Università degli Studi di Roma "La Sapienza"

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Composizione automatica di servizi: l'approccio ASTRO e il Roman Model a confronto

Corso di "Seminari di Ingegneria Del Software"

Prof. Giuseppe De Giacomo

Autore

Alessandro Dionisi

Anno Accademico 2005/2006

INDICE

1 Introduzione 2

1.1 Esempio di uno scenario di composizione 3

2 Rappresentazione dei servizi 4

2.1 Non-determinismo e osservabilità 6

3 Specifica dei requisiti 7

3.1 Requisiti di composizione in ASTRO 8

3.2 Confronto con il Roman Model 9

4 Il processo di sintesi automatica 10

4.1 Formalizzazione del problema di composizione in ASTRO 12

4.1.1 La corrispondenza composizione/pianificazione 18

4.2 Confronto con il Roman Model 20

5 Conclusioni 22

6 Riferimenti 23

1 INTRODUZIONE

L'ampia diffusione di applicazioni service-oriented e in particolare dei web services, all'interno di numerose organizzazioni, ha introdotto nel campo della ricerca tematiche interessanti su come poter sfruttare i componenti di business già esistenti, in modo di poter costruire a partire da questi e in modo automatico, nuovi servizi composti di valore aggiunto per il cliente. La composizione di servizi viene attualmente affrontata in modo manuale; il cliente specifica i suoi requisiti e il progettista si occupa di concepire un nuovo processo di business che invochi adeguatamente le componenti applicative esistenti. Come si può ben intuire, tale compito si presenta abbastanza laborioso e non privo di difficoltà. Ciò a cui si vuole arrivare al contrario, è un procedimento efficiente, affidabile e di facile uso che permetta, a partire da specifici requisiti, di comporre in modo automatico web service e in generale frammenti di applicazioni.

Il mio lavoro ha come obiettivo principale, il confronto di due differenti approcci a tale problema; il Roman Model **[ASC05]**, ideato da alcuni professori e ricercatori nostro dipartimento, e il progetto ASTRO **[ACWP05]** concepito da un differente gruppo di ricerca dell'Università di Trento, in collaborazione con la divisione SRA dell'ITC-IRST (http://sra.itc.it/). Sebbene i due orientamenti abbiano caratteristiche parzialmente somiglianti tra loro, rispetto alle specifiche di un ben determinato problema, essi possono avere comportamenti anche molto dissimili nella soluzione. Sostanzialmente è difficile trovare casi in cui entrambe le tecniche forniscono i medesimi risultati; addirittura potrebbero esservi casi risolvibili in un approccio e non nell'altro e viceversa. Il confronto sarà effettuato comparando come i differenti approcci affrontano le diverse "fasi" del processo di composizione automatica dei servizi, dalla specifica dei requisiti, al processo finale di sintesi automatica. Esso sarà focalizzato soprattutto sull'approccio ASTRO, dato che il Roman Model è stato ampiamente illustrato durante il corso di Seminari di Ingegneria del Software tenuto dal Prof. De Giacomo durante l'A.A. 2005/2006.

1.1 Esempio di uno scenario di composizione

Per tutto il seguito della relazione, verrà preso in considerazione come esempio di riferimento quello riportato in **[ACWP05]** a cui si rimanda per ulteriori dettagli. In tale esempio si vuole fornire un servizio di acquisto e distribuzione di prodotti *Purchase & Ship (W)*, a partire dai web services esistenti (per consistenza di notazione tutta la terminologia viene conservata come nell'articolo di riferimento) *Producer (W1)*, *Shipper (W2)* e *User (W3)*.

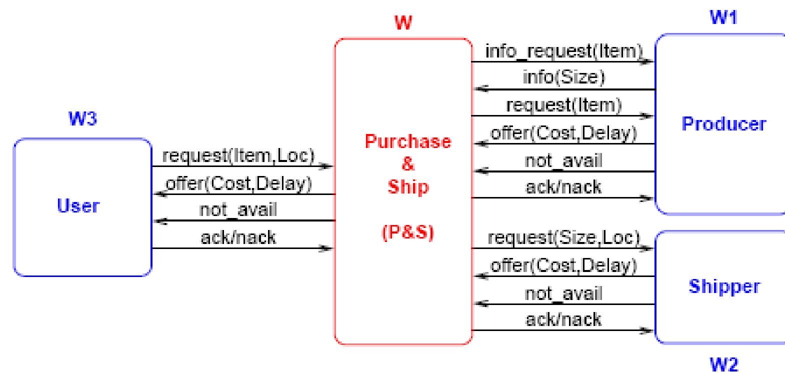

Figura 1. *L'esempio di composizione di riferimento*

Come detto precedentemente, molto probabilmente sono pochissimi i problemi risolvibili allo stesso tempo da entrambe le tecniche; tale esempio si riferisce pertanto solo all'approccio ASTRO. Per un esempio relativo al Roman Model consultare l'articolo **[ASC05]** e **[TSSC06]**.

2 RAPPRESENTAZIONE DEI SERVIZI

Il punto di partenza per un sistema di composizione automatica sono sicuramente i servizi esistenti; possiamo immaginare un progettista che abbia a sua disposizione un vasto repository di web services pubblicati (ad es. un registro UDDI di un certo ambito applicativo) e utilizzando frammenti di essi e combinando opportunamente le funzionalità offerte, riesce a soddisfare determinati requisiti. Un aspetto molto importante nella rappresentazione di tali entità, al fine di poter costruire algoritmi di composizione automatica, è individuare, oltre all'interfaccia funzionale esposta per l'invocazione, anche delle descrizioni comportamentali, come ad esempio l'insieme delle possibili conversazioni che il servizio può supportare. Esso viene visto come un programma interattivo in cui il client sceglie ogni volta la prossima azione da eseguire; ovviamente tale scelta deriva dal risultato delle azioni precedenti, ma la logica fondamentale dipende esclusivamente dal client.

Sia nell'approccio ASTRO che nel Roman Model, i web service componenti hanno la medesima rappresentazione; l'interfaccia pubblica di invocazione del servizio viene specificata con WSDL, mentre per codificare una descrizione "comportamentale" viene utilizzato un qualsiasi linguaggio per esprimere macchine a stati finiti. In ASTRO viene impiegato il linguaggio BPEL4WS **[BPELWS03]**, che permette di modellare ad alto livello le interazioni che si verificano in un singolo web service (per esempio invio e ricezione di messaggi). Nel Roman Model il comportamento viene modellato in WS-CDL **[WSCDL04]**; sempre basato su XML, esso permette di descrivere le interazioni peer-to-peer tra diversi web service, esponendo in primo luogo il comportamento osservabile da un punto di vista globale; con delle singole proiezioni è comunque ricavabile quello del singolo peer. Ovviamente tali linguaggi sono adatti a descrivere conversazioni puramente *asincrone*: ogni web service evolve indipendentemente e con velocità imprevedibile, sincronizzandosi con gli altri tramite scambio di messaggi ed evidentemente, in implementazioni reali, vengono impiegati dei buffer che consentono di non perdere i messaggi che non possono essere immediatamente processati.

Il passo successivo è quello di trasformare questa descrizione algoritmica del comportamento, in una più compatta; possiamo vedere infatti un processo BPEL4WS

o WS-CDL come un tipo specifico di macchina a stati finiti, in particolare uno *state transition system (STS)*. In generale infatti un web service può essere caratterizzato dalle operazioni (atomiche) che esso espone all'esterno, inserite opportunamente in particolari sequenze di esecuzione (conversazioni); negli STS le operazioni sono rappresentate dalle transizioni, mentre gli stati codificano le condizioni in cui i web service si trovano. Entrambe le rappresentazioni considerano solo STS con un numero finito di stati.

Definizione 1 (State transition system(STS)). *Uno state transition system Σ è una tupla* $\langle S, S^0, I, O, R, L \rangle$ *dove:*

- S *è l'insieme (finito) di stati*
- $S^0 \subseteq S$ *è l'insieme degli stati iniziali*
- I *è l'insieme finito di azioni di input*
- O *è l'insieme finito di azioni di output*
- $R \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ *è la relazione di transizione*
- $L : S \rightarrow 2^{prop}$ *è una funzione di etichettatura*
- $F \subseteq S$ *è l'insieme degli stati finali*

Sostanzialmente quindi, uno STS rappresenta il servizio come un sistema che può trovarsi in uno di diversi stati possibili (alcuni marcati come iniziali altri come finali in cui il servizio può terminare) e che può transitare in altri stati per mezzo di azioni. Tali azioni possono essere di input, output (invio e ricezione di messaggi da altri web services) o interne, ovvero il sistema evolve senza produrre output e indipendentemente dalla ricezione di input (τ-transizioni). La relazione di transizione spiega invece come passare da uno stato all'altro, al verificarsi delle azioni appena descritte. Infine, la funzione di etichettatura associa ad ogni stato l'insieme delle proprietà valide in quel determinato stato. Si assume inoltre l'ipotesi che non possano presentarsi cicli infiniti di τ-transizioni, in quanto rappresenterebbero un comportamento di un servizio che non sta interagendo con il resto del sistema, sostanzialmente una condizione di deadlock.

Come vedremo anche successivamente, la principale caratteristica che i due orientamenti hanno in comune è vedere i servizi componenti come un insieme di azioni atomiche anziché come entità "monolitiche"; sono tra i pochissimi approcci che, nella procedura composizione, vanno a smembrare i servizi combinando tra loro solo alcuni "frammenti", al fine di soddisfare il target dell'utente.

2.1 Non-determinismo e osservabilità

Per cogliere appieno il significato del non-determinismo possiamo pensare ad un web service per il pagamento con carta di credito; dopo aver invocato l'operazione il sistema può trovarsi indifferentemente nello stato `payment_OK` o `payment_FAILED`. Tale transizione non è completamente sotto il controllo né del client né dell'orchestratore; quando si effettua la composizione si debbono considerare le possibili scelte fatte non deterministicamente dai servizi componenti. Questo è possibile facendo in modo che, anche senza il pieno controllo dei servizi componenti, tuttavia l'orchestratore riesca ad ottenere piena osservabilità dello stato raggiunto, in quanto dopo l'esecuzione dell'azione esso può intuire quale transizione è stata correttamente eseguita.

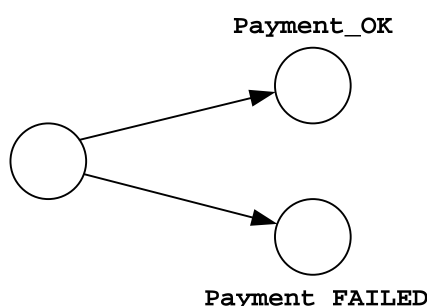

Figura 2. *Non-determinismo dei servizi*

Nel Roman Model, inizialmente, i servizi componenti vengono modellati con STS deterministici **[ASC05]**, successivamente in **[CSNOB05]** la sintesi automatica viene estesa al caso di STS non-deterministici. In realtà, si potrebbe avere uno STS non-deterministico anche per quanto concerne la richiesta del client; in tali casi tuttavia, gli autori parlano di non-determinismo *angelico*, in contrapposizione a quello *diabolico* che si verifica quando l'orchestratore non ha il pieno controllo dei

servizi disponibili.

3 SPECIFICA DEI REQUISITI

Prima di poter affrontare il processo di sintesi automatica del servizio composto, oltre alla descrizione comportamentale dei servizi esistenti, debbono essere definiti in maniera adeguata i requisiti del servizio che l'utente vuole realizzare. Da questo punto del processo i due approcci si distinguono sostanzialmente, in quanto, come vedremo successivamente, gli obbiettivi della composizione sono differenti; nel Roman Model, a partire da n servizi (descritti come STS) e dalla specifica di un web service che si desidera realizzare (anch'esso uno STS), se ne genera in modo automatico un n+1-esimo che adempie allo schema richiesto e in cui ogni azione eseguita è delegata in maniera adeguata ai web services componenti. Diversamente nel modello ASTRO si vuole che i servizi esistenti, incluso quello che rappresenta il comportamento del client, cooperino in maniera tale che siano soddisfatti determinati requisiti, espressi in una particolare logica. Nella seguente figura vengono mostrate, a grandi linee, le varie fasi eseguite dai due approcci nel processo

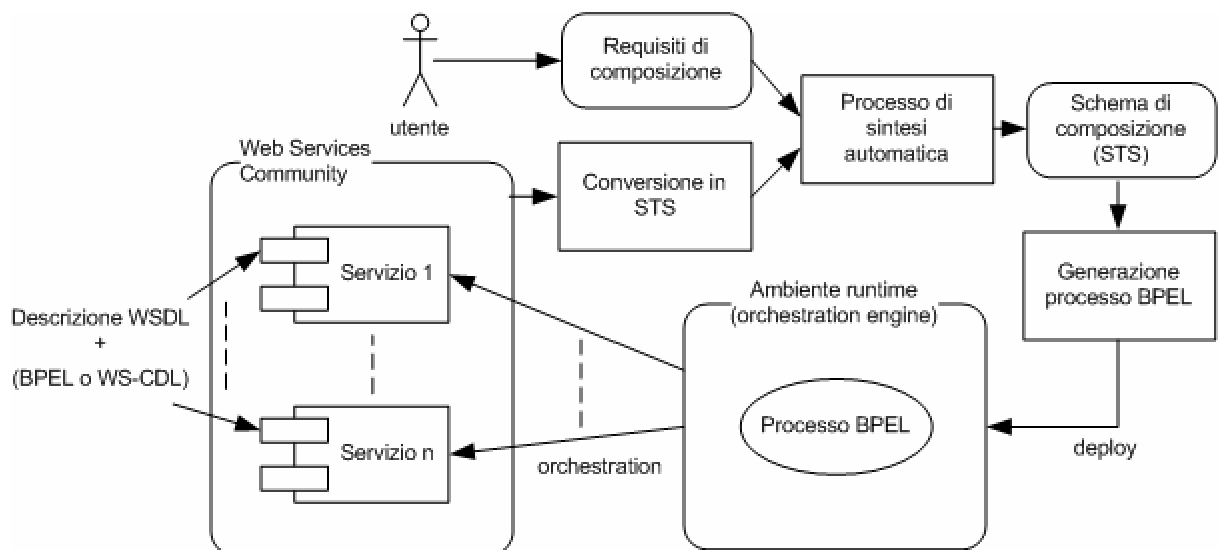

Figura 3. *Il processo di composizione automatica*

di composizione; successivamente si cercherà di spiegare in che modo esse si distinguono. In particolare il prossimo paragrafo sarà dedicato alla descrizione dei

requisiti Astro, mentre il successivo al confronto con il Roman Model.

3.1 Requisiti di composizione in ASTRO

La specifica dei requisiti in ASTRO viene effettuata tenendo conto che un web service deve essere modellato come un'entità con comportamenti non deterministici e comunque parzialmente osservabili; ciò è determinato dal fatto che la scelta dell'azione successiva da eseguire ogni volta viene fatta dal client e dal fatto che sono presenti τ-transizioni negli state transition systems, che consentono al servizio di evolvere autonomamente.

Come vedremo successivamente, la composizione viene modellata come un problema di pianificazione, diverso dai problemi di pianificazione classica. In quel caso infatti abbiamo piena conoscenza del mondo (piena osservabilità), le azioni sono deterministiche e i goal sono definiti per mezzo di semplici congiunzioni di proposizioni. Nel caso di nostro interesse i requisiti non possono essere limitati a obiettivi di raggiungibilità di stati, ma devono piuttosto essere espressi come *extended goals*, in grado di catturare condizioni esistenti sui percorsi dell'intero piano. I requisiti in ASTRO vengono descritti nel linguaggio EaGLe (descritto in **[PLEG02]**), che rispetto a logiche temporali come CTL e LTL permette di specificare un tipo di soddisfacibilità "best-effort"; è possibile indicare un obiettivo principale (main goal) e alcune condizioni che devono invece verificarsi in caso di fallimento (exception handling). Il linguaggio fornisce costrutti per esprimere condizioni che il sistema deve garantire di raggiungere o mantenere, o in alternativa effettuare solo un tentativo, prevedendo delle proprietà da soddisfare in caso di fallimento. Prendendo sempre come riferimento l'esempio riportato in **[ACWSP05],** i requisiti di composizione risultano come qualcosa del tipo:

try to "sell items at home";
upon failure,
do "never a single commit".

dove *sell items at home* significa che si desidera che il servizio P&S ha raggiunto la condizione in cui l'utente ha confermato il suo ordine e il servizio ha confermato i rispettivi ordini ai web services *Producer* e *Shipper*. Tuttavia in caso di fallimenti di qualsiasi tipo, il sistema non deve eseguire nessun *commit*, né sul *Producer* né sullo

Shipper. La formalizzazione in EaGLe delle condizioni precedenti viene espressa con la seguente formula ρ da soddisfare:

```
TryReach
user.pc=SUCC∧producer.pc=SUCC∧shipper.pc=SUCC∧
user.offer_delay=add_delay(producer.offer_delay,
shipper.offer_delay)∧user.offer_cost=add_cost(
producer.offer_cost)
Fail DoReach
user.pc=FAIL∧producer.pc=FAIL∧shipper.pc=FAIL
```

Come possiamo osservare, tramite questo tipo di formule è possibile specificare preferenze sui requisiti (ad esempio primari e secondari) e rispetto a CTL, garantiscono maggiore espressività.

3.2 Confronto con il Roman Model

Nel Roman Model viene largamente impiegato il concetto di *community ontology*, definita principalmente per mezzo di un insieme finito (alfabeto) di azioni provenienti dai servizi che ne fanno parte; esse sono ricavate considerando le operazioni descritte nel WSDL e astraendo dai parametri formali. Utilizzando quest'astrazione, tutti i servizi della comunità comprendono lo stesso significato per ogni singola azione dell'alfabeto e sono descritti tramite dei "mapping" sulla community ontology, come avviene nell'approccio LAV (Local-as-view).

Figura 4. *La community ontology*

Sostanzialmente, il servizio da comporre viene definito per mezzo di uno STS (finito) che specifica il comportamento che si vuole ottenere, utilizzando l'alfabeto di azioni dei servizi componenti. A titolo di esempio, possiamo pensare ad un progettista che, con uno strumento grafico, selezioni le varie operazioni messe a disposizione dalla comunità e disegni con un particolare grafo connesso, che andrà a rappresentare lo state transition system di ciò che si vuole realizzare.

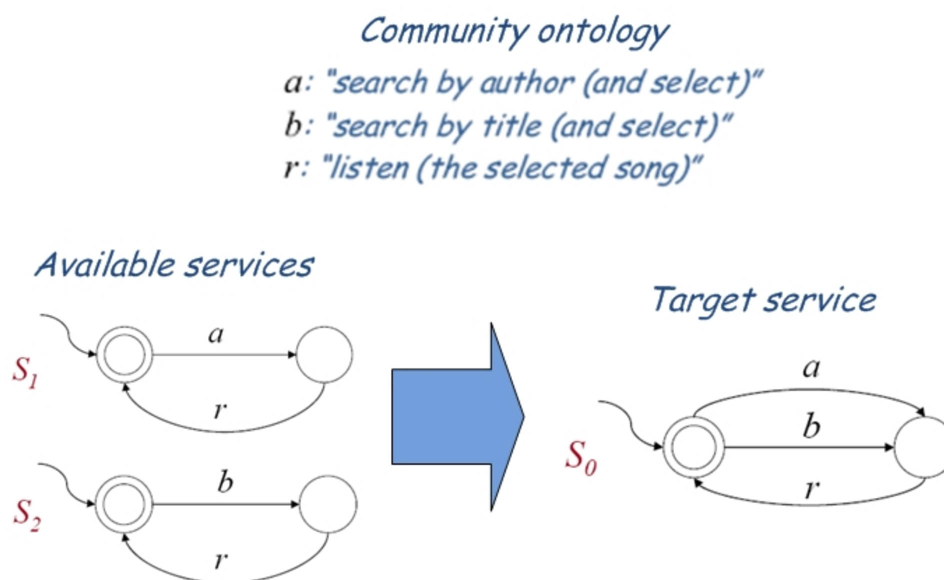

Figura 5. *Specifica del servizio composto*

In **[TSSC06],** è mostrato un esempio di questo procedimento: a partire dalle azioni messe a disposizione dai due servizi componenti S_1 e S_2, si vuole costruire il servizio S_0.

4 IL PROCESSO DI SINTESI AUTOMATICA

A questo punto del processo di composizione automatica, avendo a disposizione le rappresentazioni dei servizi esistenti e dei requisiti, possiamo constatare di possedere tutti gli input necessari per affrontare il problema. Benché i due approcci si differenzino molto nella soluzione di quest'ultimo, l'obiettivo finale è invece simile: sintetizzare un programma per l'orchestratore, in grado di schedulare le varie azioni dei servizi componenti e controllarne l'interleaving durante le fasi di esecuzione concorrente, al fine di soddisfare il target del client. Nel caso di ASTRO la

composizione è vista come un problema di pianificazione, in cui il risultato finale è un piano di esecuzione nel quale, a partire da alcune condizioni iniziali, esso specifica l'insieme di azioni da eseguire per raggiungere il goal. In questo tipo di approccio, una volta creato il piano e quindi il relativo processo BPEL, il client non ha più influenza sul flusso di esecuzione, che può essere visto quindi come un processo "batch". Ciò che deve accadere nel caso di eventuali variazioni durante l'esecuzione è stabilito a priori, specificando, come visto prima, una formula logica che contenga un main goal ed alcuni task secondari (come ad esempio eccezioni). Contrariamente, nel Roman Model, il servizio composto è a tutti gli effetti un nuovo servizio che riceve in tempo reale le richieste da parte del client, che in questo caso ha il pieno controllo del flusso di esecuzione; ovviamente ognuna di queste richieste deve essere delegata correttamente al web service componente idoneo a soddisfarla.

Nel Roman Model per coordinare l'orchestratore si cerca di soddisfare una formula logica, espressa in DPDL (una logica per verificare proprietà di programmi), in cui sono codificati i servizi esistenti, il target ed altre condizioni aggiuntive; se tale formula è soddisfatta, si ottiene in output, un particolare STS "etichettato", che contiene al suo interno le informazioni fondamentali per l'orchestratore; l'insieme delle azioni che devono essere eseguite e il rispettivo servizio responsabile dell'azione.

Diversamente, in ASTRO viene si scelto di sfruttare tecniche di pianificazione basate sull'approccio "Planning as Model Checking" **[PMCEG01]**, concepite per poter lavorare anche in domini non-deterministici, con condizioni di parziale osservabilità ed "extended goals". Una volta definito il dominio $\mathcal{D}$ (come visto precedentemente per mezzo di state transition systems) e il goal ρ (formula espressa in logica EaGLe), il pianificatore genera un piano π da cui in seguito si ricava lo STS del *mediatore* (una forma di orchestratore), che si occuperà di inviare/ricevere le invocazioni riguardanti i servizi componenti.

In entrambe le tecniche l'ultima fase prevede la traduzione dell'orchestratore, espresso nei modelli compatti sopra descritti, in un processo BPEL4WS concreto che può essere eseguito su un ambiente runtime, come ad esempio Active BPEL.

Nei paragrafi successivi si cercherà di esplorare in profondità il problema di composizione, analizzandolo da un punto di vista più formale.

4.1 Formalizzazione del problema di composizione in ASTRO

Una volta trasformati i web services componenti $W_1 \dots W_n$ rispettivamente negli STS $\Sigma_{W_1} \dots \Sigma_{W_n}$, viene costruito un STS che codifica il prodotto parallelo $\Sigma_{\|}$, il quale rappresenta tutti i possibili comportamenti ed evoluzioni concorrenti dei singoli servizi W_i, senza nessun controllo o interazione con il servizio che sarà generato W. Inoltre $\Sigma_{\|}$, nel problema di pianificazione, andrà a costituire il dominio $\mathcal{D}$ su cui il planner costruirà il piano per raggiungere il goal ρ. Introdotti tali concetti è possibile osservare, in figura 6, una visione d'insieme del processo di composizione ASTRO.

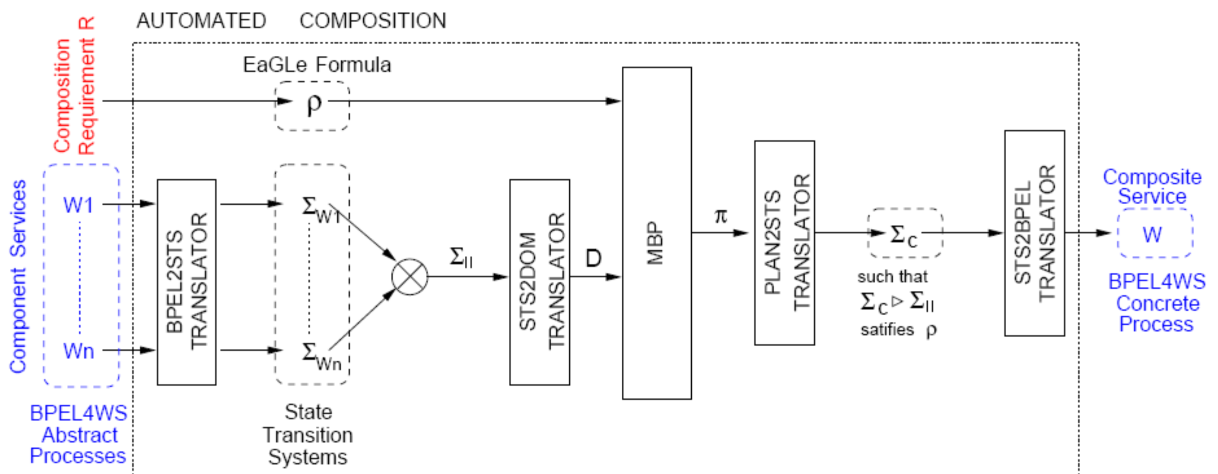

***Figura 6**.Overview del processo di composizione ASTRO*

I moduli STS2DOM TRANSLATOR, PLAN2STS TRANSLATOR e STS2BPEL TRANSLATOR si occupano ovviamente di eseguire le dovute conversioni di rappresentazione. Definiamo ora in maniera formale il prodotto parallelo tra due STSs, Σ_1 e Σ_2.

Definizione 2 (Prodotto parallelo). *Siano* $\Sigma_1 = \langle S_1, S_1^0, I_1, O_1, R_1, L_1 \rangle$ *e* $\Sigma_2 = \langle S_2, S_2^0, I_2, O_2, R_2, L_2 \rangle$ *due STS in cui vale la condizione* $(I_1 \cup O_1) \cap (I_2 \cup O_2) = \varnothing$. *Il*

prodotto parallelo $\Sigma_1 \| \Sigma_2$ *tra* Σ_1 *e* Σ_2 *è definito come:*

$$\Sigma_1 \| \Sigma_2 = \left\langle S_1 \times S_2, S_1^0 \times S_2^0, I_1 \cup I_2, O_1 \cup O_2, R_1 \| R_2, L_1 \| L_2 \right\rangle$$

dove:

$$\left\langle (s_1, s_2), a, (s_1^{'}, s_2) \right\rangle \in (R_1 \| R_2) \text{ se } \left\langle s_1, a, s_1^{'} \right\rangle \in R_1;$$

$$\left\langle (s_1, s_2), a, (s_1, s_2^{'}) \right\rangle \in (R_1 \| R_2) \text{ se } \left\langle s_2, a, s_2^{'} \right\rangle \in R_2;$$

e

$$\left(L_1 \| L_2 \right)\left(s_1, s_2 \right) = L_1(s_1) \cup L_2(s_2).$$

Di conseguenza $\Sigma_{\|}$ è definito come $\Sigma_{W_1} \| \ldots \| \Sigma_{W_n}$. Notare che quanto appena stabilito vale se e solo se gli input/output di Σ_1 e Σ_2 sono disgiunti; in caso contrario la definizione di prodotto parallelo deve essere adeguatamente modificata.

Il problema di composizione automatica consiste nel costruire un mediatore (chiamato nel seguito anche *controller*) Σ_c che riesca a controllare $\Sigma_{\|}$ assolvendo i requisiti espressi nella formula EaGLe ρ. Di seguito viene ora introdotta la definizione di un STS Σ_c che controlla ($\triangleright$) un altro STS Σ.

Definizione 3 (Sistema controllato). *Siano* $\Sigma = \left\langle S, S^0, I, O, R, L \right\rangle$ *e* $\Sigma_c = \left\langle S_c, S_c^0, I_c, O_c, R_c, L_\varnothing \right\rangle$ *due STS dove* $L_\varnothing(s_c) = \varnothing$ *per tutti gli* $s_c \in S_c$. *Lo STS* $\Sigma_c \triangleright \Sigma$ *che descrive il comportamento di* Σ *quando controllato da* Σ_c*, è definito come:*

$$\Sigma_c \triangleright \Sigma = \left\langle S_c \times S, S_c^0 \times S^0, I, O, R_c \triangleright R, L \right\rangle$$

dove:

$$\left\langle (s_c, s), \tau, (s_c^{'}, s) \right\rangle \in (R_c \triangleright R) \text{ se } \left\langle s_c, \tau, s_c^{'} \right\rangle \in R_c;$$

$$\left\langle (s_c, s), \tau, (s_c, s^{'}) \right\rangle \in (R_c \triangleright R) \text{ se } \left\langle s, \tau, s^{'} \right\rangle \in R;$$

$$\left\langle (s_c, s), a, (s_c^{'}, s^{'}) \right\rangle \in (R_c \triangleright R), \text{ con a} \neq \tau, \text{ se } \left\langle s_c, a, s_c^{'} \right\rangle \in R_c \text{ e } \left\langle s, a, s^{'} \right\rangle \in R.$$

In altre parole si richiede che l'input di Σ_c coincida con l'output di Σ e viceversa. Inoltre anche sei i sistemi sono connessi in questo modo, le transizioni in $\Sigma_c \triangleright \Sigma$ sono da considerarsi sempre di tipo input/output; ciò per riuscire a distinguere le τ-transizioni appartenenti a Σ_c o Σ da quelle derivanti dalla comunicazione tra Σ_c e Σ. Una intuizione grafica della relazione esistente tra il controller Σ_c e il prodotto parallelo di tutti i servizi componenti $\Sigma_{||}$, risulta quella in figura 6.

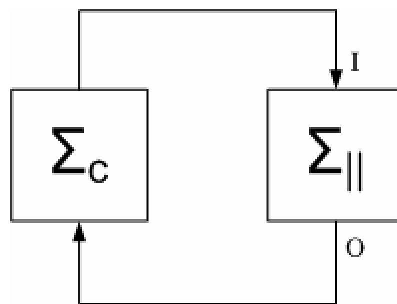

Figura 6. *Rappresentazione grafica della relazione* $\Sigma_c \triangleright \Sigma_{||}$

Ovviamente un STS Σ_c può non essere adeguato a controllare un altro sistema Σ. A dire il vero è necessario garantire che, per ogni transizione di output di Σ_c, allora Σ sia in grado di accettarla e viceversa. Nel seguito si definiranno le condizioni sotto il quale uno stato *s* di Σ è in grado di accettare un messaggio, sempre considerando un ambiente asincrono che astrae da eventuali code di messaggi. Si assume che uno stato *s* può accettare un messaggio *a* se esiste qualche successore *s'* di *s* in Σ, raggiungibile da *s* attraverso una catena di τ-transizioni, tale che *s* può eseguire una transizione di input etichettata con *a*. Viceversa se *s* non possiede tale successore *s'* e un messaggio di input è inviato a Σ, allora viene raggiunta una situazione di deadlock.

Nel seguito verrà utilizzato il concetto di τ-chiusura con τ-chiusura(s) denotiamo l'insieme degli stati raggiungibili da *s* attraverso una sequenza di τ-transizioni e con τ-chiusura(T) (dove $T \subseteq S$) l'unione di tutte le τ-chiusura(s) per tutti gli $s \in S$.

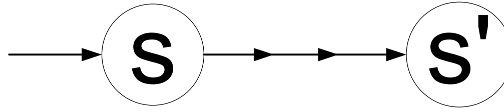

Figura 7. *Possibilità di evoluzione per uno stato*

Definizione 4 (Controller deadlock-free). *Siano* $\Sigma = \langle S, S^0, I, O, R, L \rangle$ *e* $\Sigma_c = \langle S_c, S_c^0, I_c, O_c, R_c, L_\varnothing \rangle$ *un controller per* Σ. Σ_c *è deadlock-free per* Σ *se tutti gli stati* $(s, s_c) \in S \times S_c$ *che sono raggiungibili dagli stati iniziali di* $\Sigma_c \triangleright \Sigma_{||}$ *soddisfano le seguenti condizioni:*

- *se* $\langle s, a, s' \rangle \in R$ *con* $a \in O$*, allora c'è qualche* $s_c^{'} \in \tau\text{-chiusura}(s_c)$ *tale che* $\langle s', a, s_c^{''} \rangle \in R_c$ *per qualche* $s_c^{''} \in S_c$*;*

e

- *se* $\langle s_c, a, s_c^{'} \rangle \in R_c$ *con* $a \in I$*, allora c'è qualche* $s' \in \tau\text{-chiusura}(s)$ *tale che* $\langle s', a, s'' \rangle \in R$ *per qualche* $s'' \in S$.

Per definire formalmente quando $\Sigma_c \triangleright \Sigma_{||}$ soddisfa *ρ*, occorre definire le *esecuzioni* di $\Sigma_c \triangleright \Sigma_{||}$. Per fare ciò bisogna tenere in conto che $\Sigma_{||}$ modella un dominio che è solo parzialmente osservabile da Σ_c, ovvero Σ_c non può conoscere quale esattamente sia lo stato di ogni servizio componente $\Sigma_{W_1} \ldots \Sigma_{W_n}$. Nell'esempio di riferimento possiamo individuare due forme di incertezza; la prima dovuta alle transizioni non-deterministiche (dall'esecuzione di `checkAvailable` possiamo passare indistintamente a `prepareOffer` o `prepareNotAvail`). La seconda dovuta alla natura prettamente asincrona dei web services non permette di determinare esattamente a che punto dell'esecuzione interna esso si trovi; ciò è possibile solo quando ci si scambiano dei messaggi, al completamento di transizioni di input/output. Nel definire le esecuzioni di $\Sigma_c \triangleright \Sigma_{||}$ si considera tale incertezza tenendo conto che, ad ogni passo di esecuzione, abbiamo un insieme di stati ugualmente plausibile in cui il sistema può trovarsi. Questo insieme di stati è

denominato *belief* (convinzione). La belief iniziale è l'insieme degli stati S^0 di Σ; tale insieme viene aggiornato ogni volta che Σ esegue una transizione osservabile dall'esterno (ad esempio di input/output). Formalmente possiamo definire l'evoluzione della belief nel seguente modo:

Definizione 5 (Evoluzione della belief). *Sia* $B \subseteq S$ *la belief di uno STS* $\Sigma = \langle S, S^0, I, O, R, L \rangle$ *che si trova nello stato s. Si definisce evoluzione di B dopo l'azione a, come la belief* $B' = Evolve(B,a)$*, dove:*

$$Evolve(B,a) = \{ s' : \exists s \in \tau\text{-chiusura(B)}. \langle s,a,s' \rangle \in R \}.$$

Ovvero $s' \in B'$ se, e solo se, esiste uno stato s raggiungibile da B eseguendo una sequenza di τ-transizioni, e tale che $\langle s,a,s' \rangle \in R$.

Per adempiere ad un determinato goal quindi, occorre verificare se la belief *B* ottenuta ad un certo passo di esecuzione soddisfa una certa proprietà di stato *p*. Nel nostro caso *B* soddisfa *p* se ogni $s \in B$ vale *p*.

Definizione 6 (Proprietà soddisfatta da una belief). *Sia* $\Sigma = \langle S, S^0, I, O, R, L \rangle$ *uno STS,* $p \in Prop$ *una proprietà per* Σ*, e* $B \subseteq S$ *una belief. Diciamo che B soddisfa* p $(B \models p)$ *se valgono le seguenti condizioni. Siano* $s_0, s_1 \ldots s_n$ *tali che* $s_0 \in B$ *,* $\langle s_i, \tau, s_{i+1} \rangle \in R$ *e o* s_n *non ha transizioni uscenti oppure esiste* s_{n+1} *tale che* $\langle s_n, a, s_{n+1} \rangle$ *con* $a \neq \tau$*. Allora* $p \in L(s_i)$ *per qualche* $0 \leq i \leq n$.

E' ora possibile definire lo STS che rappresenta le esecuzioni di $\Sigma_c \triangleright \Sigma_{||}$, e più in generale, di un STS Σ. Esso può essere chiamato STS "*belief-level*", in quanto i suoi stati sono belief e le sue transizioni descrivono le evoluzioni delle belief.

Definizione 7 (Sistema belief-level). *Sia* $\Sigma = \langle S, S^0, I, O, R, L \rangle$ *uno STS. Il corrispondente STS belief-level è* $\Sigma_B = \langle S_B, S_B^0, I, O, R_B, L_B \rangle$*, dove:*

- S_B *sono le belief di* Σ *raggiungibili dalla belief iniziale* S^o*;*
- $S_B^o = \{S^o\}$*;*
- *le transizioni* R_B *sono definite come segue: se* $Evolve(B,a) = B' \neq \emptyset$ *per qualche* $a \in I \cup O$*, allora* $\langle B,a,B' \rangle \in R_B$*;*
- $L_B(B) = \{p \in Prop : B \models_\Sigma p\}$.

E' opportuno notare che un STS belief-level è un tipo molto limitato di STS, in quanto possiede un solo stato iniziale, non sono presenti τ-transizioni e per tutte le belief *B* e azioni *a* esiste al massimo una belief *B'* tale che $\langle B,a,B' \rangle \in R_B$. Nel seguito, ogniqualvolta l'STS belief-level Σ_B soddisfa il goal ρ, si scriverà $\Sigma_B \models \rho$.

Il problema di composizione di web service può essere formalmente così definito:

Definizione 8 (Problema di composizione). *Siano* $\Sigma_1, \ldots \Sigma_n$ *un insieme di state transition system e* ρ *dei requisiti di composizione. Il problema di composizione per* $\Sigma_1, \ldots \Sigma_n$ *e* ρ *consiste nel trovare un controller* Σ_c *che sia deadlock-free e tale che* $\Sigma_B \models \rho$*, dove* Σ_B *è l'STS belief-level di* $\Sigma_c \triangleright (\Sigma_1, \ldots, \Sigma_n)$.

La definizione di quando il goal ρ è soddisfatto è fornita in termini di esecuzioni che $\Sigma_c \triangleright \Sigma_{||}$ può eseguire. Così, per esempio, se $\rho =$ `DoReach p`, con `p` una condizione sullo stato, allora è necessario controllare che tutte le esecuzioni di $\Sigma_c \triangleright \Sigma_{||}$ raggiungano prima o poi una configurazione che soddisfi la condizione `p`.

Una volta generato lo STS Σ_c, esso viene tradotto in un processo eseguibile BPEL dal modulo STS2BPEL TRANSLATOR di figura 5. Ovviamente tale conversione deve produrre un programma BPEL ben strutturato piuttosto che, ad esempio, uno basato su salti a etichette.

4.1.1 La corrispondenza composizione/pianificazione

Per poter sintetizzare lo STS Σ_c, come introdotto precedentemente, deve essere impostato un problema di pianificazione con extended goals, che tenga conto del non-determinismo. Come si osserva in figura 6, una volta costruito il prodotto parallelo dei servizi componenti, $\Sigma_{\|}$, esso viene trasformato in un dominio $\mathcal{D}$ adeguato per il Model Based Planner, un sistema progettato per eseguire sintesi di piani in domini non-deterministici (descritto in **[PMCEG01]** e **[PMC02]**). In particolare, per poter impiegare in modo più efficiente algoritmi basati sul symbolic model checking, il problema può essere tradotto in uno corrispondente, con piena osservabilità sugli stati; questo accade ragionando nello spazio delle belief, dato che esse vengono aggiornate ogni volta che il sistema effettua una transizione osservabile (ovvero di input o di output). Nella trasformazione, ogni stato del dominio di pianificazione corrisponde ad una intera belief dello STS di partenza $\Sigma_{\|}$, mentre le azioni di output vengono trasformate da transizioni in stati . Dato che, in generale, il dominio $\mathcal{D}$ viene costruito a partire di STS non-deterministici, il piano π che si ottiene in output dal MBP sarà un piano condizionale (sostanzialmente un programma con if e while); da esso si ottiene, tramite il modulo PLAN2STS TRANSLATOR, lo STS Σ_c che ricalca esattamente la struttura di esecuzione relativa a $\mathcal{D}$ e π (nel seguito Σ_c è indicato anche come Σ_π, per sottolineare che il controller è ricavato dal piano π). In questo caso, dato che si ritorna nel linguaggio degli STS, occorre descrivere adeguatamente anche le transizioni di output.

Se $\Sigma_\pi \models \rho$, si dice che π è un piano valido per ρ sullo STS $\Sigma_{\|}$. Il seguente lemma, indica la corrispondenza esistente tra il piano che viene sintetizzato e il relativo STS che ne viene ricavato successivamente.

Lemma 9 (Controller/piano eseguibile). *Sia $\Sigma_{\|}$ uno STS deadlock-free, Σ_B il corrispondente STS belief-level (Definizione 7) e il dominio $\mathcal{D}$. Sia π un piano per $\mathcal{D}$ e sia Σ_π il corrispondente STS. Se π è eseguibile su $\mathcal{D}$ allora Σ_π è eseguibile su $\Sigma_{\|}$.*

Occorre sottolineare che, in generale, il controller ottenuto dal piano potrebbe non

essere deadlock-free; tuttavia esistono delle condizioni su $\Sigma_{\|}$ che garantiscono, non solo che Σ_{π} sia deadlock-free, ma che Σ_{π} soddisfa i requisiti ρ su $\Sigma_{\|}$, ogni volta che π è soluzione del goal ρ sul dominio $\mathcal{D}$. Come indicato dagli autori, le condizioni che riescono a garantire questa proprietà prevedono che non esiste una belief in cui si possano verificare contemporaneamente input e output e che quando è il turno del controller, il dominio deve essere in grado di processare tutti i messaggi che esso può inviare. Questo corrisponde all'assunzione che, durante l'interazione con un web service, possiamo sapere se esso è in attesa di un'invocazione oppure sta inviando una risposta ad una precedente; nella maggior parte dei web service reali possiamo conoscere, in un determinato momento, quali sono le invocazioni eseguibili. Da ciò si può comprendere come tale approccio possa essere impiegato in una buona parte di scenari applicativi.

Viene di seguito mostrato il teorema che permette di affermare che la composizione automatica di servizi può essere impostata come un problema di planning in un dominio pienamente osservabile (nelle belief), specificando extended goals.

Teorema 10 (Equivalenza piano/controller). *Sia* $\Sigma_{\|}$ *uno STS deadlock-free,* Σ_{B} *il corrispondente STS belief-level (Definizione 7) e il dominio* $\mathcal{D}$ *. Se il piano* π *è una soluzione per il dominio* $\mathcal{D}$ *e per il goal* ρ , e Σ_{π} *è lo STS corrispondente a* π *, allora vale* $\Sigma_{\pi} \triangleright \Sigma_{\|} \models \rho$ *. Inoltre, se esiste qualche* Σ_{c} *tale che* $\Sigma_{c} \triangleright \Sigma_{\|} \models \rho$ *, allora esiste qualche* π *che è soluzione per il dominio* $\mathcal{D}$ *e goal* ρ (e quindi vale $\Sigma_{c} \triangleright \Sigma_{\|} \models \rho$ *).*

Da questo teorema si ricava che l'algoritmo utilizzato è sound e complete; se esiste una composizione dei servizi disponibili (quindi esiste un piano π su $\mathcal{D}$ per il goal ρ) allora viene restituito un controller tale che $\Sigma_{\pi} \triangleright \Sigma_{\|} \models \rho$. Viceversa, ogni volta che esso trova l'esistenza della composizione (e quindi un controller che soddisfa il goal) allora quella è la decisione corretta, dato che esiste un piano π su $\mathcal{D}$ che soddisfa ρ .

4.2 Confronto con il Roman Model

Prima di descrivere l'approccio scelto per affrontare la composizione automatica, conviene chiarire un aspetto importante nella modellazione dei servizi, ovvero la distinzione tra *external schema* (E^{ext}) ed *internal schema* (E^{int}) di un generico servizio E. In sintesi, un servizio può essere visto da due punti di vista differenti, quello del client e quello del sistema che lo renderà un processo eseguibile a tutti gli effetti. Al client interessa principalmente il comportamento del servizio, ovvero l'insieme delle azioni atomiche della community opportunamente combinate per

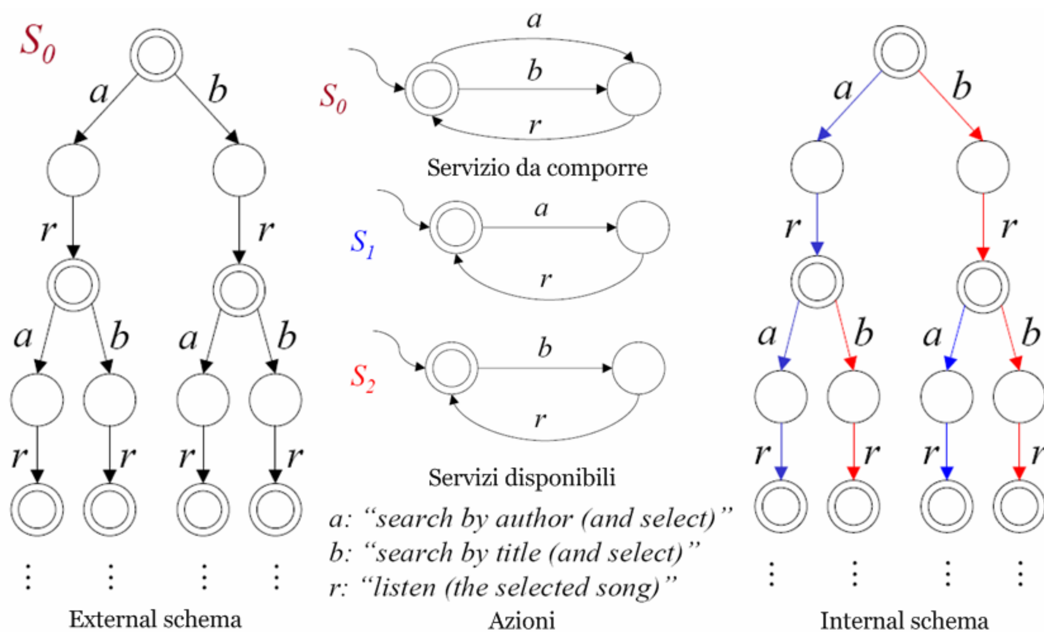

Figura 7. *External e internal schema del dell'esempio in* **[TSSC06]**

formare delle conversazioni; ciò caratterizza l'external schema. L'elemento di valore per il sistema invece, è conoscere a chi spetta la responsabilità di esecuzione di ogni azione, ovvero sapere a quale particolare servizio della community essa è stata delegata; tali informazioni sono contenute nell'internal schema. Tramite un'operazione di unfolding, uno STS può essere rappresentato tramite un *execution tree* infinito, ovvero uno STS bisimile[1] a quello originario, che rappresenta

---

[1] La bisimulazione **[TSSC06]** è una relazione di equivalenza tra STS in cui uno simula l'altro e viceversa. Due sistemi sono bisimili se è possibile creare una corrispondenza tra le azioni dei due sistemi, in modo tale che essi non possono essere distinti da un osservatore esterno.

l'evoluzione di tutte le possibili esecuzioni del web service. In figura precedente sono rappresentati gli execution tree del servizio S_0 da creare, rappresentanti l'external e l'internal schema. In particolare, osservando i colori (il blu associato a S_1, il rosso a S_2) dell'internal schema, si possono stabilire le responsabilità di esecuzione di ogni possibile azione del servizio composto.

Il problema di composizione può essere visto nel modo seguente: cercare un'etichettatura dell'execution tree del target service E in modo tale che ogni etichetta sulla singola azione specifichi in modo appropriato il servizio della community C delegato nell'esecuzione della stessa. In altre parole il problema può essere definito come la sintesi automatica di un internal schema E^{int} per E, dato l'external schema E^{ext}, che sia composizione di E rispetto a C. Una volta sintetizzato E^{int}, esso viene utilizzato dall'orchestration engine che si occupa di interagire con i servizi componenti, invocando di volta in volta le azioni adeguate.

Trovare l'esistenza della composizione, nel Roman model, diversamente da ASTRO significa ridursi ad un problema di soddisfacibilità di una formula Φ espressa in DPDL, in cui sono codificati i servizi esistenti, il target ed altre condizioni aggiuntive. Essa ha la seguente struttura:

$$\Phi = Init \wedge ([u]\Phi_0 \wedge_{i=1,\dots,n} [u]\Phi_i \wedge [u]\Phi_{aux})$$

dove *Init* indica gli stati iniziali di partenza, Φ_0 rappresenta il servizio da comporre E_0, $\Phi_{1\dots n}$ rappresentano i servizi componenti $E_{1\dots n}$ e Φ_{aux} identifica alcune condizioni ausiliarie (u indica un'abbreviazione per $(\cup_{a\in A} a)^*$, dove a sono le azioni, e $[u]\Phi$ indica che tutte le u-transizioni portano in uno stato in cui vale Φ). Le singole formule sono chiarite con maggiore dettaglio in **[ASC05]**.

Il risultato fondamentale di quanto detto precedentemente, si trova nel seguente teorema, che fornisce le condizioni necessarie e sufficienti per l'esistenza della composizione.

Teorema 11. *Sia Φ la formula DPDL che codifica il problema di composizione. Essa è soddisfacibile se e solo se esiste la composizione del servizio E_0 rispetto a $E_{1\dots n}$.*

Dato che la dimensione della formula Φ è polinomiale nella dimensione degli STS di E_0 ed $E_{1...n}$ e la soddisfacibilità in DPDL è EXPTIME-completo, dal teorema 11 segue che la composizione può essere risolta in EXPTIME. Come si può vedere più facilmente in **[TSSC06]**, gli unici termini che vengono specificati dopo la verifica di soddisfacibilità, sono proprio i predicati $moved_i$ che codificano i comandi con cui l'orchestratore coordina i servizi componenti.

Sfruttando la *tree model property*[2] si ottiene un albero infinito che rappresenta il modello della formula Φ; da esso viene estratto lo STS del servizio composto con un numero finito di stati (sfruttando questa volta la *small model property*[3]). Infine, in tempo polinomiale, è possibile minimizzare questo STS ottenuto, utilizzando algoritmi per la minimizzazione di MFSM.

5 CONCLUSIONI

In questa relazione sono state evidenziate le fasi principali del processo di composizione automatica di web service, considerando due diverse soluzioni al problema: l'approccio ASTRO e il Roman Model. In entrambi i servizi esistenti sono modellati state transition systems (finiti), costituiti da un insieme di azioni atomiche opportunamente mappate sulla community ontology, secondo una strategia LAV; essa è costruita con orientamento verso il client e in maniera indipendente dai servizi che mano a mano vengono aggiunti alla comunità. La richiesta di composizione viene fatta utilizzando l'alfabeto della community e il sistema cerca di realizzarla combinando adeguatamente frammenti (le azioni) dei servizi esistenti.

Tuttavia, l'obiettivo reale della composizione delle due tecniche è differente: in ASTRO il servizio del client viene inserito insieme a quelli esistenti cercando di sintetizzare un controller che sia in grado "pilotarlo" . Come si è visto, nel Roman Model invece, l'obiettivo è costruire un nuovo servizio, composto da un'opportuna configurazione di azioni della community. Per ottenere questi risultati, nel primo

---

[2] Ogni modello di una formula può essere rappresentato come un albero infinito con elementi del dominio come nodi e le funzioni parziali che rappresentano le transizioni come archi

[3] Ogni formula soddisfacibile ammette un modello finito, di dimensione al più esponenziale nella dimensione della formula stessa.

approccio viene impostato un problema di pianificazione con extended goals, espressi in logica EaGLe, in cui si sfruttano algoritmi basati su Symbolic Model Checking. Una volta sintetizzato il piano, esso dirige i servizi componenti tentando di raggiungere un goal primario e se ciò non accade, a causa di condizioni eccezionali, garantisce di soddisfare altri goal secondari. In questo modo il client non può influenzare il flusso di esecuzione, trovandosi a far parte di un vero e proprio processo batch.

Nel Roman Model invece, a partire dallo STS del servizio che si vuole ottenere (ottenuto mettendo insieme le azioni della community), si costruisce una formula in DPDL che se soddisfatta codifica in sé i comandi necessari per "orchestrare" il servizio (i predicati $moved_i$). In questo caso il client ha il pieno controllo sul flusso di esecuzione, dato che tale servizio può interagire direttamente con quest'ultimo. In entrambi i casi comunque, la composizione è basata su un modello di tipo *hub-and-spoke* **[ELBC05]**; possiamo infatti individuare la figura di un mediatore principale, attraverso cui transitano tutte le interazioni tra i servizi componenti.

In ambedue gli approcci, oltre alla rappresentazione e alla modellazione di web service con WSDL e BPEL4WS, in successive pubblicazioni la trattazione viene estesa alla composizione di *semantic web services*, descritti ad esempio in OWL-S o WSMO.

6 RIFERIMENTI

[ACWSP05] Bertoli P., Pistore M. and Traverso P. *Automated Composition of Web Services by Planning in Asynchronous Domains*. In Proc. of *ICAPS'05*.

[ACEEB03] Berardi D., Calvanese D.,De Giacomo G., Lenzerini M., and Mecella M. *Automated Composition of e-Services that Exports their Behavior*. In Proc. of *ICSOC'03*.

[ASC05] Berardi D., De Giacomo G. and Mecella M. *Automatic Service Composition based on Behavioral Descriptions*. In Proc. of *IJCIS'05*.

[AWCSCT06] Berardi D., Calvanese D.,De Giacomo G., Lenzerini M., and Mecella M. *Automatic Web Service Composition: Service-tailored vs. Client-tailored Approaches*. In Proc. of *AISC'06*.

[BPELWS03] Andrews T., Curbera F., Dolakia H., Goland J., Klein J., Leymann F., Liu K., Roller D., Smith D., Thatte S., Trickovic I. e Weeravarana S. *Business Process*

Execution Language for Web Services (ver. 1.1). *Microsoft & IBM*. 2003.

[CSNOB05] Berardi D., Calvanese D., De Giacomo G. and Mecella M. *Composition of Services with Nondeterministic Observable Behavior*. In Proc. of *ICSOC'05.*

[ELBC05] Benedikt M., Christophides V., Hull R. and Su J. *E-Services: A Look Behind the Curtain*. In Proc. of *PODS'03.*

[PLEG02] Dal Lago U., Pistore M. and Traverso P. *Planning with a Language for Extended Goals*. *ITC-IRST*. 2002.

[PMCEG01] Pistore M. and Traverso P. *Planning as Model Checking for Extended Goals in Non-Deterministic Domains*. *ITC-IRST*. 2001.

[PMC02] Bertoli P. Pistore M. and Roveri M. *Planning as Model Checking*. http://sra.itc.it/tools/mbp/AIPS02-TUT-We1_slides.pdf. *AIPS'02 Tutorial.*

[PMWSC05] Pistore M. , Barbon F., Bertoli P., Shaparau P. and Traverso P. *Planning and Monitoring Web Service Composition. In Proc of ICAPS'04.*

[SUCAR05] Berardi D., Calvanese D., De Giacomo G., Lenzerini M. and Mecella M. *Synthesis of Underspecified Composite e-Services based on Automated Reasoning. In Proc. of ICSOC'04.*

[TSSC06] De Giacomo G. *Transition Systems and Service Composition. Seminari di Ingegneria del Software 2005/2006.* http://www.dis.uniroma1.it/~degiacom/didattica/semingsoft/materiale/3-servizi/2-TS&ComposizioneServizi-2up.pdf.

[WSCCSOP03] J. Koehler and B. Srivastava. Web Service Composition – Current Solutions and Open Problems. *In Proc. Of ICAPS'03.*

[WSCDL04] D. Burdett, N. Kavantzas and G. Ritzinger. Web Services Choreography Description Language (WS-CDL) 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/. *W3C Working Draft, 2004.*