Università degli Studi di Salerno

TECNICHE AUTOMATICHE PER LA CORRETTEZZA DEL SOFTWARE

ANALYSIS OF INTERACTING BPEL WEB SERVICES

Mario Villani

Anno Accademico 2016/2017

INTRODUZIONE

Un obiettivo comune dei **web services** consiste nell'avere una collezione di servizi software accessibili tramite protocolli standard.
Le funzionalità di questi servizi devono essere «scoperte» in modo automatico e integrate in applicazioni oppure composte per creare dei servizi molto più complessi.

In questo paper verranno individuati i comportamenti comuni a tutti i **web services** al fine di sviluppare un modello che permetta di analizzare e **verificare** le loro proprietà.

Il tool sviluppato dagli autori permette di verificare specifiche espressioni in linguaggio **BPEL (Business Process Execution Language).**

INTRODUZIONE

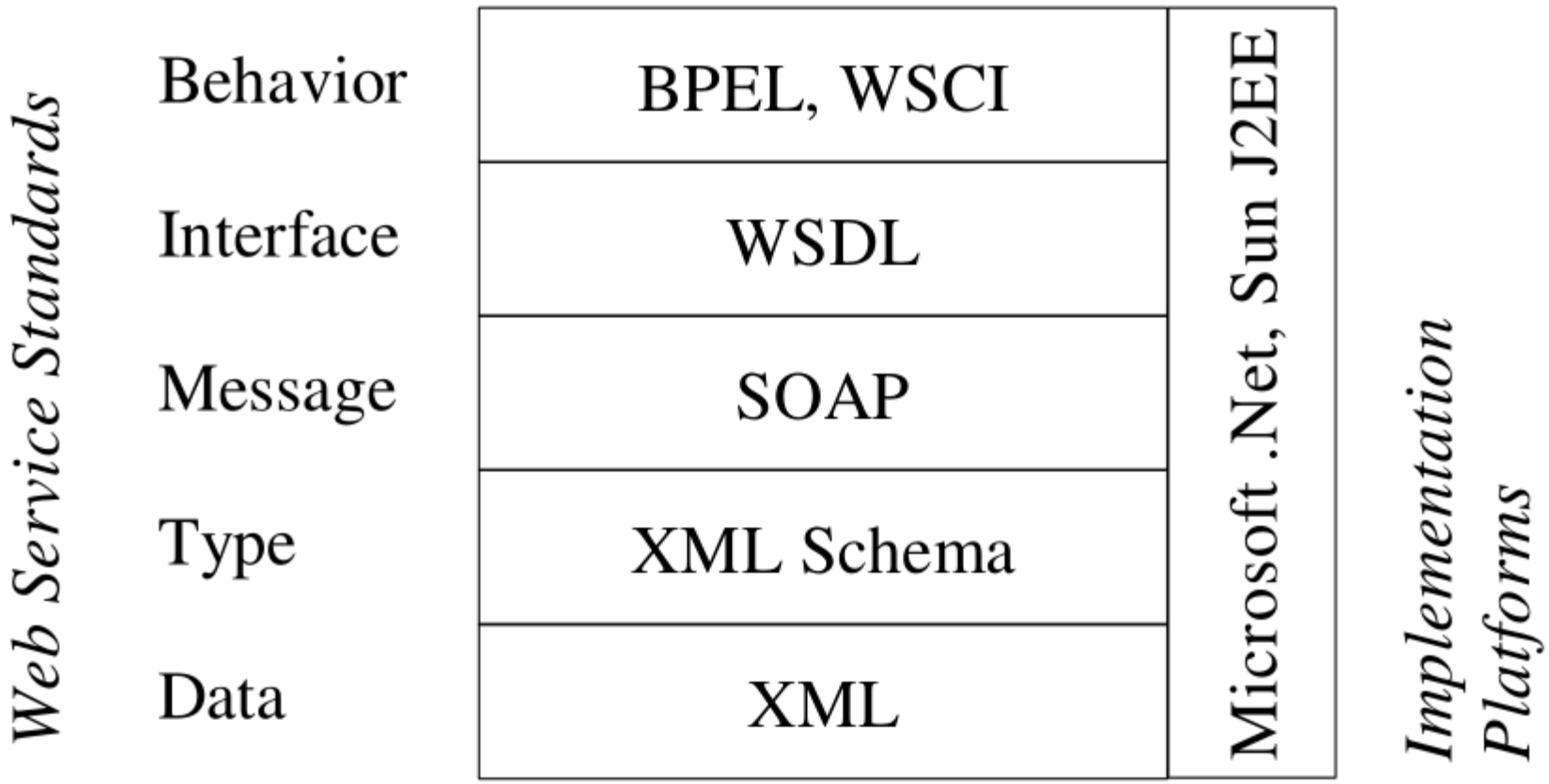
Il tool sviluppato:

- Trasforma i web services **BPEL** in un modello caratterizzato da **guarded automata;**
- Trasforma il modello **guarded automata** in linguaggio **Promela,** permettendo la verifica del web service utilizzando **SPIN.**

L'utilizzo del modello **guarded automata** come rappresentazione intermedia dei web services rappresenta un'architettura modulare ed estendibile per il tool:

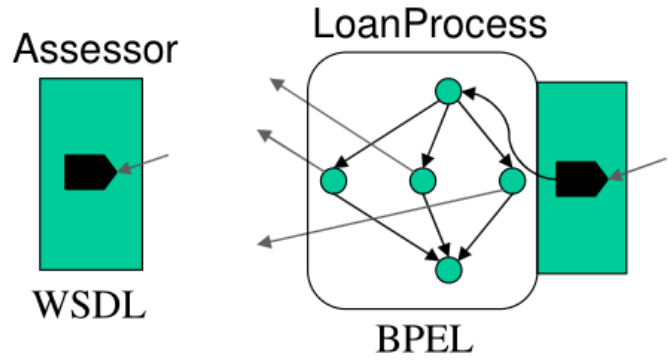
- può essere esteso per supportare più linguaggi al di fuori di XML;
- può essere esteso per supportare altri tool di model checking.

WEB SERVICES STANDARD STACK



INTERAZIONE TRA WEB SERVICES

L'interfaccia dei web services può essere espressa utilizzando **WSDL,** ma **non** fornisce alcuna informazione sul suo comportamento (figura 1).

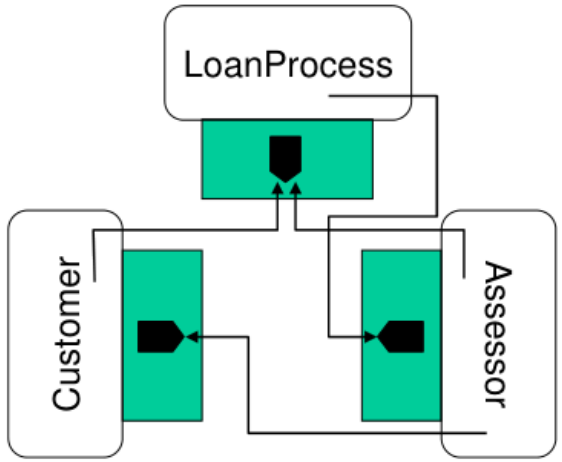


Consideriamo un web service per l'approvazione di prestiti (figura 2). Il cliente utilizza una porta (specificata da **WSDL**) ed è presente una specifica del comportamento utilizzando **BPEL**.

Per verificare un web service potrebbero essere usate pre e post condizioni su servizi «atomici».

BPEL risulta essere vantaggioso poiché fornisce diversi costrutti come **sequenze di operazioni, condizioni, parallelismo.**

INTERAZIONE TRA WEB SERVICES



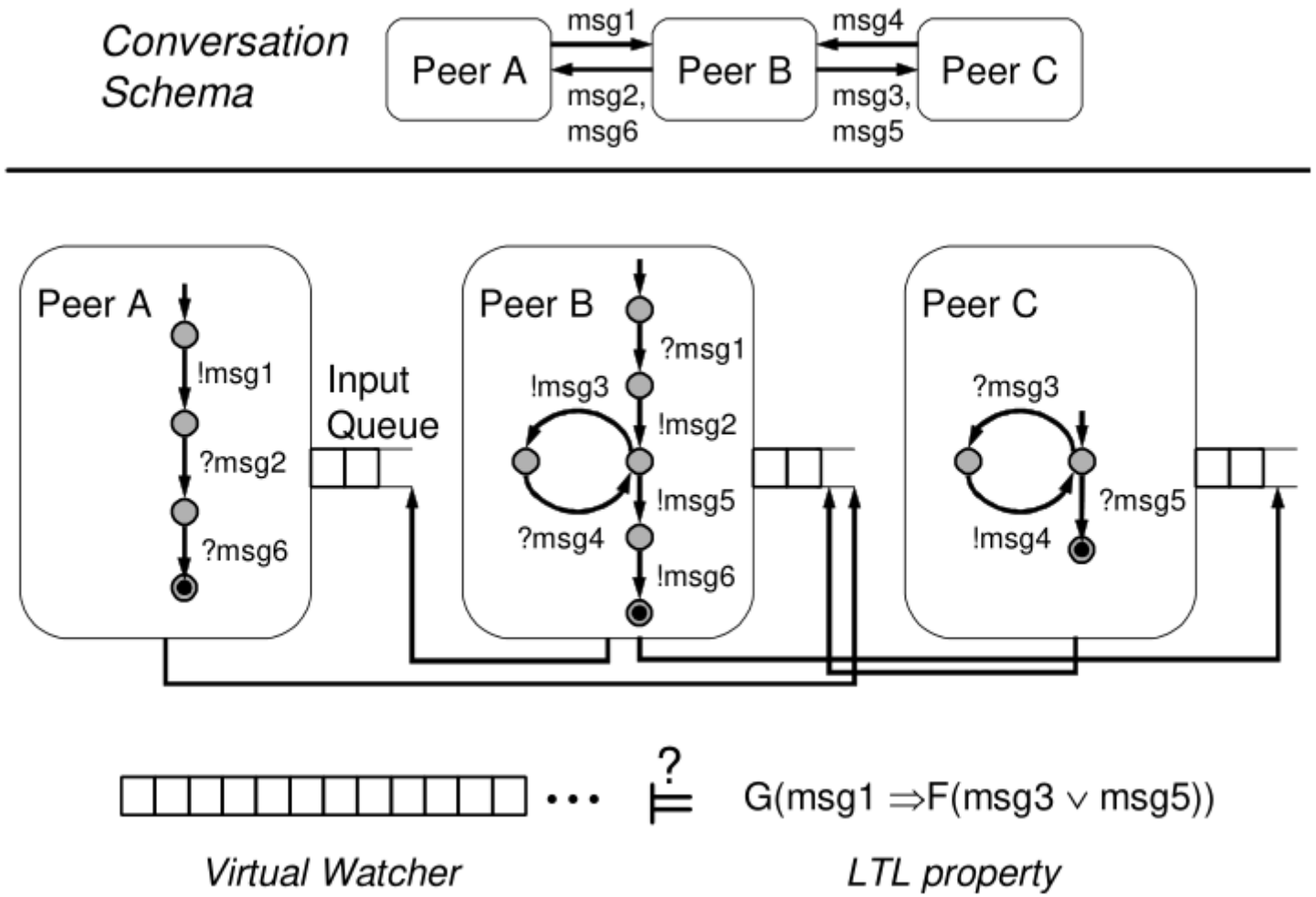
Non c'è nessun web service che possa tener traccia dello stato «globale» di tutta l'esecuzione del servizio.

MODELLO PER WEB SERVICES

Un **web service composito** è caratterizzato da un **conversation schema** che specifica il set di peer e i messaggi trasmessi tra i peer e da un set di **guarded automata** che specificano il comportamento di ogni peer.

- Data la natura asincrona della comunicazione, ciascun peer possiede una coda (**FIFO**) per memorizzare i messaggi;
- Esiste un **virtual watcher** che registra le sequenza di messaggi appena vengono inviati dai peer;
- La sequenza di messaggi registrata dal watcher è chiamata **conversazione.**
  Le **conversazioni** sono fondamentali per poter applicare proprietà di tipo **LTL** su tutte le tipologie di web services.

MODELLO PER WEB SERVICES



MODELLO FORMALE PER WEB SERVICES

Formalmente, un *web service composito* è *una tupla* $S = < (P, M), A_1, \dots, A_n >$

- (P, M) è il *conversation schema;*
- $n = |P|$ (n è il numero di *peers*);
- A_i è un'implementazione del *peer* $p_i \in P$;
- P è un set finito di *peers*, M è un set finito di *tipologie di messaggi;*
- Ogni *messaggio* di *tipo* $c \in M$ è trasmesso **esclusivamente** tra una coppia di *peers*.

Nel modello, ogni messaggio è un documento XML e i *tipi dei messaggi* corrispondono a *XML Schemas*.

MODELLO FORMALE PER WEB SERVICES

- ∀ messaggio $c \in M$ definiamo $DOM(c)$ come l'insieme di tutti i documenti XML di *tipo* c;
- Definiamo l'alfabeto dei messaggi come $\Sigma = U_{c \in M} \times \mathrm{DOM(c)}$;
- Ogni elemento $m \in \Sigma$ è un *messaggio;*
- Definiamo $TYPE(m) \in M$ per denotare il *tipo* di un messaggio m;
- m è definita *istanza* di $TYPE(m)$.

GUARDED AUTOMATA PEERS

$\forall p_i$ la sua implementazione A_i è un *guarded automaton* così strutturato:

$$(M_i^{in}, M_i^{out}, L_i, T_i, s_i, F_i, \Delta_i)$$

- M_i^{in}, M_i^{out} sono i *tipi di messaggi* in arrivo (uscita) per l'automa p_i;
- L_i è il set di *variabili locali* per l'automa p_i (sono documenti XML);
- T_i, s_i, F_i sono rispettivamente il *set di stati,* lo *stato iniziale* e *il set di stati finali;*
- Per ogni variabile locale $l \in L_i$, utilizziamo $DOM(L)$ per denotare tutti i *documenti XML* che matchano alla dichiarazione del tipo l.
  In modo analogo alle *variabili locali*, i *tipi* delle *variabili locali* non sono altro che degli schemi XML.

GUARDED AUTOMATA PEERS

Ciascuna transizione $\tau \in \Delta_i$ dell'*automa guarded* per il peer p_i ha uno stato sorgente $q_1 \in T_i$ e uno stato destinazione $q_2 \in T_i$. Una transizione può avere tre forme:

- *Local-transition:* $\tau = (q_1, \mathrm{g}, q_2)$ dove g è la guardia della transizione. La transizione cambia lo stato dell'automa da q_1a q_2 e aggiorna le sue variabili locali in base a g;
- *Receive-transition:* $\tau = (q_1, ?\,\mathrm{a}, q_2)$ dove $a \in M_i^{in}$. La transizione cambia lo stato dell'automa da q_1a q_2 e rimuove il messaggio ricevuto (di tipo a) dalla coda d'ingresso del peer p_i;
- *Send-transition:* $\tau = (q_1, (!\,\mathrm{b}, \mathrm{g}), q_2)$ dove b $\in M_i^{out}$. La transizione cambia lo stato dell'automa da q_1a q_2 e appende il messaggio inviato (di tipo b) alla coda d'ingresso del peer ricevente.

GUARDED AUTOMATA PEERS

I messaggi inviati non sono ricevuti (o consumati) in modo sincrono: sono *instantaneamente* aggiunti alla coda d'ingresso appropriata.

Un messaggio viene ricevuto solo dopo che è spostato all'inizio della coda.

Una *guardia* consiste in una *condizione* ed un *set di assegnamenti;* le transizioni di *ricezione* non hanno guardia perché consumano direttamente il messaggio dalla loro coda.

GUARDED AUTOMATA PEERS: PREDICATI

- Data una *send-transition:* $\tau = (s, (!\, b, g), t)$ la guardia g è un predicato di questa forma: $\mathrm{g}(m, \vec{m} \times \vec{l})$ *dove* m è il messaggio da inviare, $\vec{m}$ contiene l'utlima istanza di ogni tipo di messaggio che è ricevuto o inviato ed $\vec{l}$ contiene i valori delle variabili locali in L_{i}
- Data una *local-transition:* $\tau = (s, g, t)$ la guardia g è un predicato di questa forma: $\mathrm{g}(\vec{l}, \vec{m} \times \vec{l})$, dove il primo vettore $(\vec{l})$ denota il valore delle variabili locali dopo la transizione, mentre $\vec{m} \times \vec{l}$ denota ivalori dei messaggi e delle variabili locali appena prima che la transizione venga eseguita.

CONVERSAZIONI

Formalmente, dato un web service $S = < (P, M), A_1, \dots, A_n >$, una *configurazione globale di* S è una tupla $(3n + 3)$ della forma:

$$(Q_1, \vec{l}_1, \mathrm{t}_1, \dots, Q_n, \vec{l}_n, t_n, w, \vec{s}, \vec{r})$$

- Per ogni $j \in [1 \dots n], Q_j \in \Sigma^*$ è il contenuto della coda di input del peer p_j

  $\vec{l}_j$ è il vettore che denota le variabili locali di p_j

  t_j è lo stato di p_j

- $w \in \Sigma^*$ è la sequenza dei messaggi registrata dal *virtual watcher*

- $\vec{s}, \vec{r}$ memorizzano le ultime istanze *ricevute* ed *inviate* di ciascun *tipo di messaggio*

CONVERSAZIONI

Possiamo subito definire una relazione di derivazione tra due configurazioni:

$\gamma \rightarrow \gamma'$ sse $\exists p_i$ e transizione $\tau \in \Delta_i$ t.c. eseguendo la transizione τ in una configurazione γ si ottiene una configurazione γ' (*)

Inoltre, definiamo il concetto di *run* di S come una sequenza finita di configurazioni

$$\gamma = \gamma_0, \gamma_1, \gamma_2, \dots, \gamma_{|\gamma|-1}$$

che deve soddisfare le condizioni seguenti:

- $\left(\epsilon, \perp, s_1, \dots, \epsilon, \perp, s_n, \epsilon, \overrightarrow{\perp}, \overrightarrow{\perp}\right)$ deve essere la configurazione iniziale;
- Per ogni $0 \leq i < |\gamma|, \gamma_i \rightarrow \gamma_{i+1}$ (vedi proprietà *)
- $\gamma_{|\gamma|-1} = \left(\epsilon, \vec{l}_1, s'_1, \dots, \epsilon, \vec{l}_n, s'_n, w, \vec{s}, \vec{c}\right)$ dove per ogni peer p_i, $s'_i \in F_i$

CONVERSAZIONI

Le sequenze di messaggi osservate dal watcher non sono altro che le *conversazioni* del nostro web service.

- $w \in \Sigma^*$ è una conversazione del web service S se esiste una *run* $\gamma = \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_{|\gamma|-1}$ su S t.c. il valore del *virtual watcher* nella configurazione finale $\gamma_{|\gamma|-1}$ è proprio w
- $L(S)$ denota il set di conversazioni di S

Utilizziamo **LTL** per esprimere le proprietà sulle conversazioni.
Ogni proposizione atomica può essere di due tipi:

- c, dove c è un tipo di messaggio ($c \in M$)
- $c.pred$, dove $c \in M$ e $pred$ è un predicato sui contenuti di c

CONVERSAZIONI

Un messaggio m **soddisfa** una proposizione atomica ψ se

$m \vDash \psi$ if

- $TYPE(m) = \psi$ se $\psi \in M$
- $TYPE(m) = c \wedge \mathrm{pred}(\mathrm{m}) = \mathrm{true}$ se $\psi \in c.pred$

Dato un web service S e una proprietà LTL ϕ

$S \vDash \phi$ sse per tutti i $w \in L(S)$ $w \vDash \phi$

INDECIBILITÀ DELLA VERIFICA DEI WEB SERVICES

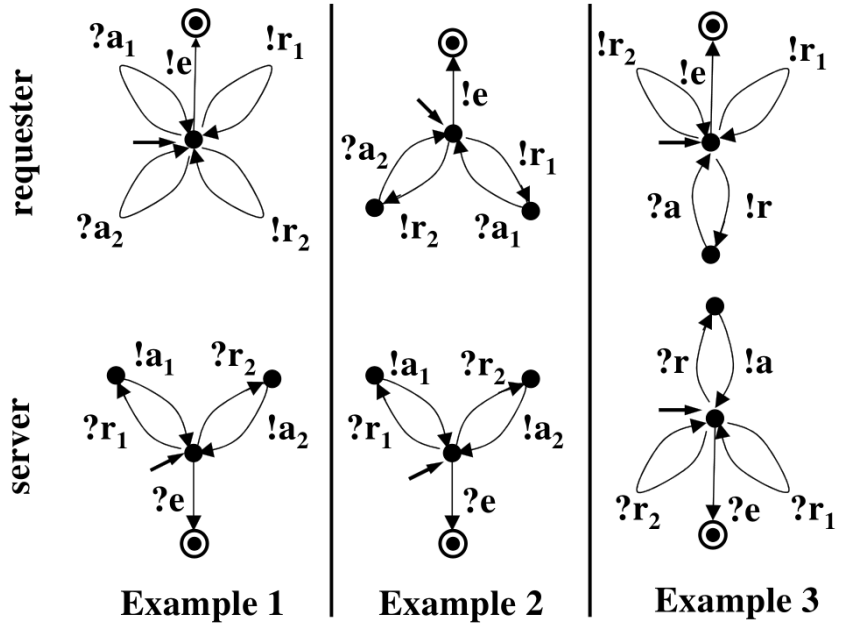
- Un web service S ha un contenuto finito se per ogni tipo di messaggio $c \in M, DOM(c)$ è un set finito e per ogni variabile locale l $l \in L_i$ di ogni peer $DOM(l)$ è un set finito.

Poiché la comunicazione avviene in modo asincrono, le *input queues* potrebbero avere una dimensione eccessivamente grande. Dato S ed una proprietà LTL ϕ:

Non è possibile stabilire se una proprietà LTL è verificabile su un web service S che abbia contenuto finito

SINCRONIZZABILITÀ

Consideriamo i tre esempi in figura:

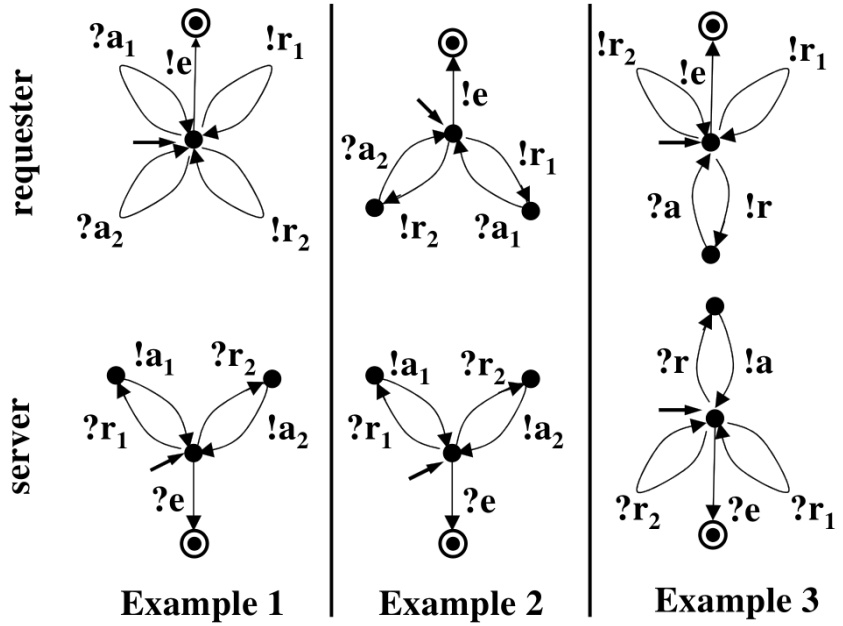


- Nell'esempio 1 il requester può inviare un numero arbitrario di messaggi prima che il server li processi.
  Viene prodotto un set non regolare del tipo $(r_1|r_2|a_1|a_2)^*e$ e la grandezza delle code è unbounded;
- Nell'esempio 2 requester e server si muovono in modo coordinato, producendo conversazioni del tipo $(r_1r_2|a_1a_2)^*e$, fornendo un numero finito di stati per il checking (c'è almeno un messaggio in coda per ogni esecuzione);

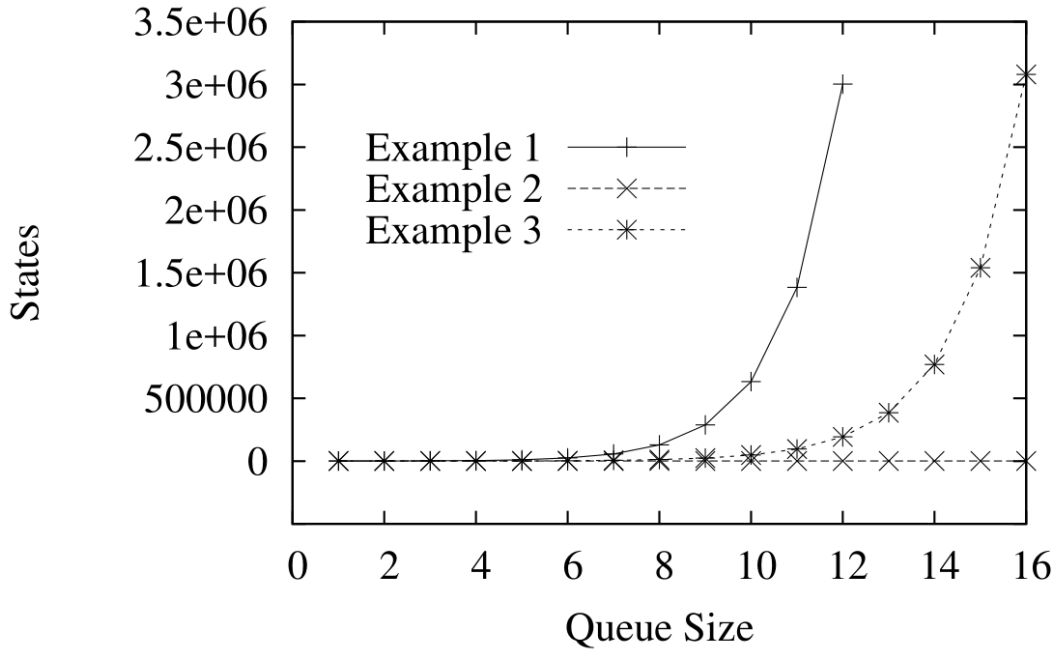
SINCRONIZZABILITÀ

- L'esempio 3 ha un set di conversazioni di tipo regolare, ma, come l'esempio 1, ha uno spazio degli stati infinito perché il numero di messaggi nelle code di input è unbounded.

Segue una verifica sperimentale direttamente su SPIN.



SINCRONIZZABILITÀ



Possiamo fare qualcosa per verificare WS con situazioni analoghe a quelle dell'esempio?

Per l'esempio 1, purtroppo, possiamo solo effettuare una verifica parziale.

COMUNICAZIONE SINCRONA

Intuitivamente, la semantica sincrona **forza ciascun peer a consumare i messaggi immediatamente,** quindi **non abbiamo più bisogno di una coda in ingresso.**
Ridefiniamo quindi le configurazioni utilizzando la semantica per *la comunicazione sincrona:*

sc-configuration: $(\overrightarrow{l_1}, t_1, \dots, \overrightarrow{l_n}, t_n, w, \vec{s}, \vec{r})$

configurazione standard: $(Q_1, \overrightarrow{l_1}, t_1, \dots, Q_n, \overrightarrow{l_n}, t_n, w, \vec{s}, \vec{r})$

- La definizione di derivazione γ viene modificata in modo che una *transizione di tipo send possa essere eseguita con una operazione di receive che matcha;*
- *Un web service è sincronizzabile se* $L_{syn}(S) = L(S)$.

COMUNICAZIONE SINCRONA

Chiaramente, se un web service è sincronizzabile, i risultati delle verifiche saranno gli stessi della sua versione asincrona con code unbounded.

Come stabiliamo se un web service è sincronizzabile?

A partire dalla definizione dei suoi *guarded automaton:*

$$A_i = \left(M_i^{in}, M_i^{out}, L_i, T_i, s_i, F_i, \Delta_i\right)$$

Costruiamo delle versioni semplificate (*skeleton*)

$$sk(A) = \left(M_i^{in}, M_i^{out}, T_i, s_i, F_i, \Delta_i'\right)$$

L_i sono state rimosse e ogni transizione $\tau' \in \Delta_i'$ viene eliminata rimuovendo la guardia della transizione corrispondente τ.

COMUNICAZIONE SINCRONA

Dato un set di *guarded automata,* la loro composizione è sincronizzabile se sono soddisfatte queste proprietà sui loro *skeletons*:

- Synchronous compatibility: il prodotto cartesiano degli *skeletons* produce un automa che non contiene uno stato dove un peer p_i è pronto ad inviare un messaggio al peer p_j ma p_j non può riceverlo;
- Autonomy: per ogni peer p_i ed ogni stato s_j di p_i deve valere almeno una di queste condizioni:
  - Le prossime transizioni di s_j devono essere solo operazioni di invio;
  - Le prossime transizioni di s_j devono essere solo operazioni di ricezione;
  - s_j è uno stato finale o può raggiungere uno stato finale tramite ϵ-transition.

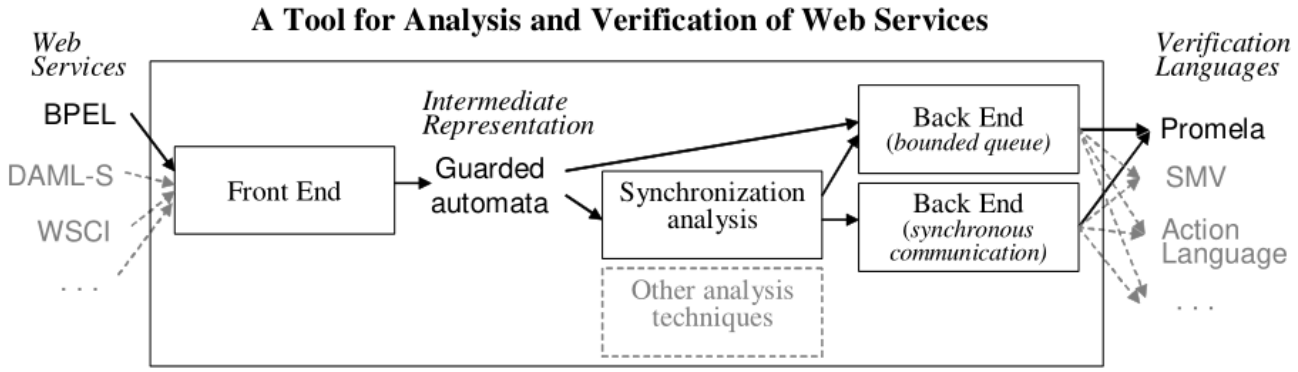
COMUNICAZIONE SINCRONA

- Lossless composition: marchiamo lo stato iniziale di ogni *skeleton* dei peers come stato finale e costruiamo il prodotto cartesiano di ogni *skeleton*. La proiezione del prodotto cartesiano ad ogni peer deve essere equivalente allo scheletro originale (con lo stato iniziale marcato come stato finale).

Teorema: Sia $S = < (P, M), A_1, \dots, A_n >$ un web service composito; se i suoi scheletri $sk(A_1), \dots, sk(A_n)$ soddisfano le tre condizioni di sincronizzabilità allora S è sincronizzabile.

Intuitivamente, quando un web service soddisfa le tre condizioni, vuol dire che la *input queue* di ogni peer è vuota appena il peer invia un messaggio. Su questa osservazione mostriamo che per ogni conversazione esiste una esecuzione corrispondente dove i peer comunicano in modo sincrono.

TOOL



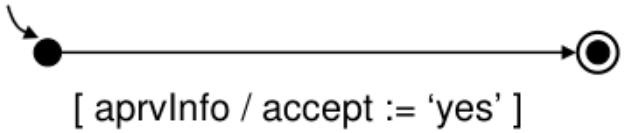
- Il tool (**WSAT,** Web Services Analysis Tool) effettua un' analisi per determinare se il web service risulta sincronizzabile: se il risultato è **positivo,** viene generata una specifica Promela con un modello semplificato (comunicazione sincrona), **altrimenti** si applica un bound sulla grandezza delle code;
- Il web service può essere poi verificato utilizzando SPIN (viene generato un file .pml).

TRADUZIONE BPEL -> GUARDED AUTOMATA

- Appena riceviamo le specifiche BPEL, la lista dei peer P della tupla per lo schema $S\ (P, M)$ è già nota. Le tipologie di messaggio sono estratte dai file WSDL.
- Per ogni input/output/fault di un'operazione di ogni porta e ogni *service link* andiamo a stabilire una *tipologia di messaggio*.

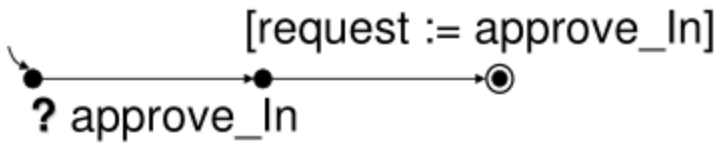
Ogni costrutto BPEL si può tradurre con un automa con un singolo stato iniziale e finale.

TRADUZIONE BPEL -> GUARDED AUTOMATA

BPEL	Sample Code	Translation
assign	<assign ...> <copy> <from="yes"/> <to var="aprvInfo" part="accept"/> </copy> </assign>	

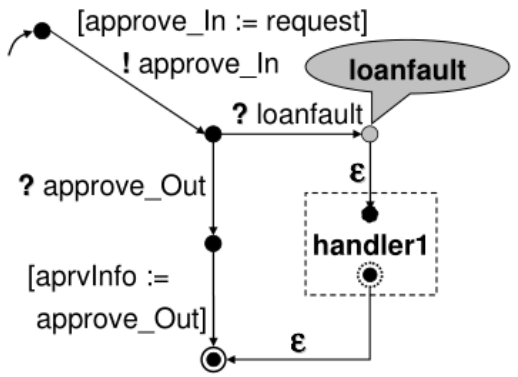
- Lo statement *assign* è tradotto con un automa dove la guardia g è una espressione Xpath che va a fare un semplice assegnamento.

TRADUZIONE BPEL -> GUARDED AUTOMATA

BPEL	Sample Code	Translation
receive	<receive ... operation="approve" variable="request" />	

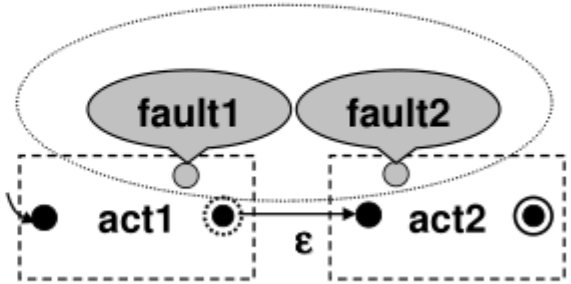
- Lo statement *receive* è tradotto con un automa a due stati, dove la prima transizione riceve il messaggio e la seconda ha una guardia g che assegna la variabile. *approve_In* è il nome dell'operazione che il client deve invocare.

TRADUZIONE BPEL -> GUARDED AUTOMATA

BPEL	Sample Code	Translation
invoke	<scope> <invoke ... operation="approve" invar="request" outvar="aprvInfo /> <catch ... faultname="loanfault" < ... handler1 ... /> </catch> </scope>	

- Lo statement *invoke (invocazione di procedura sincrona)* può avere delle eccezioni da catchare. Per ogni fault dichiarato nel blocco *invoke,* il guarded automata deve possedere una transizione verso il fault handler.

TRADUZIONE BPEL -> GUARDED AUTOMATA

BPEL	Sample Code	Translation
sequence	<sequence ... > < ... act1 ... > < ... act2 ... > </sequence>	

- Lo statement *sequence (attività in sequenza)* connette lo stato finale con quello iniziale dell'automa successivo; lo stato finale di ogni automa viene rimosso, andando a conservare solo quello relativo all'ultima attività.

TRADUZIONE AUTOMA -> PROMELA

Creati i *guarded automata,* li traduciamo in una specifica Promela, che consiste in un set di processi concorrenti, uno per ciascun *guarded automata*.

```
/* channels */
chan ch_loanaprv = [8] of {mtype, creditInfo, appeal};
chan ch_customer= [8] of {mtype, aprvInfo};
```

- Una variabile di tipo **channel** è utilizzata per simulare la sua coda di ingresso, dove il primo parametro indica il tipo di messaggio che si sta trasmettendo e gli altri due sono i messaggi che il peer può ricevere.

TRADUZIONE AUTOMA -> PROMELA

- Ogni automa è tradotto in un *process type (proctype) di Promela*.
- *state* traccia lo stato corrente dell'automa
- Il corpo dell'automa è un single loop
- Ogni transizione dell'automa è tradotto in un branch del blocco if.
- In ogni iterazione del loop, la prima condizione abilitante di ogni guardia di transizione è valutata e il risultato è salvato nella var. booleana corrispondente.

```
proctype loanaprv(){
 mtype state;
 /* definition of local variables */
 creditInfo request; ...
 /* definition of auxiliary variables used
    to evaluate XPath expressions */
 bool bVar_0, ...

 do::
  /* evaluation of transition conditions */
  ... bCond1 = true; ...

  /* nondeterministically select transitions to fire */
  if
   /* transition t1: s1 -> s2, ?aprv_In */
   ::state == m_loanaprv_s1 && bCond1 &&
     ch_loanaprv ? [m_aprv_In] ->
      atomic{
       ch_loanaprv ?
          m_aprv_In, aprv_In_r, stub_appeal;
       state = m_loanaprv_s2 ;
      }

   /* transition t2: s2 -> s3, !aprv_Out,
     [cond2 => aprv_Out//accept = 'yes' ] */
   ::state == m_loanaprv_s2 && bCond2 ->
     atomic{
      aprv_Out.s_accept = m_yes;
```

TRADUZIONE AUTOMA -> PROMELA

Come si può tradurre una espressione XPath in Promela?

Supponiamo di avere un messaggio *aprv_In* e supponiamo che possegga un elemento *preferred_term* che è una lista di tipo (APR/termLength).

Supponiamo inoltre che la lista non possa superare un numero di 10 elementi.

Consideriamo di avere l'istruzione XPath seguente:

```
aprv_In / preferred_term / termLength == 6
```

TRADUZIONE AUTOMA -> PROMELA

Il codice risultante è:

```
bool bResult = false;
int i=0;
do
:: i<10 ->
   if
   :: aprv.preferred_term[i].termLength == 6
      -> bResult = true
   :: else -> skip;
   fi;
:: else -> break;
od;
```

GRAZIE PER L'ATTENZIONE