto tramite i seguenti diagrammi delle classi UML (vedi anche [1]):

5.4.1 Concetto di actor

Una parte del Meta-Modello di *Tropos* riguarda il concetto di "actor", mostrato nel diagramma 1 come diagramma UML delle classi. Un actor è rappresentato come una classe UML, che può avere da uno a n goals. Questo concetto è espresso attraverso una relazione tra le classi actor e goal. La stessa cosa si verifica con la classe belief.
Una dependency è una relazione quaternaria, rappresentata come una classe UML.

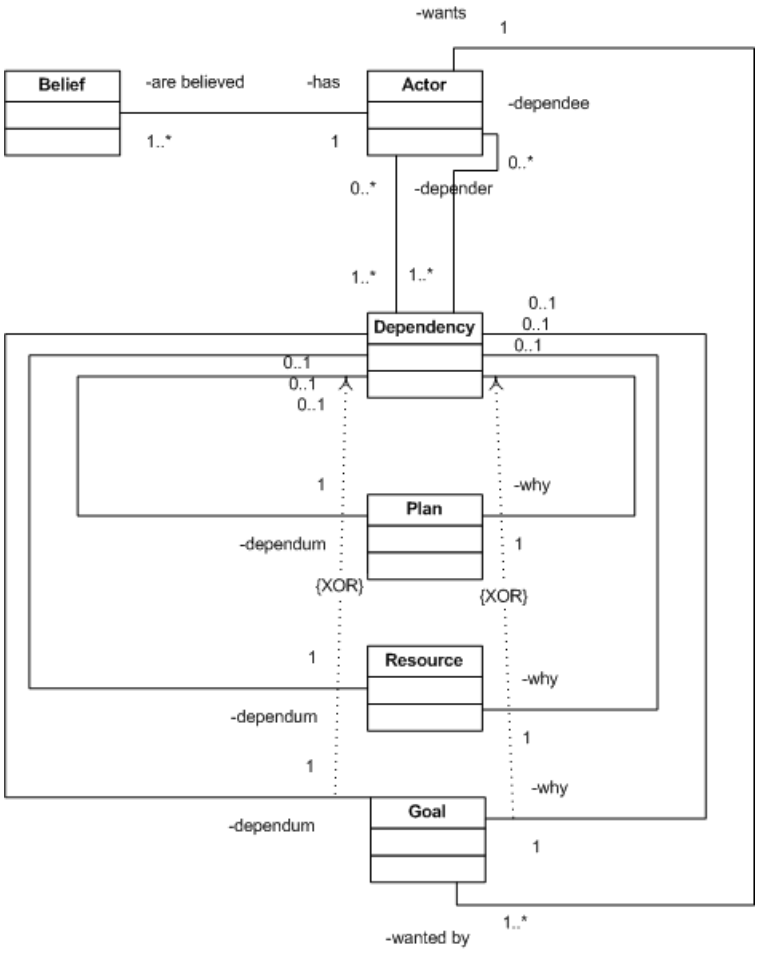


Diagramma 1. Modellazione del concetto di actor

5.4.2 Concetto di goal

Il concetto di goal è rappresentato attraverso la classe UML goal, come mostrato nel diagramma 2.
La distinzione tra hardgoal e softgoal è stata realizzata con una specializzazione in sotto-classi.
I goals possono essere analizzati dal punto di vista di un actor, attraverso una means-end analysis, una contribution o una AND/OR decomposition.
Una means-end analysis è una relazione ternaria definita tra un actor, un goal (end) e un plan, una resource o un goal(means).
Una contribution è una relazione ternaria tra un actor e due goals. Essa cerca di identificare goals che possono contribuire positivamente o negativamente al raggiungimento dell'altro goal della relazione. La contribution può essere contrassegnata con una metrica qualitativa, attraverso i simboli +, ++, -, --, secondo l'importanza del contributo dato.
Una AND/OR decomposition è anch'essa una relazione ternaria definita come una AND o una OR decomposition di un goal radice in sub-goals.

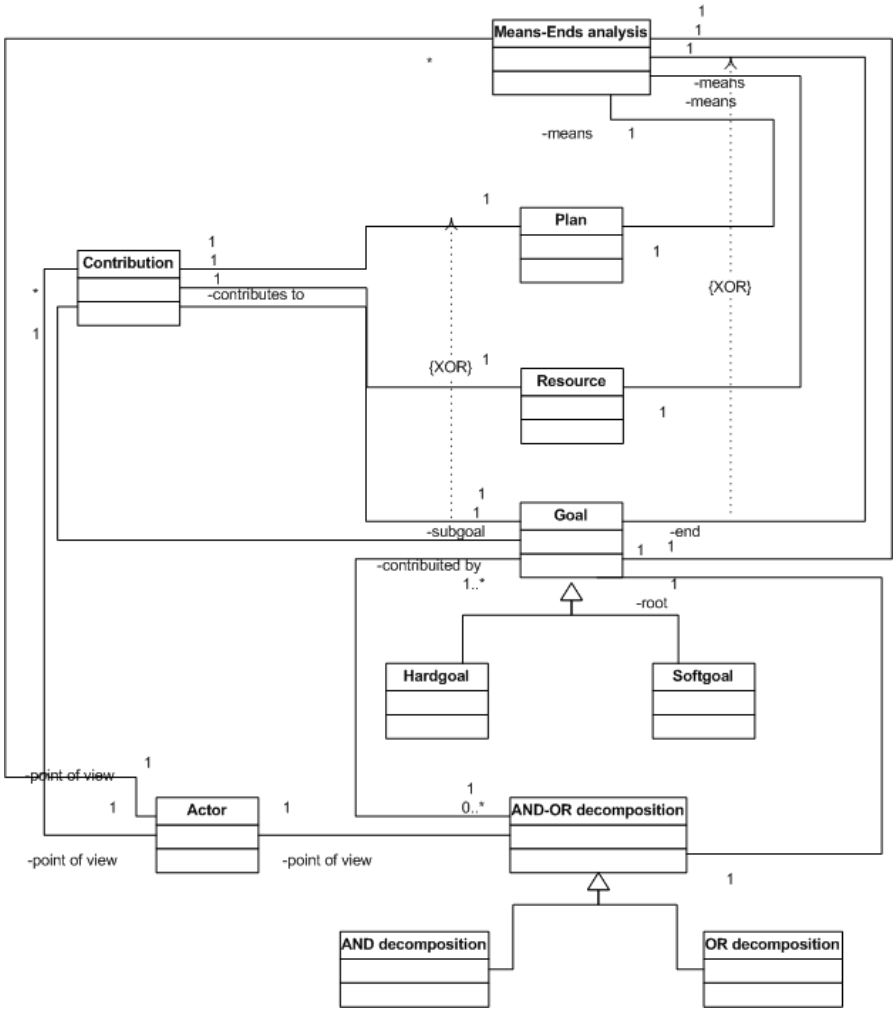


Diagramma 2. Modellazione del concetto di goal

5.4.3 Concetto di plan

Il concetto di plan, in *Tropos*, è specificato attraverso un diagramma delle classi UML, come descritto nel diagramma 3.
La means-end analysis e la AND/OR decomposition, definite precedentemente per i goals, possono essere applicate anche ai plans; in particolare, l'AND/OR decomposition permette di modellare la struttura dei plans.

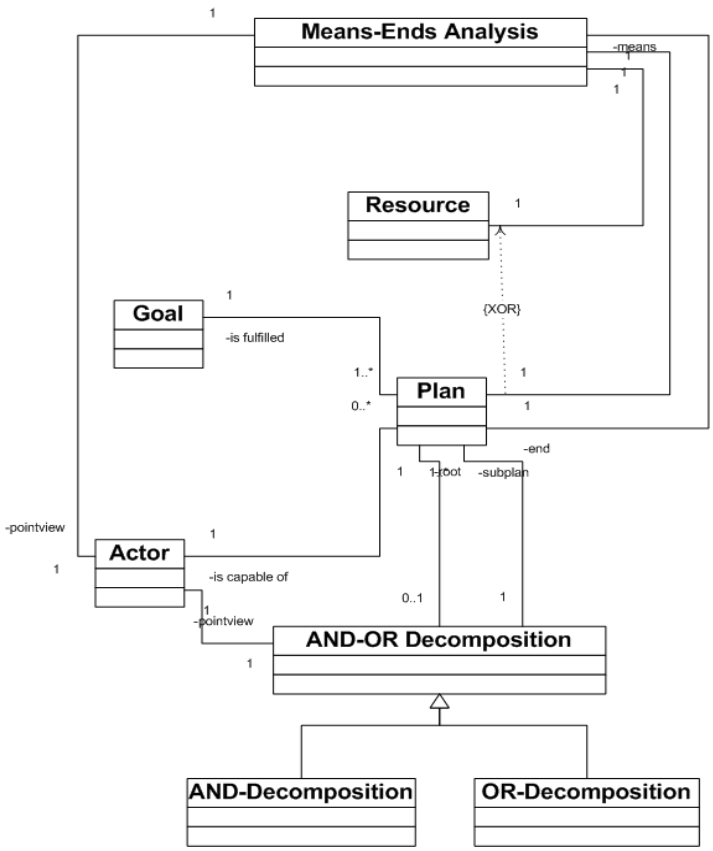


Diagramma 3. Modellazione del concetto di plan

6 Il processo di sviluppo

6.1 Il processo di sviluppo come algoritmo

Il processo di sviluppo è basato sull'analisi dei goals dei differenti actors, e verrà descritto, in questo paragrafo, attraverso un algoritmo (per altri algoritmi vedi anche [3]).
Il processo inizia identificando un certo numero di actors, ad ognuno dei quali viene assegnata una lista dei goals radice (includendovi, possibilmente, anche i softgoals), cioè gli obiettivi principali che ogni actor vuole che vengano soddisfatti (p.es. possiamo assimilare questi obiettivi alle richieste del committente, ma non solo). Ogni goal radice è analizzato dal punto di vista del corrispondente actor, e, se vengono generati sub-goals, questi possono: o essere delegati interamente ad altri actors, oppure, sarà l'actor stesso a intraprendere una serie di relazioni (dependencies) con altri per il loro soddisfacimento. Talvolta, questa parte del processo, può identificare nuovi actors, a cui delegare i goals e\o i compiti individuati dalla precedente analisi.
Il processo si considera completo, quando tutti i goals sono stati analizzati, e per ognuno di questi, sono stati individuati tutti gli actors necessari al loro raggiungimento.
Di seguito viene descritto l'algoritmo del processo di sviluppo; con la notazione *nomeLista(actor)*, verrà indicato che ogni singolo actor, avrà una lista del tipo *nomeLista*, mentre con *nomeLista* si intendono liste "globali", che non si riferiscono cioè ad un particolare actor, e di cui vi sarà una sola istanza. Si assuma che la lista degli actors (*actorList*) contenga un numero finito di actors, e che la lista dei goals di ciascun actor sia memorizzata nella corrispondente *goalList(actor)*.

Inoltre, si assuma che l'*agenda(actor)* contenga la lista dei goals che quell'actor può soddisfare autonomamente (senza l'aiuto di altri actors), insieme al plan che è stato selezionato per ogni goal. Inizialmente, l'*agenda(actor)* è vuota.
La *dependencyList* contiene una lista delle dependencies tra gli actors, mentre la *capabilityList(actor)* include coppie <goal,plan> che indicano i mezzi coi quali un actor può raggiungere un particolare goal. Infine, in *goalGraph* viene memorizzata una rappresentazione grafica dei goals generati dal processo di sviluppo. Inizialmente *goalGraph* contiene tutti i goals radice degli actors inizialmente individuati, e il cui soddisfacimento non richiede interazioni con altri actors. Durante la descrizione dell'algoritmo, le variabili sopra elencate, saranno trattate come se fossero variabili globali, che possono quindi essere direttamente utilizzate o modificate dalle procedure che andiamo a descrivere di seguito.

```
global actorList, goalList, agenda, dependencyList, capabilityList, goalGraph;
procedure rootGoalAnalysis (actorList, goalList, goalGraph)
  begin
    rootGoalList = nil;
    for actor in actorList do
      for rootGoal in goalList(actor) do
      rootGoalList = add(rootGoal, rootGoalList);
      rootGoal.actor = actor;
      end;
    end;
  end;
  concurrent for
    rootGoal in rootGoalList do
    goalAnalysis(rootGoal, actorList)
  end concurrent for ;
  if not [satisfied(rootGoalList,goalGraph)] then fail;
end procedure
```

La procedura *rootGoalAnalysis* conduce un'analisi dei goals parallela per ogni goal radice. In principio, l'analisi dei goals radice, è condotta per tutti i goals inizialmente associati con gli actors della *actorList*. In un secondo momento, sono introdotti altri goals radice, provenienti dall'individuazione dei goals che un actor non può soddisfare da solo, e che quindi delega, per il suo ottenimento, ad un altro actor (sia esso già esistente o meno).
Da notare che l'istruzione **concurrent for** effettua una chiamata parallela alla *goalAnalysis* per ogni elemento della *rootGoalList*. Inoltre, vengono effettuate tante più chiamate alla *goalAnalysis* quanto più vengono aggiunti goals radice alla *rootGoalList*.
Concurrent for termina quando vengono trattati tutti i goals radice.
Il predicato *satisfied* verifica se tutti i goals radice nel *goalGraph* siano soddisfatti (questo predicato è calcolato attraverso un algoritmo label propagation).
RootGoalAnalysis è eseguita se esiste un insieme di selezioni non-deterministiche nella corrente esecuzione delle procedure di *goalAnalysis* che stanno cercando di soddisfare tutti i goals radice.
La procedura *goalAnalysis* conduce, in parallelo, analisi dei goals per ogni sub-goal di un dato goal radice. Inizialmente, il goal radice è posto in una *pendingList*, quindi **concurrent for** seleziona parallelamente goals dalla *pendingList* e per ognuno decide, in modo non deterministico, se debba essere espanso come un goal che può essere soddisfatto dall'actor stesso, se debba essere delegato ad un altro per il suo ottenimento, o se sarà classificato come insoddisfacibile (*'denied'*).
Quando un goal è espanso, sono aggiunti vari sub-goals alla *pendingList*, e il *goalGraph* viene aggiornato per includere i nuovi goals e le loro relazioni con i goals radice.
Da notare che la selezione di un actor per la delega di un goal è un fattore non deterministisco, così come la creazione di un altro actor.

Le operazioni non deterministiche, sono state evidenziate, nell'algoritmo, sottolineando e mettendo in grassetto i caratteri corrispondenti. Queste sono i punti dove il progettista del sistema software andrà a usare la sua "creatività" per costruire il system-to-be.
Durante l'*early requirements*, questo processo analizza i goals, inizialmente individuati, degli actors dell'ambiente operativo ("stakeholders").
A un certo punto (nella *late requirements*), il sytem-to-be viene introdotto come un altro actor, al quale vengono delegati alcuni dei sub-goals che sono stati generati dalla precedente analisi.
Durante l'*architectural design*, sono introdotti i sub-actors del system-to-be, ai quali vengono delegati i goals del system-to-be precedentemente individuati. Da questo punto in poi, si passa a definire dettagliatamente l'architettura software (*detailed design*), e a realizzare la sua implementazione (*implementation*).

6.2 Il processo di sviluppo come trasformazioni

Un modo alternativo di illustrare il processo di sviluppo, è quello di utilizzare semplici trasformazioni, da applicare iterativamente, che permettono di aggiungere, a ogni ciclo, dettagli in termini di requisiti funzionali e non [2]. Ciò permette di ottenere, da un primo approssimativo problem statement redatto col cliente, un modello particolareggiato del system-to-be, dal quale si passerà alla realizzazione dell'architettura software (fasi *Architectural Design* e *Detailed Design*), degli agenti del system-to-be, e del codice vero e proprio.
Tali semplici trasformazioni, possono essere viste come generici patterns di regole di trasformazione, che trovano maggior impiego nella prima fase, la *early requirements*.
Sono divisibili secondo due criteri, il primo pone l'attenzione sulle entità del meta-modello che coinvolge (Actors, Goals e Softgoals), mentre il secondo si basa sul ruolo che l'applicazione delle trasformazioni gioca nel processo.
Seguendo il primo criterio avremo le seguenti trasformazioni:

- Actor Aggregation e Actor Generalization per gli Actors;
- Goal Decomposition, Precondition Goals, Goal Delegation e Goal Generalization per i Goals;
- Softgoal Contribution, Softgoal Decomposition, Softgoal Delegation e Softgoal Generalization per i Softgoals.

Adottando il secondo avremo:

- Top-Down (d'ora in poi TD), quando l'engineering requirements analizza un elemento concettuale di alto livello (actor, goal, softgoal), aggiungendovi dettagli in termini di relazioni (come quelle di specializzazione, di decomposizione, ecc.), che gli permettono di spezzettare il problema in vari sotto-problemi più facilmente gestibili, decidendo egli stesso il livello di dettaglio nel quale scendere;
- Bottom Up (d'ora in poi BU) permette di aggregare elementi concettuali più specifici, per esprimere quale sia il loro contributo (gerarchico, composizionale, funzionale o non funzionale) verso gli altri più generici elementi concettuali.

Descriviamo di seguito le trasformazioni, indicando, tra parentesi, nelle relative tabelle, se rispettino i criteri TD o BU.

6.2.1 Trasformazioni riguardanti i Goals

Sono trasformazioni che ci permettono di effettuare un'analisi dei goals, indicando le relazioni che possono instaurarsi tra goals e goals, o tra goals e actors; l'utilizzo di tale analisi può anche portare, se il caso lo richiede, all'introduzione di nuovi elementi concettuali che non erano stati precedentemente individuati, in tal caso i diagrammi vengono arricchiti con questi oggetti.

6.2.1.1 Goal Decomposition

Consente la trasformazione di goals in sub-goals and/or, intendendo nel primo caso che solo il raggiungimento di tutti i sub-goals permette il soddisfacimento del superiore goal (cioè del goal che ha subito la decomposizione), mentre nel secondo caso, che basta il soddisfacimento di almeno uno di questi.
Graficamente, possiamo rappresentare queste trasformazioni con la seguente tabella:

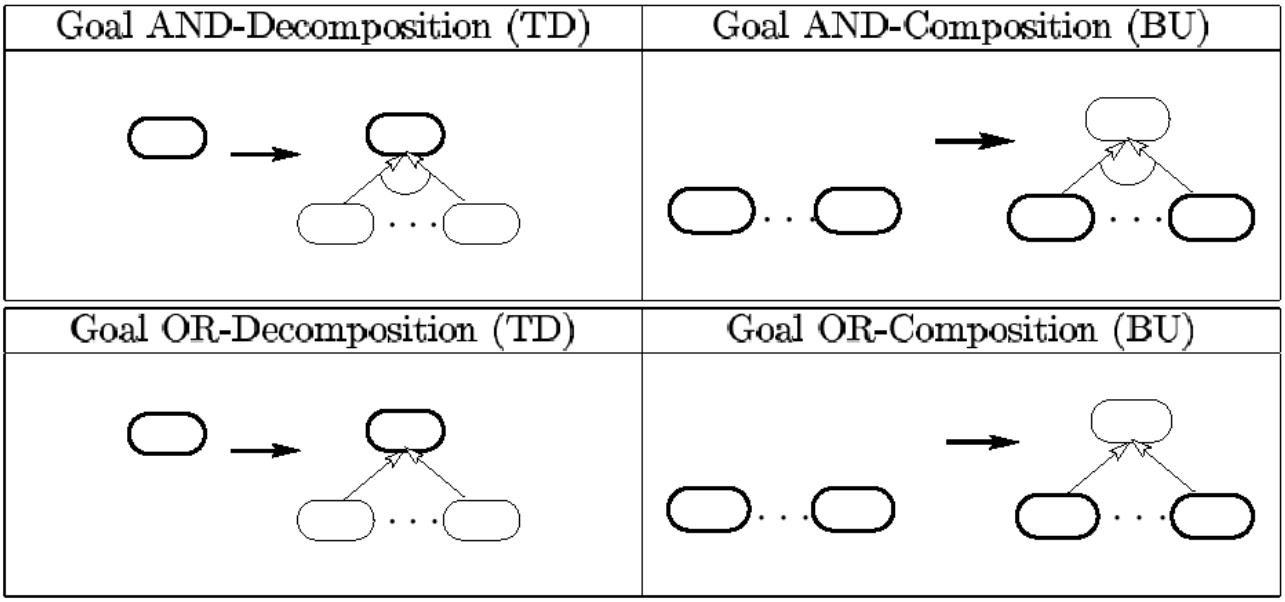


Tabella 4. Goal Transformation: The Goal Decomposition

6.2.1.2 Precondition Goals

Questa trasformazione permette di elencare un insieme di precondizioni necessarie (ma non sufficienti) al raggiungimento del goal di più alto livello, precondizioni espresse in termini di altri goals. Tipicamente, a una prima goal analysis ottenuta dall'applicazione di questo tipo di trasformazione, nella *early requirements*, ne succede un'altra, che la rende più completa e dettagliata, nella *late requirements*.
Di seguito riportiamo la tabella della Precondition Goals:

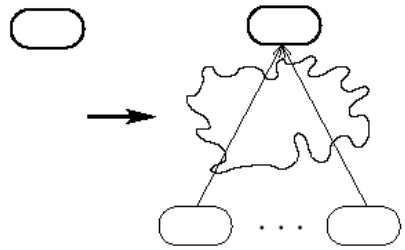
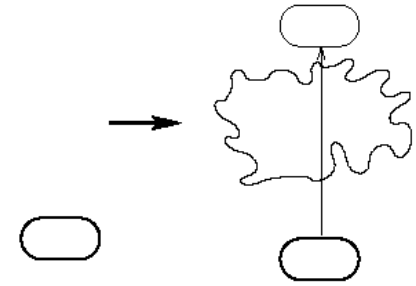
Precondition Goals (TD)	Precondition Goals (BU)
	

Tabella 5. Goal Transformation: The Precondition Goals

6.2.1.3 Goal Delegation

E' utilizzata per esprimere l'assegnamento, o il cambiamento della responsabilità del raggiungimento di un goal, da parte di un actor. In *Tropos*, come abbiamo già visto, inizialmente si definisce ogni goal o come *dependum* tra due actors, o come "desiderio" di un singolo. Nel primo caso, avremo una delega del goal, allorché il *dependee* si rivolgerà ad un terzo actor per il soddisfacimento di quel goal; avremo invece, nel secondo caso, un assegnamento, quando l'actor, per veder soddisfatto il proprio goal, instaurerà un rapporto di dipendenza con un altro actor (*dependee*).
Di seguito riportiamo la tabella della Goal Delegation:

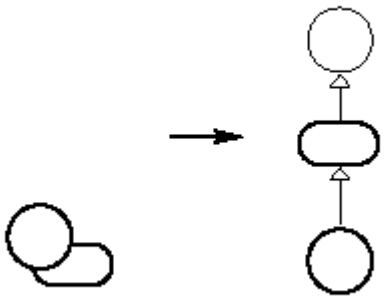
Goal Delegation


Tabella 6. Goal Transformation: The Goal Delegation

6.2.1.4 Goal Generalization

In alcuni casi può essere utile applicare una trasformazione Goal Specialization o Goal Generalization; in questi casi i goals depender e dependee, vengono, di default, ereditati sia nelle trasformazioni TD sia in quelle BU.
Di seguito riportiamo la tabella della Goal Generalization:

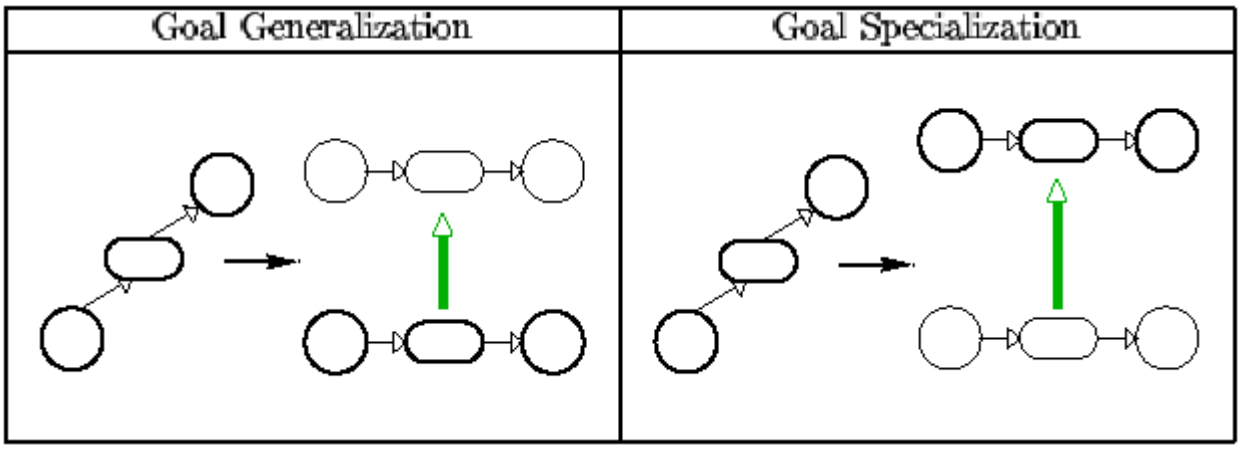


Tabella 7. Goal Transformation: The Goal Generalization

6.2.2 Trasformazioni riguardanti i Softgoals

Per l'analisi dei softgoals, sono disponibili alcune trasformazioni simili a quelle elencate per i goals. In più, però, è eseguita anche un'analisi del contributo che un singolo softgoal può dare a un goal.

6.2.2.1 Softgoal Contribution

La trasformazione Softgoal Contribution permette di specificare se un goal o softgoal contribuisce al raggiungimento di qualche altro softgoal (BU) o, se c'è qualche goal o softgoal che contribuisce (TD) positivamente o negativamente, al soddisfacimento di un softgoal.
Di seguito riportiamo la tabella della Softgoal Contribution:

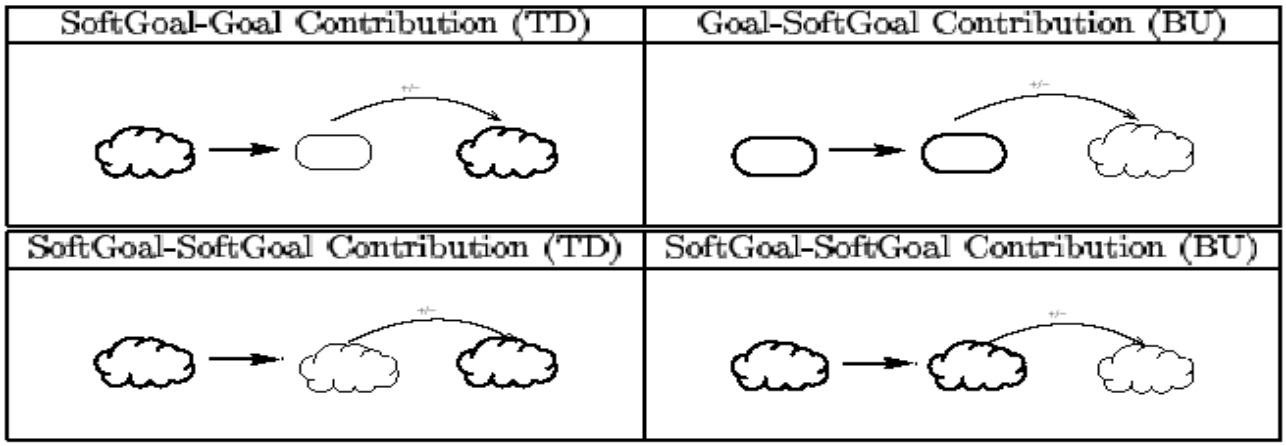


Tabella 8. Softgoal Transformation: The Softgoal Contribution

6.2.2.2 Softgoal Decomposition

Le trasformazioni di questo tipo, permettono di eseguire un'analisi dei softgoals per individuare sub-softgoals (o super-softgoals).
Mentre con la trasformazione Decomposition Goal, si ottenevano condizioni necessarie e sufficienti per il raggiungimento del goal, adesso il soddisfacimento dei goals individuati dalla

decomposizione, non può essere considerata una condizione sufficiente, quanto piuttosto un contributo (positivo) al raggiungimento del softgoal.
Di seguito riportiamo la tabella della Softgoal Decomposition:

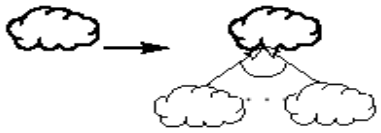
SoftGoal-AND Decomposition (TD)	SoftGoal-AND Composition (BU)
	
SoftGoal-OR Decomposition (TD)	**SoftGoal-OR Composition (BU)**
	

Tabella 9. Softgoal Transformation: The Softgoal Decomposition

6.2.2.3 Softgoal Delegation

Questa trasformazione è identica alla Goal Delegation, salvo il fatto che viene applicata ai softgoals.
Di seguito riportiamo la tabella della Softgoal Delegation:

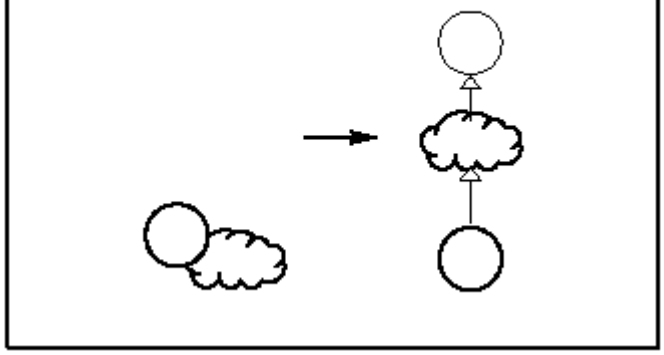
SoftGoal Delegation


Tabella 10. Softgoal Transformation: The Softgoal Delegation

6.2.2.4 Softgoal Generalization

La stessa osservazione fatta al precedente paragrafo, può essere qui ripetuta.
Di seguito riportiamo la tabella della Softgoal Generalization:

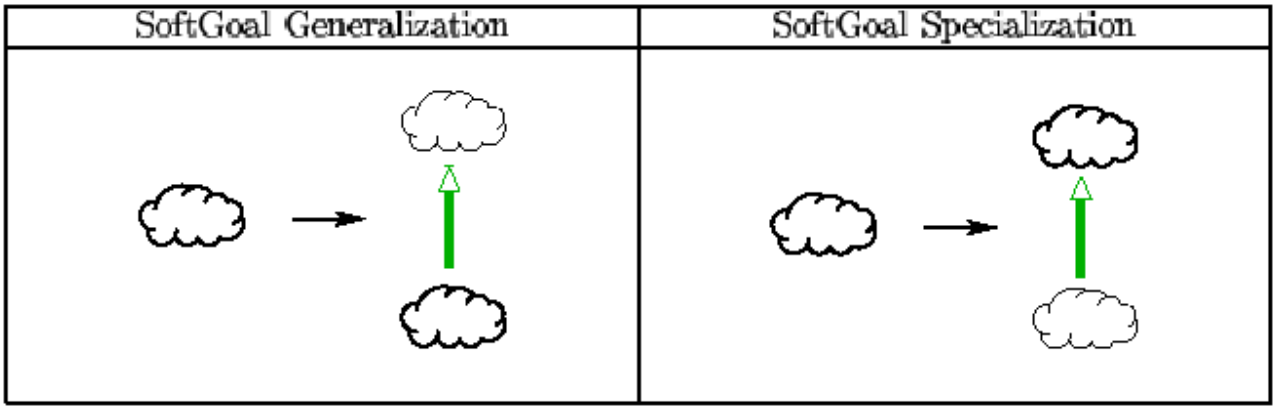


Tabella 11. Softgoal Transformation: The Softgoal Generalization

6.2.3 Trasformazioni riguardanti gli actors

Le trasformazioni riguardanti gli actors sono solo l'Actor Aggregation e l'Actor Generalization.

6.2.3.1 Actor Aggregation

Questa trasformazione consiste nel riconoscere che differenti actors fanno parte di una stessa organizzazione o di un medesimo sistema. Il raggruppare gli actors permette, in un primo momento, di assegnare o delegare goals o softgoals a tutto il gruppo, per poi, in un secondo momento, individuare quali degli actors del gruppo, dovranno provvedere al loro soddisfacimento.
Di seguito riportiamo la tabella dell'Actor Aggregation:

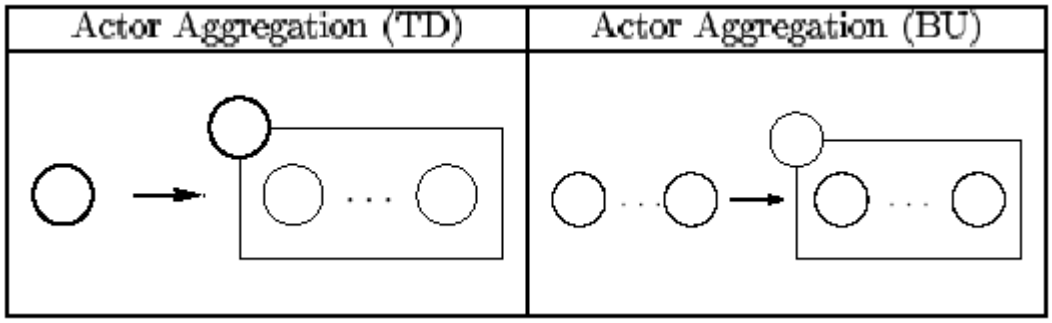


Tabella 12. Actor Transformation: The Actor Aggregation

6.2.3.2 Actor Generalization

Questa trasformazione può essere utilizzate per introdurre l'ereditarietà tra gli actors, se, p. es., un actor "padre", viene specializzato in due o più actors "figli", i goals del primo verranno ereditati dai secondi, nel senso che anche i "figli" tenderanno al soddisfacimento di quei goals, oltre ad eventuali goals specifici che il "padre" non possiede.
Di seguito riportiamo la tabella dell'Actor Generalization:

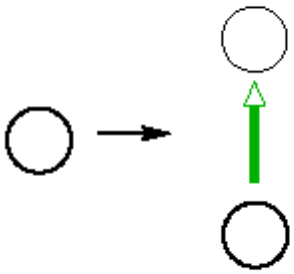
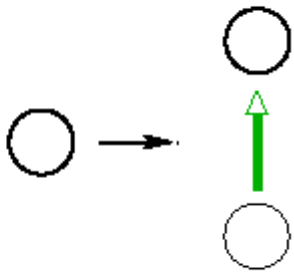
Actor Generalization	Actor Specialization
	

Tabella 13. Actor Transformation: The Actor Generalization

7 Bibliografia

[1]. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia and J. Mylopoulos, “Tropos: An Agent-Oriented Software Development Methodology”.

[2.] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia and J. Mylopoulos, “Modeling early requirements in Tropos: a transformation based approach.

[3]. F. Giunchiglia, J. Mylopoulos and A. Perini, “The Tropos Software Development Methodology: Processes, Models and Diagrams”.

[4]. M. Cossentino, L. Sabatucci and V. Seidita, “SPEM description of the PASSI process rel. 0.3.4.

[5]. Foundation for intelligent physical agents, “Method Fragment Definition”.

[6]. M. Cossentino and V. Seidita, "Composition of a New Process to Meet Agile Needs Using Method Engineering".

[7]. OMG, "Software Process Engineering Metamodel Specification".

[8]. A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos, “A knowledge level software engineering methodology for agent oriented programming”.

[9]. F. Sannicolò, A. Perini and F. Giunchiglia, “The Tropos modeling language. A user guide”.

[10]. Katia P. Sycara “Multiagent Systems”.

[11]. OMG, “OMG Unified Modeling Language Specification, version 1.3 Alpha Edition, January 1999”.

[12]. M. Kolp, P. Giorgini, and J. Mylopoulos, ‘‘An goal-based organizational perspective on multi-agents architectures,’’ in Procedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), Seattle, WA, August 2001.