Università di Roma "La Sapienza"

Tesina per il Corso di Seminari
di Ingegneria del Software
Proff. G.De Giacomo, M.Mecella,
R.Rosati

Anno 2006-2007

Il Progetto ASTRO nella Web Service Composition: analisi e confronto con il Roman Approach

Autore:
Alessandro Pagliaro

A mio Padre,
a cui devo tutto.
Trasformerò il dolore in forza
e conserverò per sempre il Suo ricordo.

Sommario

Introduzione

Il corso di Seminari di Ingegneria del Software, collocato nel biennio specialistico di Ingegneria Informatica, mira a fornire ai suoi studenti una panoramica sulle tecnologie "di frontiera", relative a determinate problematiche attuali del settore ancora ben lontane dall'essere risolte, che i ricercatori di tutto il mondo stanno esplorando, cooperando tra loro per il progresso della comunità scientifica.
Gli argomenti affrontati nel corso hanno spaziato dalla teoria delle queries congiuntive all'integrazione di dati, dalla composizione di servizi al territorio largamente inesplorato del semantic web.

All'interno di questa gamma di aree di ricerca ho scelto di concentrarmi sulla Service Composition, ovvero la disciplina che cerca di realizzare complessi e-Services (cioè programmi interattivi che forniscono, solitamente tramite il web, un qualche servizio all'utente) a partire da e-Services già esistenti; come si vedrà il problema non è affatto banale, e gli approcci seguiti dai vari gruppi di ricerca nel mondo sono molto vari, e si concentrano su aspetti di natura molto differente.

Questa relazione si occuperà in primo luogo di descrivere i risultati ottenuti dal gruppo di ricerca composto dall'Università di Trento e l'ITC-IRST, il Centro di Ricerca Scientifica e Tecnologica della Fondazione Bruno Kessler; più in particolare, alcuni dei nomi più importanti nel progetto sono M. Pistore, P. Traverso e P. Bertoli.
L'approccio di Service Composition delineato dal suddetto gruppo è conosciuto come il Progetto ASTRO; si parlerà sia della logica teorica dietro l'approccio, sia dei risultati concreti ottenuti dal gruppo di ricerca.
La relazione si prefigge inoltre un secondo scopo: quello di descrivere brevemente l'approccio alla WS-Composition del cosiddetto Roman Group, team di ricerca formato dai professori universitari che hanno tenuto il corso di Seminari di Ing. del Software (G.De Giacomo, M.Mecella, D.Berardi,. D. Calvanese), e di sottolineare analogie e differenze con l'ASTRO approach in modo da presentare al lettore un efficace confronto comparativo riguardo questa tanto interessante quanto ancora aperta area di ricerca.

Parte I: Il Problema della Web Service Composition

Sezione 1.1 - WS Composition Basics

L'argomento centrale di questa relazione è illustrare possibili metodologie per affrontare e risolvere il complesso problema della Service Composition.
Questa prima parte introduttiva delinea brevemente gli aspetti principali del problema in esame.

Un *Composite Service* è un Web Service che offre servizi da una sua interfaccia come qualsiasi altro Web Service (nel seguito si utilizzerà anche il termine "e-Service" per riferirsi ai Web Services), sebbene dal punto di vista implementativo i servizi offerti siano il risultato di un'opportuna interazione con altri Web Services, indipendenti tra loro e non pensati a priori per cooperare in un e-Service comune. Il *problema della Service Composition* è il termine generico per indicare una metodologia che ha per fine l'implementazione di Composite Services.

Per realizzare un Composite Service è indispendabile disporre di strumenti adeguati, esaminando il problema ad un alto livello di astrazione ed automatizzando il più possibile operazioni di basso livello, in modo che il progettista possa concentrarsi sulla Business Logic dell'applicazione.
Nella Web Service Composition non vogliamo realizzare un'integrazione fisica di moduli software, ma piuttosto, dati dei requisiti opportunamente espressi, trovare un piano d'esecuzione in cui siano indicati quali e-Services invocare, in che ordine farlo e come gestire condizioni di errore e imprevisti.
In linea di massima gli strumenti necessari per risolvere il problema sono:

- un linguaggio per la *rappresentazione comportamentale* di Web Services, in modo da modellare efficacemente il loro flow d'esecuzione e le funzionalità che offrono; in questa ottica i Web Services possono essere visti come *programmi interattivi e web-based che esportano il loro comportamento in termini di una descrizione astratta e formale*, utile ai nostri scopi; si noti che possiamo usare tali linguaggi comportamentali anche per rappresentare il Composite Service (anche detto "Target Service").

- una *logica di composizione*, ovvero un procedimento generale che partendo dai requisiti (Business Requirements) e dagli e-Services di partenza (Component Services) realizzi il Composite Service finale in una qualche forma eseguibile; questo punto rappresenta il cuore di un approccio per la WS Composition, una sorta di algoritmo di base.

- un *ambiente di sviluppo*, preferibilmente ricco di componenti GUI, che aiuti il progettista a creare il servizio ad alto livello, automatizzando la metodologia definita dalla logica di composizione e operazioni low-level come la creazione di files XML.

- un *composition engine* per eseguire e monitorare le istanze della composizione trovate.

Definiamo *conversazione* un'interazione con un e-Services consistente nell'esecuzione sequenziale di più operazioni, in un particolare ordine.

Una *coreografia* è un piano per la coordinazione di più conversazioni, volta ad un preciso scopo d'insieme; i Service Component che partecipano alla coreografia non hanno bisogno di sapere quale sarà l'effetto finale, devono solo assicurarsi di fare la loro parte.
Per *sintesi* di un Composite Service si intende la costruzione delle specifiche necessarie all'esecuzione del servizio a partire da requirements ben definiti; tali specifiche sono conosciute come *Composition Schema*.
L'*orchestrazione* è, infine, la gestione runtime dell'esecuzione del Composite Service (scheduling di invocazioni, gestione di errori, monitoring).

Sezione 1.2 - Web Service Representation

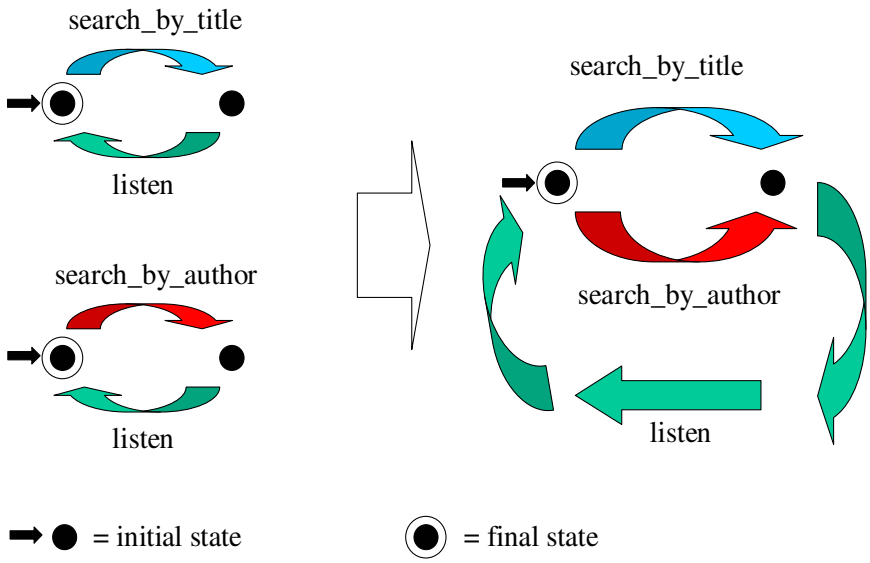
Come già accennato in precedenza, la composizione di e-Services è interessata al comportamento di questi ultimi, ovvero al *set di azioni che esportano attraverso le loro interfacce, più i possibili constraints aggiuntivi nelle conversazioni legali*.
Un metodo efficace per rappresentare Web Service behaviors, e che sarà largamente utilizzato nei capitoli successivi, consiste nell'usare Finite State Machines (o in breve FSMs). Nel seguito viene fornita la definizione di Transition System, una FSM "tailored" per rappresentare e-Services, e viene mostrato come si possa utilizzare per i nostri scopi.

Definizione: Transistion System
Un Transition System TS è una tupla $< S, S^0, A, \delta, F >$, dove:

- S è l'insieme degli stati;
- S^0, sottoinsieme di S, è l'insieme di stati iniziali
- A è l'insieme di azioni;
- δ è la relazione di transizione da S x A in S;
- F, sottoinsieme di S, è l'insieme di stati finali.

L'idea è che gli stati del TS siano "stati stabili" nell'esecuzione, del Web Service, ed azioni che arrivano dall'esterno, come la ricezione di un messaggio, o azioni che sono iniziate dal Web Service, come invio di messaggi, o ancora passi di computazione interni al servizio, siano tutti eventi scatenanti cambiamenti di transizione tra stati; si noti che nel caso di transizioni derivate da computazioni interne si parla anche di τ*-transitions*.
In genere si assume che all'avvio un e-Service si trovi in uno stato iniziale, e l'esecuzione possa terminare legalmente in qualunque stato etichettato come finale.

Nel seguito si fornisce un semplice esempio di due Component Services e n Composite Service che può essere costruito mediante una composizione dei due servizi base.
Come è ovvio, le azioni (in questo caso tutte avviate dall'esterno) *search_by_author* e *search_by_title* servono a localizzare un file musicale, mentre l'azione *listen* serve ad ascoltare tale brano.

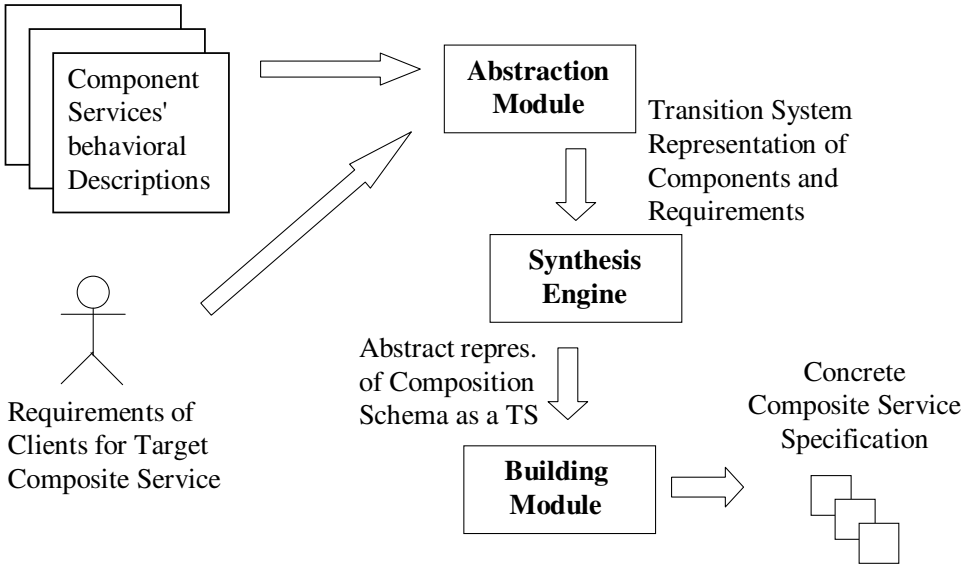


Sezione 1.3 - Our Focus

Lo scopo di questa relazione è duplice: innanziuttto si vuole porre le basi del problema della Web Service Composition per illustrare la metodologia d'approccio sviluppata all'interno del Progetto ASTRO, uno sforzo congiunto di ricerca e sviluppo dell'Università di Trento e l'ITC-IRST; oltre a ciò si vogliono delineare i punti essenziali dell'approccio nato dal Roman Group dell'Università La Sapienza di Roma in modo da giungere ad un confronto comparativo tra le due metodologie.

Entrambi gli approcci mirano alla realizzazione pratica del workflow descritto dalla seguente figura:

The General WS Composition Workflow



Nel prossimo capitolo verrà esaminato in dettaglio l'approccio del Progetto Astro alla WS-Composition, fornendo una overview generale dei suoi componenti e aree di interesse, ed approfondendo in dettaglio l'argomento per quanto riguarda la logica dietro il processo di composizione; il capitolo 3 fornirà un breve riepilogo sulle caratteristiche principali del Roman Approach, che è stato uno degli argomenti trattati durante le lezioni di Seminari di Ingegneria del Software nel cui contesto si pone la presente relazione; il capitolo 4 cercherà di evidenziare i parallelismi e le differenze tra i due approcci, Astro e Roman model, per un istruttivo confronto comparativo; il capitolo conclusivo, il 5, si occuperà invece dell'esame del tool sviluppato e reso disponibile dal team Astro, fornendo indicazioni per l'installazione, l'esecuzione e l'uso delle varie componenti coinvolte per soddisfare i nostri scopi di Web Service Composition.

Parte II: L'Approccio ASTRO

Sezione 2.1 - ASTRO Project Overview

Il Progetto Astro è un'iniziativa di ricerca congiunta riguardo l'integrazione di Web Services, sia intra- che inter-organizzazione, promossa dall'Università di Trento e l'ITC-IRST, il Centro di Ricerca Scientifica e Tecnologica della Fondazione Bruno Kessler.
Il suo scopo principale è il favorire l'adozione worldwide di Web Services compositi prestando attenzione a metriche fondamentali quali efficacia, flessibilità, facilità d'uso, basso costo ed efficienza temporale: ciò si traduce per forza di cose in una serie di obiettivi più concreti, che sono elencati qui di seguito.
Astro vuole fornire:

- un framework generale per la composizione automatica di servizi, che sarà discusso estensivamente in questo capitolo;
- dei tools concreti di realizzazione del framework, utilizzanti una larga serie di tecnologie che si sono affermate come il futuro della Web Service composition;
- supporto software per l'intero ciclo di vita delle applicazioni, dalle prime fasi di design fino al monitoraggio e verifica a runtime;
- evitare di delegare allo sviluppatore dei compiti noiosi, complessi ed error-prone, in modo da permettergli di concentrarsi in modo trasparente e user-centered sulla logica dell'applicazione ad un alto livello di astrazione.

Le aree toccate dalla ricerca del Progetto Astro si possono classificare in una tassonomia di 5 grandi macro-domini:

I. *Business Requirements* : questo settore mira allo sviluppo di un framework per rappresentare efficacemente la definizione di strategie, obiettivi e business requirements aziendali, con particolare riguardo anche alle interazioni tra differenti business processes; quest'area sarà oggetto dei paragrafi 2.2 e 2.3 del presente capitolo.
II. *Service Synthesis* : il settore della sintesi offre un modello per ottenere dei servizi compositi in maniera generale ed efficiente, nonché supportata da una teoria di fondo che garantisce la correttezza e l'affidabilità dei risultati; questo ramo ha una controparte pratica incarnata dai tools eseguibili per la composizione di e-services; della logica che costituisce il cuore dell'approccio Astro si parlerà in dettaglio nel paragrafo 2.4, mentre il tools verrà esplorato nel capitolo 5.
III. *Service Verification* : il tool offre anche strumenti di supporto per controllare se i requirments definiti sono violati dal servizio risultante ottenuto; la trattazione dettagliata di questo argomento esula dal focus della relazione, e ne verranno forniti alcuni cenni nel paragrafo 2.5.
IV. *Service Monitoring* : il corrispettivo a runtime della service verification, anche di questo si parlerà nel paragrafo 2.5.
V. *Semantics* : parte degli sforzi di ricerca sono volti all'adozione di supporto per integrare semantic web services, rendendo il tools interoperabile con OWL-S e WSMO; anche per questa parte l'argomento esula dai nostri scopi, e se ne accennerà nel paragrafo 2.5.

Le seguenti sezioni affrontano in dettaglio la metodologia di approccio per la WS-Composition, partendo dalla modellazione dei requirements per arrivare al web service finale, illustrando passo passo la logica dietro la composizione senza però mai perdere il contatto con il riscontro pratico.

Sezione 2.2 - Notions on BPEL4WS and EAGLE languages

Prima di iniziare la trattazione dei business requirements in Astro, è necessario fornire qualche nozione basilare su due importanti strumenti di ausilio alla definizione del problema, entrambi standard molto validi per la definizione di business processes nell'ambito dei Web Services: BPEL e EAGLE. Entrambi gli strumenti, soprattutto BPEL, sono estensivamente utilizzati all'interno del framework Astro, e nel seguito ne viene fornita una panoramica necessaria alla comprensione della relazione. Per maggiori dettagli si consulti **[BPEL_Spec]** e **[DPT02]**.

BPEL

BPEL Overview

L'acronimo BPEL sta per Business Process Execution Language, ed è un linguaggio appositamente creato per la definizione ed esecuzione di processi i cui passi di esecuzione possono rappresentare invocazioni a Web Services. Il linguaggio è basato su XML (ovvero un file BPEL è a tutti gli effetti un xml file con dei costrutti particolari e processabile da tools appositi) ed è il fulcro centrale su cui orbita la realizzazione dell'intero Astro toolset: sia parte dei composition requirements, sia l'output eseguibile finale sono BPEL files.

Abstract and Concrete Processes

Una distinzione chiave tra i files BPEL è quella tra Abstract e Concrete files. Entrambi descrivono un (composite) e-service via xml, ma mentre un Abstract process definisce solo il comportamento visibile "dall'esterno" dello scambio di messaggi tra web services (ricordiamo che, nell'ambiente asincrono della WS Composition, ogni interazione tra due distinti e-services avviene soltanto tramite message exchange), un Concrete process è un file a tutti gli effetti eseguibile, dettaglia anche le internal evolutions dei servizi (le τ-transitions) e può concretizzarsi in un processo deployable su un BPEL engine e monitorabile.
Detto ciò, non sorprenderà il sapere che i processi abstract sono usati, nell'approccio Astro, per la descrizione dei Component Services, mentre il fine ultimo è la costruzione del concrete BPEL process eseguibile che incarna il target composite service.

A few Details on BPEL constructs

Concludiamo questa breve parentesi su BPEL, necessaria per comprendere appieno il meccanismo di composizione, con dei sintetici cenni ai costrutti BPEL più importanti ed usati dal tool Astro:

- un abstract process definisce un set di message exchanges tra web services, senza però definire la internal business logic;
- un concrete process definisce la business logic di un servizio servendosi di *activities* constituenti, *partners* coinvolti nel servizio, *message exchange* necessario e procedure di *exception handling*.

Per quanto riguarda le BPEL activities, esse possono essere primitive o strutturate.

Le attività primitive degne di nota includono:
- *invoke* : per invocare un Web Service;

- *receive* e *reply* : per ricevere messaggi da una sorgente esterna o inviarli verso l'esterno;
- *wait* ed *empty* : per rimanere inattivi, rispettivamente per un certo periodo di tempo e indefinitamente;
- *assign* : per assegnare valori alle variabili interne che costituiscono lo stato dell'e-service;
- *throw* : per lanciare eccezioni.

Attività strutturate includono:
- *sequence* : per eseguire una catena di azioni sequenziali;
- *switch* : simile al noto costrutto informatico, effettua una singola decisione basandosi su una variabile;
- *pick* : per "ascoltare" i cambiamenti su un dato set di eventi; non appena accade un certo evento, viene scelta ed eseguita una certa azione (legato al non-determinismo);
- *while* : per il tradizionale ciclo di iterazioni;
- *flow* : per gestire esecuzioni parallele (Astro lo usa in modo limitato nella versione corrente 3.4).

EAGLE

Come si vedrà tra breve, il problema della WS-Composition affrontato dal team Astro è essenzialmente un problema di planning, e per di più sotto ipotesi realistiche che rendono la questione niente affatto banale: non-determinismo nell'evoluzione degli stati ed obiettivi (requirements, business goals) che non si riducono a problemi di reachability ma coinvolgono connotazioni come "cercare di fare tutto il possibile per".
Per tali ragioni, nella modellazione di requirements, il team Astro ha preferito non utilizzare logiche temporali quali CTL che offrono quantificatori temporali potenti ma comunque insufficienti a modellare concetti come "fare del proprio meglio per" o "in caso non ci si riesca, allora è preferibile fare quest'altra cosa". Per una panoramica più dettagliata sulle motivazioni che hanno portato a tali scelte e i risultati pratici ottenuti, si consulti **[DPT02]**.

L'adozione di EAGLE come linguaggio per esprimere "Extended Goals" (di cui parlerà meglio nel prossimo paragrafo) è la conseguenza a questa linea di pensiero; fornire una completa trattazione del linguaggio è molto al di fuori dei nostri scopi, ed anche delle notre necessità, in quanto anche solo pochi cenni possono essere sufficienti per comprendere le sezioni successive.

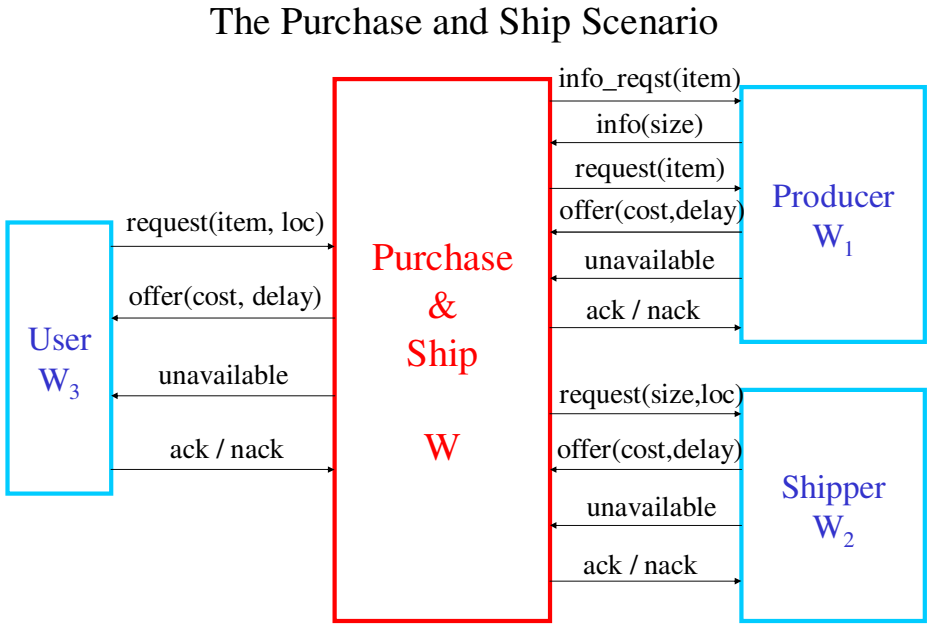
Formule EAGLE saranno utilizzate per esprimere business goals del target composite service; esse consistono in blocchi contenenti formule proposizionali che intuitivamente definiscono dei particolari stati che il sistema può raggiungere; ogni blocco è associato ad un particolare operatore che definisce la funzione stessa di quel blocco all'interno del sistema.
Ad esempio, gli operatori comprendono *TryReach* e *FailDoReach*, due costrutti che specificano rispettivamente il "tenta di raggiungere questo stato del sistema" e "in caso non fosse possibile raggiungere tale stato, portati assolutamente in quest'altro stato". Altri operatori possono esprimere sequenzialità (*A then B*), iterazioni (*repeat A*), tentativi o garanzia di mantenere vero un certo stato (*TryMaint A* oppure *DoMaint A*). Ancora, per una trattazione approfondita si veda **[DPT02]**.
Nel nostro contesto è sufficiente capire l'uso dei costrutti EAGLE TryReach e DoReach, e il motivo per cui sono importanti e più utili di CTL.

Sezione 2.3 - Component Services and Business Requirements in ASTRO

Armati delle conoscenze acquisite nel precedente paragrafo, possiamo introdurre una prima, generale versione del nostro problema: dato un set di Component Services $W_1,...,W_n$, espressi come Abstract BPEL Processes, e data una specifica di Composition Requirements come EAGLE formula, vogliamo generare automaticamente un nuovo servizio W, il nostro target composite service, che utilizza i component services esistenti e soddisfa i nostri composition requirements.

Assumiamo inoltre di essere in un dominio *asincrono*, con dei component services che offrono *partial observability* (ovvero non espongono le loro operazioni interne ma solamente le interazioni con l'esterno) e in presenza di business requirements come *extended goals*, cioè specifiche di stati "desiderabili", che non possono evitare di prendere in considerazione esecuzioni impreviste e fallimenti (rifiuto di accettare un costo, o un delivery time, o indisponibilità di merce, ad esempio) e quindi prendono in esame condizioni di intensità differente (come try vs reach) e preferenze tra differenti possibili alternative.

Case Study: Purchase & Ship
Nel seguito faremo riferimento ad un esempio pratico per favorire la chiarezza nell'esposizione dell'argomento: consideriamo un servizio composito di acquisto e consegna di merce (Purchase&Ship, anche abbreviato come P&S), illustrato graficamente in figura:



L'esempio è piuttosto autoesplicativo e non dovrebbe essere difficile da comprendere: abbiamo uno scenario in cui vogliamo comporre due differenti e-services, il Producer e lo Shipper, in modo da riuscire ad acquistare dei beni a distanza ed ottenere la consegna a domicilio; per questo sarà neessario disporre di un terzo WS che interagisce con il Composite Service P&S W, che pertanto dovrà trattare in tutto con tre component services, W_1, W_2 e W_3. Possiamo notare come le operazioni richieste da User non sono direttamente offerte da Producer e Shipper, ma mediate

da P&S, che interagisce con i due servizi base interrogandoli su dimensioni, prezzi e ritardi di produzione/consegna necessari, ordinando beni e comunicando esiti delle operazioni.
Evidenziamo inoltre come molte cose possano non andare a buon fine: un prodotto potrebbe essere non disponibile, o non consegnabile, o ancora il cliente potrebbe rifiutare l'offerta finale; tuttavia, non vogliamo nemmeno che il Composite Service permetta operazioni quali acquisti presso il Producer mentre User ha rifiutato l'offerta, o Shipper ha risposto negativamente: siamo in sostanza in presenza di extended goals, in cui vogliamo che l'operazione vada a buon fine, salvo imprevisti, e che in caso di fallimento non vengano eseguite operazioni dannose.

Component Service Representation as STSs
Abbiamo già visto nel capitolo I come la rappresentazione via FSM sia un ottimo modo per descrivere e-services dal punto di vista comportamentale.
Nell'approccio Astro i component services che costituiscono parte dell'input al composition problem vengono forniti come Abstract BPEL Processes, cioè possiedono uno stato (le variabili interne del processo), un insieme di *partners*, cioè di BPEL processes cooperanti, una descrizione di interazioni con l'esterno e la possibilità di evolvere internamente, utilizzando tutti i costrutti BPEL esaminati nel paragrafo 2.2.
La metodologia di sintesi prevede inizialmente la *traduzione di tutti i Component Services espressi come Abstract BPEL Processes in STS*. Un modulo software apposito, BPEL2STS, si occupa di tradurre i files .bpel in files .smv.
Gli STS definiti in Astro distinguono possibili stati, e cambiamenti tra stati avvengono attraverso azioni, le quali possono essere classificate in azioni di input (ricezione di messaggi), azioni di output (invio di messaggi) e τ-transitions, ovvero azioni di evoluzione interna e non visibile alle entità esterne.
Riportiamo di seguito una ridefinizione di STS in chiave Astro, molto simile a quella riportata nel capitolo I ma tailored secondo i concetti e le definizioni proprie dell'approccio in esame.

Definizione: Astro State-Transition Sytem (STS)
Un Transition System Σ è una tupla $< S, S^0, I, O, R, L >$, dove:

- S è l'insieme finito degli stati;
- S^0, sottoinsieme di S, è l'insieme di stati iniziali
- I è l'insieme finito di *input actions* (cioè message receive);
- O è l'insieme finito di *output actions* (cioè message send);
- R è la relazione di transizione da $S \times (I \cup O \cup \{\tau\}) \rightarrow S$;
- L è funzione di labeling, e associa ad ugni stato un set di proprietà soddisfatte dallo stato. Formalmente, detto *Prop* l'insieme delle proprietà, $L: S \rightarrow 2^{Prop}$

Vengono affettuate alcune assunzioni sulla modellazione in STS di Component Services: l'assenza di loops infiniti su τ-actions e l'impossibilità che uno stato abbia origine sia da input che da output transitions.
Accenniamo anche al fatto che il modulo di traduzione, BPEL2STS non supporta tutti i costrutti BPEL, ad esempio nell'ultima versione 3.4 i costrutti "Scope" e "Fault" non sono supportati; tuttavia il range di operatori attualmente disponibili permette un certo livello di complessità.
Ricordiamo infine come lo stato di un STS dipenda dalle sue variabili interne, così come le transizioni definite da R dipendono da queste stesse variabili; perché il file .smv che incarna l'STS dei Component Services sia trattabile, vengono definiti ranges finiti per le variabili in gioco.

Composition Requirements as EAGLE Formulas for Extended Goals

Alla luce di quanto detto finora riguardo l'espressione di business requirements e la necessità di utilizzare formule EAGLE, questa parte dovrebbe essere di facile comprensione.
Nel presente esempio, in ogni esecuzione del servizio il nostro scopo è quello di raggiungere il goal generale "try to sell items at home"; come già accennato, non è *garantito* che l'operazione abbia successo, a causa di imprevisti "legali" che potrebbero accadere, come l'indisponibilità in inventario dell'oggetto richiesto, una destinazione che lo Shipper non è disposto a trattare, o un rifiuto del cliente su un' offerta finale troppo costosa. In sostanza vogliamo che il sistema consideri il completamento dell'operazione come una circostanza altamente desiderabile, e faccia tutto il possibile per soddisfarla.
Il secondo punto fondamentale è che non vogliamo completare operazioni "singole" senza che l'intera transazione sia andata a buon fine (non vogliamo acquistare un item dal Producer quando lo Shipper non è disposto a consegnarlo!), quindi in sostanza desideriamo una politica error-handling di "upon failure, *do* never a single commit"; notiamo come la forza della direttiva di requirements è questa volta "do", e non "try", vogliamo cioè garanzie sullo stato risultante.
Siamo quindi pronti a definire i nostri requirements, prima in linguaggio naturale, quindi ad un più basso livello di astrazione, come EAGLE formula:

1. *Try* to "sell items at home";
2. Upon failure {
   do "never a single commit";
   }

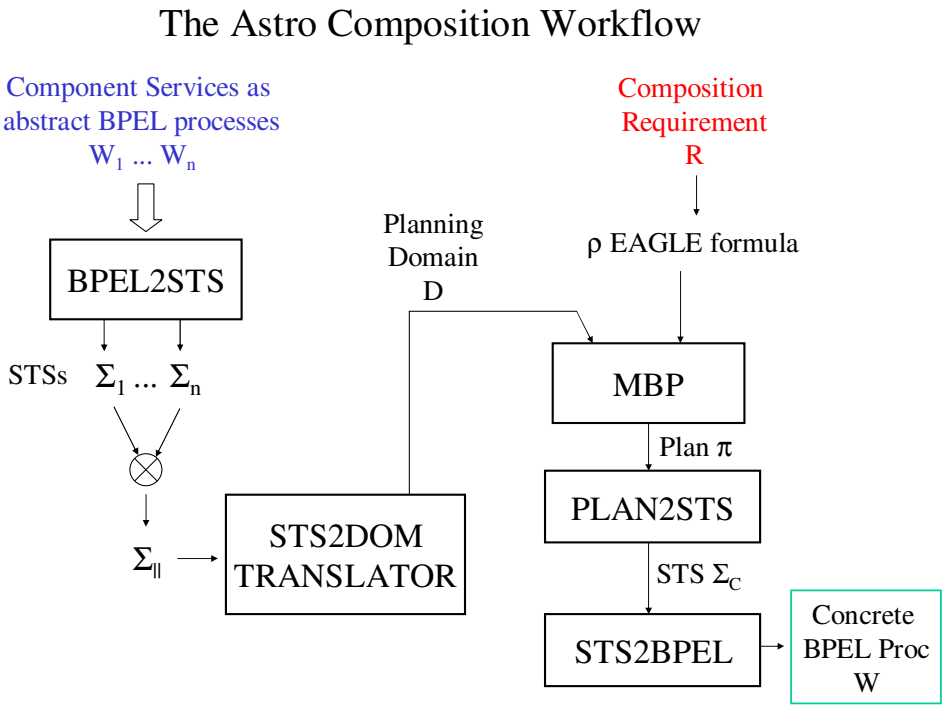
Riportiamo quindi la formalizzazione in EAGLE, legata alla rappresentazione in .smv dei Component Service STSs:

TryReach
user.pc = success && producer.pc = success && shipper.pc = success
&&
user.offer_delay = add_delay(producer.offer_delay + shipper.offer_delay)
&&
user.offer_cost = add_ cost (producer.offer_ cost + shipper.offer_ cost)
Fail DoReach
user.pc = failure && producer.pc = failure && shipper.pc = failure

Le variabili "pc" usate si riferiscono al "program counter" dell'STS, che ne individua lo stato e gioca un'importante parte nelle transizioni; comunque, l'esame in dettaglio dell'encoding in .smv è fuori dal nostro scope.

Sezione 2.4 - The ASTRO Composition Problem

Questo paragrafo è il cuore della sintesi per WS-Composition realizzata in ambito Astro, e mostrerà come sia possibile giungere dai requisiti appena definiti al Concrete BPEL Process eseguibile che implementa il Composite Service desiderato W.
Evidenziamo come il nostro focus ora giace nella logica di fondo dell'approccio, e non nel mapping implementativo, che verrà discusso e raffrontato a questa parte nel capitolo V; pertanto, se ad esmpio ci riferiremo ad un requirements goal ρ, per ora non accenneremo a come tale goal sia definito nel tool.
Di seguito è riportata un'immagine che può essere considerata come la versione in Astro del workflow-obiettivo mostrato nel paragrafo 1.3, e che sarà il nostro riferimento lungo questa sezione.



Procediamo quindi con l'analisi del workflow che costituisce la backbone dell'approccio Astro.

Composition Problem Inputs
La composizione parte da due input: uno è il set di Abstract BPEL Processes che descrivono il comportamento "visibile dall'esterno" dei Component Services su cui dobbiamo costruire, e ci riferiremo a tali processi come $W_1...W_n$; il secondo input è definito dalla formulazione ad alto livello dei requirements R della composizione, come definiti nel precedente paragrafo.

Initial Input Processing
Gli input considerati subiscono un trattamento iniziale: i processi BPEL sono trasformati in rappresentazioni STS tramite un apposito modulo software BPEL2STS: otteniamo così gli STS $\Sigma_1...\Sigma_n$ che descrivono il comportamento dei Component Services.
I composition requirements invece sono espressi tramite una EAGLE formula ρ, nel modo che abbiamo visto precedentemente.

Further Processing: the Parallel Product
Il prossimo passo è manipolare le versioni STS $\Sigma_1...\Sigma_n$ per ottenere un nuovo STS, chiamato $\Sigma_{||}$, che è formalmente definito come il *Prodotto Parallelo* di $\Sigma_1...\Sigma_n$. Intuitivamente possiamo pensare al Prodotto Parallelo come ad un STS che combina tutte le possibili evoluzioni dei Web Services componenti. Di seguito forniamo la definizione formale di Parallel Product tra due STSs:

Definizione: Parallel Product tra due STSs Σ_1 e Σ_2
Siano $\Sigma_1 = < S_1, S_1^0, I_1, O_1, R_1, L_1 >$ e $\Sigma_2 = < S_2, S_2^0, I_2, O_2, R_2, L_2 >$ due STSs tali che $(I_1 \cup O_1) \cap (I_2 \cup O_2)$ = insieme vuoto.
Il Prodotto Parallelo $\Sigma_1 || \Sigma_2$ tra Σ_1 e Σ_2 è definito come:

$$\Sigma_1 \| \Sigma_2 = < S_1 \times S_2 , S_1^0 \times S_2^0 , I_1 \cup I_2 , O_1 \cup O_2 , R_1 \| R_2 , L_1 \| L_2 >$$

dove:

- $< (s_1, s_2), a, (s_1', s_2) >$ appartiene a $(R_1 \| R_2)$ se $< s_1, a, s_1' >$ appartiene a R_1;
- $< (s_1, s_2), a, (s_1, s_2') >$ appartiene a $(R_1 \| R_2)$ se $< s_2, a, s_2' >$ appartiene a R_2;
- e inoltre $(L_1 \| L_2) (s_1, s_2) = L_1 (s_1) \cup L_2 (s_2)$.

Arriviamo quindi alla definizione di $\Sigma_{\|}$, che è nient'altro che il Parallel Product di $\Sigma_1 ... \Sigma_n$, cioè $\Sigma_1 \| \Sigma_2 \| ... \| \Sigma_n$. L'ipotesi nella definizione di insiemi di Input e Ouput disgiunti è ragionevole in quanto stiamo considerando Component Services in generale non correlati.

A Glimpse of our Goal
Possiamo ora accennare al nostro scopo finale utilizzando la terminologia adottata finora: la realizzazione del Composite Service a partire dai nostri requirements si concretizza nel trovare un STS Σ_C, che soddisfa particolari proprietà: esso deve "muoversi" all'interno di $\Sigma_{\|}$ per "controllare" i component services, allo stesso tempo rispettando il goal ρ ed evitando di porsi in stati pericolosi, come ad esempio un deadlock.
Per procedere nella nostra metodologia definiamo innanzitutto la nozione di *Sistema Controllato*.

Controlled System STS, for Σ_C controlling Σ
Nell'ambito di Sistema Controllato ci sono due attori, entrambi STS; chiameremo questi due attori Σ_C e Σ, facendo in modo che il primo controlli il secondo. Ciò dà origine ad un nuovo STS, il Sistema Controllato, che descrive in che modo il Controller Σ_C controlli l'esecuzione di Σ.

Definizione: Controlled System
Siano $\Sigma = < S, S^0, I, O, R, L >$ e $\Sigma_C = < S_C, S_C^0, O, I, R_C, L_0 >$ due STSs tali che la funzione di labeling di Σ_C sia nulla per ogni stato s_C, cioè $L_0 (s_C)$ = insieme vuoto per ogni s_C in S_C.
Il nuovo STS $\Sigma_C \mid > \Sigma$ descrive il comportamento di Σ controllato da Σ_C, ed è definito come:

$$\Sigma_C \mid > \Sigma = < S_C \times S , S_C^0 \times S^0 , I , O , R_C \mid > R , L >$$

dove:

- $< (s_C, s), \tau, (s_C', s') >$ appartiene a $(R_C \mid > R)$ se $< s_C, \tau, s_C' >$ appartiene a R_C;
- $< (s_C, s), \tau, (s_C , s') >$ appartiene a $(R_C \mid > R)$ se $< s , \tau, s ' >$ appartiene a R;
- $< (s_C, s), a, (s_C', s') >$ appartiene a $(R_C \mid > R)$, con a diverso da τ, se $< s_C, a, s_C' >$ appartiene a R_C e inoltre $< s , a, s ' >$ appartiene a R;

Notiamo come l'insieme di azioni di input dell'uno coincida con l'insieme di azioni di output dell'altro; possiamo esprimere illustrare sinteticamente la relazione di transizione del Controlled System notando come: 1 - transizioni interne nel controller permettono transizioni interne di entrambi gli STSs nel Controlled System; 2 - transizioni interne nel controlled STS non permettono transizioni interne per il Controller; 3 - transizioni esterne sono possibili soltanto su azioni che coincidono dei due STSs, per uno di input e per l'altro di output. Notiamo inoltre come il controller Σ_C non abbia funzione di labeling associata.
La nozione di controllo deriva principalmente dalla corrispondenza input-output delle azioni: l'input del Controller è l'output dell'STS controllato, ovvero il controllato fornisce all'output le informazioni generate, mentre l'output del Controller è l'input del controllato, ovvero il Controller "istruisce" l'STS controllato sulle prossime azioni.

Adequate vs Inadequate Controllers: the Deadlock-Free Controllers

Le assunzioni di sistema asincrono in cui ci poniamo pongono di fronte a noi una difficoltà: non tutti gli STS controllers per un dato STS sono adatti ai nostri scopi: vorremmo evitare deadlocks, più precisamente vorremmo che ogniqualvolta il Controller "invii" un messaggio in output all'STS controllato, questo sia pronto a ricevere tale messaggio, eventually.
Dobbiamo pertanto definire un sottoinsieme dei Controllers possibili, tale che l'STS controllato possa ricevere gli input forniti, eventualmente dopo una catena, arbitrariamente lunga ma finita, di τ-transitions.
Per i nostri scopi è utile anche definire la nozione di τ-closure(s) di uno stato, che è semplicemente il set di stati in un STS raggiungibili da s tramite transizioni interne. Possiamo anche definire τ-closure(S) l'unione delle chiusure per ogni stato s in S.
Definiamo dunque il concetto di deadlock-free controller, tagliando fuori dalla nostra sfera d'interesse i Controllers non adeguati.

Definizione: Deadlock-Free Controller w.r.t the controlled STS
Siano $\Sigma = < S, S^0, I, O, R, L >$ e $\Sigma_C = < S_C, S_C^0, O, I, R_C, L_0 >$ due STSs tali Σ_C è un controller per Σ**.**
Σ_C è detto "deadlock-free per Σ" se per ogni stato (s_C, s) in S_C x S raggiungibile dagli stati iniziali del Controlled System $\Sigma_C \mid > \Sigma$, sono soddisfatte le seguenti proprietà:

- se in R compare una transizione del tipo < s, a, s' > con 'a' azione di *output*, allora esiste uno stato s_C' appartenente alla τ-closure(s_C) tale che in R_C compaia la transizione $< s_C', a, s_C'' >$ per qualche s_C'' appartenente a S_C;
- se in R_C compare una transizione del tipo $< s_C, a, s_C' >$ con 'a' azione di *input*, allora esiste uno stato s' appartenente alla τ-closure(s) tale che in R compaia la transizione < s', a, s" > per qualche s" appartenente ad S;

Esaminando con attenzione la definizione vediamo come la prima proprietà imponga che, se l'STS effettua una qualunque transizione di output ("invia" un messaggio al Controller), allora per il Controller esisterà una input transition raggiungibile dallo stato presente per accettare quell'input lanciato; la seconda proprietà è esattamente la duale della prima.
Una volta definito come possiamo ottenere un controller per il prodotto parallelo di N STSs, e avendo mostrato quali tipologie di Controllers ci interessano, passiamo alla questione del soddisfacimento della formula ρ che esprime i composition requirements.

Towards meeting the Composition Requirements: the Belief Evolution
Per affrontare la successiva parte del problema, mettiamo per prima cosa in corrispondenza il concetto di STS e la soddisfacibilità di un goal ρ.
Intuitivamente abbiamo un goal definito in EAGLE e un STS che è il nostro Controlled System, $\Sigma_C \mid > \Sigma_{\parallel}$; l'STS in questione ha una serie di possibili esecuzioni, e se vogliamo un goal di tipo **doMaint p**, saremmo esclusivamente interessati a sistemi controllati tali che per ogni esecuzione, la proprietà p è vera in tutti gli stati. Un requirement di tipo **doReach** imporrebbe che per ogni esecuzione si raggiungesse, eventually, una configurazione in cui p vale.

Per affrontare tale problema è necessario definire tutte le possibili esecuzioni del sistema controllato; tuttavia, essendo in un ambiente solo *parzialmente osservabile*, nel senso che il Controller *non ha* osservabilità sulle transizioni *interne* dell'STS controllato, abbiamo bisogno di qualcosa per aggirare il problema. Ricordiamo che siamo in un ambiente *asincrono*, e quindi non sappiamo quando e se i messaggi che aspettiamo o abbiamo inviato arriveranno, e inoltre *non deterministico*, in quanto stiamo interagendo con e-services non correlati, e quindi ad esempio

non possiamo aspettarci che il WS Producer abbia accesso diretto alle variabili interne di Shipper, e di conseguenza non può prevedere l'esito delle azioni possibili.

Cerchiamo di risolvere il problema considerando, per l'STS d'interesse afflitto da assunzioni di asincronicità, non-determinismo e osservabilità parziale, un nuovo STS, che esprime tutti gli stati ugualmente plausibili raggiungibili passo per passo dall'STS iniziale, secondo le informazioni in nostro possesso.
Un *Belief*, o *Belief State*, è un set di stati che può essere raggiunto a partire da un noto set di sati in nostro possesso; l'idea è che il Belief iniziale coincida con l'insieme di stati iniziali, poiché siamo certi che inizialmente avremo la configurazione iniziale, ed il Belief è aggiornato ogni volta che il sistema evolve tramite una external (e quindi osservabile) transition, input o output.
Definiamo allora il concetto di Belief Evolution introducendo la funzione *Evolve(B,a)*:

Definizione: Belief Evolution
Sia Σ un STS e B un Belief , con B sottoinsieme di S.
Definiamo la *Belief Evolution* di B a causa dell'azione 'a' come un nuovo Belief B' = Evolve(B, a) etale che:

> Evolve(B, a) = { s' | exists s appartenente alla τ-closure(B) con
> < s, a, s' > appartenente ad R }

Ora vedremo come l'uso del Belief e la sua evoluzione possa permetterci di controllare la soddisfacibilità di proprietà.

Satisfiability of Properties: the Belief-Level System
Abbiamo introdotto il concetto di Belief per far fronte alle nostre ipotesi generali e per descrivere una "configurazione" del nostro Controlled System. Poiché siamo interessati alle proprietà che le configurazioni possibili verificano, è necessario definire quando un Belief B soddisfa una proprietà p.
Non c'è dubbio che se tutti gli stati che B contiene soddisfano p, allora B soddisfa p; tuttavia, nel caso B contenesse uno stato che non soddisfa p, il nostro scenario è complicato ulteriormente dalla presenza di transizioni interne, che potrebbero portarci in altri stati che soddisfano p senza che il Belief evolva; pertanto, se in un Belief B non tutti i suoi stati soddisfano p, ma esiste una catena di τ-transitions tale che a partire da tali stati si raggiungono altri stati che soddisfano p, allora B soddisfa p. Ne consegue che B non soddisfa p se non tutti i suoi stati la soddisfano e per ogni sequenza di τ-transitions possibile da tali stati, non si raggiungono stati che soddisfano p.
Forniamo di seguito la definizione formale:

Definizione: Belief satisfying a property
Sia $\Sigma = < S, S^0, I, O, R, L >$ un STS, p appartenente a *Prop* una proprietà per Σ, e B sottoinsieme di S un Belief.
Si dice che B soddisfa p ($B \models_{\Sigma} p$) se è verificata la seguente condizione:
si consideri la sequenza di stati $s_0, s_1, \ldots, s_n$, tali che s_0 appartiene a B, le transizioni non osservabili del tipo $< s_i, \tau, s_{i+1} >$ appartengono ad R, e per quanto riguarda s_n, esso o non ha transizioni in uscita o esiste una n+1-esima transizione s_{n+1} del tipo $< s_n, a, s_{n+1} >$ con a diverso da τ. Allora p *deve appartenere* ad $L(s_i)$ per qualche indice i tra 0 ed n.

Una volta definito il concetto di soddisfacibilità di proprietà con l'ausilio delle Beliefs, possiamo descrivere tutte le esecuzioni di un generico STS Σ (che nel nostro problema è il Controlled System $\Sigma_C \mid > \Sigma_{||}$) tramite un STS che chiamiamo "*Belief-Level System*", i cui nodi sono i possibili Beliefs di S, mentre gli archi rappresntano le possibili Belief Evolutions.

Definizione: Belief-Level System
Sia $\Sigma = < S, S^0, I, O, R, L >$ un STS. Il corrispondente Belief-Level STS $\Sigma_B = < S_B, S_B^0, I, O, R_B, L_B >$ è definito nel seguente modo:

- S_B è l'insieme di Beliefs di Σ raggiungibile dall'insieme di Beliefs iniziali S_B^0;
- $S_B^0 = \{ S^0 \}$;
- se Evolve(B, a) = B', con B' diverso dall'insieme vuoto, per qualche azione a di input o ouput, allora < B, a, B' > appartiene a R_B;
- L_B (B) = { p appartenenti a *Prop* | B $|=_\Sigma$ p }.

Il grande vantaggio ottenuto dal passaggio al Belief-Level System deriva dai pesanti constraints che lo caratterizzano: infatti Σ_B possiede un unico stato iniziale, non possiede alcuna τ-transition, e per tutti i Beliefs B e le azioni a esiste al massimo un unico Belief B' tale che < B, a, B' > appartenga a R_B.
In questo scenario possiamo riconsiderare la soddisfacibilità di un goal r, ma stavolta in un *dominio non-deterministico e completamente osservabile*. Otteniamo un problema di planning che può essere risolto con tecniche affermate come il Symbolic Model Checking.

Definition of the Astro Composition Problem
Di seguito riportiamo la definizione formale del problema della composizione in Astro:

Definizione: Astro Composition Problem
Siano $\Sigma_1, ..., \Sigma_n$ un insieme di STSs, e ρ un composition requirement.
Il problema di composizione per $\Sigma_1, ..., \Sigma_n$ e ρ è il problema di trovare un Controller Σ_C che è deadlock-free e tale che $\Sigma_B |= \rho$, dove Σ_B è il Belief-Level System di $\Sigma_C |> (\Sigma_1 \| ... \| \Sigma_n)$.

Una volta trovato il Controller, il modulo software STS2BPEL to traduce in un Concrete BPEL Process eseguibile.

The Passage to the Planning problem in Fully Observable, Nondeterministic Domains
Questa sezione mostra come dalla definizione di Composition Problem possiamo portarci nelle condizioni di applicare in modo efficace tecniche di planning al problema, in condizioni di piena osservabilità grazie al Belief-Level, e risolvere il tutto tramite Symbolic Model Checking.

Innanzitutto viene effettuata la creazione di un Dominio D ottenuto a partire dal Belief-Level System considerato Σ_B per Σ, e si ottiene una tupla $D = < S, S^0, A, T, L >$, in cui:

- gli stati dell'insieme S sono coppie $< s_B, o >$ che includono stati del Belief-level System più output dell'ultima transizione effettuata; un carattere speciale '*' è usato in caso l'ultima transazione non abbia avuto output;
- A è l'insieme di azioni, unicamente di input, ed include anche il carattere '*' per modellare le transizioni di output di T;
- T è la funzione di transizione da una coppia <B, o> ad una coppia <B', o'>, scatenate da azioni * se output transitions e da azioni a != * in caso di input transiztions, secondo le transizioni definite in R_B;
- L è funzione di labeling per una coppia <B, o>, e si riduce al labeling di B.

Disponendo di un goal e di un planning domain D possiamo ricercare un piano π; i dettagli del problema di planning sono riportati in **[PT01]**, nel seguito se ne forniscono alcuni cenni.

Un piano per D e ρ è una tupla $\pi = < C, c^0, \alpha, \varepsilon >$, cioè un insieme di contesti d'esecuzione C, dei quali c^0 è quello iniziale, e due funzioni di evoluzione, una per le azioni α: C x S $\rightarrow$ A, e una per i contesti ε: C x S $\rightarrow$ C.
L'esecuzione di un piano su un dominio è definito in termini di configurazioni, ovvero coppie <contesto, stato>, in cui le azioni eseguite e i nuovo contesti raggiunti dipendono dalle apposite funzioni definite all'interno del piano.
Siamo interessati a piani *eseguibili*, ovvero piani in cui partendo dagli stati iniziali, per ogni azione compiuta sia definito il dominio successivo corrispondente; la *struttura d'esecuzione* di un piano è costituita da tutte le configurazioni raggiungibili, ognuna con delle proprietà soddisfatte in accordo alla funzione di labeling.
A partire dalla struttura d'esecuzione possiamo costruire un STS Σ_π che descrive le esecuzioni del piano π relativamente al dominio D e alle proprietà soddisfatte, reintegrando nell'STS le informazioni sulle output transitions.
L'STS così ottenuto dal piano non è in genere deadlock-free, ma sotto determinate ipotesi sull'STS originario Σ si può garantire che il Σ_π ottenuto sia deadlock-free, e inoltre si può garantire che *Σ soddisfi ρ se Σ_π soddisfa ρ nel dominio D corrispondente a Σ*. Le pre-condizioni necessarie per questa importante proprietà sono:

- non esistono Belief-states in cui sono possibili sia input che outputs contemporaneamente;
- quando è il turno del Controller, è garantito che il dominio sarà in grado di processare qualunque messaggio inviato dal Controller.

Sotto tali ipotesi il sistema Σ si dice *controllabile*.

Tali ipotesi restrittive sono comunque ragionevoli: la condizione di non avere in/out Beliefs corrisponde all'assunzione di sapere sempre se un WS con cui stiamo interagendo sta aspettando una nostra invocazione o sta per inviare un messaggio in risposta; la seconda condizione richiede che in ogni momento sappiamo quali sono le invocazioni valide accettate da un WS, e ciò è effettivamente verificato.

Nel seguito ci accingiamo a concludere l'analisi della logica teorica dietro il Composition Problem in Astro, mostrando la corrispondenza tra il problema della sintesi e quello della pianificazione SMC.

Lemma: Controller/Plan Executability
Sia Σ un deadlock-free STS, Σ_B il suo Belief-Level System e D il corrispondente planning domain. Sia inoltre π un piano per D, e Σ_π l'STS corrispondente a π.
Si ha che se π è eseguibile su D, allora Σ_π è eseguibile su Σ.

Lemma: Controller/Plan Equivalence
Sia Σ un deadlock-free STS, Σ_B il suo Belief-Level System e D il corrispondente planning domain. Sia inoltre π un piano per D, e Σ_π l'STS corrispondente a π.
Si ha che se π è soluzione su D per il goal g, allora è vero che $(\Sigma_\pi \mid> \Sigma) \mid= g$.
Inoltre, se esiste un STS Σ_C tale che $(\Sigma_C \mid> \Sigma) \mid= g$, allora esiste un piano π che è soluzione per il goal g su D.

Con l'ultimo lemma abbiamo dichiarato di poter utilizzare affermati algoritmi di planning per fully observable, nondeterministic domain con extended goals per risolvere in modo efficiente il problema della composizione automatica.

Sezione 2.5 - Other Aspects of ASTRO Composition Efforts

Oltre alle aree di Business Requirements e Composition Synthesis, il Progetto Astro ha sviluppato tools e metodologie per trattare anche Monitoring, Verification e Semantics.
Sebbene il focus della tesina sia rivolto principalmente all'aspetto di sintesi automatica, forniamo comunque alcuni cenni al riguardo.

Monitoring in Astro
Per Monitoring si intende il controllo, effettuato a runtime, che permette di verificare, per una data proprietà, se questa è rispettata o violata, o di riscontrare la presenza di anomalie, o ancora di controllare se una certa sequenza specifica di eventi si è verificata.
Il tool wsMonitor (wMon) permette, data una proprietà o situazione d'interesse, la generazione automatica di Java code da eseguire a runtime per monitorare i processi.

Verification in Astro
Mentre il monitoring si occupa di error detection a runtime, gli efforts del team Astro in Verification vogliono fornire allo sviluppatore di Composite Web Services strumenti per controllare che goals e constraints siano soddisfatti a design time.
Il tool wsVerify accetta in input il materiale per la composizione (Abstract BPELs dei component services, Concrete BPEL del target service, composition requirements) e in caso scopra errori mostra uno scenario di esecuzione ("error flow") in cui le specifiche sono violate cosicché lo sviluppatore possa correggere il suo lavoro.

Semantics in Astro
In linea con lo sforzo per un ambiente di programmazione semantico, in cui gli strumenti utiilzzati non si limitino ad eseguire ciecamente ordini, ma "comprendano" i nostri desideri e scopi, nel Progetto Astro sono in corso iniziative per estendere ai linguaggi OWL-S e WSMO i processi di Monitoring, Verification e Synthesis. Lo scopo è passare da specifiche semantiche dei Web Services a processi eseguibili.

Parte III: L'Approccio del Roman Group

Nella parte del corso di Seminari di Ingegneria del Software dedicata alla Service Composition è stato introdotto il cosiddetto Roman Approach, ovvero il lavoro dei professori dell'Università di Roma "La Sapienza" (Berardi, De Giacomo, Mecella, Calvanese) volto a definire una metodologia per la Automatic Service Composition.
Nel seguito viene sinteticamente riepilogato il Roman Approach, vengono evidenziati i punti principali per preparare il terreno alla parte di confronto comparativo di cui tratterà il capitolo IV.

Sezione 3.1 - Roman Model Basics: the Community

Al centro del Roman Approach c'è l'enfasi sull'importanza delle *singole azioni* che un e-service può eseguire (o, utilizzando la terminologia degli autori, "i comportamenti e le azioni che il web service *esporta* verso l'esterno"); è ovvio che parlando di azioni che un servizio può compiere, se vogliamo mantenere un adeguato livello di astrazione (il che è una necessità, nel complesso problema della composizione) dobbiamo centrare la nostra attenzione su una descrizione *comportamentale* dei component services, nonché del target service, utilizzando formalismi utili come quello descritto nel capitolo introduttivo.

Il vero fulcro dell'approccio è il concetto di *Service Community*, ovvero una sorta di "Federazione di Component Web Services" che *esportano* verso la community e all'esterno il proprio insieme di azioni, che vengono *comprese e condivise* da tutti i membri della Community.
Il join di un nuovo membro alla Community si traduce *nell'esportazione delle proprie azioni in termini del Commmon Action Alphabet* della comunità.

La scelta per la modellazione dei servizi nel Roman Model è caduta sulle FSMs non deterministiche (anche chiamate "Transition Systems"), delle tuple $< \Sigma, S, S_0, \delta, F >$, dove:

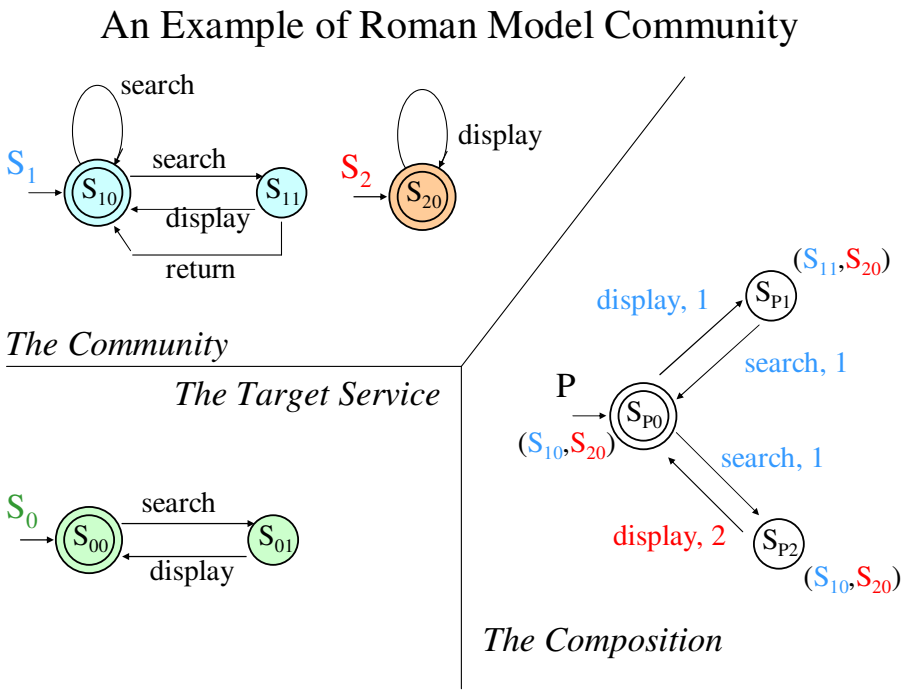
- S è l'insieme di stati;
- S_0 è l'insieme di stati iniziali, F quello di *possibili* stati finali;
- δ è la relazione di transizione;
- Σ è l'insieme di azioni condivise dai membri della community, un alfabeto comune.

Qual è allora l'added value della Community? Condividendo azioni con altri membri, ogni servizio, durante una composizione, può *delegare l'esecuzione di parte delle proprie azioni ad altri servizi che "possono sostituirlo"*. Per effettuare ciò è necessario che l'entità che si occupa della coordinazione (l'Orchestrator) abbia *full observability* sugli stati dei component services, e abbia la possibilità di interrogarli in proposito a runtime. L'assuzione di osservabilità è ragionevole in questo contesto in quanto il set di azioni condivise dalla community permette tale scenario.

Il target service definito dall'utilizzatore del composite service è un STS *deterministico* (in quanto si assume che l'utente sappia ciò che vuole fare nei vari passi di esecuzione), nel senso che ad ogni azione dell'utente corrisponde un unica transizione di stato, ed è costruito a partire dalle azioni esportate dai membri; l'utente può effettuare operazioni a suo piacimento, ed interagisce con il servizio composito a runtime, dove potrebbe decidere di interrompere le

operazioni in uno stato finale arbitrario o continuare, cioè il servizio composito non è un'entità statica e pre-programmata, ma user-centered e gestita a runtime dal programma di orchestrazione. Dal punto di vista del client del Composite Service il comportamento definito è dato *dall'External Schema*, un unfolded execution tree che caratterizza le histories di invocazione delle azioni atomiche che compongono il servizio; per l'Orchestratore ciò che ha importanza è l'Internal Schema, del tutto simile all'external ma con archi etichettati con la coppia <azione, performer> invece della sola azione scelta: il programma di orchestrazione deve poter garantire la sequenza di operazioni richieste assegnando a runtime l'azione da eseguire ad un opportuno servizio.

Nel seguito si riporta un esempio di Community su piccola scala e un servizio composito utilizzatore, nonché l'STS che gestisce l'orchestratore.



Sezione 3.2 - PDL Reduction of the Composition Problem

Definiti i concetti alla base del Roman Approach, accenniamo alla logica di fondo che permette di individuare l'esistenza o meno di una composizione in modo sound and complete, esaminando anche gli aspetti computazionali che emergono.

Il problema della composizione viene ridotto alla soddisfacibilità di una formula DPDL (Deterministic Propositional Dynamic Logic), un linguaggio per eseguire reasoning su programmmi.
Per una introduzione dettagliata a DPDL si consulti **[IJCIS05]**; in questa sede ci concentriamo su come DPDL venga utilizzato per rispondere a tre domande fondamentali:

1. Possiamo sempre controllare che una composizione effettivamente esista?
2. Se esiste, il suo Transition System è finito?
3. Se si, come si può computare la composizione?

L'encoding in DPDL del problema di composizione risulta polinomiale rispetto alle dimensioni dei component Transition Systems; la fomaula DPDL finita che caratterizza il problema ha la forma:

$\Phi = \text{Init} \;\&\&\; [u] \; (\Phi_0 \;\&\&\; (\Phi_1 \;\&\&\; \Phi_2 \;\&\&\; ... \;\&\&\; \Phi_n) \;\&\&\; \Phi_{aux})$

"Init" descrive lo stato iniziale dei servizi in gioco; F0 è l'encoding del target service TS, realizzato esprimendo i vari vincoli che formano la FSM con l'aiuto degli operatori modali DPDL; ognuno dei Φ_i descrive l'evoluzione di un Component Service, con particolare attenzione al suo stato e alle relazioni con gli altri servizi componenti: esso include dei predicati "$moved_i$" condizionali per esprimere l'inattività del servizio in caso un altro servizio abbia svolto quella particolare azione; la formula finale ausiliaria si occupa di constraints relativi agli stati iniziali e finali, e alla "liveness" della procedura.

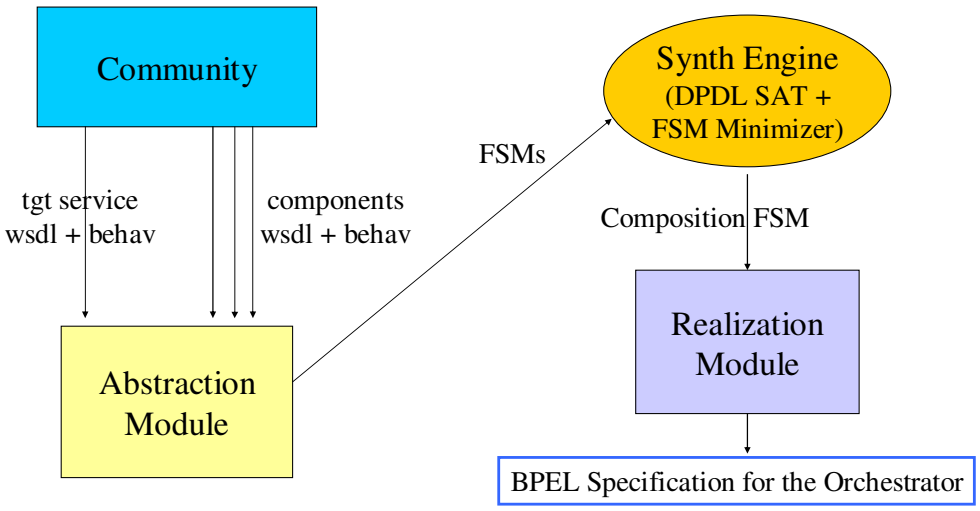
Possiamo rispondere alle tre domande poste precedentemente grazie ad un teorema che afferma che *"la composizione esiste se e soltanto se la formula Φ è soddisfacibile"*, il che è decidibile in EXPTIME. Prima di rispondere alle domande accenniamo alla "small model property" di DPDL: ogni formula DPDL soddisfacibile ammette un modello finito di dimensione al massimo esponenziale rispetto alla dimensione della formula.
Pertanto:

1. La composizione è controllabile poiché l'encoding del problema è decidibile in EXPTIME;
2. Un finito TS di composizione esiste grazie alla "small model property" di DPDL;
3. La composizione può essere computata da uno small model della DPDL formula.

Sezione 3.3 - Roman Model Synthesis Workflow

Mostrato come sia possibile, nel Roman approach, trattare il problema della WS Composition in modo sound and complete, riportiamo il workflow generale del capitolo I in chiave Roman Model.

The Roman Model Composition Workflow



Il workflow consiste nel codificare innanzitutto il problema in DPDL in tempo polinomiale, quindi controllare in tempo esponenziale la soddisfacibilità, ed in caso positivo ricavare uno small model, da cui estraiamo una Mealy FSM che rappresenta l'Internal Schema da fornire all'orchestratore; la FSM viene minimizzata prima di essere usata per costruire l'orchestration program.

Sezione 3.4 - The Current State of Advancement

Al momento della stesura del presente lavoro, il Roman Group dispone di un solido terreno di partenza su cui costruire, e tuttavia non è ancora avvenuto il passaggio ad una forma finale per le fasi implementative dell'approccio.
In **[IJCIS05]** viene descritto lo stato di avanzamento di un prototype module, chiamato *ESC*, che implementa parte del workflow descritto nel paragrafo precedente.
Correntemente la struttura dell'Abstraction Module dipende dal linguaggio scelto per rappresentare le descrizioni di input dei Component Services, mentre gli sforzi realizzativi si sono concentrati principalmente sul module di Synthesis Engine, che ha prodotto buoni risultati anche senza un lavoro di fine tuning e ottimizzazione. Il modulo per la traduzione del Composite Service in un linguaggio per l'orchestratore, come ad esempio Concrete BPEL, non è ancora ultimata.

Parte IV: Il Confronto tra i Due Approcci

Delineati nei precedenti capitoli gli aspetti principali dei due approcci, cerchiamo di mettere in risalto le differenze e le analogie tra i due modi di vedere e risolvere il difficile problema della Service Composition.

Sezione 4.1 - Similarities and Differences in the various Stages of the Composition Workflow

The Composition Problem
Come argomentato in **[AISC06]**, sia l'approccio Astro sia il Roman approach condividono una visione del problema della Service Composition di tipo *Client-Tailored*, cioè incentrata sulle esigenze dell'utente. A differenza di molti approcci nel mondo della ricerca orientati alla composizione *Service-Tailored*, in cui le funzionalità composte offerte sono subordinate alle procedure offerte dai servizi esistenti, nell'approccio Client-Tailored l'utente specifica il suo target service, ed entità di sintesi ed orchestrazione si occupano di utilizzare frammenti dell'esecuzione dei Component Services per soddisfare le aspettative dell'utente.
Questo è certamente il caso di Astro e del Roman Group: in Astro il target service è specificato come un e-service (modellato tramite Abstratc BPEL Process) e la sintesi consiste nel creare un processo concreto per permettere l'interazione con i componenti rispettando dei vincoli di business requirements, mentre nel Roman model l'utente può addirittura assemblare a piacimento le azioni offerte dalla Community come un bambino assembla blocchi per le costruzioni, portando all'estremo l'atteggiamento client-tailored, ancora più dell'approccio Astro.

Una differenza tra le due metodologie per considerare il problema nel suo insieme è anche data dall'attenzione all'aspetto pratico ed implementativo: mentre il Roman group ha inizialmente costruito solide basi teoriche per definire e risolvere il problema, senza legarsi a nessuna tecnologia pratica e perseguendo un goal molto ambizioso ed affascinante la cui incarnazione non è ancora interamente definita, gli sforzi del progetto Astro sono dichiaratamente volti in primo luogo a fornire un tool efficiente, scalabile e pratico per aiutare il progettista/implementatore a risolvere il problema in questione: le funzionalità sono iniziate con un core primitivo, da subito legato a specifici linguaggi e standard (come BPEL), che è stato via via raffinato sia in performances che in ampiezza, ed è ad oggi funzionante, sebbene in continuo cambiamento.

Component Services, Requirements and the Background Architecture
Riguardo la modellazione dell'input, le due metodologie presentano una similarità nell'utilizzare entrambe delle FSM per modellare i comportamenti dei Component Services e del target service, ma nel corso della relazione abbiamo delineato anche importanti differenze.
Senza dubbio l'aspetto più evidente che differenzia i due approcci è la presenza o meno di una struttura di fondo su cui fare assunzioni: il Roman Model presuppone l'esistenza della Community e di tutto ciò che la Community comporta, mentre Astro non accenna a nessuna architettura pre-esistente, a parte i servizi offerti dai Component Services isolati.

Tale importante differenza lascia le sue tracce anche sulle assunzioni che vengono fatte per le caratteristiche dei servizi coinvolti: Astro presuppone un ambiente asincrono e partially observable, in cui solo le interazioni esterne sono note (e ciò causa la necessità del passaggio al Belief level in planning) mentre la Community permette full observability al programma orchestratore, il quale non potrebbe schedulare azioni efficacemente a runtime senza poter ispezionare in dettaglio lo stato dei vari membri.

Esistono anche altre differenze nei requirements derivate dalla presenza o meno di un'architettura di fondo: l'alfabeto di creazione del target service, ad esempio, differisce nei due approcci. Nel Roman Model esso coincide con il set di azioni reso disponibile dalla Community, azioni componibili a piacimento, mentre in Astro il target service è costruito sfruttando un joint effort tra le funzionalità offerte dai component services (le azioni esposte dai vari "frammenti", viste simili ad interfacce remote) e l'espressione dei business requirements in un linguaggio di alto livello per extended goals, per costruire un sistema sì specificato dall'utente e quindi client-tailored, ma più "statico" rispetto al Roman Model.

In sostanza la presenza di un'architettura di fondo dona vari benefici, quali assunzioni semplificative, regolarità, maggiore semplicità nella definizione del target service, ma per contro richiede uno sforzo aggiuntivo di gestione e consistenza (ad esempio il join di nuovi membri alla community).

Abstraction & Synthesis
La figura del modulo software dedito all'astrazione degli input in qualche forma di FSM è comune a entrambe le metodologie, sebbene vi siano delle lievi differenze nella struttura delle tuple, derivate dalle diversità delineate nei punti precedenti: mentre in Astro si ragiona in termini di "proprietà soddisfatte in un certo stato" per conciliare gli STS con il controllo dei requirements, e si distingue tra input, output actions e τ actions per differenziare transizioni osservabili dalle non osservabili, nel Roman model compare il set Σ di azioni condivise dalla Community, e l'esplicita presenza di "stati finali", ovvero stati in cui il client può *scegliere* se terminare o meno.

Gli approcci alla sintesi sono molto differenti: DPDL satisfiability in EXPTIME e ricerca di una soluzione al problema come Mealy FSM minimizzato nel Roman model, contro creazione di un dominio di planning e di un piano per il dominio che soddisfa un goal, il tutto risolto via Symbolic Model Checking affrontando il problema al Belief-level.
I test pratici eseguiti dal team Astro su varie dimensioni del problema (numero e complessità dei component services, ad esempio) riportano risultati buoni nei casi di media complessità, che corrispondono alla maggioranza dei WS di oggi, e riportano un sostanziale miglioramento rispetto a test precedenti eseguiti su versioni più primitive e limitate del tool (dell'ordine di 1/50 del tempo impegato). Questo deriva dalle efficienti tecniche di planning esistenti per risolvere le tipologie di problemi a cui Astro ha ricondotto il proprio.
Anche il Roman Group riporta in **[IJCIS05]** un cenno ai test condotti su Communities di al massimo 10 membri, ognuno con una complessità dell'ordine di 10-20 stati, per i moduli di Astrazione e Sintesi, con buoni risultati anche senza una forma finale ed ottimizzata del tool.

Final Result of the Composition
In questo settore l'approccio del Roman Group è ancora Work-In-Progress, e mira a voler fornire un programma per l'uso dell'orchestratore, per funzioni di "online Service fragment coordination", in linea con lo schema totalmente client-tailored perseguito; gli scopi del team Astro sono piuttosto diversi, poiché mirano a produrre automaticamente un processo BPEL eseguibile che rappresenta una sorta di nuovo servizio più statico: una volta effettuata la

composizione e definiti gli obiettivi l'utente non ha alcun controllo sull'esecuzione, bensì dispone di un nuovo servizio composito. Il problema è in un certo senso affrontato in modo ortogonale, da punti di vista differenti ma ugualmente interessanti e senza dubbio challenging.

Sezione 4.2 - A Graphical Glance of the two Approaches in today's Landscape of Service Composition

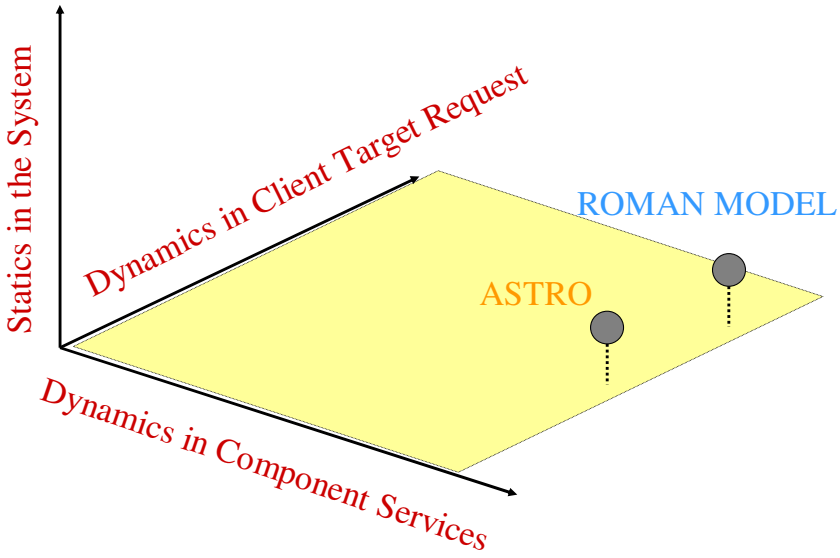
In vari articoli del settore è riportato un interessante grafico, si consulti ad esempio **[AISC06]**. Esso consiste in una sorta di piano tridimensionale, i cui tre assi individuano tre diversi aspetti dell'approccio alla Web Service Composition, e quindi la posizione di un punto in questo "spazio della composizion research" individua la politica perseguita da un gruppo di ricerca in questo ambito.

I tre assi indicano:

- *Statics in the System* : questa dimensione indica il livello di modellazione di proprietà statiche, l'alfabeto di azioni usato, la modellazione di input/output; in genere gruppi di ricerca concentrati su sforzi di modellazione degli input con Semantics approaches saranno alti su questa dimensione;
- *Dynamics in Component Services* : questa dimensione indica il livello di complessità per la modellazione dei Component Services; chi li modella con articolate descrizioni comportamentali via FSMs sarà più in alto di chi considera i servizi come azioni atomiche e monolitiche ("frammentazione" dei componenti);
- *Dynamics in Client Service Request* : la dimensione tratta in un certo senso la "client-tailorability" dell'approccio, in quanto esprime la sofisticatezza nella definizione del target service; più l'utente ha influenza sulla personalizzazione e la gestione del Composite Service, più l'approccio sarà alto nella scala.

Riportiamo nel grafico la posizione dei due approcci in esame:

Our Approaches in the WS Composition 3D Space



Come possiamo vedere, e questo è una conferma di tutto ciò che è stato detto finora, le due metodologie condividono essenzialmente sia la modellazione di propretà statiche del sistema sia la rappresentazione dei Component Services come TSs comportamentali; la grossa differenza nello spazio è data dalla target request e dalla libertà del client nella composizione finale: l'architettura Community-based del Roman Model colloca tale approccio molto in alto, in posizione maggiore rispetto al Progetto Astro, per il quale il target service è un aggiuntivo servizio statico, sebbene composito.

Parte V: Il Toolset Sviluppato nell'ambito del Progetto ASTRO

Questo capitolo delineerà in quale modo le tecniche e le metodologie discusse finora relativamente all'approccio Astro hanno preso vita sotto forma di un toolset per il lifecycle management di composite Web Services. La versione dell'Astro toolset esaminata è la 3.4.

Sezione 5.1 - Software Components forming the Astro Suite

L'Astro toolset (anche chiamato Astro Suite) 3.4 è formato da numerosi componenti software, alcuni sviluppati interamente dal team Astro, altri sono programmi di terze parti con le quali il toolset interagisce.
Segue un elenco dei vari componenti, accompagnato da brevi spiegazioni sul loro ruolo:

- *Java 1.5.x* - La JVM è un componente essenziale del toolset, poiché molte parti di esso sono scritte in Java e producono/usano files Java;
- *Eclipse IDE 3.2.2* - L'ambiente di sviluppo Eclipse è stato scelto dal team Astro come la colonna portante di tutta la sezione grafica del toolset; molte componenti della Suite sono state sviluppate come Eclipse plugins, e la creazione dei files necessari alla composizione è stata portata avanti grazie alle funzionalità offerte dalle estensioni di Eclipse dedicate alla creazione e al lifetime di BPEL processes, estensioni come ActiveWebFlow o ActiveBPEL Designer;
- *Tomcat Server 5.5.x* - Il server Tomcat è utilizzato per il deployment e running dei BPEL processes che incarnano i Web Services offerti; esso è utilizzato in tandem con un BPEL engine;
- *ActiveBPEL Engine 2.0* - L'ActiveBPEL engine è un prodotto freeware che permette di eseguire deployment e running di processi BPEL su un application server; la sua installazione è particolarmente semplice, poiché consiste nell'aggiungere dei componenti alla root directory di un application server (Tomcat nel nostro esempio); la sua attivazione è contestuale a quella del server su cui vive; il deployment di un BPEL process avviene eseguendo un packaging del file .bpel, wsdl relativi e files di deployment .xml e .pdd in un file .bpr, che viene copiato nella directory /bpr del server e rilevato automaticamente e deployed quando il server è running; il deployed process può essere monitorato via Web browser ed invocato da programmi client appositi per l'invocazione di Web Services.
- *Graphical Editing Framework & Graphical Modeling Framework Eclipse plugins* - questi due insiemi di plugins per Eclipse sono necessari per far funzionare l'aspetto visuale delle applicazioni Astro;
- *Astro wsToolset 1.8.0 - wsTranslator 0.14.0* - il package wsToolset è composto da quattro programmi necessari a vari stadi della composizione, attivabili da command line; il programma wsTranslator èl'importantissimol modulo adibito alle traduzioni dei files di coreografia (.chor) in vari formati di STS, ad esempio files .smv o Spin, per poi realizzare il prodotto parallelo dei Component Services e preparare il terreno per il planning via Model Checking, generando quindi il dominio D.

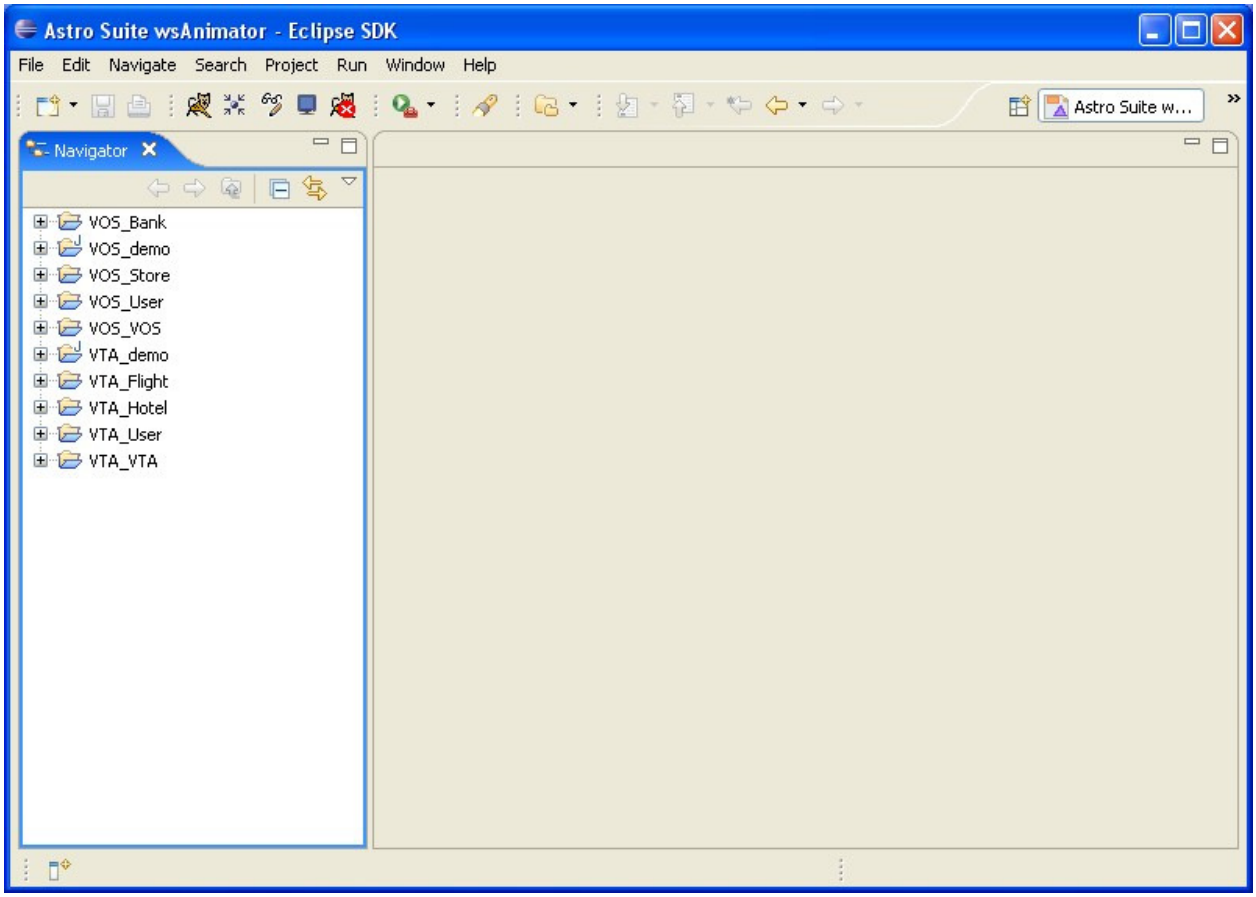
- *Astro wsToolset 1.8.0 - synTools 0.13.1* - il package synTools contiene due programmi, *wmon* e *wsynth*; il primo è adibito al monitoring dei processi BPEL, e quindi alla generazione del codice Java che controlla a runtime il verificarsi di eventi d'interesse e fa rapporto all'utente nelle schermate di monitoring dei processi (accessibili via browser); la seconda applicazione, wsynth, è la responsabile del vero e proprio processo di sintesi che ricava il piano π che soddisfa il goal ρ su dominio D e restituisce il file concrete BPEL eseguibile e con un certo livello di ottimizzazione per realizzare il composite service obiettivo;
- *Astro wsToolset 1.8.0 NuSMV 2.2.5* - prodotto da terze parti, NuSMV è essenziale per eseguire operazioni di model checking su STSs, il che è al cuore dell'approccio Astro;
- *Astro wsMonitor 1.6.0* - si è gia accennato a come l'installazione del BPEL Engine permetta di utilizzare una schermata accessibile via browser per monitorare i deployed BPEL processes; questa applicazione Astro è un'estensione a Tomcat che aggiunge funzionalità ulteriori di monitoring online all'interfaccia dell'Engine; del codice Java viene pre-generato ed eseguito a runtime dal wsMonitor per cercare situazioni insolite o di errore e fare rapporto all'utente via browser; è quindi sostanzialmente un'estensione dell'interfaccia offerta dal BPEL engine per il monitoraggio dei processi;
- *Astro wsRequirement 0.2.0 Eclipse plugin* - questa plugin permette di integrare i files di input necessari alla composizione (Abstract BPEL processes for component and target services, EAGLE requirements) creando in output un unico file xml con estensione .chor (file "di coreografia" che caratterizza completamente il problema) da dare in pasto a wsTranslator e wSynth per il processo di composizione; oltre alla creazione del file .chor, la plugin di Eclipse permette anche di analizzare le sue proprietà (come i component services, i "main" e "recovery" goals, gli interessi nel monitoring ecc) tramite un'efficace GUI;
- *Astro wsChainManager 2.4.0 Eclipse plugin* - la plugin permette di eseguire, a partire da un file .chor omnicomprensivo, vari servizi di composizione automatica, verification offline e preparazione per l'online monitoring, attivabili tramite la pressione di un singolo tasto; il nome deriva dallo stile di esecuzione: le varie funzionalità sono delle *catene* di chiamate ai vari componenti dell'Astro Suite, per realizzare il workflow discusso nel capitolo II;
- *Astro wsAnimator 0.0.7 Eclipse plugin* - la plugin è dedicata alla simulazione dei composite services impiegando lo stile grafico di ActiveWebFlow/ActiveBPEL Designer per mandare in esecuzione diverse tipologie di scenari di simulazione; usa files grafici con estensioni .adf;
- *Astro wsUseCases 1.0.0 Eclipse plugin* - questa plugin è semplicemente un insieme di folders che rappresentano due esempi di demo di composizione, chiamati VOS e VTA, rispettivamente dedicati a scenari "classici" come l'acquisto User-Store-Bank e la prenotazione User-Hotel-Flight; i folders contengono tutto l'occorrente per testare tutte le funzionalità offerte; la documentazione che accompagna le demo, tuttavia (e in generale anche il resto dei tools), è pressoché *inesistente*.

Sezione 5.2 - Composition Execution and Mappings

In questo paragrafo parleremo di come sia possibile eseguire una composizione automatica e di come i vari componenti dell'approccio Astro, trattato nella sua teoria di fondo nel capitolo II, trovino un'implementazione pratica nel wsToolset.

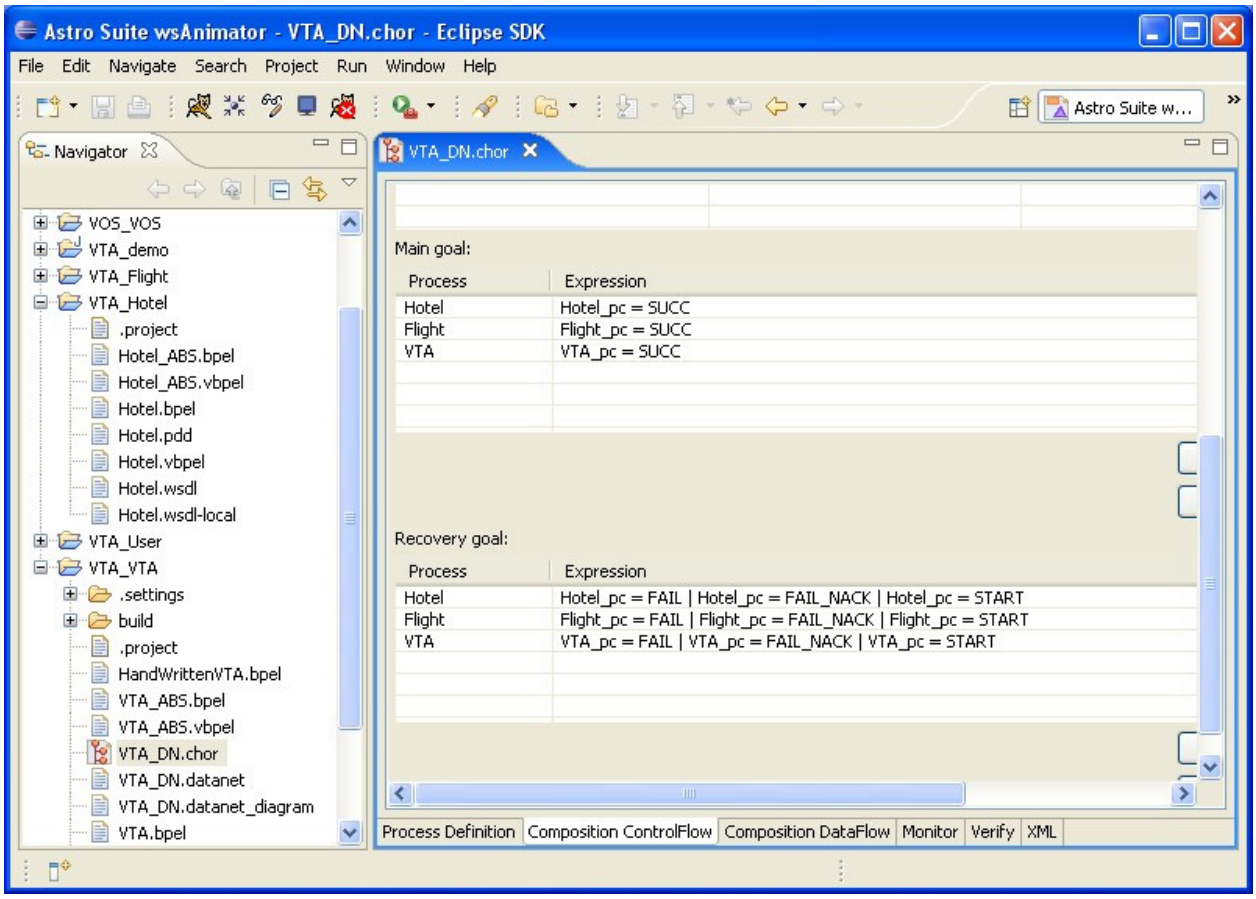
Problem Inputs

Una volta installati tutti i componenti descritti nel precedente paragrafo, e supponendo che i files di input alla composizione siano pronti, ci si posiziona in Eclipse, si caricano nel workspace i progetti di interesse (in figura sono uno per ogni Component Service, ed uno per il Composite), visibili nella Navigator Perspective, e si rendono disponibili sulla toolbar i pulsanti per le varie funzionalità dell'Astro toolset.



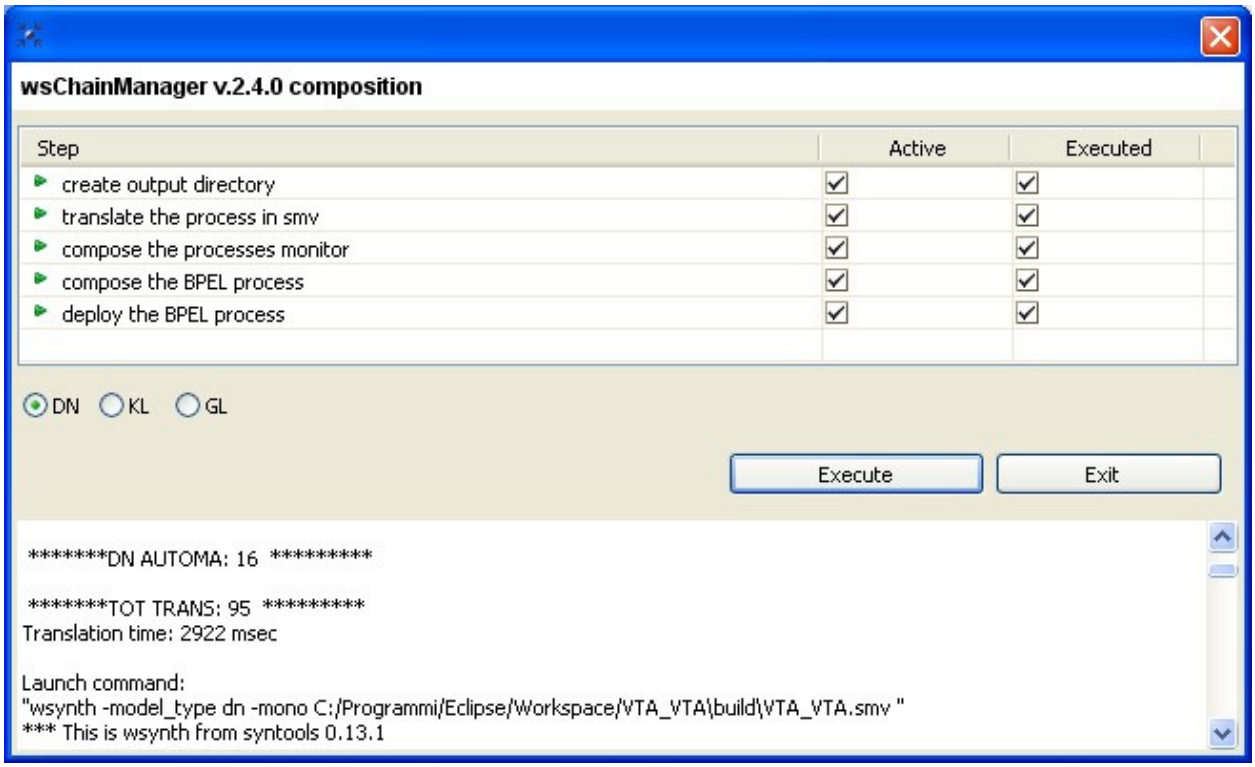
Notiamo sulla toolbar in alto una serie di 5 buttons, rispettivamente per: avviare Tomcat, iniziare la catena di Process Composition, avviare la Process Verification (offline), preparare le procedure di Process Monitoring, effettuare lo shutdown di Tomcat.

Gli input al problema sono dati dalle descrizioni astratte dei processi componenti e del target service, più il business goal della forma "try to reach main goal - upon failure, reach this other goal" (in generale: prova a portare a termine l'operazione, e in caso fallisca assicurati di non eseguire nessuna azione "pericolosa", come ad esempio un ordine di acquisto o un pagamento).
Le descrizioni dei processi sono fornite come Abstract BPEL Processes (files xxx_ABS.bpel) e corrispondenti wsdl files (xxx.wsdl), e tutti gli input sono riuniti insieme tramite un Wizard sviluppato da Astro per la creazione di files di coreografia (.chor files) grazie alla plugin wsRequirements; la plugin, oltre alla creazione degli onnicomprensivi files .chor (che includono le descizioni dei processi, il main goal, i recovery goals, le proprietà d'interesse da monitorare, quelle da verificare, ed altro) offre anche una UI per ispezionare a fondo il file coreografico, come mostrato in figura.

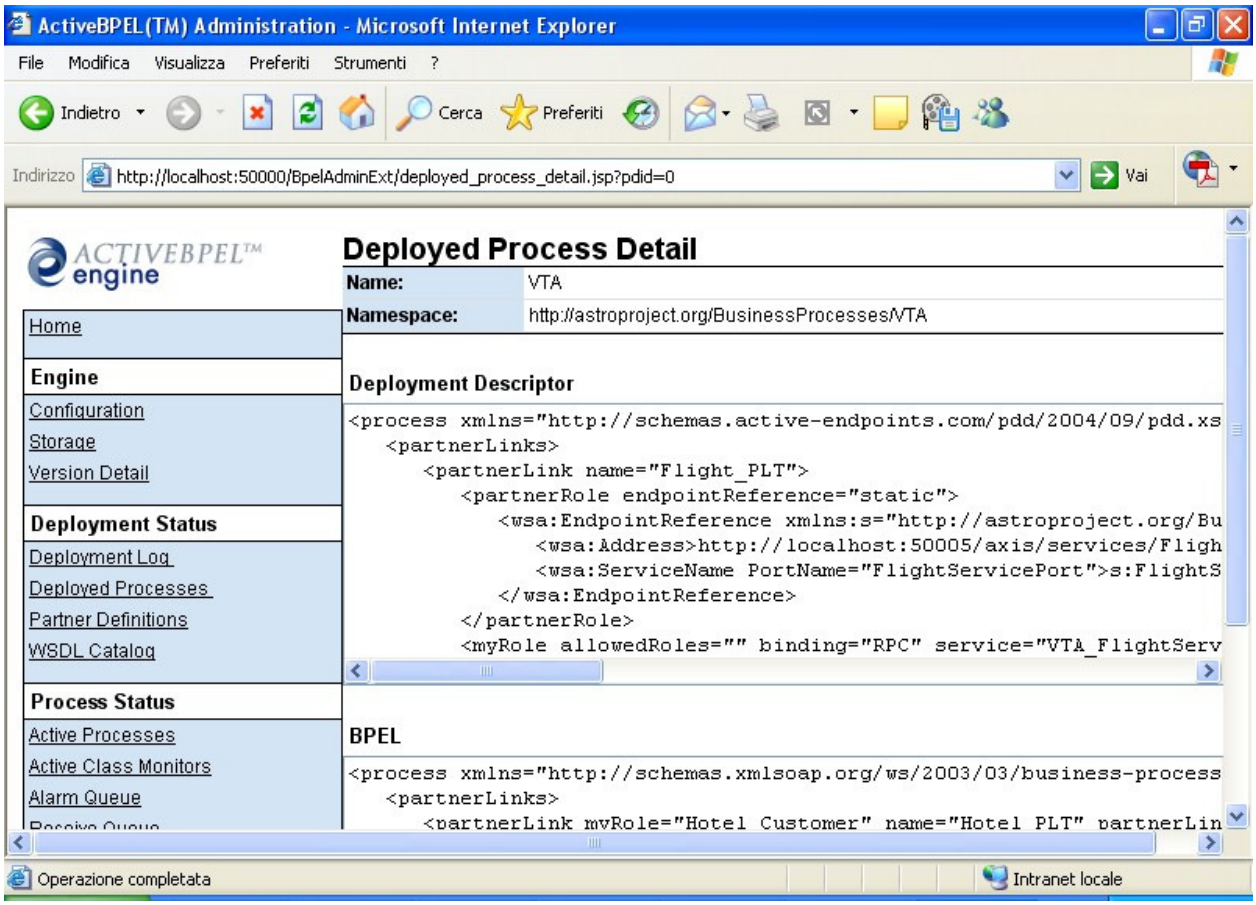


Nella Navigator view a sinistra possiamo vedere le composizioni dei folders (in questo caso contengono molto più del necessario, ad esempio il file HandWrittenVTA.bpel è un file eseguibile per il composite service scritto a mano da un programmer esperto del team Astro, e la sua funzione è essere comparato al file generato dalla composizione automatica); notiamo che il file VTA_DN.chor, che incarna i requirements della demo VTA (Flight-Hotel Reservation Service - Virtual Travel Agency) è stato selezionato ed aperto grazie a wsRequirements.
La schermata a destra è uno scorcio della UI offerta da wsRequirements per ispezionare i files .chor, più precisamente sono mostrati i goals (main e recovery) della composizione, all'interno della view "Composition Workflow" (notare i pannelli di switch per le views in basso a destra); l'interfaccia permette anche di modificare il file .chor on the fly (si ricorda che è anch'esso un XML file).

Composition Synthesis
Per avviare la composizione tramite wsChainManager si deve semplicemente avviare il Tomcat Server tramite toolbar, ed una volta che l'inizializzazione di Tomcat e del BPEL Engine è completata, si seleziona il file .chor (e tale azione rende attivi i tasti per composizione, monitoring e verification) e si invoca la funzionalità di Service Composition, che conduce ad una checklist di steps da affrontare; la message box in basso riporta dati sull'esecuzione, eventuali problemi e tempo impiegato, dimensioni delle strutture dati in gioco ecc. Ciò che il chainManager fa è costruire il dominio D dal prodotto parallelo e risolvere il problema di planning via Model Checking, utilizzando i programmi NuSMV, wsTranslator, wSynth descritti precedentemente.



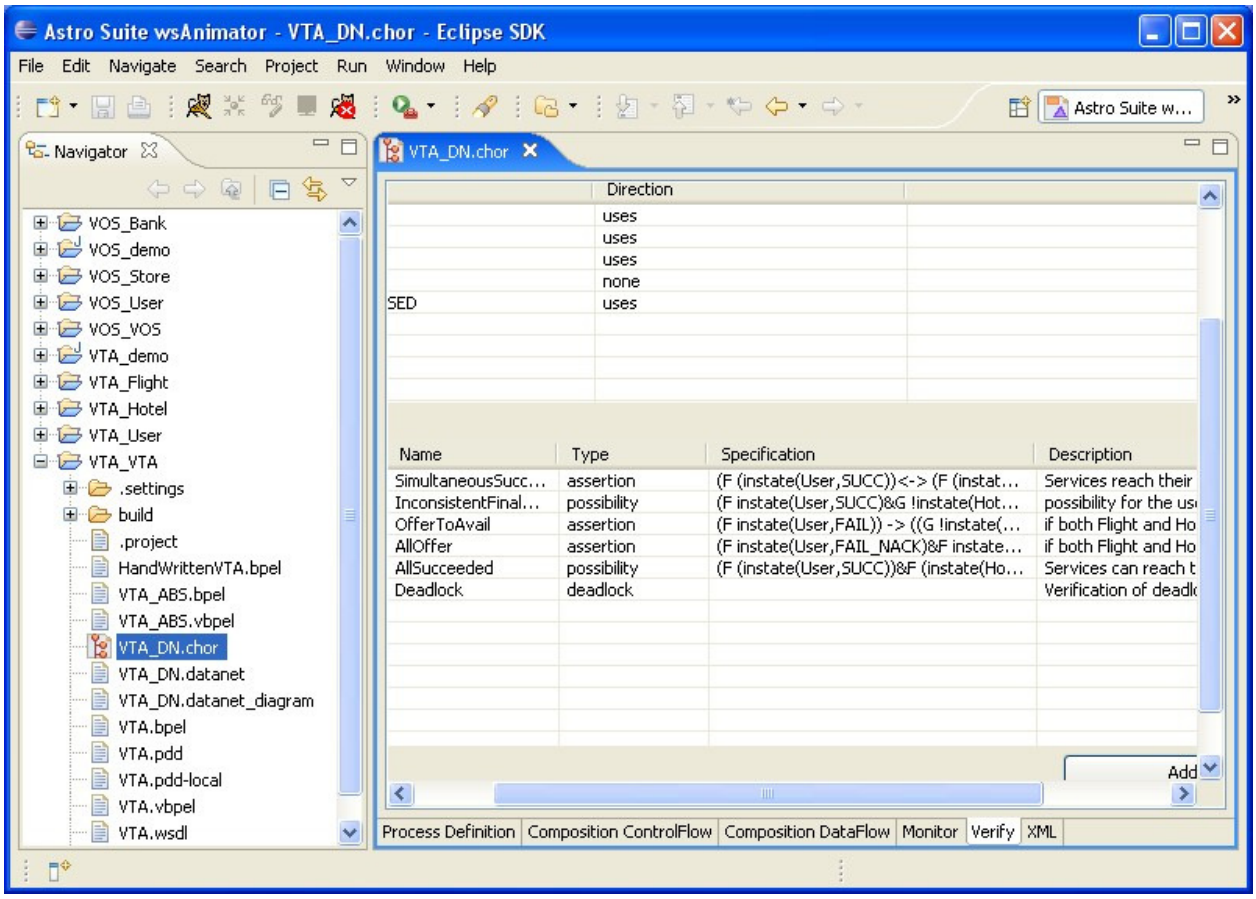
Nella finestra del ChainManager possiamo distinguere la checklist delle operazioni in alto, e la message box in basso. L'output finale della composizione è il file Concrete BPEL eseguibile su ActiveBPEL Engine, che viene posto nel folder del progetto con nome xxx.bpel; inoltre il chainManager effettua automaticamente il packaging del file, con gli annessi necessari, nel formato .bpr deployable, ed esegue il deployment su Engine (il che onsiste semplicemente in una copia del file .bpr nel folder /bpr di Tomcat, che poi rileva automaticamente il processo); nella figura seguente è mostrata l'interfaccia offerta dal BPEL Engine (ed estesa da wsMonitor) che mostra l'avvenuto deployment del processo composito VTA. La pagina è accessibile via browser tramite una URL del tipo "http://<computer-host-name>:50000/BpelAdminExt/"; nel nostro caso: "http://localhost:50000/BpelAdminExt/".



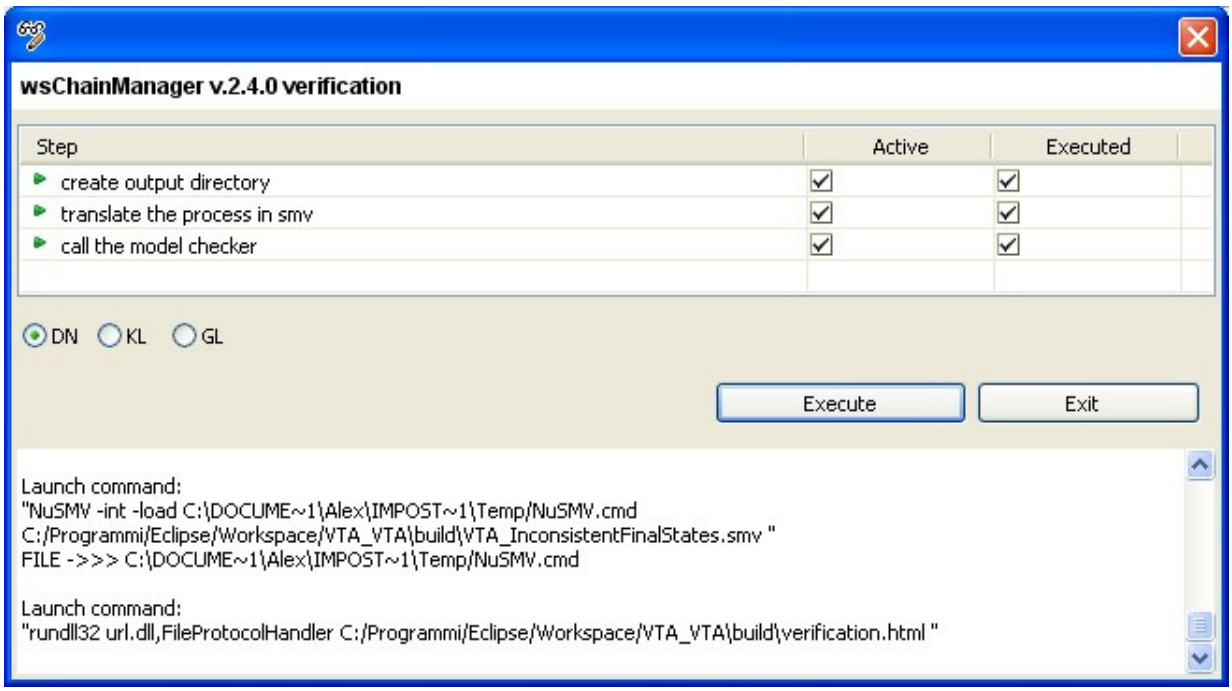
Sezione 5.3 - Process Offline Verification

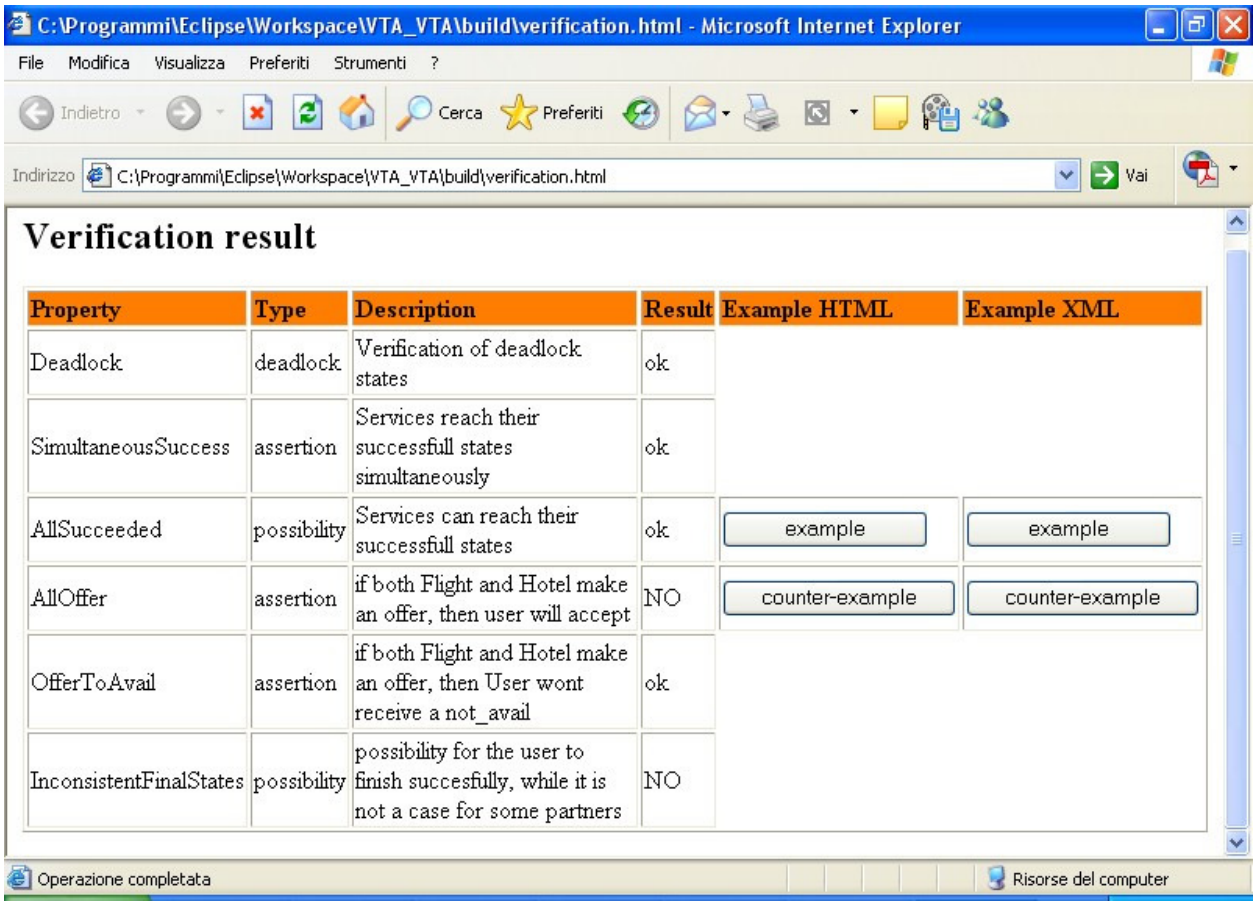
L'Astro Suite offre funzionalità per verificare proprietà del modello costruito (ovver il file .chor) semplicemente a partire dal file di coreografia.
Tali proprietà da verificare seguono la linea dell'uso di metodi formali nelle fasi di design del software, e possono includere esempi di success scenarios, ricerche di deadlocks, assertions da confutare riguardo a cosa può e non può succedere sotto specifiche ipotesi e molto altro.

In figura è riportato uno scorcio dalla schermata di ispezione del file .chor, sezione "Verify" di wsRequirements, per illustrare le proprietà che erano di interesse nella verifica.

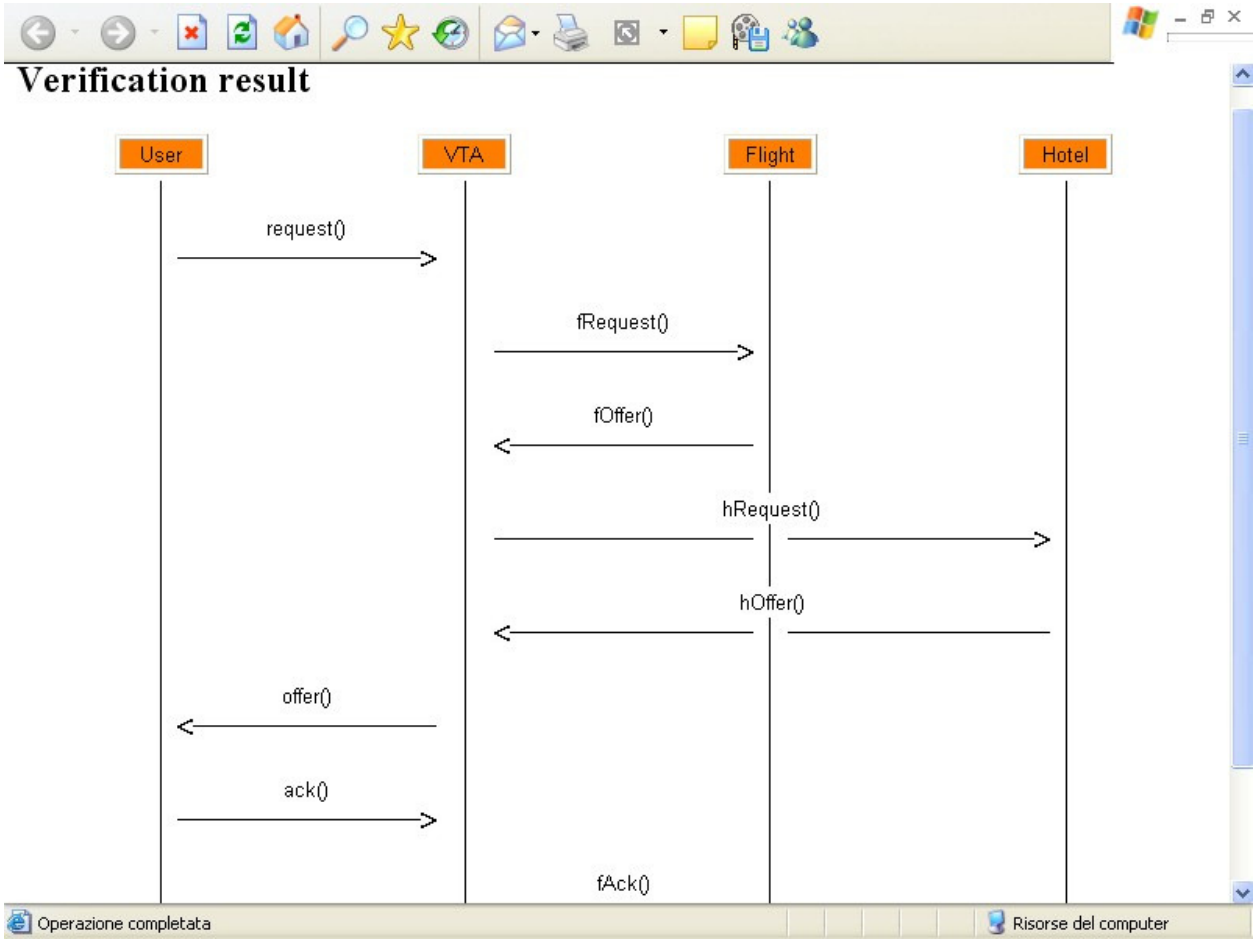


Trattandosi di una procedura offline, possiamo avviare l'analisi tramite ChainManager e ricevere risposta immediatamente, tramite una schermata Web apposita:



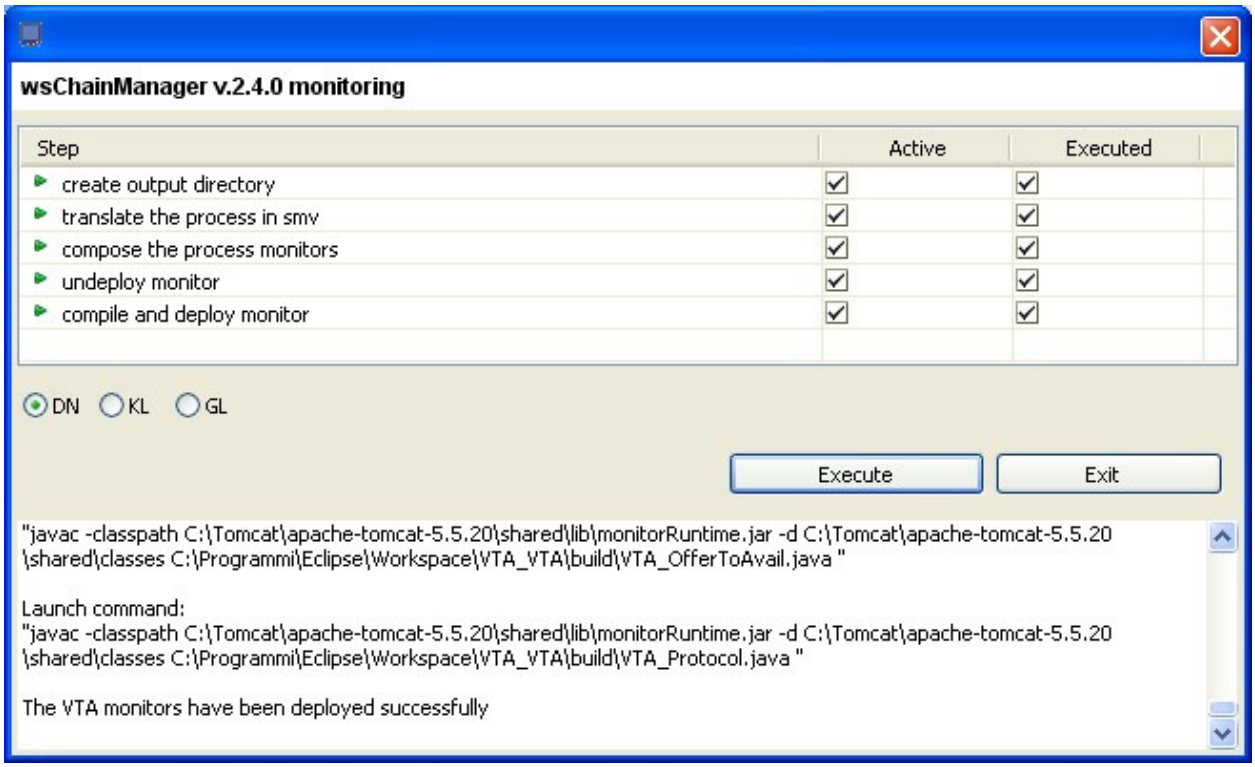


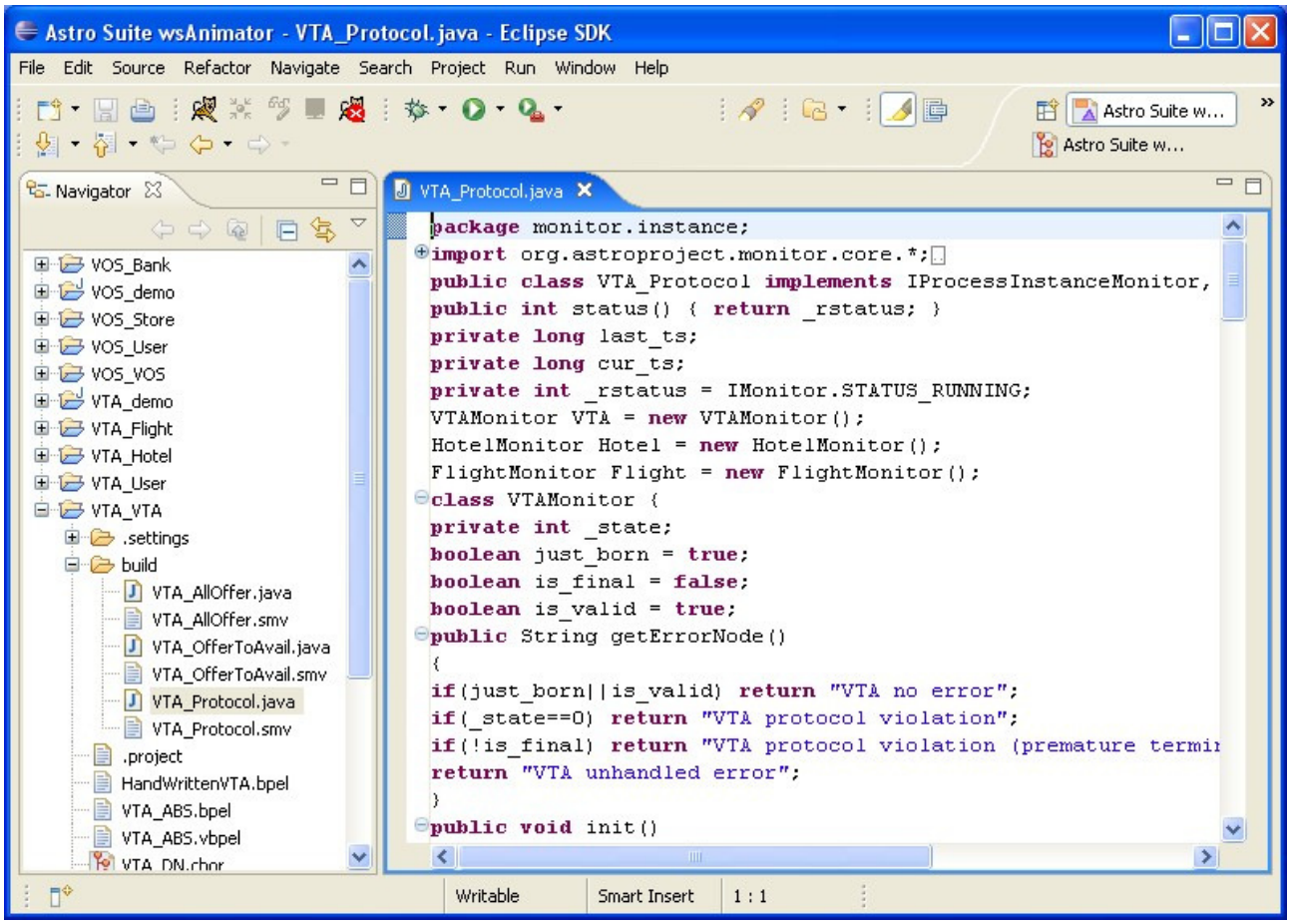
Selezionando esempi e controesempi forniti, ci viene mostrato uno scenario ad alto livello stile UML Sequence Diagram:

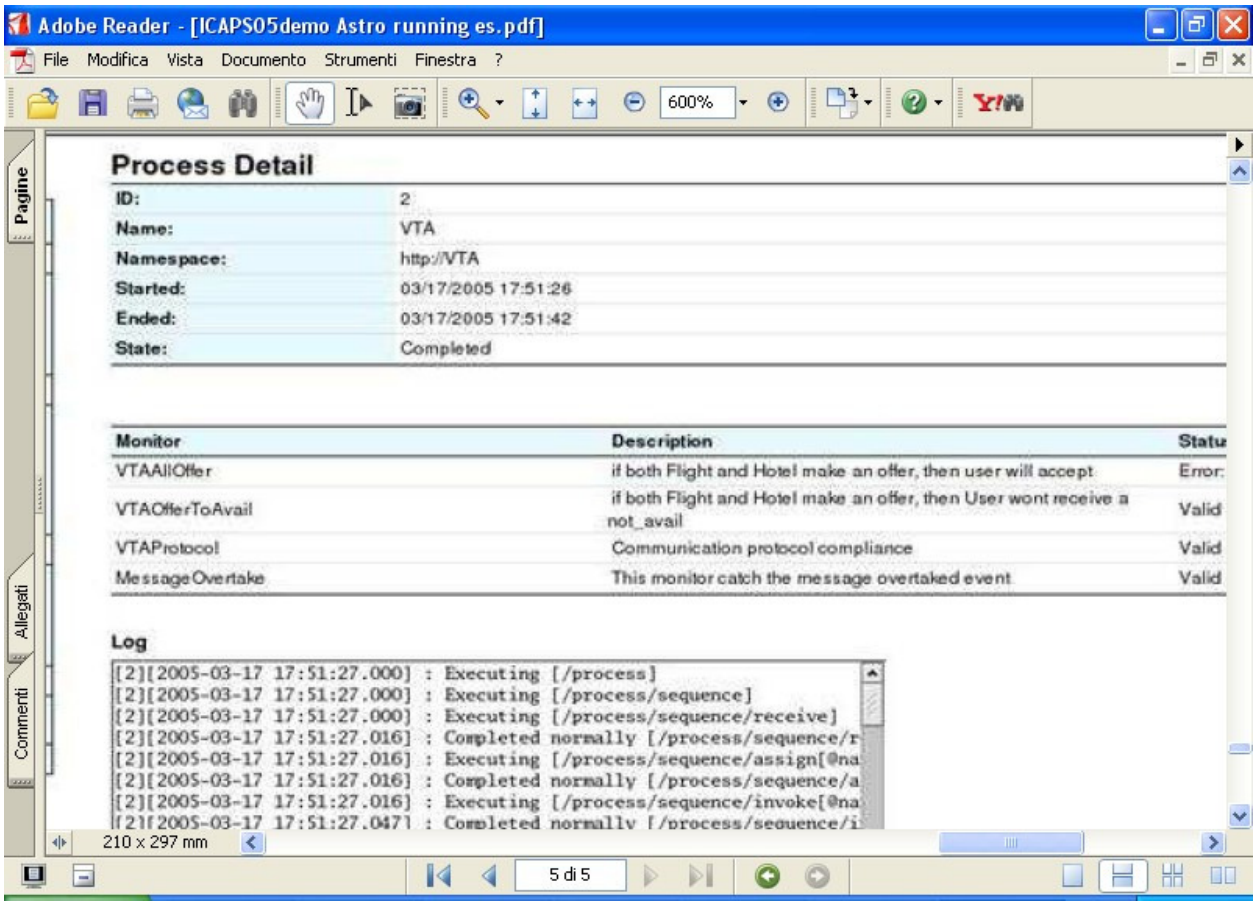


Sezione 5.4 - Online Process Monitoring

La funzionalità di Monitoring è il corrispettivo online della Verification, e vuole fornire rapporti all'utente su stati anormali dell'esecuzioni di processi BPEL while running.
Tramite la consueta schermata wsChainManager vengono creati automaticamente dei files Java (i cui obiettivi sono definiti ancora una volta nel file .chor) i quali vengono messi in ascolto su esecuzioni del processo d'interesse per monitorarlo e fornire informazioni all'utente; dal momento che utilizzano l'interfaccia BPEL per fornire tali rapporti, e poiché l'ActiveBPEL Engine è estraneo alla creazione ed esecuzione di questi monitors, si è resa necessaria una espansione all'interfaccia dell'Engine per includere supporto ai programmi monitors (questo è lo scopo dell'applicazione wsMonitor).
Nel seguito sono riportate immagini del wsChainManager per il Monitoring, di uno dei Java monitor files generati, e dei monitor in esecuzione.

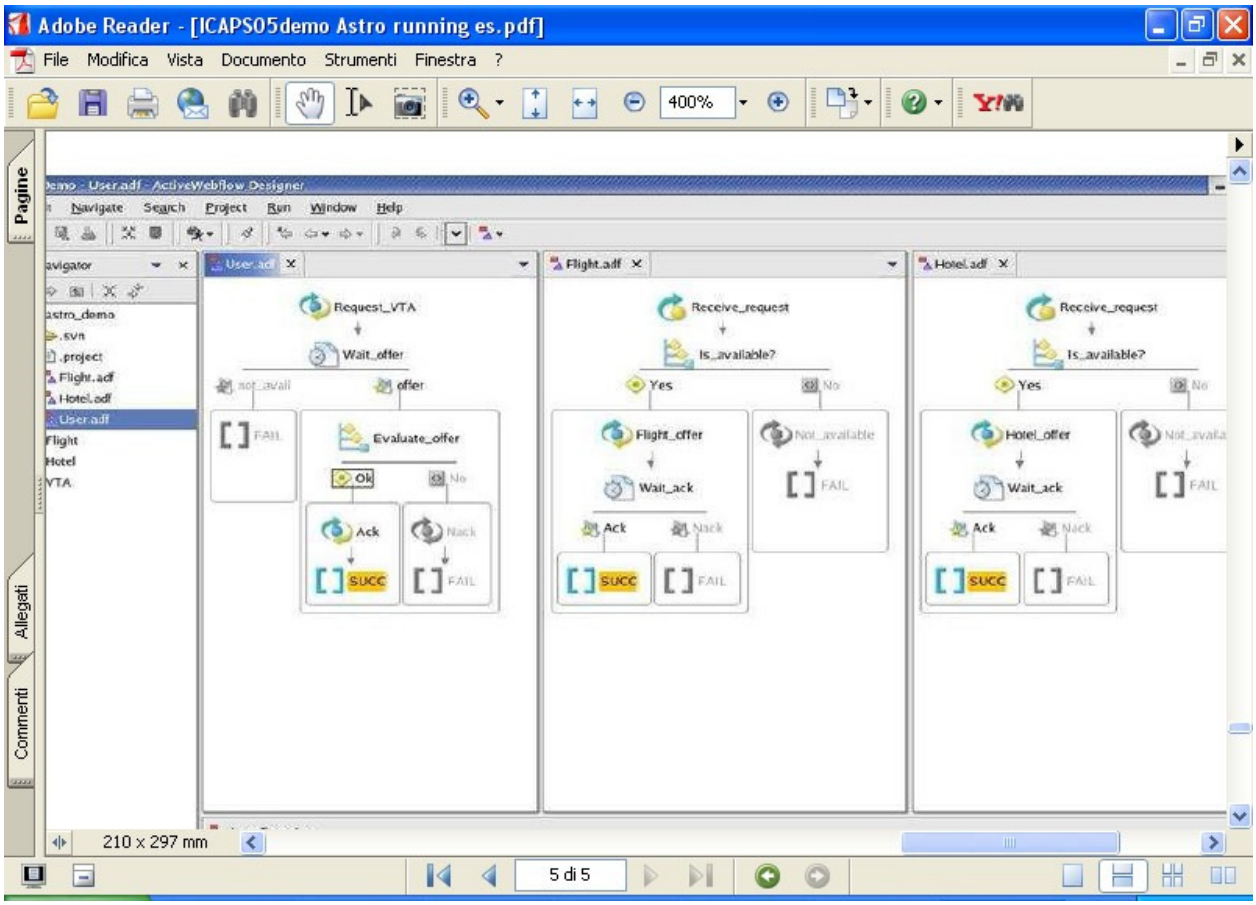

wsChainManager v.2.4.0 monitoring
Step
Active
Executed
create output directory
translate the process in smv
compose the process monitors
undeploy monitor
compile and deploy monitor
DN
KL
GL
Execute
Exit
"javac -classpath C:\Tomcat\apache-tomcat-5.5.20\shared\lib\monitorRuntime.jar -d C:\Tomcat\apache-tomcat-5.5.20
\shared\classes C:\Programmi\Eclipse\Workspace\VTA_VTA\build\VTA_OfferToAvail.java "
Launch command:
"javac -classpath C:\Tomcat\apache-tomcat-5.5.20\shared\lib\monitorRuntime.jar -d C:\Tomcat\apache-tomcat-5.5.20
\shared\classes C:\Programmi\Eclipse\Workspace\VTA_VTA\build\VTA_Protocol.java "
The VTA monitors have been deployed successfully


Astro Suite wsAnimator - VTA_Protocol.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Navigator
VTA_Protocol.java
VOS_Bank
VOS_demo
VOS_Store
VOS_User
VOS_VOS
VTA_demo
VTA_Flight
VTA_Hotel
VTA_User
VTA_VTA
.settings
build
VTA_AllOffer.java
VTA_AllOffer.smv
VTA_OfferToAvail.java
VTA_OfferToAvail.smv
VTA_Protocol.java
VTA_Protocol.smv
.project
HandWrittenVTA.bpel
VTA_ABS.bpel
VTA_ABS.vbpel
package monitor.instance;
import org.astroproject.monitor.core.*;
public class VTA_Protocol implements IProcessInstanceMonitor,
public int status() { return _rstatus; }
private long last_ts;
private long cur_ts;
private int _rstatus = IMonitor.STATUS_RUNNING;
VTAMonitor VTA = new VTAMonitor();
HotelMonitor Hotel = new HotelMonitor();
FlightMonitor Flight = new FlightMonitor();
class VTAMonitor {
private int _state;
boolean just_born = true;
boolean is_final = false;
boolean is_valid = true;
public String getErrorNode()
{
if(just_born||is_valid) return "VTA no error";
if(_state==0) return "VTA protocol violation";
if(!is_final) return "VTA protocol violation (premature termi
return "VTA unhandled error";
}
public void init()
Writable
Smart Insert
1 : 1



Sezione 5.5 - Process Execution Simulation

Mediante la plugin wsAnimator, ed utilizzando un formato file partcolare .adf, è possibile mandare in esecuzione diversi scenari pre-programmati, simile a scenari di simulazione in ActiveBPEL Designer ma customizzati da team Astro per Composite Services. In figura è riportato un esempio di esecuzione andata a buon fine, in cui tutti i processi hanno terminato su "Success".


Adobe Reader - [ICAPS05demo Astro running es.pdf]
File Modifica Vista Documento Strumenti Finestra ?
400%
Pagine
Allegati
Commenti
Navigate Search Project Run Window Help
User.adf
Flight.adf
Hotel.adf
Request_VTA
Wait_offer
offer
Evaluate_offer
Ok
Ack
Nack
SUCC
FAIL
Receive_request
Is_available?
Yes
Flight_offer
Wait_ack
Hotel_offer
210 x 297 mm
5 di 5

Conclusioni

In questa relazione abbiamo esplorato a fondo il lavoro svolto nell'ambito del Progetto Astro per una ricerca di un approccio per Automated Web Service composition, mettendo in risalto tanto la logica teorica su cui si fonda l'approccio, tanto gli ambiziosi e importanti risultati pratici conseguiti dal team Astro.
Abbiamo inoltre ricordato i punti salienti dell'approccio del Roman Group, legato al medesimo problema, i cui autori sono i docenti universitari responsabili del Corso di Seminari di Ingegneria del Software in seno a cui nasce il presente lavoro, e abbiamo comparato i due approcci studiati, mettendo in risalto analogie e differenze, esaminando le metodologie ed i loro differenti punti di vista anche nel panorama della ricerca internazionale odierna.

Bibliografia

- Alonso, Casati, Kuno, Machiraju; 2004; "Web Services - Concepts, Architectures and Applications"
- G.De Giacomo, M.Mecella; Slides dal corso di Seminari di Ingegneria del Software 2006-07
- **[ICAPS05]** Pistore, Traverso, Bertoli; 2005; "Automated Composition of Web Services by Planning in Asynchronous Domains"
- **[ICAPS05demo]** Trainotti, Pistore et al.; 2005; "ASTRO: Supporting Composition and Execution of Web Services"
- **[IJCIS05]** Berardi, Calvanese, De Giacomo, Lenzerini, Mecella; 2005; "Automatic Service Composition based on Behavioral Descriptions"
- **[AISC06]** Berardi, De Giacomo, Mecella, Calvanese; 2006; "Automatic Web Service Composition: Service-Tailored vs Client-Tailored Approaches"
- **[ICSOC05]** Berardi, Calvanese, De Giacomo, Mecella; 2005; "Composition of Services with Nondeterministic Observable Behavior"
- **[PT01]** Pistore, Traverso; 2001; "Planning as Model-Checking for Extended Goals in Non-Deterministic Domains"
- **[DPT02]** Dal Lago, Pistore, Traverso; 2002; "Planning with a Language for Extended Goals"
- **[P_BER]** Berardi et al.; "Automatic Composition of e-Service that export their Behavior"
- The ASTRO Project Website; http://www.astroproject.org
- **[BPEL_Spec]** The BPEL4WS Specification, v1.1; http://dev2dev.bea.com/technologies/webservices/BPEL4WS.jsp
- The ActiveBPEL Engine, http://www.activebpel.org
- The ActiveBPEL Designer v4 User's Guide (http://www.active-endpoints.com)