UNIVERSITÀ DEGLI STUDI DI FIRENZE
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA SPECIALISTICA IN SCIENZE E TECNOLOGIE DELL'INFORMAZIONE

Model checking probabilistico di orchestrazioni di servizi

Relatore:
Prof. Rosario Pugliese

Candidato:
Luca Cesari

Correlatore:
Dott. Francesco Tiezzi

Anno Accademico 2010-2011

Luca Cesari: *Model checking probabilistico di orchestrazioni di servizi*

Corso di Laurea Specialistica in Scienze e Tecnologie dell'Informazione
Anno Accademico 2010-2011

A mio nonno Luigi

Sommario

Questa tesi illustra progetto e relativa implementazione di uno strumento che abilita la verifica di specifiche di sistemi di servizi, descritti nel linguaggio B*lite*, tramite il model checker probabilistico Prism.

Le motivazioni del lavoro descritto in questa tesi nascono dalla necessità di poter verificare il comportamento di un sistema di servizi rispetto alle caratteristiche desiderate, senza dover affrontare le difficoltà di scrittura della specifica del sistema nel linguaggio accettato dal model checker. Ciò poiché i linguaggi di specifica accettati dai model checker sono solitamente molto semplici e non permettono una specifica diretta di costrutti astratti complessi come quelli necessari alla definizione di un servizio. Sarebbe quindi necessaria una prima fase per decidere come esprimere i costrutti di alto livello nel linguaggio del model checker e poi un'altra fase per comporre le traduzioni parziali in un modello che rappresenti un'astrazione corretta della specifica. In sintesi, queste operazioni risultano molto complesse, scoraggiando l'utente dall'effettuare una verifica formale del sistema.

L'obiettivo di questo lavoro è di semplificare ed automatizzare questa serie di fasi e permettere all'utente di procedere direttamente con la verifica del sistema. Il linguaggio scelto per la definizione delle specifiche è B*lite*. B*lite* è un linguaggio formale per la definizione e orchestrazione di servizi, che mette a disposizione dell'utilizzatore un insieme di costrutti semplici senza però ridurre il potere descrittivo rispetto a linguaggi come WS-BPEL. L'adozione di B*lite* come base per la descrizione dei sistemi di servizi ha permesso di utilizzare B*lite*C. B*lite*C è uno strumento software che permette la generazione di codice WS-BPEL da una specifica B*lite*, oltre a prendersi cura della generazione di tutti i documenti accessori necessari al deploy e all'esecuzione di un processo WS-BPEL su di un engine WS-BPEL.

Lo strumento software che proponiamo è B*lite*2Prism, un modulo di B*lite*C per la generazione di specifiche Prism da descrizioni B*lite*. Dunque con questo nuovo modulo B*lite*C oltre a permettere l'esecuzione su engine WS-BPEL di sistemi di servizi scritti in B*lite*, abilita anche una fase di verifica del loro comportamento. Ovviamente la fase di traduzione è trasparente all'utente, che dovrà semplicemente scrivere la specifica B*lite* e le proprietà da testare sul modello tradotto da B*lite*2Prism. In questo modo l'utente è alleggerito dall'incarico di formalizzare e scrivere il modello associato alla specifica B*lite* e dovrà semplicemente occuparsi di verificare le proprietà di interesse.

Ringraziamenti

Desidero ringraziare per il supporto e per la compresione dimostrata in questo periodo di stesura della tesi il mio relatore Prof. Pugliese e il correlatore Dott. Tiezzi. Entrambi mi hanno aiutato e seguito nonostante gli impegni e il periodo non proprio congeniale.

Un grande ringraziamento va a mia madre Marina e mio padre Marco per avermi sostenuto e creduto nelle mie capacità permettendomi di portare a compimento questa carriera di studio.

Uno speciale ringraziamento va a mia sorella, mia nonna Irma e la mia fidanzata Silvia per il sostegno psicologico e la pazienza dimostrata in questo periodo di stesura della tesi.

Ringazio il mio compagno di ventura Geri Tollkuçi che mi ha aiutato, sostenuto, ma sopratutto sopportato in questo stessante periodo.

Per ultimi, ma non per importanza, ringrazio tutti i miei colleghi di studio Karen, Olta, Ela e Massimiliano per aver studiato e condiviso con me questi anni di studio.

Potrei aver dimenticato di ringraziare qualcuno che, comunque, ha contato molto per me in questi anni. Mi scuso della possibile dimenticanza e confido nella sua comprensione.

Indice

Capitolo 1

Introduzione

Con l'aumento dell'utenza di Internet si è andato ad evidenziare sempre di più il successo del World Wide Web, che progressivamente si è popolato di risorse e funzionalità sempre diverse. L'affermazione di Internet nell'uso comune ha portato alla definizione di nuove tecnologie per sfruttare in modo sempre più innovativo questa risorsa. Sono state infatti definite tecnologie quali: e-business, per la vendita di beni; e-learning, per la fruizione di insegnamento a distanza; e-government, per l'accesso facilitato ai servizi per il cittadino; e-healt, per l'automatizzazione dei servizi di tipo medicale, ma anche altri modelli analoghi per altri contesti.

Questo processo di innovazione ha evidenziato sempre di più l'importanza di una tecnologia che permettesse l'interazione umana, ma anche l'utilizzo automatico di risorse. Questa esigenza ha portato il World Wide Web ad evolversi in un'Architettura Orientata ai Servizi.

Un'*Architettura Orientata ai Servizi* (o SOA, dal nome inglese Service Oriented Architecture) è un'architettura distribuita composta da uno o più nodi che mettono a disposizione funzionalità e risorse dette "servizi" fruibili attraverso lo scambio di messaggi. Un servizio è assimilabile ad un'entità autonoma, che può essere pubblicata, scoperta e assemblata con altre, per costruire applicazioni complesse. Il servizio, inteso in chiave architetturale, permette di disaccoppiare l'implementazione della risorsa dalla modalità di interazione con essa, consentendo così all'utente di utilizzarla senza conoscere i dettagli relativi alla realizzazione. In un'architettura SOA, i servizi possono ricoprire tre ruoli: il provider, il richiedente e il registro. I *provider* offrono funzionalità e pubblicano descrizioni di servizi su *registri* per abilitare la scoperta automatica e permetterne l'invocazione da parte dei *richiedenti*.

Ad oggi, sebbene esistano varie interpretazioni di SOA, quella maggiormente utilizzata descrive i servizi attraverso *servizi web*. Un servizio web è un sistema software

progettato per garantire interoperabilità tra nodi eterogenei di una stessa rete. Le interfacce dei servizi web sono definite attraverso WSDL (Web Service Description Language [36]), un linguaggio di descrizione dei servizi web basato su XML (eXtensible Markup Language [38]). Il documento WSDL associato ad un servizio ha una duplice funzione, da un parte permette al fornitore di specificare le funzionalità offerte senza fornire dettagli relativi all'implementazione e dall'altra specifica all'utilizzatore come contattare il servizio.

I nodi di un'architettura SOA basata su servizi web inviano messaggi XML attraverso il protocollo SOAP (ma possono essere usati anche altri protocolli quali JAX-RPC [24], XML-RPC [35], REST [14]), che sono trasportati sulla rete attraverso protocolli come HTTP [20], SMTP [21], FTP [19] o XMPP [22]. Per garantire funzionalità specializzate possono essere utilizzati protocolli come: UDDI [28] per l'elencazione dei servizi, WS-Security [30] per l'autenticazione e WS-Reliability [29] per la definizione di messaggi affidabili (ad esempio quelli necessari a movimenti di denaro sul web).

Con il diffondersi dell'architettura SOA composta da servizi web, si è delineata la necessità di comporre e gestire i servizi in modo più articolato, erogandone di nuovi senza modificare quelli esistenti. Da questa necessità sono nate le specifiche WS-CDL (Web Service Choreography Description Language [37]) e WS-BPEL (Web Service Business Process Execution Language [3]) che regolano due diversi aspetti dello stesso problema. La prima definisce le regole con le quali i vari servizi si dovranno comportare; la seconda invece fornisce i costrutti per generare nuovi servizi componendo quelli già esistenti. Un programma scritto in WS-BPEL è detto *processo*, ed è eseguito su appositi engine WS-BPEL che mettono a disposizione un ambiente di esecuzione del programma. L'*engine* è un elemento fondamentale per l'esecuzione di codice WS-BPEL, perché fornisce al programmatore un'implementazione delle complesse funzionalità dei costrutti WS-BPEL, che altrimenti non troverebbero corrispondenza immediata in un normale linguaggio di programmazione. Esistono molti engine che permettono l'esecuzione di un processo WS-BPEL, tra i più noti troviamo ActiveVOS [2] di Active Endpoints, Apache ODE [5] del progetto Apache, ORACLE Bpel Process Manager [31] di Oracle, WebSphere [18] di IBM.

La definizione dello standard WS-BPEL in linguaggio naturale, cioè senza una specifica formale della semantica, e la mancanza di una convenzione per la fase di deploy, ha portato allo sviluppo di engine che, a volte, interpretano la stessa specifica in modo diverso ed utilizzano documenti differenti per la fase di deploy. Tutto ciò limita la portabilità di un processo WS-BPEL, cioè la possibilità di eseguirlo su un qualunque engine. Infatti, oltre all'evidente problema dell'utilizzo di documenti diversi per la fase di deploy, vi è il problema ben più grave della mancata garanzia di ottenere lo stesso

comportamento del processo. Tali inconvenienti sono stati ampiamente e dettagliatamente descritti in [26] e [27].

L'ampia gamma di costrutti messi a disposizione dallo standard WS-BPEL permette all'utilizzatore di definire un servizio in modi diversi relativamente al contesto in cui dovrà lavorare. Questa capacità rende difficile la scrittura di codice WS-BPEL perché, oltre a dover scrivere il codice utilizzando una sintassi XML, l'utente deve avere una discreta conoscenza dello standard per creare un servizio corretto. Per aiutare l'utente nella fase di stesura del codice, sono state create delle interfacce grafiche che permettono la semplificazione della progettazione in WS-BPEL, come ad esempio Active Designer [1] di Active Endpoints, Intalio Designer [23] di Intalio o il designer di Eclipse [11]. Questi strumenti permettono di definire facilmente processi molto semplici (con poche operazioni), ma quando la complessità dei processi aumenta ben presto la definizione grafica diventa di difficile gestione e comprensione.

In un precedente lavoro [8] è stata proposta una soluzione ai problemi che rendono difficile la progettazione e lo sviluppo di applicazioni WS-BPEL, tramite lo sviluppo di uno strumento software, B*lite*C. B*lite*C accetta in ingresso la specifica di un servizio scritto in B*lite* [26], un linguaggio in stile imperativo semplice e intuitivo, e restituisce in uscita la traduzione in WS-BPEL utilizzando solo un sottoinsieme dei costrutti dello standard, per facilitarne la comprensione. Questa soluzione non perde il potere descrittivo iniziale, infatti la maggior parte dei costrutti WS-BPEL omessi possono comunque essere codificati in B*lite*.

Utilizzando B*lite*C è possibile progettare e scrivere un nuovo servizio concentrandosi solo sul problema da risolvere piuttosto che sulla scelta dei dettagli. La definizione testuale proposta da B*lite*C permette di mantenere semplice la comprensione dei processi che svolgono funzionalità complesse, risolvendo così il problema riscontrato nell'utilizzo di interfacce grafiche. Inoltre, B*lite*C automatizza il processo di deploy sugli engine WS-BPEL generando automaticamente tutti i documenti necessari e organizzandoli nel formato accettato dall'engine. In particolare, B*lite*C è sviluppato per permettere il deploy di una specifica B*lite* su vari engine, fornendo così una soluzione alla mancanza di standard per i documenti caratterizzanti questa fase; questa capacità rende B*lite*C più versatile rispetto ad altri strumenti di supporto grazie alla capacità di adattamento ai vari engine.

Lo scopo di questa tesi è quello di estendere B*lite*C in modo da permettere, oltre al deploy dei servizi su di un engine, anche l'analisi del comportamento e delle interazioni dei processi attraverso l'utilizzo di un model checker.

Il model checking negli ultimi anni si è diffuso in varie aree come tecnica di verifica di sistemi; infatti il model checking abilita la verifica automatica di proprietà su di un

modello che descrive il sistema in esame. Le proprietà, sono descrizioni di condizioni desiderate (o non desiderate) che il modello deve (o non dovrebbe) soddisfare. Quando una proprietà è testata su di un modello si può presentare una delle due seguenti situazioni: la proprietà è verificata, dunque la verifica ha successo; la proprietà non è verificata quindi il model checker fornisce un controesempio che indica una situazione nella quale il modello si comporta in modo indesiderato.

Tipicamente i modelli sono rappresentati attraverso automi a stati finiti, dove le transizioni tra uno stato e l'altro modellano l'evoluzione del sistema. Gli automi solitamente sono generati in modo automatico da alcune specifiche di alto livello come reti di Petri [33], Promela [12] o PEPA [15]. Questa astrazione è aggiunta per permettere all'utente di concentrarsi sulla descrizione del modello piuttosto che su come comporre gli automi. Questa modalità permette inoltre di modificare il modello aggiungendo dettagli al procedere dell'analisi, evitando così di dover aver chiaro fin dall'inizio il tipo di analisi da effettuare sul modello. Le proprietà sono comunemente espresse attraverso logiche temporali capaci di esprimere relazioni temporali tra eventi.

In questa tesi utilizzeremo però una variante del model checking classico chiamata *model checking stocastico*. Il model checking stocastico è un metodo per il calcolo della probabilità che un evento si verifichi durante l'esecuzione di un sistema. Tipicamente un model checker classico fornisce una risposta Booleana come risultato della verifica di una proprietà su di un modello. Il model checking stocastico condivide con quello classico l'analisi di raggiungibilità del sistema di transizione, ma in aggiunta esso permette il calcolo della probabilità di questi eventi attraverso appropriati metodi numerici e analitici. In particolare, con il model checking classico è possibile verificare proprietà come *la condizione non è mai verificata nel modello* oppure *la condizione è sempre rispettata dal modello*, ma non è possibile conoscere con quale probabilità una condizione può verificarsi. Inoltre in molti contesti non è possibile descrivere il modello con un metodo deterministico a causa della complessità del fenomeno da rappresentare. Ad esempio con modelli probabilistici è possibile quantificare il tasso di successo dopo l'esecuzione di un'azione di fallimento, che nel contesto di un protocollo multimediale potrebbe rappresentare il tasso di successo della trasmissione quando la perdita si verifica ogni 100 frame trasmessi.

Ovviamente questo carattere probabilistico non può essere garantito utilizzando le rappresentazioni a sistemi di transizione del model checking classico, infatti sono utilizzate le catene di Markov, in particolare: MDP (Markov Decision Process), DTMC (Discete Time Markov Chain) e CTMC (Continuos Time Markov Chain).

I linguaggi di specifica delle proprietà sono logiche temporali probabilistiche solitamente ottenute da logiche temporali standard aggiungendo al quantificatore di cammino

una nozione probabilistica. In questo modo, oltre a garantire la capacità di esprimere relazioni temporali tra eventi, è aggiunta l'abilità di verificare la probabilità degli eventi.

Il lavoro descritto in questa tesi permette di sfruttare il model checker stocastico Prism [25, 32] per effettuare l'analisi delle specifiche B*lite*. Il modulo di B*lite*C che si occupa della trasformazione in Prism nel resto della trattazione sarà indicato come B*lite*2Prism.

Utilizzando B*lite*2Prism un utente potrà scrivere una specifica B*lite* che descrive il comportamento di un insieme di servizi e analizzarla prima di passare alla fase di sperimentazione del comportamento dei servizi su di un engine. In questo modo l'utente potrà verificare se la specifica è priva di *deadlock*, rispetta proprietà di *liveness* e *accessibilità*, oltre a proprietà più specifiche, ancor prima di effettuare il deploy su di un engine WS-BPEL.

Per predisporsi alla verifica l'utente dovrà semplicemente occuparsi della scrittura di una specifica B*lite* ben formata, senza preoccuparsi dei dettagli di traduzione in Prism, che sarà generata automaticamente da B*lite*2Prism. Effettuata la generazione l'utente caricherà la specifica generata in Prism e si concentrerà sulla scrittura delle proprietà che il modello dovrà rispettare.

È da evidenziare che la versione di B*lite*C che accompagna questa estensione è una versione migliorata rispetto a quella che accompagnava il precedente lavoro [8]. Le modifiche effettuate per migliorare B*lite*C sono molte, quindi ci limiteremo a citare solo quelle più importanti.

Rispetto alla precedente versione, B*lite*C supporta una sintassi delle specifiche in ingresso più semplice e chiara aggiungendo nuovi costrutti che completano la copertura della sintassi B*lite*. Grazie alla maggiore copertura dei costrutti B*lite* adesso è possibile definire più servizi per ogni file di specifica, evitando così di disperdere la definizione di uno stesso sistema in vari file separati. Oltre alle funzionalità accessibili dall'adozione di B*lite* sono state aggiunte funzionalità per il supporto ad XPath e per il *logging* di messaggi durante l'esecuzione di un processo da un engine.

Come per la precedente versione ogni specifica da tradurre in WS-BPEL necessita di una breve parte di configurazione, che in questa versione è stata estesa per supportare la definizione di più processi nella stessa specifica. Infatti ora è possibile definire più configurazioni nella stessa specifica abilitando così la compilazione di specifiche multiple.

La nuova versione estende, inoltre, la compatibilità degli engine supportati per il deploy da B*lite*C, infatti è adesso possibile eseguire i processi B*lite* anche sull'engine open source Apache ODE. Questa estensione rende B*lite*C una soluzione efficace al problema della portabilità dei processi WS-BPEL da un engine ad un altro.

Infine, B*lite*C è stato corredato con un *editor*, B*lite*Pad, per semplificare la stesura

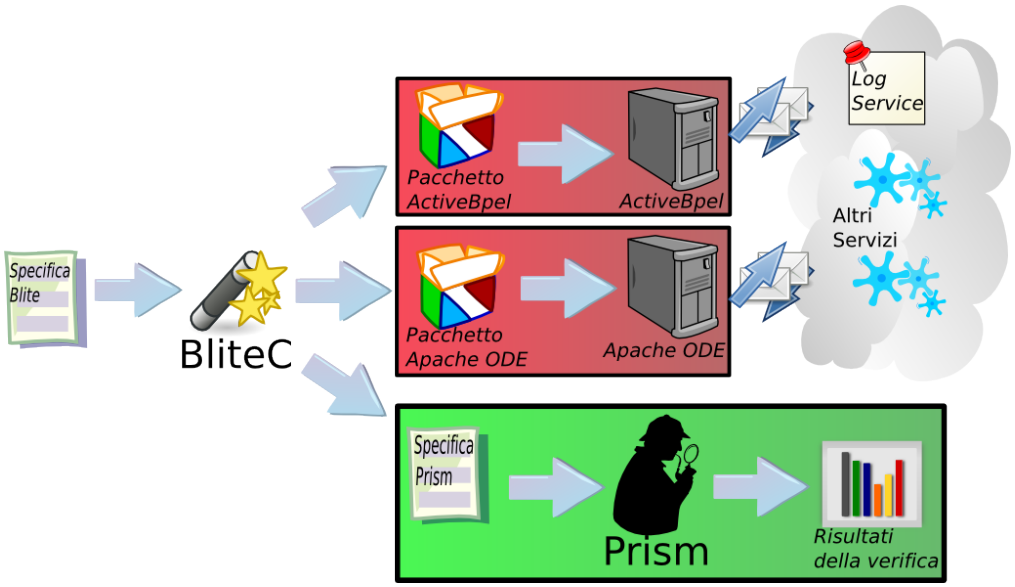


Figura 1.1: B*lite*C - funzionamento generale

del codice B*lite* e da un servizio che abilita il salvataggio dei messaggi di log inviati dalle specifiche B*lite* che utilizzano il costrutto *log*.

La tesi è stata organizzata come segue.
Nel Capitolo 2 sono introdotti i principali standard necessari alla definizione dei servizi web, una breve descrizione dei costrutti WS-BPEL utilizzati da B*lite*C, una descrizione del linguaggio B*lite* e una breve introduzione ai modelli stocastici usati nella tesi e relative nozioni preliminari. Nel Capitolo 3 è presentata una breve introduzione del funzionamento di B*lite*C. Nel Capitolo 4 è dettagliato il lavoro effettuato per predisporre B*lite*C alla generazione di specifiche Prism. Nel Capitolo 5 sono proposti alcuni esempi significativi per evidenziare l'utilità della traduzione e dall'analisi in Prism. Infine, il Capitolo 6 riporta alcune osservazioni conclusive.

Capitolo 2

Nozioni Preliminari

Nelle sezioni seguenti forniremo alcune nozioni preliminari necessarie alla comprensione del lavoro svolto, introdurremo i principali modelli stocastici ed infine presenteremo il model checker utilizzato nel lavoro.

2.1 Lo standard per l'orchestrazione di servizi web WS-BPEL

WS-BPEL (Web Service Business Process Execution Language [3]) è uno standard OASIS che permette di costruire processi di business utilizzando in modo trasparente servizi messi a disposizione da vari fornitori attraverso WSDL, attraverso un linguaggio basato su XML.

WSDL è un linguaggio standard W3C che permette di esprimere le funzionalità offerte da una risorsa, il metodo di invocazione e la risposta di quest'ultima senza dover entrare in dettagli relativi all'implementazione. Nella pratica WSDL permette di fornire un'interfaccia alle risorse messe a disposizione da un fornitore.

Ogni processo WS-BPEL sarà corredato da un documento WSDL per rendere note all'utilizzatore tutte le informazione necessarie all'utilizzo.

Il linguaggio WS-BPEL fornisce costrutti che permettono l'interazione con servizi web, l'esecuzione di attività in modo concorrente o sequenziale, la manipolazione di dati e la gestione del flusso, compreso il trattamento di casi particolari dell'esecuzione attraverso eccezioni e compensazioni.

Un programma WS-BPEL è chiamato *processo* ed è il risultato della composizione di attività messe a disposizione dal linguaggio. Ogni processo per comunicare con un servizio utilizza un *partner link*, che consiste in un'astrazione delle informazioni di comunicazione con un servizio, necessarie all'esigenza di astrazione del processo WS-

BPEL. Le definizioni di tipo dei partner link non possono essere dichiarate all'interno del processo, ma devono essere dichiarate nel WSDL che accompagna il processo.

Per la specifica di processo, il linguaggio WS-BPEL mette a disposizione due tipi di attività: quelle di *base* e quelle *strutturate*. Nella trattazione introdurremo le attività utilizzate da B*lite*C per la traduzione da B*lite* a WS-BPEL. Per una trattazione completa consigliamo la lettura della specifica WS-BPEL fornita da OASIS [3].

Le attività di base considerate sono introdotte di seguito.

L'attività <receive> attende un messaggio in input da parte di un clienti per una operazione fornita dal processo. La <receive> gioca un ruolo molto importante nel ciclo di vita di un processo, infatti insieme al costrutto <pick>, può istanziare un processo di business in WS-BPEL e inizializzare le variabili di correlazione attraverso le quali è possibile instradare i messaggi alle varie istanze dello stesso processo. Le precedenti operazioni devono essere corredate da un partner link, un'operazione e da una variabile.

L'attività <reply> è usata per rispondere a richieste precedentemente accettate attraverso un'attività <receive>. Questa operazione ha significato solo nella comunicazione request-response.

L'attività <invoke> permette al processo di contattare un servizio web per richiedere l'esecuzione di un'operazione.

L'attività <assign> copia valori compatibili tra una sorgente e una destinazione. L'attività <empty> non effettua nessuna attività. L'attività <exit> termina immediatamente l'istanza del processo che esegue l'attività. L'attività <throw> lancia un'eccezione indicando una condizione di errore. Le attività <assign>, <throw>, <exit> sono dette *short-lived activity*, in quanto hanno un tempo di esecuzione molto breve, e per questo saranno eseguite anche in caso di eccezioni.

Le attività strutturate, ovvero le attività che permettono di organizzare i precedenti costrutti in modo più articolato, sono introdotte di seguito.

L'attività <sequence> permette di definire una collezione di attività da eseguire sequenzialmente, il costrutto <flow> permette l'esecuzione delle attività indicate in modo concorrente. L'attività <while> è usata per ripetere l'esecuzione di una data attività fintanto che la condizione specificata è verificata. L'attività <if> permette l'esecuzione di un'attività tra due possibili, sulla base del risultato restituito dalla valutazione di una condizione. L'attività <pick> è utilizzata per attendere l'arrivo di un messaggio tra un insieme di messaggi possibili. Se uno di questi è ricevuto, l'attività ad esso associata è eseguita e le altre possibilità sono scartate. Quando l'attività selezionata termina, anche il costrutto <pick> è completato.

L'attività <scope> è utilizzata per definire attività annidate che condividono variabili e partner link, e hanno un trattamento specializzato per quanto riguarda le attività di

compensazione e fallimento. Ovviamente tutte le variabili definite all'interno del costrutto hanno solo valore locale. Un'attività di fallimento rappresenta l'insieme di operazioni che il processo effettuerà nel caso sia lanciata una eccezione, invece le attività di compensazione sono delle operazioni da effettuare nel caso tutte le altre terminino senza errori.

Per eseguire un processo dobbiamo utilizzare appositi *engine*. L'engine è un "interprete" che trasforma i costrutti astratti del linguaggio WS-BPEL in operazioni eseguibili da un elaboratore. Vista l'astrazione di un processo WS-BPEL, l'engine necessita di un insieme di documenti che gli permettano di configurare tutti i dettagli relativi alle interazioni con il "mondo esterno", come ad esempio i partner link. La fase nella quale l'engine effettua la combinazione delle informazioni tra i documenti di configurazione e il processo è detta fase di *deploy*.

Quando il deploy di un processo è effettuato, l'esecuzione è abilitata, ovvero è possibile creare un'*istanza* del processo. Un'istanza può essere creata solo attraverso un'attività di ricezione, che inizializzerà i valori delle variabili del processo. Ovviamente, la scelta di inizializzazione solo attraverso attività di ricezione è dovuta al contesto di lavoro del processo. Il processo infatti risulta agli utilizzatori come un servizio web, quindi, l'unica motivazione per l'istanziazione del processo è una richiesta da parte di un partner esterno.

Su di un engine, uno stesso processo può avere più istanze attive per poter servire più richieste ricevute dall'esterno. Eseguire più istanze però introduce un problema riguardo allo smistamento dei messaggi; infatti l'engine dovrà instradare ogni messaggio all'istanza giusta del processo. A questo problema si possono trovare varie soluzioni; la soluzione adottata da WS-BPEL, semplice ed elegante, consiste nella dichiarazione di parti del messaggio che permettano di distinguere un'istanza da un'altra in relazione al valore ivi contenuto. Questo sistema fornisce un metodo di *correlazione* sia per i messaggi entranti sia per quelli uscenti. Come per i partner link, le dichiarazioni delle parti di messaggio che fungeranno da chiave per la correlazione, dovranno essere incluse nel WSDL che accompagna il processo e non nella specifica del processo.

2.2 B*lite*: una variante "snella" di WS-BPEL

Il linguaggio B*lite* è una versione semplificata di WS-BPEL modellata sulle sue caratteristiche fondamentali, quali: partner link, terminazione dei processi, correlazione dei messaggi, transazioni a lungo termine e attività di compensazione. Non sono invece considerate alcune attività, quali timeout, attività di gestione di eventi e terminazione,

Attività di base	$\mathsf{b} ::= \mathsf{inv}\,\ell^i\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{x} := \mathsf{e}$ $\mid \mathsf{empty} \mid \mathsf{throw} \mid \mathsf{exit}$	invoke, receive, assegnazione empty, throw, exit
Attività strutturate	$\mathsf{a} ::= \mathsf{b} \mid \mathsf{if(e)\{a_1\}\{a_2\}} \mid \mathsf{while(e)\{a\}}$ $\mid \mathsf{a_1}\,;\mathsf{a_2} \mid \sum_{j\in J}\mathsf{rcv}\,\ell_j^r\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$ $\mid \mathsf{a_1} \mid \mathsf{a_2} \mid [\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	base, condizionale, iterazione sequenza, pick ($\lvert J\rvert > 1$) parallelo, scope
Attività di avvio	$\mathsf{r} ::= \mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \sum_{j\in J}\mathsf{rcv}\,\ell_j^r\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$ $\mid \mathsf{r}\,;\mathsf{a} \mid \mathsf{r_1} \mid \mathsf{r_2} \mid [\mathsf{r} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	receive, pick sequenza, parallelo, scope
Servizi	$\mathsf{s} ::= [\mathsf{r} \bullet \mathsf{a}_f] \mid \mu \vdash \mathsf{a} \mid \mu \vdash \mathsf{a}\,,\,\mathsf{s}$	definizione, instanza, multiset
Deployment	$\mathsf{d} ::= \{\mathsf{s}\}_\mathsf{c} \mid \mathsf{d_1} \| \mathsf{d_2}$	deployment, composizione

Tabella 2.1: Sintassi di B*lite*

dipendenze esplicite fra attività concorrenti e sofisticate forme di manipolazione dei dati; ciò permette di mantenere semplice e gestibile la specifica.

B*lite* fornisce una descrizione formale del deploy di servizi utilizzando partner link, definizioni di servizi e insiemi di correlazione. I ruoli che partecipano ad una interazione tra servizi sono indicati esplicitamente attraverso partner link e partner, mentre aspetti relativi al deploy, quali ad esempio i *binding* del documento WSDL associato, sono tralasciati. Questo è visibile nelle interazioni Request-Response dove i partner link indicano due partner perché la parte richiedente deve fornire una operazione di callback attraverso la quale il provider potrà rispondere. I partner link in comunicazioni One-Way, invece, sono indicati da un unico partner perché soltanto una delle due parti fornisce tutte le operazioni da utilizzare.

Oltre alle invocazioni asincrone, WS-BPEL fornisce un costrutto per invocazioni sincrone di un servizio remoto. Questo costrutto forza il chiamante ad attendere una risposta dal servizio invocato, che effettua una coppia di operazioni *receive-reply*. In B*lite* questo comportamento è simulato da una coppia di *invoke-receive* eseguite dal client e da una coppia di attività *receive-invoke* eseguite dal fornitore del servizio. Questo approccio non comporta nessuna perdita di espressività infatti le interazioni sincrone condividono il nome dell'operazione, quelle asincrone invece hanno nomi di operazioni diverse.

2.2.1 Sintassi

La sintassi di B*lite* è riportata in Tabella 2.1. I dati potranno essere condivisi attraverso *variabili* (x, x', ...). L'insieme di valori (v, v', ...) è stato volutamente lasciato non specificato, assumeremo comunque che includa perlomeno un insieme di *nomi di partner*

(p, q, ...) e un insieme di *nomi di operazioni* (o, o′, ...). Inoltre, con u indicheremo partner e variabili, e con w valori e variabili.

Le *Espressioni* (e, e′, ...) sono anch'esse lasciate non specificate, ma devono contenere almeno valori e variabili. La notazione $\bar{\cdot}$ è utilizzata per indicare tuple di oggetti, ad esempio $\bar{x}$ denota una tupla di variabili $\langle x_1, \ldots, x_h \rangle$ (con $h \geq 0$). Assumiamo che le variabili nella stessa tupla siano distinte due a due. La notazione speciale $\tilde{\cdot}$ indica tuple di uno o due oggetti, ad esempio $\tilde{p}$ denota $\langle p_1, p_2 \rangle$ oppure $\langle p_1 \rangle$. Le tuple possono essere costruite utilizzando l'operatore di concatenamento $\cdot : \cdot$, che ad esempio per $\langle p, u \rangle : \langle x_1, \ldots, x_h \rangle$ restituisce $\langle p, u, x_1, \ldots, x_h \rangle$.

Le attività di base sono le operazioni fondamentali per il funzionamento di un processo. L'invocazione $\mathsf{inv} \cdots$ è il costrutto di comunicazione necessario ad interrogare servizi; quest'attività deve essere accompagnata da un partner link necessario alla comunicazione e una variabile da cui estrarre i valori da inviare al servizio. I partner link ℓ^i utilizzati nelle operazioni di invoke possono essere sia $\langle u, p \rangle$ che $\langle u \rangle$, dove u è il partner che fornisce l'operazione e p è il partner utilizzato per riceve i messaggi di risposta. Il partner ricevente deve essere staticamente noto, e dunque non può essere una variabile.

La ricezione $\mathsf{rcv} \cdots$ è il costrutto che permette di ricevere messaggi da utilizzatori esterni, necessario anche all'istanziazione del processo. Come per il costrutto di invocazione anche questo necessita di un partner link per acquisire le informazioni sul canale di comunicazione e di una variabile su cui salvare i dati ricevuti. I partner link ℓ^r delle attività di ricezione possono essere sia $\langle p, u \rangle$ che $\langle p \rangle$, dove p è il partner che fornisce l'operazione ed u è il partner usato per spedire i messaggi di risposta. I partner per ricevere i messaggi devono essere noti al momento della scrittura del codice, invece i partner usati per rispondere possono essere determinati dinamicamente.

Le altre attività di base sono: l'assegnamento $\cdot := \cdot$, attività necessaria all'impostazione dei valori di una variabile; l'attività vuota empty è un'attività che funge da segnaposto; throw genera eccezioni attivando le attività di fallimento e compensazione; exit terminazione forzata permette di far terminare il processo con successo.

Le *attività strutturate* sono costrutti che organizzano in modo articolato le attività di base. La scelta condizionale $\mathsf{if}(\cdot)\{\cdot\}\{\cdot\}$, permette di eseguire delle attività in modo esclusivo, il ciclo $\mathsf{while}(\cdot)\,\{\cdot\}$ invece, permette di eseguire delle attività fin quando l'espressione della guardia risulta verificata.

La composizione sequenziale $\cdot\,;\cdot$, e la composizione parallela $\cdot \mid \cdot$ permettono rispettivamente di organizzare le attività di base in modo sequenziale o in modo parallelo. Daremo precedenza all'attività di sequenzializzazione rispetto all'attività di composizione e scelta esterna, ad esempio $a_1\,;a_2 \mid a_3\,;a_4$ sarà equivalente a $(a_1\,;a_2) \mid (a_3\,;a_4)$ e $a_1\,;a_2 + a_3$ equivarrà a $(a_1\,;a_2) + a_3$.

L'attività pick $\sum_{j\in J} \mathsf{rcv} \cdots ; \cdot$, permette di rimanere in attesa, riceve messaggi su una serie di partner link in attesa di una richiesta, permettendo però soltanto alla prima che è invocata di evolvere. Ovviamente, ogni attività di ricezione della pick è corredata con i parametri tipici dell'attività di ricezione.

L'attività di scope $[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c]$ raggruppa un'attività primaria a con un'attività di gestione dei fallimenti a_f e una di compensazione a_c. Ogni volta che all'interno di uno scope saranno omesse le attività di compensazione e di fallimento saranno rispettivamente inserite al loro posto una attività empty e una throw.

I servizi permettono di organizzare le *attività strutturate* in una specifica di servizio. Le *Attività di avvio* r invece, sono attività strutturate che inizialmente possono eseguire solo attività receive per permettere l'istanziazione del processo.

Le attività di *Deploy* sono composizioni finite di multinsiemi di *istanze* $\mu \vdash \mathsf{a}$, contenenti non più di una *definizione* $[\mathsf{r} \bullet \mathsf{a}_f]$ associata ad un insieme di *variabili di correlazione* c. Una definizione di servizio fornisce uno scope di "alto livello" (che non può essere compensato) e può offrire una scelta di ricezioni alternative su attività multiple di inizializzazione. Ogni istanza di servizio $\mu \vdash \mathsf{a}$ ha il suo stato (privato) μ. Uno stato è una funzione (parziale) che associa variabili in valori. Scriveremo $\{\mathsf{x} \mapsto \mathsf{v}\}$ per indicare che "la variabile x è associata al valore v". Lo stato ottenuto aggiornando μ con μ', scritto $\mu \circ \mu'$, è induttivamente definito da: $\mu \circ \mu'(\mathsf{x}) = \mu'(\mathsf{x})$ se $\mathsf{x} \in dom(\mu')$ (dove $dom(\mu)$ denota il dominio di μ) e $\mu(\mathsf{x})$ altrimenti. Lo stato vuoto è denotato da $\emptyset$.

Per ulteriori delucidazioni rimandiamo alla lettura dell'articolo [26].

2.2.2 Esempio: Carta Virtuale

La specifica B*lite* di un fornitore di carte virtuali ($\mathsf{d}_{virtualcard}$) e del possessore di una carta virtuale (d_{owner}) è riportata nel seguente esempio.

Gli attori che vengono composti nel sistema sono due $\mathsf{d}_{virtualcard}$ d_{owner}.

$$virtualPay \triangleq \quad \mathsf{d}_{virtualcard} \parallel \mathsf{d}_{owner}$$

Il deploy $\mathsf{d}_{virtualcard}$, presente in Figura 2.1 definisce un servizio fornitore carte prepagate che attende l'invio del numero di carta e la quantità di denaro da caricare sulla nuova carta prepagata; dopo questa operazione permette al cliente di effettuare prelievi fino all'esaurimento del credito.

Il deploy d_{owner} definisce un servizio cliente che invia una richiesta di creazione al fornitore di carte virtuali, poi effettua una serie di prelievi fino all'esaurimento del credito. Come si può vedere dalla Figura 2.2 il servizio è implementato attraverso l'ausilio di

$$
\begin{aligned}
&\mathsf{d}_{virtualcard} \triangleq \quad \{\mathsf{s}_{virtualcard}\}_{\mathsf{c}_{id}} \\
&\mathsf{s}_{virtualcard} \triangleq \quad [\\
&\qquad \mathsf{rcv} \ \ \langle \mathsf{p}_{createcard}, \mathsf{x}_{card} \rangle \ \mathsf{o}_{newcard} \ \langle \mathsf{x}_{id}, \mathsf{x}_{cash} \rangle \ ; \\
&\qquad \mathsf{if} \, (\mathsf{x}_{cash} \ > \ 0) \\
&\qquad \{ \\
&\qquad\quad \mathsf{x}_{resp} := \text{''}\mathit{Carta\ Virtuale\ n.}\ x_{id}\ \mathit{creata\ con\ successo}\text{''} \ ; \\
&\qquad\quad \mathsf{inv} \ \ \langle \mathsf{x}_{card} \rangle \ \mathsf{o}_{newcard} \ \langle \mathsf{x}_{id}, \mathsf{x}_{resp} \rangle \\
&\qquad \} \\
&\qquad \mathsf{else} \\
&\qquad \{ \\
&\qquad\quad \mathsf{x}_{resp} := \text{''}\mathit{Carta\ Virtuale\ n.}\ x_{id}\ \mathit{non\ creata.\ Credito\ Insufficiente}\text{''} \ ; \\
&\qquad\quad \mathsf{inv} \ \ \langle \mathsf{x}_{card} \rangle \ \mathsf{o}_{newcard} \ \langle \mathsf{x}_{id}, \mathsf{x}_{resp} \rangle \\
&\qquad \} \, ; \\
&\qquad \mathsf{while}(\mathsf{x}_{cash} > 0) \\
&\qquad \{ \\
&\qquad\quad \mathsf{rcv} \ \ \langle \mathsf{p}_{vcard}, \mathsf{x}_{clt} \rangle \ \mathsf{o}_{getcash} \langle \mathsf{x}_{id}, \mathsf{x}_{wdr} \rangle \, ; \\
&\qquad\quad \mathsf{if}(\mathsf{x}_{cash} \geq \mathsf{x}_{wdr}) \\
&\qquad\quad \{ \\
&\qquad\qquad \mathsf{x}_{cash} := \mathsf{x}_{cash} - \mathsf{x}_{wdr} \, ; \\
&\qquad\qquad \mathsf{x}_{resp} := \text{''}\mathit{Prelievo\ di}\ \mathsf{x}_{wdr}\ \mathit{Eur\ accettato}\text{''} \\
&\qquad\quad \} \\
&\qquad\quad \mathsf{else} \\
&\qquad\quad \{ \\
&\qquad\qquad \mathsf{x}_{resp} := \text{''}\mathit{Prelievo\ di}\ \mathsf{x}_{wdr}\ \mathit{Eur\ non\ accettato}\text{''} \\
&\qquad\quad \} \, ; \\
&\qquad\quad \mathsf{inv} \ \ \langle \mathsf{x}_{clt} \rangle \ \mathsf{o}_{getcash} \ \langle \mathsf{x}_{id}, \mathsf{x}_{resp} \rangle \\
&\qquad \} \\
&\quad]
\end{aligned}
$$

Figura 2.1: B*lite* - specifica del fornitore di carte prepagate

una istanza, in modo da avviare subito la creazione e non attendere un altro servizio per l'attivazione.

Nei precedenti esempi sono stati usati alcuni operatori addizionali che non fanno parte della specifica B*lite*. Questi sono l'operatore . che permette la composizione di stringhe e gli operatori di addizione + e sottrazione − che permettono di manipolare i valori presenti nelle variabili.

2.3 Concetti di base di teoria delle probabilità

La descrizione dei modelli stocastici necessita l'illustrazione di alcuni concetti di teoria della probabilità quali: variabili aleatorie; funzioni di distribuzione; processi stocastici.

Definizione 1. Si definisce *esperimento casuale C* ogni avvenimento (provocato diretta-

$$
\begin{array}{lll}
\mathsf{d}_{owner} & \triangleq & \{\mathsf{s}_{owner}\}_{\mathsf{c}_{id}} \\
\mathsf{s}_{owner} & \triangleq & \{\mathsf{x}_{id} \mapsto 42\}, \{\mathsf{x}_{cash} \mapsto 500\}, \{\mathsf{x}_{g} \mapsto 120\}, \{\mathsf{x}_{wdr} \mapsto 120\} \\
 & & \vdash \\
 & & [\\
 & & \quad \mathsf{inv}\ \langle \mathsf{p}_{createcard}, \mathsf{x}_{card} \rangle\ \mathsf{o}_{newcard}\ \langle \mathsf{x}_{id}, \mathsf{x}_{cash} \rangle\ ; \\
 & & \quad \mathsf{while}(\mathsf{x}_{g} \leq \mathsf{x}_{cash}) \\
 & & \quad \{ \\
 & & \quad\quad \mathsf{inv}\ \langle \mathsf{p}_{vcard}, \mathsf{x}_{clt} \rangle\ \mathsf{o}_{getcash}\ \langle \mathsf{x}_{id}, \mathsf{x}_{wdr} \rangle\ ; \\
 & & \quad\quad \mathsf{x}_{g} := \mathsf{x}_{g} + \mathsf{x}_{wdr} \\
 & & \quad \} \\
 & &]
\end{array}
$$

Figura 2.2: B*lite* - specifica del cliente del fornitore di Virtual Card

mente o indirettamente dall'uomo) suscettibile a manifestarsi secondo una pluralità di alternative o *eventi elementari*.

Definizione 2. Lo *spazio campionario* Ω è l'insieme dei possibili risultati di un esperimento casuale C. Indicheremo con *evento* un insieme $E \subseteq \Omega$ e con *evento elementare* $\omega \in \Omega$ un possibile risultato dell'esperimento.

L'evento corrispondente all'intero spazio campionario è definito come *evento certo*. L'evento corrispondente all'insieme vuoto, invece, è detto *evento impossibile*.

Definizione 3. Sia Ω uno spazio campionario non vuoto. Una famiglia $\mathcal{F}$ di eventi in Ω (una qualsiasi collezione di sottoinsiemi $F_i \in \Omega$) è detta *σ-algebra* se e solo se:

1. $\Omega \in \mathcal{F}$;
2. $F \in \mathcal{F} \Rightarrow \bar{F} \in \mathcal{F}$;
3. $F_i \in \mathcal{F}\ \forall i \in \mathbb{N} \Rightarrow \bigcup_{i=1}^{\infty} F_i \in \mathcal{F}$;

dove $\bar{F}$ denota il complementare di F in Ω (cioè $\bar{F} = \Omega - F$).

Gli elementi della σ-algebra sono chiamati *insiemi misurabili*. $(\Omega, \mathcal{F})$ è chiamato *spazio misurabile*.

Definizione 4. Dato uno spazio misurabile $(\Omega, \mathcal{F})$, una *misura di probabilità* su di esso è una funzione $\mathbb{P} : \mathcal{F} \to \mathbb{R}^{\geq 0}$ tale che:

1. $\mathbb{P}(\emptyset) = 0, \mathbb{P}(\Omega) = 1$
2. per ogni famiglia $\{F_i \mid F_i \in \mathcal{F}\} \subseteq \mathcal{F}$ tale che $k \neq h \Rightarrow F_k \cap F_h = \emptyset$ vale

$$\mathbb{P}\left(\bigcup_i F_i\right) = \sum_i \mathbb{P}(F_i)$$

Le precedenti proprietà sono chiamate “Assiomi di Kolmogorov”.

Definizione 5. Uno *spazio di probabilità* di un esperimento casuale C è la tupla $(\Omega, \mathcal{F}, \mathbb{P})$, dove:

- Ω è uno spazio campionario,
- $(\Omega, \mathcal{F})$ è uno spazio misurabile,
- $\mathbb{P}$ è una misura di probabilità.

Definizione 6. Dato $(\Omega, \mathcal{F}, \mathbb{P})$ con $E, F \in \mathcal{F}$ tale che $\mathbb{P}(F) > 0$, definiremo *probabilità condizionata* di E al verificarsi di F

$$\mathbb{P}(E \mid F) = \frac{\mathbb{P}(E \cap F)}{\mathbb{P}(F)}$$

Dalla precedente formula possiamo dedurre

$$\mathbb{P}(E|F) \cdot \mathbb{P}(F) = \mathbb{P}(E \cap F) = \mathbb{P}(F|E) \cdot \mathbb{P}(E)$$

Definizione 7. Due eventi E e F, sono *stocasticamente indipendenti* se e solo se:

$$\mathbb{P}(E \cap F) = \mathbb{P}(E) \cdot \mathbb{P}(F)$$

Definizione 8. Siano $(\Omega_1, \mathcal{F}_1)$ e $(\Omega_2, \mathcal{F}_2)$, due spazi misurabili. Una applicazione $f : \Omega_1 \to \Omega_2$ viene detta *funzione misurabile* se la controimmagine di ogni elemento di $\mathcal{F}_2$ è in Ω_1, ossia se f^{-1} trasforma insiemi misurabili di Ω_2 in insiemi misurabili di Ω_1.

Definizione 9. Dato uno spazio campionario Ω su cui è definita una misura di probabilità $\mathbb{P}$, una *variabile aleatoria* è una funzione misurabile dallo spazio campionario a uno spazio misurabile.

Una variabile aleatoria, in pratica, è una variabile i cui valori dipendono da un insieme di esperimenti casuali.

Definizione 10. La *funzione di distribuzione* F_X di una variabile aleatoria X è definita come la funzione

$$F_X(x) = \mathbb{P}(X \leq x).$$

Dove con $\mathbb{P}(X \leq x)$ intendiamo la probabilità che la variabile aleatoria X assuma un valore minore o uguale ad x.
Nel caso di variabile aleatoria discreta avremo

$$F_X(x_n) = \sum_{i=0}^{n} \mathbb{P}(X = x_i)$$

dove $\mathbb{P}(X = x_i)$ è la probabilità che X assuma il valore x_i.

Nel caso di variabile aleatoria continua avremo invece:

$$F_X(x) = \int_0^x f(t)dt,$$

dove la funzione $f(x)$, la derivata di $F_X(x)$, è chiamata *densità di probabilità* della variabile aleatoria X.

2.4 Modelli Probabilistici

Definizione 11. Un *processo stocastico* è una famiglia $\{X(t)\}$ di variabili aleatorie indicizzate sul parametro $t \in I$ e definite su uno stesso spazio campionario, dove

- t ha il significato di tempo,
- I di intervallo di tempo,
- lo spazio campionario denota l'insieme dei possibili valori (o stati) di $X(t)$.

Definizione 12. Un processo stocastico è chiamato *processo di Markov* se lo stato successivo dipende solo dallo stato precedente. Più formalmente

$$\mathbb{P}\left(X(t_k) = x_k \mid X(t_0) = x_1, X(t_1) = x_1, \ldots, X(t_{k-1}) = x_{k-1}\right) = P\left[X(t_k) = x_k \mid X(t_{k-1}) = x_{k-1}\right]$$

per

- $k \in \mathbb{N}$,
- $t_0, \ldots, t_k \in I$ con $t_0 < t_1 < \ldots < t_k$
- $x_0, \ldots, x_k$ appartenenti ad un qualche spazio di campionamento.

Un processo di Markov con spazio campionario discreto è chiamato *catena di Markov* (MC). In funzione al tipo di comportamento rispetto al tempo possiamo parlare di:

- Catene di Markov a Tempo Discreto (DTMC)
- Catene di Markov a Tempo Continuo (CTMC)

2.4.1 Catene di Markov a Tempo Discreto (DTMC)

In questa sezione definiremo in dettaglio le DTMC e introdurremo una logica temporale probabilistica per verificare proprietà su queste ultime.

Definizione 13. Una DTMC è una tupla $\mathcal{D} = (S, s_0, P)$ dove:

- S è l'insieme degli stati;
- s_0 è lo stato iniziale;
- $P : S \times S \to [0, 1]$ è la *matrice di probabilità delle transizioni* tale che:

$$\sum_{s' \in S} \mathbb{P}(s, s') = 1$$

  per tutti gli stati $s \in S$.

Ogni elemento $P(s, s')$ della matrice di probabilità delle transizioni indica la probabilità di effettuare una transizione dallo stato s allo stato s', cioè per ogni $k \geq 0$:

$$P(s, s') = \mathbb{P}\left(X(t_k) = s' \mid X(t_{k-1}) = s\right)$$

Gli stati terminali, ovvero gli stati dai quali il sistema non può più evolvere in un altro stato, possono essere modellati aggiungendo un "cappio" (cioè una transizione che ha come partenza e arrivo lo stesso stato).

Definizione 14. Una DTMC *etichettata* è una tupla (S, s_0, P, L) dove:

- (S, s_0, P) è una DTMC;
- $L : S \to 2^{AP}$ è una *funzione di etichettamento* che mappa ogni stato in un insieme di proposizioni atomiche appartenenti all'insieme AP.

Cammini e misure di probabilità

Un'esecuzione di un sistema modellato con una DTMC è rappresentata da un *cammino*. Formalmente, un cammino ω è una sequenza non vuota di stati $s_0 s_1 s_2 \ldots$ dove $s_i \in S$ e $P(s_i, s_{i+1}) > 0$ per tutti gli $i \geq 0$. Un cammino può essere sia finito che infinito.

Denoteremo con $|\omega|$ la lunghezza di ω (numero di transizioni) e con $\omega(i)$ l'i-esimo stato del cammino ω se $i \leq |\omega|$, e infine, per un cammino finito ω_{fin}, l'ultimo stato sarà indicato con $last(\omega_{fin})$. Diremo che un cammino finito ω_{fin} di lunghezza n è un *prefisso* di un cammino infinito ω se $\omega_{fin}(i) = \omega(i)$ per $0 \leq i \leq n$. L'insieme di tutti i cammini finiti ed infiniti con nodo iniziale s sono denotati rispettivamente da $Path_s^{fin}$ e $Path_s$. Se

non specificato esplicitamente i cammini utilizzati nella trattazione sono da intendere infiniti.

Per studiare il comportamento probabilistico di una DTMC dobbiamo determinare la probabilità che un preciso cammino sia scelto. Questo può essere effettuato definendo, per ogni stato $s \in S$ una misura di probabilità $Prob_s$ sull'insieme $Path_s$. La misura di probabilità è indotta dalla *matrice di probabilità delle transizioni P* come segue: per ogni cammino finito $\omega_{fin} \in Path_s^{fin}$, definiamo la probabilità

$$P_s(\omega_{fin}) = \begin{cases} 1 & se\ n = 0 \\ P(\omega(0), \omega(1)) \cdot \ldots \cdot P(\omega(n-1), \omega(n)) & altrimenti \end{cases}$$

dove $n = |\omega_{fin}|$.
Dato un cammino ω_{fin}, l'*insieme cilindro* $C(\omega_{fin})$ è definito come:

$$C(\omega_{fin}) \stackrel{def}{=} \left\{\omega \in Path_s \mid \omega_{fin}\ prefissodi\ \omega\right\}$$

e rappresenta l'insieme di tutti i cammini infiniti con prefisso ω_{fin}.
Sia $\sum_s$ la più piccola σ-algebra su $Path_s$ che contiene tutti gli insiemi $C(\omega_{fin})$, dove $\omega_{fin} \in Path_s^{fin}$. Definiremo la misura di probabilità $Prob_s$ su $\sum_s$ come l'unica misura per cui:

$$Prob_s(C(\omega_{fin})) = P_s(\omega_{fin}) \quad \forall \omega_{fin} \in Path_s^{fin}$$

Con queste basi possiamo quantificare la probabilità che una DTMC etichettata si comporti come voluto identificando l'insieme dei cammini che soddisfano la specifica e, assumendo che tale insieme sia misurabile, usando la misura di probabilità associata $Prob_s$.

Probabilistic Computation Tree Logic (PCTL)

Per scrivere proprietà per i DTMC etichettati, useremo PCTL [16] (Probabilistic Computation Tree Logic), un'estensione probabilistica della logica temporale CTL [10].

Definizione 15. La sintassi di PCTL è la seguente:

$$\begin{aligned} \phi &::= \ true \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}[\psi] \\ \psi &::= \ X\phi\ (Next) \\ &\quad \mid \phi \mathcal{U} \phi\ (Until) \\ &\quad \mid \phi \mathcal{U}^{\leq k} \phi\ (BoundedUntil) \end{aligned}$$

dove a è una proposizione atomica, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0, 1]$ e $k \in \mathbb{N}$.

Nella presentazione della sintassi sono state distinte le *formule di stato* ϕ dalle *formule di cammino* ψ perché le prime sono valutate sugli stati e le seconde sui cammini. Per specificare una proprietà di una DTMC useremo sempre una formula di stato: le formule di cammino occorrono solo come parametro dell'operatore $\mathcal{P}_{\bowtie p}[\psi]$. Intuitivamente uno stato s soddisfa $\mathcal{P}_{\bowtie p}[\psi]$ quando la probabilità di prendere un cammino da s che soddisfi ψ è nell'intervallo specificato da $\bowtie p$.

Per uno stato s e una formula ϕ, scriveremo $s \vdash \phi$ per indicare che s soddisfa ϕ. In modo analogo per un cammino ω che soddisfi una formula di cammino ψ scriveremo $\omega \vdash \psi$.
La semantica di PCTL sulle DTMC è la seguente.

Definizione 16. Sia $\mathcal{D} = (S, s_0, P, L)$ una DTMC etichettata. Per ogni stato $s \in S$, la relazione di soddisfacibilità $\vdash$ è definita induttivamente da:

$$\begin{aligned}
&s \vdash true && \forall s \in S,\\
&s \vdash a && \Leftrightarrow a \in L(s),\\
&s \vdash \neg\phi && \Leftrightarrow s \nvdash \phi,\\
&s \vdash \phi_1 \wedge \phi_2 && \Leftrightarrow s \vdash \phi_1 \wedge s \vdash \phi_2\\
&s \vdash \mathcal{P}_{\bowtie p}[\psi] && \Leftrightarrow p_s(\psi) \bowtie p,
\end{aligned}$$

dove

$$p_s(\psi) = Prob_s\left(\{\omega \in Path_s \mid \omega \vdash \psi\}\right).$$

Per ogni cammino $\omega \in Path_s$:

$$\begin{aligned}
&\omega \vdash X\phi && \Leftrightarrow \omega(1) \vdash \phi,\\
&\omega \vdash \phi_1 \mathcal{U}^{\leq k} \phi_2 && \Leftrightarrow \exists i \leq k \mid \omega(i) \vdash \phi_2 \wedge (\omega(j) \vdash \phi_1 \forall j < i),\\
&\omega \vdash \phi_1 \mathcal{U} \phi_2 && \Leftrightarrow \exists k \geq 0 \mid \omega \vdash \phi_1 \mathcal{U}^{\leq k} \phi_2.
\end{aligned}$$

Dalla sintassi di base possiamo derivare alcuni operatori addizionali:

$$\begin{aligned}
&false \equiv \neg true,\\
&\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2),\\
&\phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2.
\end{aligned}$$

Attraverso l'operatore "until" e "bounded until" possiamo esprimere gli operatori $\Diamond$ e $\Diamond^{\leq k}$ ("diamond" o "eventually"), comuni nelle logiche temporali. Intuitivamente $\Diamond\phi$ significa che ϕ prima o poi verrà soddisfatto; la variante vincolata $\Diamond^{\leq k}\phi$ significa che ϕ dovrà essere soddisfatta in k unità di tempo.

$$\Diamond\phi \equiv true\mathcal{U}\phi,$$
$$\Diamond^{\leq k}\phi \equiv true\mathcal{U}^{\leq k}\phi.$$

Un altro operatore logico temporale comune è $\Box$ ("box" o "always"). Se un cammino soddisfa $\Box\phi$, allora ϕ dovrà essere soddisfatta da ogni stato del cammino. In modo analogo, la variante vincolata $\Box^{\leq k}\phi$ significa che ϕ è vera per i primi k passi del cammino. È possibile esprimere $\Box$ in termini di $\Diamond$ nel seguente modo:

$$\Box\phi \equiv \neg\Diamond\neg\phi,$$
$$\Box^{\leq k}\phi \equiv \neg\Diamond^{\leq k}\neg\phi.$$

L'operatore PCTL $\mathcal{P}_{\bowtie p}[\psi]$ può essere considerato la versione probabilistica del corrispondente operatore temporale di CTL. Ad esempio, la formula PCTL $\mathcal{P}_{\bowtie p}[\Diamond\phi]$, che verifica se la probabilità di raggiungere uno stato che soddisfi ϕ è $\bowtie p$, risulta molto simile alle formule CTL $\forall\Diamond\phi$ e $\exists\Diamond\phi$. Infatti, esse asseriscono rispettivamente che *tutti* i cammini raggiungono o *almeno un* cammino raggiunge uno stato che soddisfa ϕ.

In PCTL non è possibile determinare la probabilità attuale con cui una formula è soddisfatta, ma solo quando è (o non è) raggiunto un certo vincolo: questo è necessario per assicurare che ogni formula sia valutata come Booleana. Nella pratica, questo vincolo può essere rilassato se l'operatore più esterno della formula è $\mathcal{P}_{\bowtie p}$. In tal caso possiamo omettere il vincolo $\bowtie p$ e computare la probabilità, dato che l'algoritmo per il model checking di formule PCTL procede computando la probabilità attuale e confrontandola con il vincolo; l'operazione non necessita di alcun lavoro addizionale.

2.4.2 Catene di Markov a Tempo Continuo (CTMC)

Le *Catene di Markov a Tempo Continuo* sono una estensione delle DTMC. Mentre in una DTMC le transizioni avvengono ad intervalli di tempo discreti, in una CTMC le transizioni avvengono in tempo continuo. Formalmente una CTMC è una famiglia di variabili aleatorie $\{X(t)|t \geq 0\}$ dove $X(t)$ è una osservazione effettuata al tempo t con $t \in \mathbb{R}$.

Lo spazio degli stati, cioè l'insieme di tutti i possibili valori di $X(t)$, è discreto e sarà assunto come finito. Una CTMC deve soddisfare la proprietà di Markov, che in questo caso si può esprimere come segue: per ogni $k > 0$, sequenza di intervalli di tempo $t_0 < t_1 < \ldots < t_k$ e stati $s_0, \ldots, s_k$

$$\mathbb{P}[X(t) = s_k | X(t_{k-1}) = s_{k-1}, \ldots, X(t_0) = s_0] = \mathbb{P}[X(t_k) = s_k | X(t_{k-1}) = s_{k-1}]$$

Questo significa che la probabilità di effettuare una transizione verso uno stato dipende solo dallo stato attuale e non dalle precedenti transizioni. Inoltre, la probabilità è

indipendente dal tempo passato nello stato corrente.

La sola distribuzione di probabilità continua che rispetta la proprietà di Markov è quella esponenziale, dunque nelle CTMC anziché associare ad ogni coppia di stati una probabilità, assegneremo un *rate* λ, che useremo come parametro della distribuzione esponenziale.

La probabilità di effettuare una transizione tra una coppia di stati al tempo t sarà quindi $1 - e^{-\lambda \cdot t}$. Intuitivamente il rate λ rappresenta il numero medio di volte che una transizione potrà occorrere per unità di tempo.

Definizione 17. Una CTMC è una tupla $(S, s_0, \mathcal{R})$ dove:

- S è un insieme finito di stati;
- s_0 è lo stato iniziale;
- $\mathcal{R} : S \times S \to \mathbb{R}^{\geq 0}$ è la *matrice dei rate di transizione*.

La matrice dei rate di transizione $\mathcal{R}$ assegna il rate ad ogni coppia di stati in una CTMC. Una transizione può occorrere solo tra stati s ed s' per i quali $\mathcal{R}(s, s') > 0$.

Se per uno stato s, ci sono più stati s' per i quali $\mathcal{R}(s, s') > 0$, si ricade in una situazione conosciuta come *race condition*. La determinazione dello stato successivo della CTMC è effettuata selezionando la transizione più veloce tra quelle attive (cioè con rate non nullo).

Il tempo passato in uno stato s prima che una qualunque transizione occorra è esponenzialmente distribuito con rate $E(s)$ dove:

$$E(s) \stackrel{def}{=} \sum_{s' \in S} \mathcal{R}(s, s')$$

$E(s)$ è conosciuto come *exit rate* dello stato s. È possibile determinare anche la probabilità attuale di ogni stato s', indipendentemente dal tempo.

Definizione 18. Una *embedded DTMC emb*(C) di una CTMC $C = (S, s_0, \mathcal{R})$ è una DTMC (S, s_0, P) con, $\forall s, s' \in S$,

$$P(s, s') = \begin{cases} \mathcal{R}(s, s')/E(s) & \textit{se } E(s) \neq 0 \\ 1 & \textit{se } E(s) = 0 \wedge s = s' \\ 0 & \textit{altrimenti.} \end{cases}$$

Nella CTMC uno stato può non avere nessun arco uscente: questi stati sono detti *assorbenti*.

Usando la precedente definizione, possiamo considerare il comportamento di una CTMC in un modo alternativo: la CTMC rimane in uno stato s per un tempo che è esponenzialmente distribuito con rate $E(s)$ e dopo effettua una transizione; la probabilità che la transizione sia verso s' è determinata da $P(s, s')$.

Definiremo adesso anche un'altra matrice usata per effettuare analisi sulle CTMC.

Definizione 19. La *matrice generatrice infinitesimale* per una CTMC $(S, s_0, \mathcal{R})$ è la matrice $Q : S \times S \to \mathbb{R}$ definita come:

$$Q(s, s') = \begin{cases} \mathcal{R}(s, s') & se\ s \neq s' \\ -\sum_{s'' \neq s} \mathcal{R}(s, s'') & altrimenti. \end{cases}$$

Per analizzare le CTMC le annoteremo con informazioni addizionali: useremo una funzione di etichettamento L, che mapperà gli stati in insiemi di proposizioni atomiche appartenenti ad un insieme AP.

Definizione 20. Una *CTMC etichettata* è una tupla (S, s_0, R, L) dove:

- $(S, s_0, \mathcal{R})$ è una CTMC;
- $L : S \to 2^{AP}$ è una funzione di etichettamento.

Cammini e misure di probabilità

Un cammino in una CTMC è una sequenza non vuota $s_0 t_0 s_1 t_1 \ldots$ tale che $\mathcal{R}(s_1, s_{i+1}) > 0$ e $t_i \in \mathbb{R}_{>0}$, $\forall i \geq 0$. Il valore di t_i rappresenta la quantità di tempo passato nello stato s_i. Come nelle DTMC denoteremo con $\omega(i)$ l'i-esimo stato s_i del cammino ω.
Indicheremo con $time(\omega, i)$ la quantità di tempo t_i passato nello stato s_i e con $\omega@t$ lo stato occupato al tempo t. Più formalmente $\omega@t$ seleziona $\omega(k)$ se k è il più piccolo indice per cui $\sum_{i=0}^{k} t_i > t$.
Denoteremo con $Path_s$ l'insieme di tutti i cammini che partono dallo stato s.

Se gli stati $s_0, \ldots, s_n \in S$ soddisfano $\mathcal{R}(s_i, s_{i+1}) > 0$ per tutti $0 \leq i < n$ e $I_0, \ldots, I_{n-1}$ sono intervalli non vuoti in $\mathbb{R}^{\geq 0}$, allora l'*insieme cilindro* $C(s_0, I_0, \ldots, I_{n-1}, s_n)$ è definito come l'insieme di tutti i cammini infiniti $s'_0 t_0 s'_i t_1 s'_2 \ldots$ dove $s_i = s'_i$ per $i < n$ e $t_i \in I_i$ per $i < n$.

Sia $\sum_s$ la più piccola σ-algebra su $Path_s$ che contenga tutti gli insiemi cilindro $C(s_0, I_0, \ldots, s_{n-1}, I_{n-1}, s_n)$, dove $s_0, \ldots, s_{n-1}, s_n \in S$ sono tutte le sequenze degli stati con $s_0 = s$, $\mathcal{R}(s_i, s_{i+1}) > 0$, per $0 \leq i < n$, e $I_0, \ldots, I_{n-1}$ intervalli non vuoti su $\mathbb{R}_{\geq 0}$.

La misura di probabilità $Prob_s$ su $\sum_s$ è la misura unica definita induttivamente da

$$\begin{aligned} Prob_s(C(s_0)) = & \quad 1 \\ Prob_s(C(s_0, I_0 \ldots, s_n, I_n, s_{n+1})) = & \quad Prob_s(C(s_0, \ldots, s_n)) \cdot P(s_n, s_{n+1}) \\ & \quad \cdot (e^{-E(s_n) \cdot inf I_n} - e^{-E(s_n) \cdot sup I_n}) \end{aligned}$$

con P matrice di transizione della embedded DTMC associata alla CTMC, $supI_n$ l'estremo superiore dell'intervallo I_n e $infI_n$estremo inferiore dell'intervallo I_n L'ultima espressione indica il carattere induttivo della definizione, infatti la probabilità associata a $Prob_s(C(s_0, I_0 \ldots, s_n, I_n, s_{n+1}))$ dipende dal valore della probabilità associata a:

- l'insieme cilindro contenente il cammino prefisso a s_n,
- la probabilità di effettuare una transizione dallo stato s_n a s_{n+1},
- la probabilità di effettuare la transizione da s_n a s_{n+1} nell'intervallo I_n.

Comportamento transiente e steady-state

In aggiunta alle probabilità di cammino, considereremo altre due proprietà ben note nelle CTMC: il comportamento *transiente* (la probabilità di trovarsi in uno stato del modello in un particolare istante) e il comportamento *steady-state* (il comportamento descritto dal modello a lungo termine).

La proprietà transiente $\pi_{s,t}(s')$ è definita come la probabilità, partendo da s, di trovarsi nello stato s' nell'istante di tempo t. Formalmente

$$\pi_{s,t}(s') \stackrel{def}{=} Prob_s(\{\omega \in Path_s \mid \omega@t = s'\})$$

La probabilità steady-state $\pi_s(s')$ è la probabilità di arrivare in s' partendo da s a lungo termine. Più formalmente

$$\pi_s(s') \stackrel{def}{=} \lim_{t \to \infty} \pi_{s,t}(s').$$

La distribuzione di probabilità steady-state può essere usata per inferire la percentuale di tempo che la CTMC passa in ogni stato nel lungo termine. Per le classi di CTMC che considereremo, il precedente limite esiste sempre.

Se la CTMC è *irriducibile*, ovvero ha un cammino che collega qualunque stato con qualunque altro stato, la probabilità steady-state $\pi_s(s')$ è indipendente dallo stato iniziale s.

Continuous Stochastic Logic (CSL)

Per specificare le proprietà da verificare sulle CTMC utilizzeremo la logica CSL [6, 7], una versione estesa della logica PCTL usata per le DTMC. Essa permette di analizzare sia proprietà di stato e cammino, sia proprietà di comportamento transiente e steady state.

Definizione 21. La logica PCTL è definita come segue:

$$\phi ::= true \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}[\psi] \mid \mathcal{S}_{\bowtie p}[\phi]$$
$$\psi ::= X\phi \mid \phi \mathcal{U}^I \phi \mid \phi \mathcal{U} \phi$$

dove a è una proposizione atomica, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0,1]$ e I è un intervallo in $\mathbb{R}^{\geq 0}$.

Le formule di cammino sono simili a quelle in PCTL, eccetto che per il parametro della versione vincolata dell'operatore "until" che, in questo caso, deve essere contenuto in un intervallo $I \in \mathbb{R}^{\geq 0}$.

La formula di cammino $\phi_1 \mathcal{U}^I \phi_2$ è soddisfatta se ϕ_2 è soddisfatta in qualche istante nell'intervallo I e ϕ_1 è soddisfatta in tutti gli istanti precedenti. Ci riferiremo all'operatore come "time-bounded until".

L'operatore $\mathcal{S}$ descrive il comportamento steady-state della CTMC. La formula $\mathcal{S}_{\bowtie p}[\phi]$ asserisce che la probabilità steady-state di uno stato soddisfacente ϕ rispetta il vincolo $\bowtie p$.

Per CSL, come per PCTL, scriveremo $s \vdash \phi$ per indicare che una formula ϕ è soddisfatta in uno stato s di una CTMC e indicheremo con $Sat(\phi)$ l'insieme $\{s \in S \mid s \vdash \phi\}$. In modo analogo, per una formula di cammino ψ soddisfatta dal cammino ω, scriveremo $\omega \vdash \psi$. La semantica di CSL sulle CTMC è la seguente.

Definizione 22. Sia $C = (S, s_0, \mathcal{R}, L)$ una CTMC etichettata. Per ogni stato $s \in S$ la relazione $s \vdash \phi$ è definita induttivamente da:

$$\begin{aligned}
&s \vdash true && \forall s \in S;\\
&s \vdash a && \Leftrightarrow a \in L(s);\\
&s \vdash \neg\phi && \Leftrightarrow s \nvdash \phi;\\
&s \vdash \phi_1 \wedge \phi_2 && \Leftrightarrow s \vdash \phi_1 \wedge s \vdash \phi_2;\\
&s \vdash \mathcal{P}_{\bowtie p}[\psi] && \Leftrightarrow p_s(\psi) \bowtie p;\\
&s \vdash \mathcal{S}_{\bowtie p}[\phi] && \Leftrightarrow \textstyle\sum_{s' \vdash \phi} \pi_s(s') \bowtie p.
\end{aligned}$$

dove

$$p_s(\psi) \stackrel{def}{=} Prob_s(\{\omega \in Path_s \mid \omega \vdash \psi\}).$$

Per ogni $\omega \in Path_s$:

$$\begin{aligned}
\omega \vdash X\phi \quad & \Leftrightarrow \omega(1)\ definito \wedge \omega(1) \vdash \phi; \\
\omega \vdash \phi_1 \mathcal{U}^I \phi_2 \quad & \Leftrightarrow \exists t \in I \mid (\omega@t \vdash \phi_2 \wedge \omega@x \vdash \phi_1, \forall x \in [0,t)); \\
\omega \vdash \phi_1 \mathcal{U} \phi_2 \quad & \Leftrightarrow \exists k \geq 0 \mid (\omega(k) \vdash \phi_2 \wedge \omega(j) \vdash \phi_1, \forall j < k).
\end{aligned}$$

Possiamo facilmente derivare gli operatori CSL per $\vee$ e $\rightarrow$, oltre a dedurre gli operatori temporali $\diamond$ e $\square$:

$$\begin{aligned}
\mathcal{P}_{\bowtie p}\left[\diamond\phi\right] \quad & \equiv \mathcal{P}_{\bowtie p}\left[true\mathcal{U}\phi\right]; \\
\mathcal{P}^I_{\bowtie p}\left[\diamond^I\phi\right] \quad & \equiv \mathcal{P}_{\bowtie p}\left[true\mathcal{U}^I\phi\right]; \\
\mathcal{P}_{\bowtie p}\left[\square\phi\right] \quad & \equiv \mathcal{P}_{\bar{\bowtie} 1-p}\left[\diamond\neg\phi\right]; \\
\mathcal{P}_{\bowtie p}\left[\square^I\phi\right] \quad & \equiv \mathcal{P}_{\bar{\bowtie} 1-p}\left[\diamond^I\neg\phi\right];
\end{aligned}$$

Anche se CSL non include esplicitamente operatori per analizzare il comportamento transiente, il seguente operatore può essere usato per controllare la probabilità di soddisfare una formula ϕ in un instante t:

$$\mathcal{P}_{\bowtie p}\left[\diamond^{[t,t]}\phi\right].$$

Come per PCTL, quando l'operatore più esterno di una formula CSL è $\mathcal{P}_{\bowtie p}$ o $\mathcal{S}_{\bowtie p}$, possiamo omettere il vincolo $\bowtie p$ ed otterremo il valore effettivo associato alla probabilità.

2.5 Prism Model Checker

Prism è un model checker probabilistico, ovvero uno strumento per modellare ed analizzare sistemi che agiscono in modo casuale o probabilistico. Lo strumento permette di utilizzare i modelli probabilistici DTMC, CTMC e MDP (Markov Decision Process), oltre a permettere l'estensione di questi con strutture di costo e premio.

I modelli sono descritti tramite un semplice linguaggio a stati e possono essere analizzati automaticamente da Prism utilizzando una delle seguenti logiche:

- PCTL;
- CSL;
- LTL;
- PCTL*.

Inoltre, come per i modelli, le logiche sono estendibili con verifiche di proprietà quantitative di *reward*.

I file contenenti la descrizione del modello possono includere commenti se questi sono preceduti dai caratteri `//` in stile C. Per convenzione, l'estensione del file contenente la descrizione del modello deve essere: nm (MDP), pm (DTMC) o sm (CTMC).

Gli strumenti da noi realizzati sfruttano la capacità di Prism di generare modelli CTMC e di testarli tramite la logica CSL.

2.5.1 Linguaggio di specifica

Una specifica in Prism è definita tramite un semplice linguaggio a stati ispirato al formalismo dei Moduli Reattivi di Alur e Henzinger [4], e ciò indipendentemente dal tipo di modello probabilistico utilizzato.
I componenti fondamentali del linguaggio sono i *moduli* e le *variabili*. Un *modello* è composto da uno o più *moduli* che possono interagire tra loro.
Un modulo contiene delle *variabili locali*, i cui valori definiscono in ogni istante lo stato del modulo. Lo *stato globale* del modello è definito dallo *stato locale* di tutti i moduli e dai valori contenuti nelle variabili globali (se presenti).

Il comportamento di ogni modulo è regolato da un insieme di *comandi*.

Tipi di modello

Per indicare il tipo di modello da utilizzare si dovrà includere una delle parole chiave **dtmc**, **ctmc** o **mdp**. La parola chiave è, generalmente, posta all'inizio del file, ma può occorrere in qualsiasi posizione del file (eccetto all'interno dei moduli e in altre dichiarazioni). Se non è inserita nessuna parola chiave, il modello è assunto di tipo MDP.

Moduli

Un modulo è specificato tramite le seguenti parole chiave:

Listing 2.1: Modulo

```
module name ... endmodule
```

La definizione dei moduli si divide in due parti:

- variabili;
- comandi.

Variabili

Le variabili possono essere intere e Booleane. Nel Listing 2.2 è, ad esempio, definita una variabile intera con intervallo [0..2] e valore iniziale 0.

Listing 2.2: Variabile Intera

```
x : [0..2] init 0;
```

Possono essere utilizzate anche variabili Booleane, come nel Listing 2.3.

Listing 2.3: Variabile Booleana

```
b : bool init false;
```

I nomi associati ai moduli e alle variabili possono essere riferiti come *identificatori*. Gli identificatori possono essere composti da lettere, dal carattere _ e da numeri, ma non possono iniziare con un numero, né utilizzare nomi riservati alle parole chiave.

Stati iniziali

Lo spazio degli stati di un modello probabilistico è l'insieme di tutte le possibili valutazioni delle sue variabili. La modalità più semplice per la specifica dello stato iniziale è quella di inserire il valore iniziale della variabile al momento della sua dichiarazione: se il valore iniziale è omesso dalla dichiarazione, allora è selezionato il valore minimo dell'intervallo. Questo approccio permette di avere un solo stato iniziale globale.

Una alternativa alla precedente modalità è il costrutto `init ... endinit`, che permette di costruire modelli con stati iniziali multipli definendo un predicato su tutte le variabili del modulo: ogni stato che soddisfa la condizione è uno stato iniziale.
Ad esempio potremmo ottenere la stessa inizializzazione del Listing 2.2 utilizzando il costrutto `init` come si vede nel Listing 2.4.

Listing 2.4: Costrutto init

```
init x=0 endinit
```

Alternativamente potremmo dichiarare una condizione più complessa come quella nel Listing 2.5

Listing 2.5: Inizializzazione Complessa

```
init x+y=1 endinit
```

Variabili globali

In aggiunta alle variabili locali appartenenti a singoli moduli, un modello può includere anche *variabili globali* che possono essere lette e scritte da tutti i moduli.

Come per le variabili locali, i tipi sono interi e Booleani; la dichiarazione avviene in modo analogo, con la differenza che la dichiarazione delle variabili globali deve avvenire al di fuori della specifica dei moduli. Alcuni esempi di dichiarazioni sono riportate nel Listing 2.6.

Listing 2.6: Variabili Globali

```
global g : [1..10];
global b : bool init true;
```

Un'importante restrizione sull'uso delle variabili globali è che i comandi di sincronizzazione non possono modificarle.

Comandi

Il comportamento di ogni modulo è descritto attraverso *comandi*, formati da una guardia e da uno o più aggiornamenti.

Listing 2.7: Comando

```
[] guard -> prob_1 : update_1 + ... + prob_n : update_n;
```

La *guardia* è un predicato su tutte le variabili del modello: ovvero possono essere usate sia variabili globali, sia variabili contenute in qualsiasi modulo. Gli *aggiornamenti* descrivono le transizioni che il modulo può effettuare se la guardia è soddisfatta. Una transizione è specificata fornendo un nuovo valore per le variabili del modulo. Ogni aggiornamento è contraddistinto da probabilità (nel caso delle DTMC) o da rate (nel caso delle CTMC) che saranno assegnate alle corrispondenti transizioni.
Ad esempio il comando nel Listing 2.8 descrive il comportamento del modulo quando la guardia `x=0` sarà soddisfatta, ovvero quando la variabile `x` sarà uguale a 0. Gli aggiornamenti `0.8:(x'=0)` e `0.2:(x'=1)` e le loro probabilità associate indicano che il valore di `x` rimarrà 0 con probabilità 0.8 e diverrà 1 con probabilità 0.2 (`x'` indica il valore di `x` dopo l'aggiornamento).

Listing 2.8: Comando

```
[] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
```

In caso di modelli DTMC e MDP è necessario che le probabilità nella parte destra del comando abbiano somma 1.

Un altro esempio di comando è quello riportato nel Listing 2.9, che illustra la capacità di inserire in una guardia vincoli su ogni variabile, anche non appartenente al modulo di definizione: è così possibile modellare il funzionamento di un modulo in base al comportamento di un altro modulo. Inoltre, il comando evidenzia che, quando un aggiornamento ha probabilità 1, la specifica della probabilità può essere omessa.

Listing 2.9: Comando vincolato

```
[] x=1 & y!=2 -> (x'=2);
```

Se il modulo ha due variabili `x1` e `x2`, un possibile comando di aggiornamento multiplo potrebbe essere quello nel Listing 2.10.

Listing 2.10: Comando di aggiornamento multiplo

```
[] x1=0 & x2>0 & x2<10 ->
            0.5:(x1'=1)&(x2'=x2+1)
            +
            0.5:(x1'=2)&(x2'=x2-1);
```

In caso di aggiornamenti multipli, gli elementi da aggiornare devono essere inseriti tra parentesi singolarmente e devono essere concatenati con il simbolo `&`. Se un aggiornamento non comprende alcune variabili del modulo è assunto che queste non cambino valore.

Un caso speciale di aggiornamento è la parola chiave `true` che denota un aggiornamento dove nessuna variabile è modificata.

Composizione Parallela

Il modello probabilistico corrispondente alla specifica è costruito come *composizione parallela* dei moduli.

In ogni stato del modello è presente l'insieme dei comandi attivati, ovvero i comandi appartenenti a tutti i moduli la cui guardia è soddisfatta dallo stato attuale. La scelta del comando da eseguire dipende dal tipo di modello: nei modelli MDP la scelta è *non deterministica*; nei DTMC la scelta è definita dalle distribuzioni associate; nei CTMC la scelta è probabilistica.

Non tutti i modelli permettono l'utilizzo del non-determinismo locale: le MDP e le DTMC ne ammettono l'uso, a differenza delle CTMC che modellano la situazione come una *race condition*, dove si sceglie la transizione più veloce tra quelle attive.

Ridenominazione dei moduli

La *ridenominazione* permette la duplicazione dei moduli. Un esempio è contenuto nel Listing 2.11, dove è dichiarato il modulo `M2` identico al modulo `M1` usando l'operatore di ridenominazione.

Listing 2.11: Ridenominazione

```
module M2 = M1 [ x=k, y=z ] endmodule
```

La ridenominazione è effettuata a *livello testuale*, quindi ogni identificatore usato nel modulo (`x` e `y` in questo caso) deve essere rimpiazzato con un nuovo nome non utilizzato nella specifica (`k` e `z`).

Costanti

Le costanti possono essere di tipo intero, double o Booleane. Il valore di una costante può essere definito con un literal oppure attraverso espressioni tra costanti.

La dichiarazione avviene nella specifica usando la parola chiave `const` (vedi Listing 2.12).

Listing 2.12: Costanti

```
const int radius = 12;
const double pi = 3.141592;
const double area = pi * radius * radius;
const bool yes = true;
```

Gli identificatori delle costanti sono soggetti alle stesse regole di quelli delle variabili.

Le costanti possono essere utilizzate in ogni contesto dove sia richiesto un valore: il limite superiore o inferiore di una variabile, il valore associato ad un aggiornamento oppure la condizione di una guardia.

Espressioni

Le espressioni in Prism possono contenere valori literal (ad esempio, 42, 3.141592, `true`, `false`), variabili, alcune funzioni built-in (che saranno introdotte nel seguito) e gli operatori della seguente lista:

- - (meno unario)
- *, / (moltiplicazione, divisione)

- +, - (addizione, sottrazione)
- <, <=, >=, > (operatori di relazione)
- =, != (operatori di uguaglianza)
- ! (negazione)
- & (congiunzione)
- | (disgiunzione)
- => (implicazione)
- ? (valutazione della condizione: condizione ? a : b significa “se la condizione è vera allora a altrimenti b”)

Tutti gli operatori, tranne ?, sono valutati da sinistra verso destra e la loro precedenza è l’ordine di occorrenza nella lista precedente.

È da notare che l’operatore di divisione / effettua sempre operazioni in virgola mobile. Ad esempio la divisione 22/7 restituirà 3.142857 e non 3.

Funzioni Built-in

Le espressioni possono utilizzare alcune funzioni built-in:

- `min(...)` e `max(...)` selezionano il minimo e il massimo tra due o più valori;
- `floor(x)` e `ceil(x)` approssimano rispettivamente per eccesso e per difetto;
- `pow(x,y)` calcola `x` elevato alla `y`;
- `mod(i,n)` calcola il modulo intero, cioè *i mod n*;
- `log(x,b)` calcola il logaritmo in base `b` di `x`.

Alcuni esempi di utilizzo sono nel Listing 2.13.

Listing 2.13: Funzioni Built-in

```
min(x+1, x_max)
max(a,b,c)
floor(13.5)
ceil(13.5)
pow(2, 8)
```

```
pow(9.0, 0.5)
mod(1977, 100)
log(123, 2.71828183)
```

Uso delle espressioni

Le espressioni posso essere utilizzate nel linguaggio di specifica come:

- definizione delle costanti;
- limite superiore o inferiore di un intervallo;
- valori iniziali per le variabili;
- guardie;
- probabilità e rate;
- aggiornamenti.

Ad esempio, le espressioni permettono alla probabilità di un aggiornamento di dipendere dallo stato attuale del modulo (vedi Listing 2.14).

Listing 2.14: Aggiornamento con funzioni Built-in

```
[] (x>=1 & x<=10) -> x/10 : (x'=max(1,x-1)) + 1-x/10 : (x'=min(10,x
    +1))
```

Sincronizzazione

La *sincronizzazione* è definita, come in molte algebre di processo, etichettando i comandi tramite *azioni* e combinando i moduli con una composizione parallela tipo CSP [17] in cui i moduli si sincronizzano su tutte le azioni comuni.

Le azioni sono poste all'interno delle parentesi quadre che marcano l'inizio di un comando, come è possibile vedere nel Listing 2.15.

Listing 2.15: Comando di sincronizzazione

```
[serve] q>0 -> lambda:(q'=q-1);
```

Le azioni possono essere usate per forzare due o più moduli ad effettuare transizioni simultaneamente (ossia a sincronizzarsi).

Attraverso il costrutto `system ... endsystem` è definibile la composizione dei moduli. Tale costrutto dovrà essere posto alla fine della descrizione del modello e dovrà inglobare al suo interno una espressione contenente tutti i moduli. I moduli possono comparire nell'espressione esattamente una volta e possono essere composti tramite i seguenti operatori (simili a quelli forniti da CSP):

- `M1 || M2` : sincronizza solo le operazioni comuni tra `M1` e `M2`;
- `M1 ||| M2` : composizione parallela asincrona di `M1` e `M2` (nessuna sincronizzazione, esecuzione completamente "interleaved");
- `M1 |[a,b,...]| M2` : restrizione sulla composizione dei moduli `M1` e `M2` (sincronizzazione solo sulle azioni dall'insieme $\{a, b, ...\}$);
- `M / {a,b,...}` : hiding delle azioni $\{a, b, ...\}$ nel modulo M;
- `M {a<-b,c<-d,...}` : ridenominazione delle azioni a in b, c in d, etc. nel modulo `M`.

I primi due operatori `||` e `|||` sono associativi e possono essere applicati a più di due moduli alla volta. Quando si valuta un'espressione, gli operatori di hiding e ridenominazione risultano più forti dei tre operatori di composizione parallela.
Esempi di espressioni sono:

- `(station1 ||| station2 ||| station3)| [serve] | server`;
- `( P1 | [a] | P2)/ a)|| Q`;
- `( ( P1 | [a] | P2)a<-b )| [b] | Q`.

Se nessuna composizione parallela è specificata dall'utente, si assume implicitamente una espressione della forma `M1 || M2 || ...` contenente tutti i moduli del modello.

Formule ed Etichette

I modelli possono includere *formule* per evitare la duplicazione del codice. Una formula è composta da un nome (identificatore) e da un'espressione. Il nome della formula può essere usato come abbreviazione dell'espressione associata ovunque quest'ultima sia accettata. Un esempio di formula è presente nella prima linea del Listing 2.16, la seconda linea presenta un utilizzo della formula dichiarata ed infine la terza linea rappresenta lo stesso comando ma senza l'ausilio della formula.

Listing 2.16: Formule

```
formula num_tokens = q1+q2+q3+q4+q5;
[] p1=2 & num_tokens=5 -> (p1'=4);
[] p1=2 & (q1+q2+q3+q4+q5)=5 -> (p1'=4);
```

Le formule possono essere utilizzate anche nella specifica di proprietà.

Durante la lettura del modulo, l'espansione delle formule è effettuata prima dell'operazione di ridenominazione del modulo, così sarà ridenominato solo il contenuto della formula e non la formula stessa.

I moduli Prism possono anche contenere *etichette*, che identificano un insieme di stati di particolare interesse. Le etichette possono essere usate per specificare proprietà ed essere definite sia nel modello, sia nelle proprietà da verificare.
Le etichette differiscono dalle formule perché:

- devono essere di tipo Booleano;
- devono essere scritte utilizzando le virgolette (`" ... "`).

Alcuni esempi di etichetta sono riportati nel Listing 2.17.

Listing 2.17: Etichette

```
label "safe" = temp<=100 | alarm=true;
label "fail" = temp>100 & alarm=false;
```

Esempio di specifica in Prism

Nel Listing 2.18 possiamo trovare un esempio di un semplice sistema Prism formato da due processi che devono operare in mutua esclusione.

Listing 2.18: Prism - Esempio di due processi in mutua eclusione

```
formula num_tokens = q1+q2+q3+q4+q5;

module M1

    x : [0..2] init 0;

    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);
```

```
endmodule

module M2

    y : [0..2] init 0;

    [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
    [] y=1 & x!=2 -> (y'=2);
    [] y=2 -> 0.5:(y'=2) + 0.5:(y'=0);

endmodule
```

I processi si possono trovare in uno dei tre stati: 0, 1, 2. Ogni processo può muoversi dallo stato 0 allo stato 1 con probabilità 0.2 e rimanere nello stesso stato con probabilità 0.8. Dallo stato 1 un processo può provare ad accedere alla sezione critica, stato 2, quando l'altro processo non la sta occupando. Infine, dallo stato 2, un processo può decidere di rimanere nella sezione critica o lasciarla in modo equiprobabile.

Capitolo 3

B*lite*C

B*lite*C è uno strumento di supporto alla specifica e al deploy di servizi WS-BPEL.

La versione di B*lite*C utilizzata in questo elaborato è un'evoluzione del precedente lavoro [8], infatti oltre alle modifiche apportate per l'estensione descritta in questa tesi, B*lite*C è stato modificato per aggiungere le seguenti caratteristiche:

- permettere il deploy di processi eseguibili anche su Apache ODE [5];
- supportare un servizio di log che permetta al processo di inviare messaggi durante la sua esecuzione;
- supportare XPath [9];
- rendere più semplice e strutturata la scrittura del codice tramite un linguaggio in ingresso modificato opportunamente.

In questo capitolo descriveremo brevemente il contesto del lavoro di B*lite*C: introdurremo la nuova sintassi accettata, le funzionalità avanzate fornite dallo strumento ed infine daremo una breve descrizione del suo utilizzo.

3.1 Sintassi accettata da B*lite*C

Il linguaggio accettato da B*lite*C è una versione estesa di B*lite*, che mette a disposizione costrutti per effettuare operazioni di basso livello come, ad esempio, utilizzare XPath per estrarre o impostare valori nelle variabili.

3.1.1 B*lite*C basics

Una specifica accettata da B*lite*C è formata da una serie di dichiarazioni di termini associati ad un identificatore.

Listing 3.1: Esempio di identificatore

```
id ::= blite_construct ;;
```

Ogni identificatore può essere associato a una delle seguenti azioni:

- deploy e composizione;
- start activity;
- structured activity;
- service activity.

L'identificatore potrà essere utilizzato in ogni posizione della specifica al posto del blocco di codice associato. In questo modo si evita la duplicazione di codice e si rende la specifica più leggibile e strutturata. L'unico vincolo imposto è che la specifica contenga la dichiarazione di tutti gli identificatori utilizzati in un qualunque ordine.
Un esempio è visibile nel seguente Listing 3.2.

Listing 3.2: Esempio di specifica

```
OneWayPartners ::= OneWayServer || OneWayClient ;;

OneWayServer ::= {OwServerService}{x_id};;

OneWayClient ::= {OwClientService}{x_id};;

OwServerService ::=
[
  seq
    rcv <OW_server> o_req <x_id>;
    empty
  qes
];;

OwClientService ::=
[
    seq
      rcv <init_owclient> o_init <x_id>;
      inv <OW_server> o_req <x_id>
```

```
    qes
];;
```

Deploy, Composizione e Istanze

B*lite*C permette all'utente di definire servizi in due modalità:

- singolo processo;
- processi multipli.

Per creare un solo processo è sufficiente definire un deploy, come mostrato nel Listing 3.3.

Listing 3.3: Esempio di Deploy

```
OneWayServer ::= {OwServerService}{x_id};;
```

Nel caso di specifica multipla dovrà essere definito un deploy per ogni servizio; inoltre i deploy da compilare dovranno esseri posti all'interno di un costrutto di composizione.

Ogni specifica deve contenere al più un costrutto di composizione, come mostrato nel Listing 3.4

Listing 3.4: Esempio di Composition

```
OneWayPartners ::= OneWayServer || OneWayClient ;;
OneWayServer ::= {OwServerService}{x_id} ;;
OneWayClient ::= {OwClientService}{x_id} ;;
```

Nella versione più recente di B*lite*C è possibile definire anche istanze, ovvero servizi WS-BPEL che rappresentano un processo già istanziato. Al momento il traduttore non supporta la compilazione di questi costrutti su di un motore WS-BPEL, il cui sviluppo è stato lasciato per il futuro.

Listing 3.5: Esempio di istanza

```
OwClientService ::=
let
   x_id := 42;
in
[
   inv <OW_server> o_req <x_id>
];;
```

Un'istanza si definisce attraverso l'utilizzo delle parole chiave `let ... in`, al cui interno saranno definiti i valori associati alle variabili utilizzate nel codice B*lite* seguente alla parola chiave `in`. Il costrutto fornisce un metodo per evitare l'utilizzo di una attività di ricezione per l'istanziazione di un processo; le istanze posso essere utilizzate, ad esempio, per simulare il contatto di un servizio da parte di un utente umano.

Gestione della compensazione e del fallimento

Ogni processo deve dichiarare almeno uno scope, vedi Listing 3.6 dove si definiscono le attività di fallimento e compensazione, come si vede nell'esempio seguente.

Listing 3.6: Esempio di Scope

```
Service ::=
[
 default_act,
 failure_act,
 compensation_act
];;
```

Per comodità dell'utente, il traduttore permette di lasciare non dichiarate le attività di fallimento e/o compensazione: queste saranno completate nella traduzione rispettivamente dal costrutto `throw` e da un costrutto `empty`. Vedi Listing 3.7.

Listing 3.7: Esempio di scope senza azioni di fallimento e compensazione

```
OwServerService ::=
[
 seq
    rcv <OW_server> o_req <x_id>;
    empty
 qes
];;
```

Sequenzializzazione e parallelo

Per eseguire attività in sequenza si devono utilizzare le parole chiave `seq ... qes` e inserire al loro interno le attività da sequenzializzare. Le attività nel corpo del costrutto devono essere separate da `;`, come si vede nel Listing 3.8.

Listing 3.8: Esempio di sequenzializzazione

```
seq
    act1;
```

```
    act2;
    ...
    actN
qes
```

In modo analogo alla sequenza, l'operatore parallelo si definisce con le parole chiave `flw ... wlf` (da "flow", nome dell'operatore di parallelo in WS-BPEL). Le attività da eseguire in parallelo dovranno essere separate dal carattere `|` (vedi Listing 3.9).

Listing 3.9: Esempio di parallelo

```
flw
     act1
    |
     act2
    |
     ...
    |
     actN
wlf
```

Condizionale e Iterazione

I costrutti per il controllo del flusso presenti in B*lite*C sono usati in modo analogo a quelli definiti da B*lite*.

L'operatore di scelta *condizionale* è definibile utilizzando l'usuale costrutto `if (espressione)... [else ... ]`. Un esempio di scelta condizionale è presente nel Listing 3.10.

Listing 3.10: Esempio di condizionale

```
if ( espressione )
{
    act1
    ...
}
else
{
    actN
    ...
}
```

L'operatore di *iterazione*, nel Listing 3.11, è caratterizzato dalla parola chiave `while` ed è utilizzabile in modo analogo ad altri linguaggi di programmazione.

Listing 3.11: Esempio di iterazione

```
while ( espressione )
{
    act1
    ...
}
```

Receive e Invoke

Un comando di interazione receive o invoke, vedere Listing 3.12, deve essere corredato da un partner link, un nome di operazione e da una variabile in cui salvare il contenuto ricevuto o da cui estrarre i dati da inviare.

Listing 3.12: Esempio di Receive ed Invoke

```
rcv <pl> op <var>
inv <pl> op <var>
```

Configurazione

Per poter compilare una specifica B*lite* in un processo WS-BPEL eseguibile dobbiamo fornire al traduttore alcune informazioni non deducibili dalla specifica stessa.

Nel Listing 3.13 forniamo un esempio di configurazione per commentarlo in dettaglio nel seguito.

Listing 3.13: Esempio di configurazione

```
<?blm

c1@OneWayClient
::
ADDRESSES {
  myns => "http://example";
  myaddress =>"http://localhost:8080";
  dep => OneWayServer;
}

IMPORTS { pl  => "http://example/OneWayServer.wsdl"; }

VARIABLES { <x_id> => pl:req; }

PARTNERLINKS {
  PARTNERLINK {
```

```
    TYPE  => pl:OW_serverPLT;
    PARTNER_ROLE OW_server => pl:OW_serverPT, (o_req => o_req);
  }
}
::

c2@OneWayServer
::
ADDRESSES {
  myns => "http://example";
  myaddress =>"http://localhost:8080/active-bpel/services/";
}

VARIABLES {
  <x_id> => gen:req,
            <id>,
            <xsd:integer>;
  }
::

?>
```

Le configurazioni, in B*lite*C, devono essere incluse all'interno delle parole chiave `<?blm ... ?>`, nel file contenente la specifica B*lite*.

Ogni configurazione è contraddistinta dal nome del processo cui la configurazione è associata. Ogni configurazione deve essere associato ad una specifica ancora non associata a nessun'altra configurazione.

Dopo l nome del processo cui la configurazione è associata vi sono i simboli `:: ... ::` al cui interno si deve sviluppare la parte dichiarativa.

Nel corpo della configurazione si trovano:

- informazioni sul nuovo servizio;
- definizioni dei namespace usati dal servizio;
- definizione e/o importazione delle variabili;
- definizione delle costanti;
- importazione dei partner link.

Informazioni sul nuovo servizio

Le informazioni generali riguardanti il nuovo servizio sono racchiuse nel corpo del blocco `ADDRESS`, vedi Listing 3.14. Questa parte di configurazione è obbligatoria perché contenente informazioni fondamentali per la generazione del processo da parte dello strumento.

Le informazioni obbligatorie da inserire all'interno di questo blocco sono due:

- la base del namespace da usare per il servizio;
- la base dell'indirizzo dove il servizio è reperibile.

Oltre a queste informazioni è possibile definire un campo opzionale `log` che indica l'indirizzo del servizio di log da usare nella compilazione del servizio (questo argomento verrà trattato in dettaglio nella sezione seguente).

Listing 3.14: Esempio di sezione ADDRESSES

```
ADDRESESS
{
    myns => "base_address";
    myaddresses => "base_address";
    log => "address";
}
```

Definizione dei namespace

Nella definizione di nuovi servizi spesso è necessario importare dati da namespace già esistenti. Per permettere queste importazioni esiste nella parte di configurazione una sezione chiamata `IMPORTS`. In questa sezione è associata una chiave ad ogni namespace, per permetterne l'utilizzo nelle definizioni successive, vedere Listing 3.15.

Listing 3.15: Esempio di sezione IMPORTS

```
IMPORTS
{
    ns1 => "address";
    ns2 => "address";
}
```

Come indicato nel precedente Listing, a sinistra del simbolo => è posta la chiave e a destra il namespace associato.

Nella definizione delle chiavi non può essere usata la chiave `gen` perché riservata alla generazione automatica dei tipi da B*lite*C. I namespace standard come quello di XML Schema sono importati automaticamente e non necessitano di dichiarazione.

Questa sezione non è obbligatoria e al suo interno possono occorrere un numero qualsiasi di definizioni.

Definizione e/o importazione delle variabili

Per utilizzare variabili nella specifica B*lite* è necessario associare a ciascuna di esse una dichiarazione, nella sezione `VARIABLES` della configurazione, che espliciti il tipo di dato contenuto.
Questa sezione è obbligatoria perché ogni servizio usa almeno una variabile.
In questa sezione possono essere definite due tipologie di variabili:

- variabili di processo;
- messaggi.

Le *variabili di processo* non sono tuple e quindi possono essere utilizzate dal processo WS-BPEL solo come locazione di appoggio per attività, ad esempio a sinistra di un assegnamento. La descrizione delle *variabili di processo* è effettuata inserendo il nome della variabile WS-BPEL a sinistra dell'operatore per la dichiarazione di tipo (=>) e ponendo a destra un tipo di dato definito nello XML Schema da utilizzare, vedere Listing 3.16.

Listing 3.16: Esempio di Variabile

```
x_test1 => xsd:integer;
```

I *messaggi* invece sono tuple e, oltre alle precedenti funzionalità, possono essere utilizzati dal processo B*lite* sia come locazione di salvataggio in ricezione, sia come fonte di dati per l'invio.

Listing 3.17: Esempio di Messaggio

```
<x_test2> => gen:aID, <test>, <xsd:integer>;
<x_imp1, x_imp> => pl:aImp;
```

La descrizione dei *messaggi* è diversa dalla precedente perché possiede due diverse modalità: una per impostare l'autogenerazione del messaggio e una per l'importazione del messaggio da un namespace già esistente.
Per utilizzare la funzione di *autogenerazione* (vedi prima linea del Listing 3.17) è necessario inserire a sinistra dell'assegnazione il nome della variabile e a destra il tipo del messaggio, definendo nel namespace "gen" il nome delle parti del messaggio e i relativi tipi.

L'importazione da un namespace già esistente (vedi seconda linea del Listing 3.17) utilizza solo due campi: uno per il nome e uno per il tipo.

Definizione di Costanti

In un processo WS-BPEL è possibile associare ad una variabile un *literal*, ossia una costante definita dall'utente al momento della stesura del codice. Per aggiungere queste assegnazioni anche in B*lite*, senza perdere la semplicità di lettura del codice, è stato deciso di definire queste costanti nel file di configurazione per poi richiamarle nel codice attraverso un nome. Vedere il Listing 3.18 per un esempio.

Listing 3.18: Esempio di sezione LITERALS

```
  lit => [[ <wsa:EndpointReference xmlns:wsa="http://schemas.xmlsoap.org/ws
      /2003/03/addressing" xmlns:dyn="http://example/AsyncServer/AsyncServer.
      wsdl">
            <wsa:Address>http://localhost:8080/active-bpel/services/
                custAService</wsa:Address>
            <wsa:PortType>dyn:custAPT</wsa:PortType>
            <wsa:ServiceName PortName="custAServicePort">dyn:custAService</
                wsa:ServiceName>
          </wsa:EndpointReference> ]];
}
```

Le costanti possono essere definite all'interno della sezione *LITERALS* attraverso l'assegnazione di un nome al codice XML racchiuso tra `[[` e `]]`.

Importazione dei Partner Link

I partner link sono necessari al processo WS-BPEL per interagire con servizi esterni. Nel codice B*lite* i partner link sono associati ad un simbolo che necessita di ulteriori informazioni per la traduzione in WS-BPEL.
La sezione che permette questa associazione è *PARTNERLINKS*: nel corpo della sezione devono comparire tutti i partner link utilizzati dal processo B*lite*, vedi Listing 3.19.

Listing 3.19: Esempio di sezione PARTNERLINKS

```
PARTNERLINKS {
  PARTNERLINK {
    TYPE => pl:srvAPLT;
    MY_ROLE custA => pl:custAPT;
    PARTNER_ROLE srvA => pl:srvAPT, (Request => o_req);
  }
```

```
}
```

`PARTNERLINK` contiene le informazioni di un partner link. I campi obbligatori della definizione sono:

- `TYPE`: definisce il namespace e il tipo del partner link che sarà utilizzato;
- `MY_ROLE` (campo opzionale): definisce il tipo di porta (a destra dell'assegnazione) e il nome del partner link di ricezione (a sinistra dell'assegnazione) nel codice B*lite*;
- `PARTNER_ROLE`: come il precedente, definisce il tipo di porta (a destra dell'assegnazione) e il nome del partner link dell'invocazione (a sinistra dell'assegnazione).

In tutti i campi di definizione dei ruoli può essere presente un terzo argomento che associa un nome B*lite* (a sinistra) ad una operazione definita nel documento WSDL da importare (a destra). Nel caso non sia presente, sono importate tutte le operazione con i nomi definiti nel documento associato al namespace.

3.1.2 Costrutti aggiuntivi

Presentiamo in questa sezione alcuni costrutti che fanno parte della sintassi di B*lite*C, ma non sono parte della sintassi B*lite*.

Xpath

Il supporto ad XPath è stato inserito per permettere la gestione di messaggi complessi e per rendere più potente la gestione dei dati. Questo abilita l'interazione fra un processo WS-BPEL generato da una specifica B*lite* con un qualunque servizio web disponibile in rete.

Nel caso si debba estrarre un valore legato ad una espressione XPath dovremo usare il costrutto `get`. Questo operatore può comparire alla destra di un'assegnazione e si aspetta due argomenti: la variabile su cui eseguire l'espressione e l'espressione XPath da eseguire (si veda l'esempio nel Listing 3.20).

Listing 3.20: Esempio di funzione get

```
x := get(respWeath, "/wth:GetCityWeatherByZIPResponse/wth:
    GetCityWeatherByZIPResult/wth:Description");
```

In caso di impostazione di un campo selezionato da una espressione XPath si dovrà usare l'operatore `set`. L'operatore deve comparire alla sinistra di una assegnazione e si aspetta, come l'operatore `get`, la variabile su cui eseguire l'espressione e l'espressione XPath (si veda il Listing 3.21)

Listing 3.21: Esempio di funzione set

```
set(respWeath,"/wth:GetCityWeatherByZIPResponse/wth:
    GetCityWeatherByZIPResult/wth:Description"):= "My description";
```

Servizio di Log

Il servizio di log e il rispettivo comando `log "string"` sono stati introdotti per permettere un debug più semplice dei processi WS-BPEL generati, che nella maggior parte dei casi non dispongono di un metodo intuitivo per fornire messaggi all'utente.

Per attivare questa funzionalità l'utente dovrà semplicemente effettuare il deploy del servizio fornito insieme a B*lite*C su un'installazione di Axis e inserire l'indirizzo di tale servizio nella parte di configurazione delle specifiche B*lite* che lo utilizzeranno (vedi esempio nel Listing 3.22).

Listing 3.22: Esempio configurazione Log

```
...
ADDRESESS
{
    ...
    log => "address";
    ...
}
...
```

Dopo queste operazioni di inizializzazione l'utente potrà includere la parola chiave `log` all'interno della specifica per poter evidenziare particolari stati o valori del processo, come nell'Esempio 3.23.

Listing 3.23: Esempio di utilizzo del servizio di log

```
...
log "This is an example!"
...
```

La traduzione in WS-BPEL del costrutto `log` è molto semplice, infatti viene trasformato in una invocazione one-way dell'operazione di logging del servizio di log. Tutti i dati necessari a questa operazione sono trasparenti all'utente, infatti quest'ultimo deve solo inserire l'indirizzo del servizio. Per accedere ai messaggi di di log inviati dal processo, l'utente dovrà contattare il servizio di log richiedendo l'invio dei messaggi ricevuti.

3.2 Esempio d'uso: gestione di un package

In questa sezione introduciamo brevemente una sessione di esecuzione di B*lite*C in modo da presentare in modo chiaro il processo di utilizzo dello strumento.

Per poter eseguire con successo la compilazione e la relativa esecuzione, l'utente deve disporre del seguente software:

- Java 1.6;
- B*lite*C; [1]
- Tomcat;
- ActiveVOS e/o Apache ODE;
- Axis (opzionale, per esecuzione del log service)
- un browser web.

3.2.1 Creazione

Per creare il pacchetto relativo all'engine, l'utente dovrà fornire un parametro che indichi il tipo di motore da utilizzare: `activebpel` o `apacheode`.

Il comando da eseguire per la creazione del pacchetto è quello nel Listing 3.24.

Listing 3.24: Comando di compilazione semplice

```
$ java -jar blitec.jar -o engine_name esempio.bl
```

Il precedente comando creerà il/i pacchetto/i relativo/i alla specifica nella directory contenente il file di specifica (file con estensione .bl). Nel caso l'utente voglia specificare una directory diversa basterà aggiungere l'opzione -t e fornirgli come argomento la directory desiderata. In questo caso il comando sarà quello nel Listing 3.25.

Listing 3.25: Comando di compilazione avanzato

```
$ java -jar blitec.jar -o engine_name -t user_defined_directory
    esempio.bl
```

[1] B*lite*C è disponibile in formato eseguibile presso `http://rap.dsi.unifi.it/blite/index.php/blitec-a-tool-for-developing-ws-bpel-applications/`. I sorgenti, rilasciati sono licenza GPLv3, sono reperibili presso `http://blitec.bewq.org`

3.2.2 Deploy

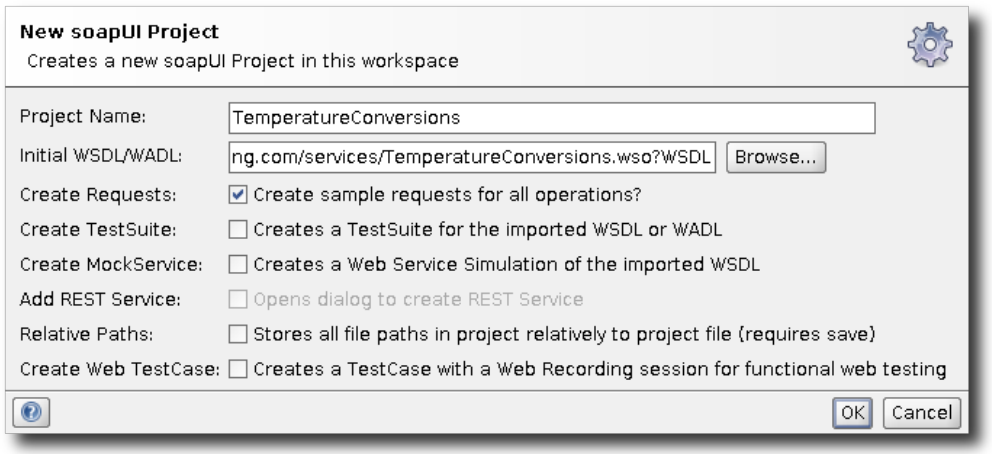
Per effettuare il deploy sarà sufficiente copiare il pacchetto nella directory di deploy dell'*engine*, oppure semplicemente indicare la directory con l'opzione -t , come riportato nel Listing 3.25.

Per verificare il successo del deploy, l'utente dovrà collegarsi con un browser alla pagina di stato dell'*engine* WS-BPEL e verificare che il nome del processo sia tra quelli di cui è stato effettuato il deploy.

3.2.3 Esecuzione

Quest'ultima fase permette all'utente di simulare un'interazione del servizio con un client.

Nell'esposizione utilizzeremo l'utility soapUI [13] per contattare i servizi creati. Per contattare il servizio, dopo aver avviato il programma, dobbiamo creare un nuovo progetto, dargli un nome, e inserire nel campo adiacente alla voce *Initial WSDL/WADL* l'URI del file WSDL del servizio.



Una volta inserite queste informazioni, è sufficiente cliccare sul pulsante *OK* per eseguire l'importazione del documento WSDL e la conseguente generazione delle richieste SOAP. Finito il processo di creazione, apparirà un'interfaccia che permette la navigazione attraverso tutti i messaggi inviabili al servizio per usufruire di una operazione.

Per inviare un messaggio SOAP dovremo semplicemente selezionare una operazione e cliccare su una *Request*. Questa operazione aprirà una nuova finestra nella parte destra dell'interfaccia, dove l'utente inserirà i valori all'interno dei campi contraddistinti dal carattere segnaposto *?*, prima di inviare il messaggio (si veda la Figura 3.1).

Effettuata quest'ultima operazione, saremo pronti ad inviare il messaggio al nostro nuovo servizio. L'invio del messaggio può essere eseguito cliccando sul pulsante "freccia"

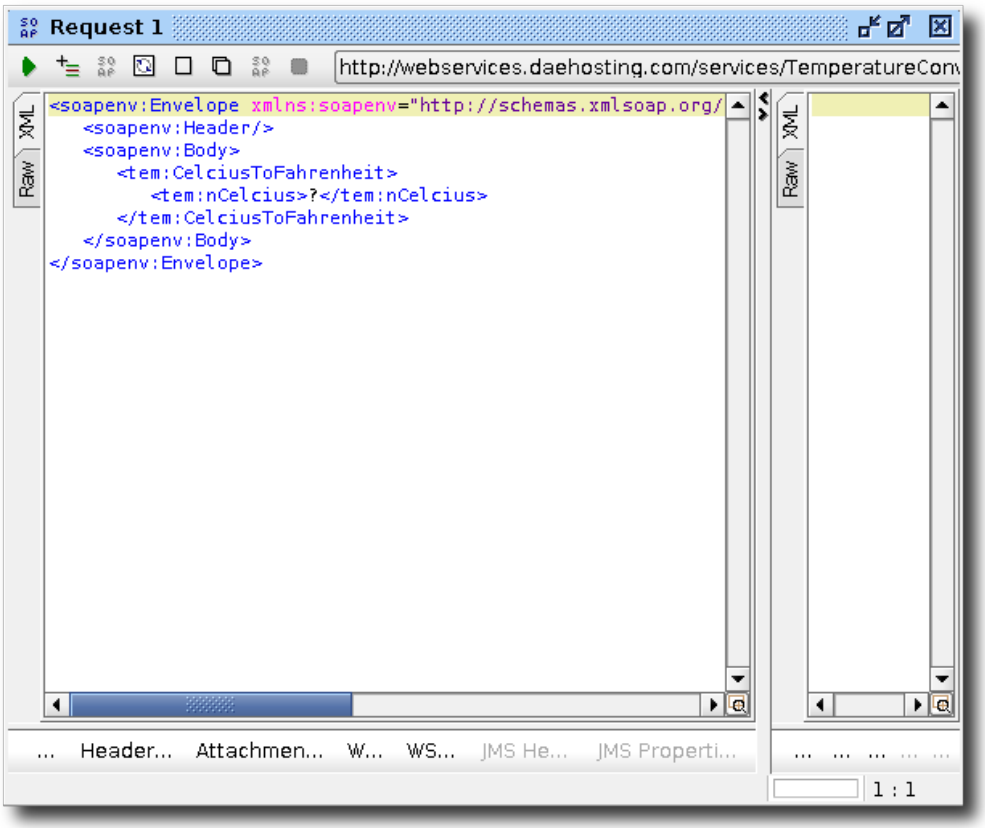


Figura 3.1: soapUI - campo da riempire

in alto a sinistra nella barra presente sopra l'editor del messaggio. Inviato il messaggio, per verificare che il servizio abbia elaborato la richiesta, dovremo tornare con il browser sulla pagina di stato dell'engine e controllare lo stato del processo (per un esempio si veda Figura 3.2).

Nel caso di invocazione di un servizio Request-Response sarà invece sufficiente attendere che l'istanza del servizio ci risponda, infatti soapUI mostrerà tale risposta accanto alla richiesta inviata (si veda Figura 3.3).

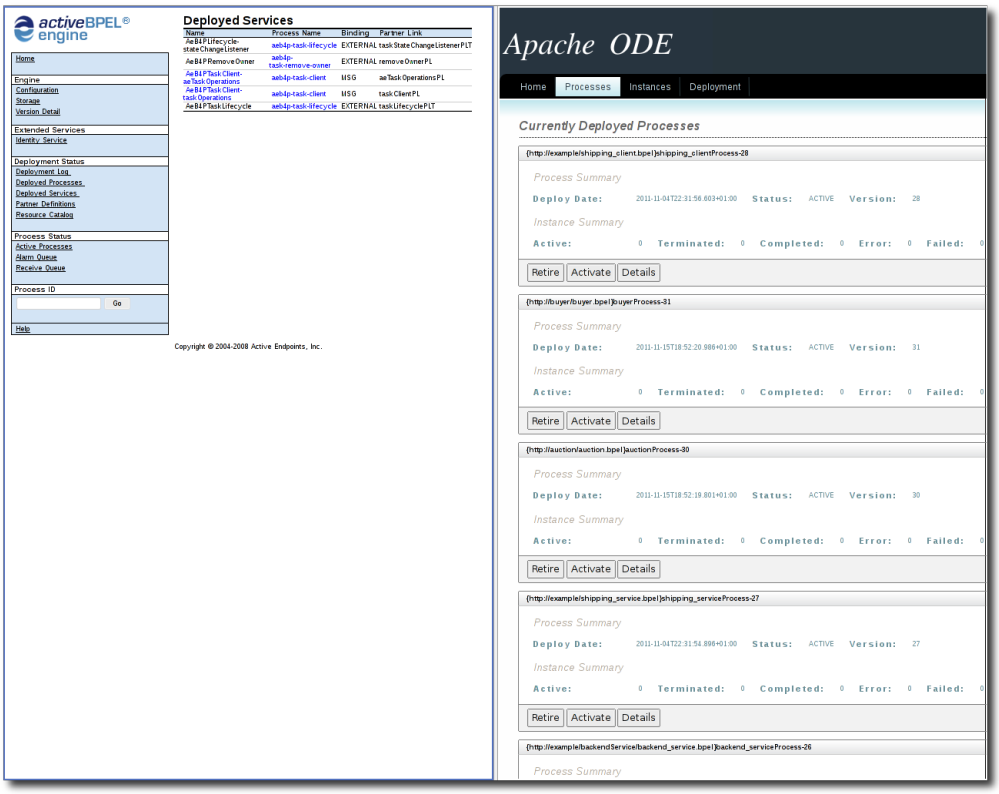


Figura 3.2: Engine supportati - schermate di deploy

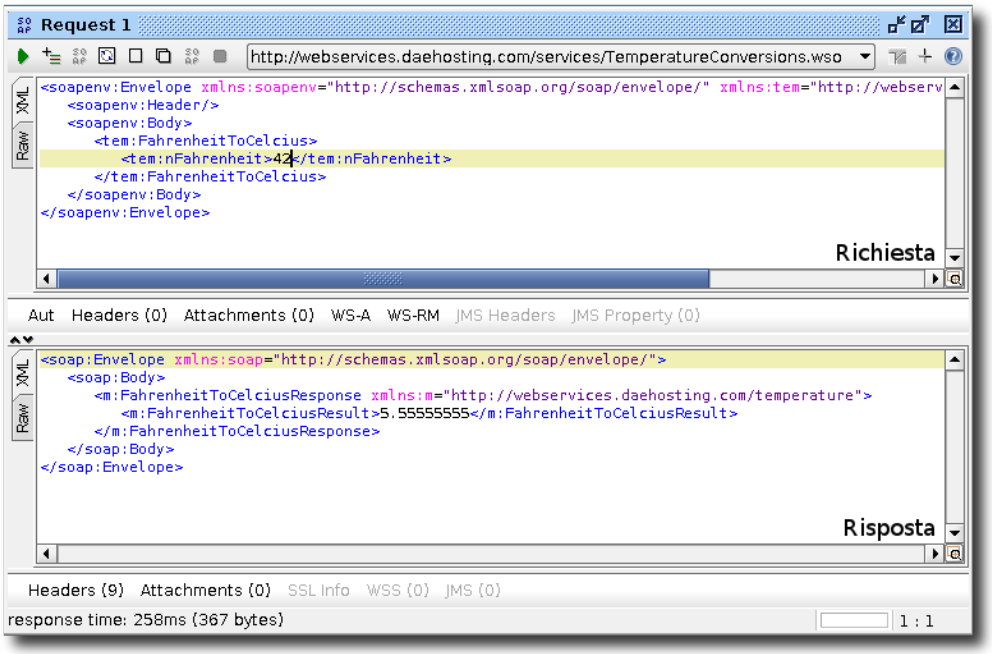


Figura 3.3: soapUI - risposta del servizio

Capitolo 4

Da B*lite*C a Prism

In questo capitolo introdurremo in dettaglio il lavoro effettuato per la realizzazione della traduzione automatica da B*lite* a Prism. Prima esporremo l'estensione del linguaggio accettato da B*lite*C e la tecnica di traduzione da B*lite* a Prism adottata nel modulo B*lite*2Prism di B*lite*C. Quindi, presenteremo un esempio di utilizzo del modulo B*lite*2Prism e una breve guida su come predisporsi all'analisi della specifica generata con Prism.

4.1 Estensione del linguaggio accettato da B*lite*C

L'estensione del linguaggio accettato da B*lite*C deve permettere da un lato l'aggiunta di informazioni con le quali personalizzare il comportamento della specifica Prism, dall'altro garantire la compatibilità con i servizi B*lite* già esistenti. Dunque non può essere modificata la sintassi dei costrutti B*lite*, né può essere aggiunta una nuova parte di configurazione addizionale.

Per rispettare queste richieste si è scelto quindi di aggiungere delle annotazioni al codice, in stile Java. Le annotazioni sono opzionali e non variano la sintassi dei costrutti, permettendo così di aggiungere informazioni nella modalità desiderata.

4.1.1 Annotazioni

L'annotazione di un costrutto è molto semplice: è sufficiente aggiungere

```
@annotazione valore,
```

prima del costrutto interessato, come si può vedere nel Listing 4.1.

Annotazione	Argomento
`@prob`	`42`
`@rate`	`42.0`
`@setname`	`"myVar"`
`@condition`	`"myVar>1"`
`@declare`	`myVar~2`
`@update`	`"myVar'=42"`
`@multi`	`"myVar<42"`

Tabella 4.1: Esempi di annotazioni disponibili

Listing 4.1: Esempio di annotazione

```
@prob 42
blite_construct
```

Le annotazioni applicabili variano da costrutto a costrutto, dunque nelle seguenti sezioni saranno indicate nel dettaglio quali annotazioni possono essere aggiunte per ogni tipo di costrutto. Tutte le possibili annotazioni sono riportate in Tabella 4.1. Nel seguito esporremo in generale il significato e l'utilizzo di ogni annotazione.

prob

L'annotazione `prob` necessita come argomento un valore nell'intervallo $[0, 100] \in \mathbb{N}$ oppure un nome di una costante ed un valore di default. Nel caso sia usato il valore numerico, questo sarà usato come probabilità di successo associata al costrutto nella generazione della specifica. Nel caso con annotazione di una stringa, sarà generata una costante che permetterà all'utente di variare il comportamento del costrutto senza dover ricompilare la specifica B*lite*. La costante risulta utile nel caso sia necessario valutare il comportamento del costrutto in vari contesti.

Il significato e il tipo di valore associabili all'annotazione variano da costrutto a costrutto, sarà quindi specificato in seguito il comportamento caso per caso. Per un esempio di questa annotazione si veda la prima riga della Tabella 4.1.

rate

L'annotazione `rate` necessita come argomento un valore $x \in \mathbb{R}$, che sarà usato per calcolare la probabilità del costrutto a cui è associato. Il significato varia da costrutto a costrutto, sarà quindi specificato in seguito caso per caso. Per un esempio di questa annotazione si veda la seconda riga della Tabella 4.1.

set-name

L'annotazione *set-name* accetta in argomento una stringa, che sarà usata come nome della variabile associata al costrutto nella generazione della specifica.

Questa annotazione permette all'utente di dare nomi personalizzati a costrutti di interesse per l'analisi, inoltre può essere usato in combinazione con altre annotazioni per modificare il comportamento del modello. Un esempio di questa annotazione è presente nella terza riga della Tabella 4.1.

condition

L'annotazione *condition*, utilizzabile come alternativa a *prob*, permette di condizionare il comportamento di un costrutto al valore di una espressione sulle variabili componenti la specifica. Per un esempio si veda la Tabella 4.1.

declare

Con l'annotazione *declare* è possibile aggiungere variabili intere Prism per personalizzare il comportamento del modello. L'annotazione accetta come argomento il nome della nuova variabile e il numero di stati che la compongono, separati dal simbolo ~. Per un esempio vedere la Tabella 4.1.

Questa annotazione è utile in combinazione all'annotazione `@condition`, infatti l'utente può creare una variabile e utilizzarla per condizionare il comportamento di uno o più costrutti.

update

L'annotazione *update*, presente nella sesta riga della Tabella 4.1, permette di aggiungere aggiornamenti personalizzati alla trasformazione in Prism dell'attività annotata. Esistono due varianti di questa annotazione, una che accetta due espressioni Prism e un'altra che ne accetta soltanto una. Il numero di argomenti annotabili dipende dal tipo di costrutto; dunque sarà specificato in seguito quali costrutti supportano l'annotazione di una espressione e quali supportano l'annotazione di due espressioni.

multi

L'annotazione *multi* permette di impostare l'esecuzione multipla del codice Prism associato al deploy B*lite* annotato. L'annotazione accetta un parametro stringa opzionale che definisce la condizione dell'iterazione contenente il servizio. In caso di mancato inserimento del parametro viene impostata una esecuzione infinita del codice associato.

Questa annotazione permette, ad esempio, di eseguire più volte il codice associato ad un deploy che viene richiamato più volte in una singola esecuzione del sistema di servizi. Un esempio di questa annotazione è presente nell'ultima riga della Tabella 4.1.

4.2 Traduzione in Prism

In questa sezione introdurremo le idee su cui si basa il processo di traduzione e commenteremo nel dettaglio la traduzione di ogni costrutto.

4.2.1 Generalità

La traduzione da B*lite* in Prism si basa su alcuni principi che rendono le operazioni di traduzione semplici e veloci, senza però alterare il comportamento del modello generato. Nel definire il processo di traduzione abbiamo cercato di tenere in considerazione che una traduzione non accurata può facilmente portare all'impossibilità di generare il CTMC associato alla specifica a causa dell'esplosione degli stati. A questo proposito si è cercato di effettuare una traduzione che non alterasse il comportamento della specifica originale e al contempo generasse codice Prism ottimizzato dal punto di vista dello spazio degli stati. Le cause principali dell'esplosione degli stati in un modello Prism sono:

- l'utilizzo di un numero elevato di moduli;
- la creazione di variabili con molti stati.

Ovviamente il problema non è risolvibile per ogni tipo di specifica B*lite*: infatti, se la specifica B*lite* include un grande numero di processi complessi, molto probabilmente ci si troverà in una situazione di esplosione degli stati. Con la nostra traduzione si è cercato di confinare questo problema a specifiche B*lite* complesse garantendo il trattamento di sistemi usuali. Inoltre, grazie alla possibilità di vincolare la specifica con variabili personalizzate, l'utente specializzando il comportamento del modello potrà ridurre il numero degli stati che lo compongono.

La traduzione delle specifiche di sistemi di servizi B*lite* in Prism ha come scopo l'analisi del flusso di controllo generale e di errore dei servizi. Per contenere la dimensione del modello associato al servizio B*lite*, le azioni di errore e fallimento sono valutate solo quando l'utente inserisce azioni personalizzate. Infatti, in un contesto di verifica, le attività di default, `empty` per la compensazione e `throw` per i fallimenti, non arricchirebbero il comportamento del modello, ma semplicemente renderebbero la specifica Prism più grande del necessario. In particolare l'attività `empty` non è altro che un segnaposto per un'azione e `throw` non fa altro che richiamare le operazioni di fallimento degli scope

racchiudenti. In dettaglio, le due attività eccezionali non sono rappresentate nel modello da due flussi separati, ma da un unico flusso che esegue prima il fallimento e poi la compensazione. Questo permette all'utente di concentrarsi sul funzionamento del servizio in caso di funzionamento corretto e sulle azioni di compensazione e fallimento quando queste sono effettivamente utili all'analisi.

I servizi B*lite* che compongono il sistema considerato nella traduzione in Prism comunicano attraverso un *endpoint* che gestisce la comunicazione. Ogni endpoint contiene un buffer di dimensione finita (15 nell'esempio in Figura 4.1) dove verranno memorizzati i messaggi inviati da ogni invoke partecipante (5 nell'esempio in Figura 4.1). Ovviamente è imposto un limite massimo ai messaggi inviabili da ogni partecipante. Per ogni messaggio consegnato, l'endpoint si occuperà di smistarlo verso una attività di ricezione disponibile, come si può vedere in Figura 4.1.

I motivi del fallimento di un processo analizzabili nei modelli generati da B*lite*2Prism sono quelli lanciati esplicitamente dall'utente, dal fallimento nell'invio o ricezione di un messaggio e dal sovraccarico di richieste su di un endpoint. In sintesi, le attività che possono far fallire un servizio sono le invocazioni e le attività di `throw`. L'utilizzo del costrutto `throw` permette di forzare il fallimento del servizio, come avviene in WS-BPEL e B*lite*. Le invocazioni possono fallire per tre cause: il numero massimo di messaggi per l'invocante è stato superato; il buffer dell'endpoint risulta saturo, ma un altro messaggio deve essere inviato; il messaggio non è inviato correttamente a causa del fallimento del mezzo di trasmissione.

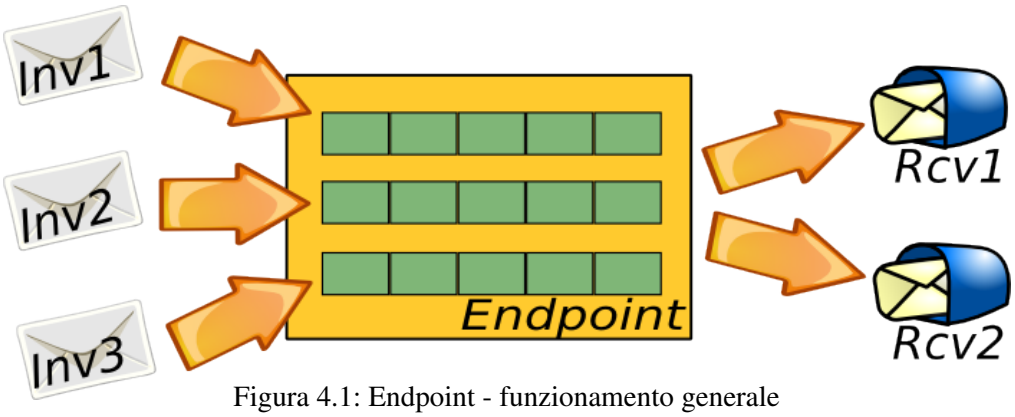


Figura 4.1: Endpoint - funzionamento generale

La specifica generata permette l'analisi di una sola istanza di ogni processo B*lite*, in quanto l'interesse è stato concentrato sulle possibili interazioni attuabili da una singola esecuzione, senza valutare i meccanismi e le problematiche dovute a più istanze, le quali sono a carico dell'engine WS-BPEL e non a carico dell'utente. Nel caso l'analisi di uno scenario richieda necessariamente la creazione di più istanze non concorrenti di un

servizio, è possibile ricorrere all'annotazione `multi` per creare un numero limitato (o illimitato) di istanze.

Un altro punto fondamentale dell'analisi è che questa non viene condotta su variabili contenenti valori, ma su una loro astrazione. L'idea alla base di questa decisione è che non è importante il significato dei valori, ma il comportamento generato da questi. Infatti per modellare il comportamento dei valori quando ciò è rilevante, all'interno dei costrutti di decisione è data all'utente la possibilità di simulare la scelta attraverso l'annotazione di una probabilità o una condizione personalizzata, come verrà esposto nel seguito.

La traduzione in Prism è incentrata sulla generazione di specifiche di tipo CTMC, in quanto la rappresentazione a tempo continuo simula in modo più fedele il comportamento dei sistemi di cui abiliteremo l'analisi.

Costrutti non considerati

Alcuni costrutti B*lite* non sono stati considerati perché non interessanti per il tipo di analisi effettuata. I costrutti non considerati nel processo di traduzione sono:

- assegnamenti;
- espressioni XPath e Booleane;
- variabili di correlazione.

Le assegnazioni sono operazioni che caratterizzano l'implementazione e, dunque, possono essere omesse dalla specifica. Le espressioni XPath sono state tralasciate per lo stesso motivo: esse sono a tutti gli effetti delle assegnazioni più articolate.

Le variabili di correlazione sono state omesse perché l'analisi è effettuata su una sola istanza per ogni processo e non sui meccanismi usati dal motore WS-BPEL per eseguire più istanze di un processo.

Le espressioni booleane possono essere rese in Prism con l'annotazione di una probabilità che simuli il comportamento del costrutto che esegue la valutazione. Dunque possono essere eliminate dai costrutti che necessitano di una traduzione.

Vincoli nella specifica

Per generare una specifica Prism che abbia le caratteristiche desiderate, la specifica B*lite* in ingresso dovrà rispettare alcuni vincoli.

Identificazione degli endpoint

Nel processo di traduzione, B*lite*2Prism seleziona i partecipanti ad una comunicazione sfruttando l'endpoint utilizzato. Un endpoint è costituito da un partner link, un'operazione e una variabile. Per permettere a B*lite*2Prism di selezionare i canali utilizzati per la

Partner ricevente	Partner Invocante
`rcv <p1,p2> op <x1,x2,x3>`	`inv <p1,p2> op <x1,x2,x3>`

Tabella 4.2: Esempio di costrutti che comunicano

comunicazione, i nomi usati per le componenti dell'endpoint dovranno essere gli stessi sia nel servizio che fornisce l'operazione, sia in quello che la utilizza. In dettaglio, questo accorgimento è stato utilizzato per evitare di dover annotare altre informazioni alla specifica B*lite*; infatti B*lite*2Prism seleziona i partner di un canale attraverso il pattern matching sui nomi degli endpoint. Un esempio di costrutti che comunicano è mostrato in Tabella 4.2.

Numero di partecipanti

Per poter generare una specifica interessante dobbiamo garantire che i processi partecipanti siano almeno due: uno attende richieste su di un endpoint e l'altro utilizza l'endpoint per iniziare una comunicazione.

Solitamente i servizi client sono rappresentati da istanze B*lite*, in modo da non dover attendere messaggi da un ulteriore attore per poter iniziare l'evoluzione.

4.2.2 Traduzione

Di seguito presenteremo la traduzione da B*lite* a Prism introducendo prima il meccanismo generale comune a tutti i costrutti, poi riporteremo il dettaglio di ogni costrutto considerato.

Per facilitare la comprensione della codifica definiremo adesso alcuni concetti di base che sfrutteremo nell'esposizione delle trasformazioni dei vari costrutti. Per la traduzione di ogni costrutto è necessario fornire in input almeno due parametri: la condizione di sequenzializzazione e le condizioni addizionali. Oltre a questi due parametri alcuni costrutti necessitano di altre informazioni che però verranno definite caso per caso. La *condizione di sequenzializzazione* è necessaria per permettere l'esecuzione sequenziale delle attività tradotte e per ovviare all'impossibilità di eseguire comandi in modo sequenziale in Prism. Questo punto verrà spiegato in dettaglio nella traduzione dell'attività di sequenzializzazione. Le *condizioni addizionali* invece sono una o più condizioni che devono essere verificate da un determinato comando per proseguire nell'esecuzione.

Per comodità, in seguito ci riferiremo con: `[[seq]]` all'ultimo parametro di sequenzializzazione; `[[add]]` all'insieme dei parametri addizionali; `[[end]]` all'insieme delle receive e invoke che interagisco in tutti gli endpoint; `[[pick]]` all'insieme delle receive contenute in ogni costrutto di pick.

Utilizzeremo la notazione '`[[nome]] <- valore`' per indicare un aggiornamento dello stato del parametro, in particolare per `[[end]]` e `[[pick]]` un'aggiornamento avverrà con la notazione '`[[nome]](indice)<- valore,valore`' perché esse contengono una lista di parametri per ogni indice.

Inoltre, per ragioni di sintesi, data un'attività B*lite* `act` useremo la forma `{{act}}` per indicare la trasformazione in Prism dell'attività B*lite*. Nella traduzione in Prism, in generale, ogni costrutto verrà tradotto attraverso l'ausilio di una variabile (ulteriori dettagli saranno forniti nel Paragrafo 4.2.2): per indicare la variabile associata ad una attività `act` utilizzeremo, con abuso di notazione, il nome `act`, nel caso che invece si voglia indicare una variabile di un costrutto interno alla traduzione si indicherà con `${{act}}[i]`, dove `i` indica l'i-esimo costrutto. Come convenzione intenderemo con `last` l'indice dell'ultimo costrutto della traduzione di una attività e contraddistingueremo il valore associato al successo di un'attività con `SUCC`.

I costrutti verranno introdotti attraverso delle tabelle che riporteranno le informazioni necessarie alla comprensione del metodo di traduzione. Non tutti i costrutti riporteranno gli stessi attributi: in caso di assenza sarà dato per assunto che l'attributo non è utile nella trasformazione. L'attributo *parametri in input* indica quali informazioni devono essere passate in ingresso alla traduzione di uno specifico costrutto. L'attributo *Traduzione - Corpo del modulo Prism* indica che cosa sarà aggiunto al modulo rappresentante il servizio, invece, l'attributo *Traduzione - Specifica* indica quali costrutti verranno aggiunti nella specifica al di fuori del modulo principale. Con l'attributo *Parametri in in uscita - Traduzione del corpo* e *Parametri in output - Termine traduzione* vengono indicate le modifiche che saranno effettuate a `[[seq]]`, `[[add]]`, `[[pick]]` e `[[end]]`, rispettivamente, durante la traduzione del costrutto e dopo la terminazione della traduzione.

Moduli e Variabili

Ad ogni costrutto B*lite* considerato dalla traduzione nella specifica assoceremo nella specifica Prism una variabile intera. La variabile avrà un numero di stati adeguato alla definizione corretta del costrutto. Ogni variabile sarà contraddistinta da un nome derivante dal costrutto rappresentato e da un numero progressivo necessario alla unicità del nome di ogni traduzione. Ad esempio, una traduzione semplicistica del costrutto condizionale potrebbe essere una variabile intera con due stati: uno per la condizione vera e uno per quella falsa, come mostrato nel Listing 4.2.

Listing 4.2: Esempio di variabile

```
int var : [0..1]
```

Deploy	**Traduzione**

`service ::= {service_body}{corr_vars}`	`global serviceStatus : [0..2] init 0;`
	
	`module service`
	`  {{service_body}}`
	`endmodule`

Parametri in Output - Traduzione del corpo	
`[[add]] <- serviceStatus=0`	

Tabella 4.3: Traduzione del deploy

Oltre agli stati menzionati precedentemente, le variabili intere associate ai costrutti hanno due stati speciali: uno stato indica la mancata inizializzazione del costrutto e l'altro la terminazione dell'esecuzione. Solitamente lo stato di mancata inizializzazione è rappresentato dal valore intero 0, l'altro stato speciale invece varia con il tipo di costrutto. Ovviamente, ogni variabile ha come valore di inizializzazione 0; questo valore è usato come controllo iniziale di ogni costrutto per impedire l'esecuzione multipla del comportamento tradotto.

Deploy

Ogni deploy è rappresentato in Prism attraverso un modulo che contiene le traduzioni delle attività nel corpo del costrutto ed il nome associato al deploy nella specifica B*lite*. Inoltre, ogni modulo è contraddistinto da una variabile globale intera che tiene traccia dello stato attuale del servizio (il nome della variabile è la concatenazione del nome del modulo con la parola *status*). La variabile può assumere tre stati:

- 0, quando il servizio è in esecuzione;
- 1, nel caso sia terminato con successo;
- 2, nel caso sia terminato con fallimento.

Il valore della variabile modifica il comportamento dei costrutti componenti il processo: i costrutti che compongono il normale flusso di esecuzione dipendono dallo stato 0, quelli che rappresentano il flusso di compensazione e fallimento dipendono dallo stato 2. In questo modo, nel caso in cui il processo sia terminato con successo nessuna altra attività potrà essere eseguita. La traduzione dell'attività di deploy è riportata in Tabella 4.3.

Sequenza

Il costrutto di sequenzializzazione B*lite* non ha una controparte nel linguaggio di specifica di Prism: infatti in Prism ogni comando è eseguito quando la condizione associata è verificata. Dunque per garantire la sequenzializzazione, in Prism, delle azioni B*lite* si

Sequenzializzazione	**Traduzione**

`seq`
`  act1;`
`  act2;`
`  act3`
`qes`
	`[] ( Act1=0 ) -> act1Updates;`
	`[] ( Act2=SUCC & Act1=0 ) -> act2Updates;`
	`[] ( Act3=SUCC & Act2=0 ) -> act3Updates;`

Tabella 4.4: Traduzione della sequenzializzazione

Parallelo	

`flw`
`  act1`
`  | act2`
`  ...`
`  | actN`
`wlf`	

Parametri in input	
`[[seq]]`: Condizione di sequenzializzazione	`[[add]]`: Condizioni addizionali
Traduzione - Corpo del modulo Prism	**Traduzione - Specifica**

`flw : [0..2] init 0`
 	
`[flwInit]`
` ([[add]] & [[seq]] & flw=0) -> flw=1;`
 	
`[flwEnd] flw=1 -> (flw'=2);`
	`module m1`
	` [flwInit] flw=0 -> true;`
	` {{act1}}`
	` [flwEnd]${{act1}}[last]=SUCC ->true;`
	`endmodule`
	`...`
	`module mN`
	` [flwInit] flw=0 -> true;`
	` {{actN}}`
	` [flwEnd]${{actN}}[last]=SUCC ->true;`
	`endmodule`

Parametri in output - Termine traduzione	
`[[seq]] <- flw=2`	

Tabella 4.5: Traduzione del parallelo

è dovuto costruire quest'ultima imponendo, dove desiderato, che l'azione successiva dipenda dal successo dell'azione precedente (quando presente).

Il costrutto di sequenzializzazione di B*lite* viene usato come guida per determinare dove applicare il metodo di sequenzializzazione. Il meccanismo è quindi implementato dalla traduzione delle varie attività contenute nel corpo della sequnzializzazione. Infatti al termine della traduzione di un'attività, viene passato in ingresso all'attività successiva la condizione di successo dell'attività appena tradotta.

Un esempio semplificato di questo metodo è in Tabella 4.4 dove è possibile vedere che, dato `act=seq ... qes`, ogni comando ha come guardia `${{act[i]}}=0` $\forall i \in 1 \dots 3$, ovvero un controllo per verificare che la variabile associata al costrutto indichi ancora un costrutto non inizializzato, e `${{act[j]}}=SUCC` per $j \in 1, 2$ che controlla se il comando precedente è terminato con successo. Gli `actiUpdates` presenti nell'esempio in Tabella 4.4 sono il risultato della traduzione delle varie attività B*lite* `acti` in Prism.

Condizionale

```
@prob p
@update update1 update2
if( conditions )
  {act1}
else
  {act2}
fi
```

Parametri in input

`[[seq]]`: Condizione di sequenzializzazione	`[[add]]`: Condizioni addizionali

Traduzione - Corpo del modulo Prism

```
con : [0..2] init 0

[] ( [[seq]] & [[add]] & con = 0 ) -> p:(con' = 1) & (update1)
                                        + 100 - p:(con' = 2) & (update2);
{{act1}}
{{act2}}
```

Parametri in Output - Traduzione del corpo	**Parametri in output - Termine traduzione**
`[[add]] <- con=1` (nel caso del corpo di if)	`[[seq]] <- (${{act1}}[last]=SUCC`
`[[add]] <- con=2` (nel caso del corpo di else)	`| ${{act2}}[last]=SUCC )`

Tabella 4.6: Traduzione dell'attività condizionale: annotazione `@prob`

Parallelo

La traduzione del costrutto B*lite* di esecuzione parallela è resa in Prism attraverso i moduli e la sincronizzazione forzata dell'ultima attività di ogni modulo. La sincronizzazione è garantita dalle *azioni* di Prism che forzano la sincronizzazione di tutte le componenti etichettate con lo stesso nome.

Con questa strategia le componenti parallele sono eseguite concorrentemente ed il costrutto di esecuzione parallela termina solo quando tutte le sue componenti terminano. Nella Tabella 4.5 possiamo vedere in modo più chiaro la traduzione del costrutto B*lite* di esecuzione parallela. La variabile intera `flw` associata al costrutto di esecuzione parallela ha tre stati: 0 è quello di mancata inizializzazione, 1 indica che il costrutto è attivo ed infine 2 indica la terminazione con successo del costrutto. Ogni componente parallela è inserita all'interno di un modulo che ha come primo comando la sincronizzazione (`[flwInit]`) con il modulo principale. Il modulo principale attende che tutti gli altri moduli abbiano terminato (`[flwEnd]`), poi imposta la variabile `flw` associata al costrutto di esecuzione parallela a 2 per indicare la terminazione.

Condizionale

L'attività condizionale è realizzata in Prism attraverso una probabilità (`@prob`) o una condizione su variabili Prism (`@cond`) che simuli il comportamento determinandone la scelta.

La Tabella 4.6 e la Tabella 4.7 ci illustrano come avviene la traduzione. Al costrutto è associata una variabile intera `con0` avente tre stati: 0 indica la mancata inizializzazione,

<table>
<tr><th colspan="2">Condizionale</th></tr>
<tr><td colspan="2">

```
@cond "condition"
if( conditions )
  {act1}
else
  {act2}
fi
```

</td></tr>
<tr><th colspan="2">Parametri in input</th></tr>
<tr><td colspan="2"><code>[[seq]]</code>: Condizione di sequenzializzazione <code>[[add]]</code>: Condizioni addizionali</td></tr>
<tr><th colspan="2">Traduzione - Corpo del modulo Prism</th></tr>
<tr><td colspan="2">

```
con : [0..2] init 0

[] ( [[seq]] & [[add]] & con = 0 & condition ) -> (con' = 1)
{{act1}}
[] ( [[seq]] & [[add]] & con = 0 & !condition ) -> (con' = 2)
{{act2}}
```

</td></tr>
<tr><th>Parametri in Output - Traduzione del corpo</th><th>Parametri in output - Termine traduzione</th></tr>
<tr><td><code>[[add]] <- con=1</code> (nel caso del corpo di if)
<code>[[add]] <- con=2</code> (nel caso del corpo di else)</td><td><code>[[seq]] <- (${{act1}}[last]=SUCC
| ${{act2}}[last]=SUCC)</code></td></tr>
</table>

Tabella 4.7: Traduzione dell'attività condizionale: annotazione `@cond`

1 indica la scelta del ramo associato alla condizione vera e 2 indica la scelta del ramo associato alla condizione falsa. Il comando Prism associato a questo costrutto è molto semplice, infatti sceglie la condizione dell'espressione attraverso la probabilità inserita dall'utente.

Se l'utente annota la probabilità (Tabella 4.6), il traduttore utilizzerà questo valore per calcolare la probabilità da associare alla condizione falsa, sottraendo a 100 il valore inserito, e assocerà alla condizione vera la probabilità inserita.

Nel caso invece che sia annotata una condizione (Tabella 4.7), che chiameremo nell'esempio `condition`, il processo di traduzione genererà due comandi, uno avente come condizione di attivazione `condition` e un altro avente come condizione di attivazione la negazione di quest'ultima, ossia `!condition`.

Il costrutto condizionale permette l'utilizzo della versione doppia dell'annotazione `@update`, vedere Tabella 4.6. Gli argomenti annotati saranno rispettivamente aggiunti, in ordine di inserimento, alla condizione soddisfatta e alla condizione non verificata.

Se l'utente non indica nessuna annotazione, il traduttore imposta le scelte in modo che siano equiprobabili.

Il comando successivo a quello condizionale, non conoscendo a priori quale dei due rami sarà eseguito, conterrà nella sua condizione il controllo sulla terminazione dell'ultimo costrutto contenuto di entrambi i rami, come si può vedere nella sezione "Parametri in Output - Termine Traduzione" in Tabella 4.6 e in Tabella 4.7.

Iterazione	

`@prob p`
`while (conditions)`
`  {act1}`	

Parametri in input	
`[[seq]]`: Condizione di sequenzializzazione	`[[add]]`: Condizioni addizionali
Traduzione - Corpo del modulo Prism	

`ite : [0..2] init 0;`
 	
`[] ([[seq]] & [[add]] & ite = 0 & ${{act}}[0]=0 & .. & ${{act}}[last]=0 )`
`  -> p:(ite' = 1) + 100 - p:(ite' = 2);`
 	
`[] ( ${{act}}[0] = SUCC & ite = 0 ) -> (${{act}}[0]' = 0);`
`  ...`
`[] ( ${{act}}[last] = SUCC & ite = 0 ) -> (${{act}}[n]' = 0);`
 	
`[] ( ${{act}}[last] = SUCC & ite = 1 ) -> (ite = 1);`	

Parametri in Output - Traduzione del corpo	**Parametri in Output - Termine traduzione**
`[[add]] <- ite=1`	`[[seq]] <- ite=2` `[[seq]] <- ${{act1}}[n]=SUCC`

Tabella 4.8: Traduzione dell'attività di iterazione

Iterazione

Il costrutto di iterazione è tradotto in Prism in modo analogo a quello di scelta condizionale: infatti, anche in questo caso, l'espressione della guardia è resa attraverso una probabilità rappresentata da un numero compreso nell'intervallo $[0 \dots 100] \in \mathbb{R}$ oppure attraverso una espressione Prism personalizzata. La traduzione risulta più complessa di quella dell'operatore condizionale perché deve essere valutata anche l'esecuzione multipla delle operazioni nel corpo del ciclo; in particolare è necessario aggiungere una transizione che controlli lo stato del costrutto e, nel caso debba essere eseguita un'altra iterazione, reinizializzi tutte le variabili utilizzate nel corpo del ciclo allo stato iniziale.

In Tabella 4.8 si nota che la variabile associata all'attività `ite` può assumere tre valori: 0 indica la mancata inizializzazione, 1 indica che la guardia del ciclo è vera e 2 indica che la guardia del ciclo è falsa. La differenza con gli altri costrutti è che in questa traduzione, al termine di ogni iterazione, la variabile è riportata allo stato non inizializzato e prima di effettuare la scelta il costrutto deve aspettare che tutti i costrutti nel corpo del ciclo siano non inizializzati (vedere rispettivamente l'ultimo comando e il primo comando nella parte di traduzione della Tabella 4.8). Per ogni costrutto nel corpo del ciclo è aggiunta un comando che reimposta la variabile ad essi associata ogni volta che la variabile del ciclo è resettata (vedere il secondo e terzo comando in Tabella 4.8).

Come nel caso del costrutto condizionale, quello di iterazione impone al costrutto che lo segue un controllo addizionale. Infatti, oltre al controllo sull'ultimo costrutto nel corpo del ciclo (`${{act}}[n]`), si dovrà effettuare un controllo sulla condizione del ciclo accertandosi che quest'ultimo sia terminato (`ite1=2`).

Canale - Receive
Parametri in input
`[[end]]`: Ricezioni e invocazioni degli endpoint `[[pick]]`: Ricezioni dei costrutti pick
Traduzione - Specifica

```
const MAX_ENDPOINT_BUFFER = 10;

global inv0MsgStatus : [0..5] init 0;

global inv1MsgStatus : [0..5] init 0;

formula endpointLength = inv0MsgStatus + inv1MsgStatus;

module endpoint

  [] ( rec0 = 1 ) -> (rec0' = 2);
  [] ( inv0MsgStatus > 0 & rec0 = 2 ) ->
     (rec0' = 3) & (inv0MsgStatus' = inv0MsgStatus - 1);

  [] ( inv0MsgStatus > 0 & inv1MsgStatus > 0 & rec1 = 2 ) ->
       0.5:(rec0' = 3) & (inv0MsgStatus' = inv0MsgStatus - 1)
     + 0.5:(rec0' = 3) & (inv1MsgStatus' = inv1MsgStatus - 1);

endmodule
```

Tabella 4.9: Traduzione del canale: attività di ricezione

Le annotazioni supportate dal costrutto di iterazione sono analoghe a quelle supportate dal costrutto condizionalei, rimandiamo quindi a Tabella 4.7 per la traduzione dell'annotazione `@cond`.

Operazioni di comunicazione

Per ottenere una traduzione semplice dei costrutti di comunicazione, la comunicazione tra due partner è stata resa attraverso endpoint (vedi Sezione 4.2.1 e Figura 4.1). Gli endpoint sono generati da B*lite*2Prism dopo aver analizzato tutti gli endpoint ed i vari partner (vedi Listing 4.9). Per ogni endpoint è generato un modulo e vengono definite le variabili necessarie alla rappresentazione dei buffer delle operazioni di invocazione partecipanti. Per ogni operazione di invocazione viene generato un buffer (nell'esempio `inv0MsgStatus` e `inv1MsgStatus`) e per ogni endpoint è generata una formula Prism che controlla il numero dei messaggi da processare nell'endpoint (`endpointLength`). La formula somma la capienza di tutti buffer delle invocazioni partecipanti. Così facendo è possibile implementare in modo semplice una capienza massima per ogni endpoint, simulando così il comportamento di un server che può processare un numero limitato di richieste in ogni istante. L'utilizzo di questa formula sarà spiegato in dettaglio nella descrizione dell'attività di invocazione.

Per permettere la personalizzazione della guardia per controllare la dimensione del buffer dell'endpoint è usata una costante Prism `MAX_ENDPOINT_BUFFER`: l'utente può così

Invocazione

```
@rate r
inv pl op [v1, ... , vN]
```

Parametri in input

`[[seq]]`: Condizione di sequenzializzazione `[[add]]`: Condizioni addizionali

Traduzione - Corpo del modulo Prism

```
inv : [0..1] init 0;

[]([[add]] & [[seq]] &
    (
    endpointLength < MAX_ENDPOINT_BUFFER_LENGHT &
    invMsgStatus < MAX_QUEUE_LENGTH
    ) & inv = 0 )
  ->
    MEDIUM_RELIABILITY:(invMsgStatus' = invMsgStatus + 1) & (inv' = 1)
   + 100 - MEDIUM_RELIABILITY:(serviceStatus' = 2);

[]([[add]] & [[seq]] &
    (
     endpointLength > MAX_ENDPOINT_BUFFER_LENGHT
     | invMsgStatus >= MAX_QUEUE_LENGTH
    ) & inv = 0 )
  -> (serviceStatus' = 2);
```

Parametri in output - Termine traduzione

```
[[seq]] <- inv0=1
[[end]](endpoint) <- r,inv
```

Tabella 4.10: Traduzione dell'attività di invocazione

variare la costante per controllare il comportamento del processo in più contesti.

Ogni invocazione invia un messaggio e lo aggiunge al proprio buffer; i messaggi saranno smistati dal canale verso una attività di recezione disponibile in quell'istante fino all'esaurimento dei messaggi contenuti nel buffer. La scelta della ricezione scelta è effettuata basandosi sul numero di receive attive nell'istante di smistamento: nel caso sia presente una sola receive è scelta quest'ultima; nel caso siano presenti più receive, invece, è effettuata una scelta probabilistica basata sul rate annotato alle receive. Una valutazione simile è effettuata anche nel caso di invoke multiple.

La dimensione dei buffer di ogni invocazione è definita attraverso una costante `MAX_BUFFER_LENGTH`, permettendo all'utente di variare la dimensione del buffer come per quello dell'endpoint.

Le attività di comunicazione permettono l'annotazione del rate. L'annotazione è necessaria per scegliere a quale receive recapitare un messaggio, nel caso vi siano più receive in ascolto sullo stesso canale. Per il costrutto di invocazione, invece, il rate permette di fornire una priorità al messaggio in caso di invoke multiple.

Se sono attive più receive o più invoke, vengono aggiunte scelte probabilistiche tra le invoke per ogni ricezione, inoltre per ogni combinazione di recezioni in ascolto è generato un comando che sceglierà a chi recapitare il messaggio in funzione ai rate annotati.

Ricezione	
`@rate r` `rcv pl op [v1, ... ,vN];`	
Parametri in input	
`[[seq]]`: Condizione di sequenzializzazione	`[[add]]`: Condizioni addizionali
Traduzione - Corpo del modulo Prism	**Traduzione - Specifica**
`[] ([[add]] & [[seq]] & rec = 0) -> (rec' = 1);`	`global rec : [0..3] init 0;`
Parametri in output - Termine traduzione	
`[[seq]] <- rec0=3` `[[end]](endpoint) <- r,rcv`	

Tabella 4.11: Traduzione dell'attività di ricezione

L'approccio utilizzato per il calcolo della probabilità di scelta è molto semplice: data una combinazione di n operazioni di comunicazion attive, si indichi con r_i il rate dell'i-esima operazione; allora calcoleremo la probabilità p_i associata al costrutto con:

$$p_i = \frac{r_i}{\sum_{j=1}^{n} r_j}.$$

In questo modo si garantisce che il peso di un costrutto di comunicazione attivo sia sempre normalizzato con il peso di tutti gli altri, evitando situazioni dove un costrutto è troppo valorizzato. Questo metodo è applicato al calcolo della probabilità di invocazioni e ricezioni.

Il costrutto di invoke è tradotto in modo semplice grazie alla presenza dei canali. Invoke è non bloccante, dunque aggiungerà, in caso di esecuzione normale, un messaggio al suo buffer nell'endpoint e poi terminerà la sua esecuzione. In caso di fallimento, il costrutto farà fallire il processo ad esso associato, ma non aggiungerà nessun messaggio al proprio buffer. Per simulare il comportamento del mezzo di trasmissione è stata inserita una costante Prism `MEDIUM_RELIABILITY` che può assumere valori $[0, 100] \in \mathbb{R}$; anche il valore di questa costante è personalizzabile da parte dell'utente in modo da valutare il comportamento del processo in vari contesti.

Dalla traduzione mostrata in Tabella 4.10 possiamo vedere che la variabile associata all'attività (nell'esempio `inv0`) è composta da due stati: lo stato 0 indica la non inizializzazione del costrutto, lo stato 1 indica il termine dell'operazione. Inoltre possiamo notare che la trasposizione Prism del costrutto è composta da due comandi: uno determina il successo dell'invio nel caso in cui il numero di messaggi contenuti nel buffer della invoke e dell'endpoint siano minori della quantità massima amessa; l'altro invece gestisce il fallimento nel caso in cui numero di messaggi ecceda la dimensione massima ammessa per il buffer del costrutto o per l'endpoint (`end0Length`). Il primo comando inoltre effettua una scelta probabilistica sull'affidabilià del mezzo di trasmissione: la probabilità di successo

<table>
<tr><th colspan="2">Pick</th></tr>
<tr><td colspan="2">

```
pck
  @rate r1
  rcv <pl> op <tuple>;act1
 +
  @rate r2
  rcv <pl> op <tuple>;act22
kcp
```

</td></tr>
<tr><th colspan="2">Parametri in input</th></tr>
<tr><td>[[seq]]: Condizione di sequenzializzazione</td><td>[[add]]: Condizioni addizionali</td></tr>
<tr><th>Traduzione - Corpo del modulo Prism</th><th>Traduzione - Specifica</th></tr>
<tr><td>

```
[]([[add]] & [[seq]] & pic0=0 & rec0=0)
                              -> (rec0'=1);
{{act1}}
[](pic0=1 & ${{act1}}[n]=SUCC )
                     -> (pic0'=2);

[]([[add]] & [[seq]] & pic0=0 & rec1=0 )
                               -> (rec1'=1);
{{act2}}
[](pic0=1 & ${{act2}}[n]=SUCC)
                     -> (pic0'=2);
```

</td><td>

```
global pic: [0..2] init 0;
global rec0: [0..3] init 0;
global rec1: [0..3] init 0;
```

</td></tr>
<tr><th>Parametri in output - Termine traduzione</th><th>Parametri in Output - Traduzione del corpo</th></tr>
<tr><td>[[add]] <- pic=1
[[add]] <- rec0=3 (per il primo ramo)
[[add]] <- rec1=3 (per il secondo ramo)</td><td>[[seq]] <- pck=2
[[pick]](pck) <- rec0,rec1</td></tr>
</table>

Tabella 4.12: Traduzione dell'attività pick

della comunicazione avrà una probabilità pari alla costante `MEDIUM_RELIABILITY` e la probabilità di fallimento sarà pari alla differenza tra 100 e la costante.

L'attività di invocazione permette l'annotazione di aggiornamenti personalizzati doppi, uno in caso di successo e l'altro in caso di fallimento. Non sono invece ammesse le condizioni personalizzate.

Il costrutto di receive, in quanto bloccante, deve attendere un partner per la comunicazione; essa dovrà quindi fermare l'evoluzione del servizio fino alla sincronizzazione con un partner. L'attesa è resa grazie al canale di comunicazione: l'attività di recezione rimane in attesa fino a che sul canale non arriva un messaggio.

Dalla traduzione esplicitata in Tabella 4.11 si nota che la variabile associata al costrutto di ricezione è formato da quattro stati: 0 per lo stato di non inizializzazione, 1 per indicare che il costrutto è attivo e attende richieste, 2 per indicare che la receive è candidata alla ricezione del prossimo messaggio entrante ed infine 3 che indica la terminazione con successo della ricezione.

Il comando associato alla ricezione è solo quello che la inizializza perché l'evoluzione è gestita dall'endpoint associato. Dalla Tabella 4.9 possiamo notare che per ogni receive l'endpoint si prende cura di far evolvere la receive dallo stato 1 (attivo) a 2 (selezionato) e, infine, a 3 (termine della comunicazione). I comandi Prism aggiunti sono due: il primo serve a variare lo stato della receive da 1 a 2 nel caso che la receive sia selezionata per la

Canale - Pick

Parametri in input

`[[end]]`: Ricezioni e invocazioni degli endpoint `[[pick]]`: Ricezioni dei costrutti pick

Traduzione - Specifica

```
const MAX_ENDPOINT_BUFFER = 10;

global inv0MsgStatus : [0..5] init 0;

global inv1MsgStatus : [0..5] init 0;

formula endpointLength = inv0MsgStatus + inv1MsgStatus;

module endpoint

 [] ( (rec0 = 1 | rec0 = 2) & (pic0 = 1 | pic0=2 ) ) -> 1:(rec0' = 0);

 [] ( inv0MsgStatus > 0 & inv1MsgStatus > 0 & rec0 = 2 & rec1 != 3 ) ->
     0.5:(rec0' = 3) & (inv0MsgStatus' = inv0MsgStatus - 1) & (pic' = 1)
   + 0.5:(rec0' = 3) & (inv1MsgStatus' = inv1MsgStatus - 1) & (pic' = 1);

endmodule
```

Tabella 4.13: Traduzione del canale: attività di pick

ricezione, il secondo, invece, varia il valore della variabile da 2 a 3 quando una invoke si è sincronizzata con successo con la receive.

Il costrutto pick è tradotto come un insieme di receive di cui solo una potrà essere eseguita prima della terminazione del costrutto. Dal dettaglio della traduzione riportato in Tabella 4.12 possiamo vedere che il costrutto pick è tradotto con una variabile Prism intera di tre stati: lo stato 0 indica che ancora nessuna recezione è stata eseguita; lo stato 1 indica che un ramo è stato selezionato (una ricezione è stata terminata con successo); lo stato 2 indica la terminazione del costrutto. La traduzione dei costrutti di ricezione nel corpo della pick è resa tramite la traduzione di un'operazione di ricezione con l'aggiunta di condizioni addizionali, quali il controllo sull'inizializzazione dell'attività pick, una receive deve prima controllare che il costrutto di pick sia non inizializzato (`pic=0`), e il controllo che l'attività precedente alla pick sia terminata con successo (`[[seq]]`).

L'evoluzione della pick, come per l'attività di ricezione, è in parte gestita dal canale. Come si può vedere nella Tabella 4.13 all'interno del modulo Prism "endpoint" la gestione da parte del canale si compone di due comandi per ogni attività di ricezione contenuta nella pick: il primo gestisce il ritorno allo stato non inizializzato della ricezione nel caso nel caso in cui un altra ricezione della pick venga scelta per la sincronizzazione(rispettivamente `pic=1` e `pic=2`); il secondo comando invece gestisce la variazione dello stato della pick da 0 (non inizializzato) a 1 (ramo selezionato), quando una sincronizzazione và a buon fine (`rec0=3` o `rec1=3`).

Nell'esempio in Tabella 4.13 si suppone di avere due operazioni di ricezione nel costrutto pick: `rec1` e `rec2`. Infine, come si può vedere in Tabella 4.12 al termine di tutti i

Exit
`exit`
Parametri in input
`[[seq]]`: Condizione di sequenzializzazione `[[add]]`: Condizioni addizionali
Traduzione - Corpo del modulo Prism

`exit : [0..1] init 0;`
 	
`[] ( [[acc]] & [[seq]] & exit=0 ) -> (serviceStatus' = 1) & (exit'=1);`

Tabella 4.14: Traduzione dell'attività exit

Throw
`throw`
Parametri in input
`[[seq]]`: Condizione di sequenzializzazione `[[add]]`: Condizioni addizionali
Traduzione - Corpo del modulo Prism

`throw : [0..1] init 0;`
 	
`[] ( [[acc]] & [[seq]] & throw=0 ) -> (serviceStatus' = 2) & (throw'=1);`

Tabella 4.15: Traduzione dell'attività throw

comandi relativi ad una componente della pick (`act1` o `act2`) troveremo un comando che sposta il valore della variabile della pick nello stato terminato, permettendo al comando successivo di potersi avviare.

Exit e Throw

Il costrutto `exit` è tradotto con un comando che varia lo stato del modulo da 0 (Esecuzione) a 1 (Terminato), come si può vedere in Tabella 4.14.

Il costrutto `throw` è tradotto con un semplice comando che sposta lo stato del modulo da 0 (Esecuzione) a 2 (Fallito), come si può vedere in Tabella 4.15.

Gestione del fallimento

Le attività di fallimento e compensazione sono supportate dal modello Prism generato. La traduzione, come detto in precedenza, prevede di inserire in sequenza le azioni da effettuare per gestire il fallimento, seguite dalle attività per la gestione della compensazione. B*lite* permette di associare ad ogni scope un'attività di fallimento e una di compensazione prima di eseguire quelle dello scope che contiene lo scope in esame. Come si può vedere in Tabella 4.16, per permettere questo tipo di approccio B*lite*2Prism genera una variabile Prism (`fau1`) intera di quattro stati che indicano rispettivamente:

- 0: lo stato non inizializzato;
- 1: lo stato inizializzato;

Modulo di Fallimento
`[act1 fh: act2 ch: act3]`
Parametri in input
`[[fault]]`: Lista dei moduli di fallimento `[[add]]`: Condizioni addizionali
Traduzione - Corpo del modulo Prism

`int fau1: [0..3] init 0`
 	
`{{act1}}`
 	
`// fau1 ----`
`[] ( [[add]] & processStatus=2 & fau0=SUCC ) -> 1:(fau1' = 1);`
 	
`// fault handler ----`
`[] ( processStatus = 2 & fau1 = 1 & ${{act2}}[0]=0 ) -> ..`
`{{act2}}`
 	
`// compensation handler ----`
`{{act3}}`
`[] ( processStatus = 2 & fau1 = 1 & ${{act3}}[last]=SUCC ) -> 1:(fau1' = 2);`

Parametri in Output - Traduzione del corpo
`[[add]] <- fau1=1` `[[add]] <- serviceStatus=2`

Tabella 4.16: Traduzione del fallimento

- 2: lo stato di fallimento;

- 3: lo stato di terminazione con successo.

Grazie a questa variabile è possibile eseguire i vari moduli di fallimento in ordine corretto: infatti ogni scope contenente scope annidati controllerà se gli scope contenuti in esso sono non falliti o terminati con successo prima di eseguire il proprio modulo di gestione del fallimento.

Nella traduzione dell'esempio si suppone che vi sia un altro modulo di fallimento `fau0` all'interno di `act1`; invece il modulo di fallimento che viene generato è chiamato `fau1`. A differenza degli altri metodi di traduzione questo non necessita di avere in ingresso le condizioni di sequenzializzazione, ma necessita solo di sapere quale sia il modulo di fallimento del quale attendere la terminazione con successo o fallimento, nel caso sia presente. Come possiamo vedere nella Tabella 4.16 nella parte di Traduzione, i comandi hanno delle condizioni per forzare l'avvio del modulo solo nel caso che il modulo da attendere sia terminato con successo. Nel caso siano presenti solo delle attività di fallimento o solo delle attività di compensazione, il metodo di traduzione procede come descritto, ma trasforma soltanto l'attività presente.

Label

Per semplificare la fase di analisi, per ogni operatore di comunicazione sono generate delle label che contraddistinguono alcuni stati interessanti. In questo modo, nella fase di analisi, per riferirsi ad un particolare stato del modello basterà utilizzare l'etichetta corrispondente, il cui nome deriva dallo stato descritto ed è quindi più semplice da utilizzare, rispetto al valore numerico assunto dalle variabili in quello stato.

Listing 4.3: Esempio di label

```
label "rec0Success" = rec0 = 3;
label "rec0Available" = rec0 = 1;
label "rec0NotInit" = rec0 = 0;
```

4.3 Esempio d'Uso

In questa sezione descriveremo un utilizzo tipico di B*lite*C per la generazione di una specifica Prism e indicheremo come quest'ultima possa essere utilizzata per testare proprietà di interesse definite dall'utente.
Gli elementi necessari per poter effettuare questa dimostrazione sono:

- Java 1.6;
- B*lite*C;[1]
- Prism;
- una specifica B*lite*.

Data una specifica B*lite* (`esempio.bl`) per generare il modello Prism è sufficiente eseguire il comando nel Listing 4.4.

Listing 4.4: Esecuzione B*lite*C

```
java -jar blitec.jar -o prism esempio.bl
```

Al termine dell'esecuzione l'utente troverà nella directory contenente il file `esempio.bl` un nuovo file `esempio.pm`, che conterrà il modello Prism.

Esiste un'alternativa al comando precedente, nel caso l'utente voglia generare il file in una directory diversa da quella di lavoro. Il comando, in questo caso, è analogo al primo ma avrà un parametro aggiuntivo, come si vede nel Listing 4.5.

[1] B*lite*C è disponibile in formato eseguibile presso `http://rap.dsi.unifi.it/blite/index.php/blitec-a-tool-for-developing-ws-bpel-applications/`. I sorgenti, rilasciati sono licenza GPLv3, sono reperibili presso `http://blitec.bewq.org`

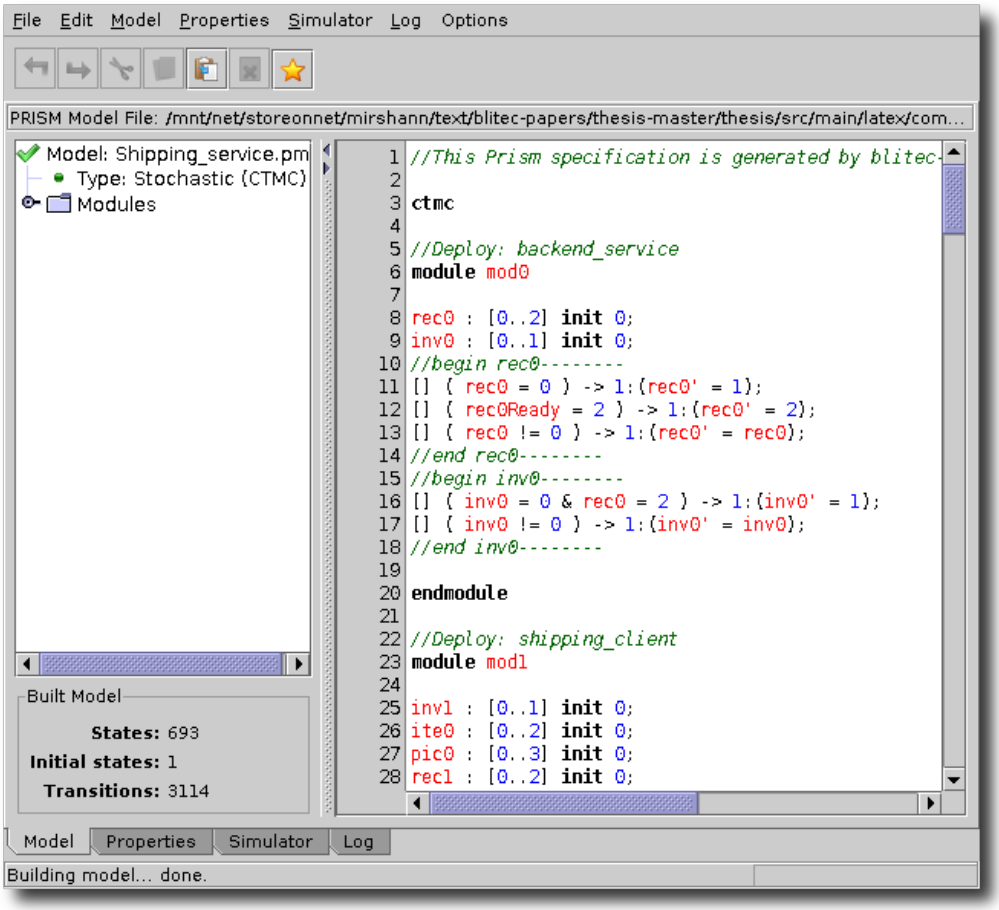


Figura 4.2: Finestra di specifica del modello

Listing 4.5: Esecuzione B*lite*C con directory personalizzata

```
java -jar blitec.jar -o prism esempio.bl -t directory_di_generazione
```

Eseguito uno dei precedenti comandi, la fase di traduzione è terminata.

4.3.1 Caricamento della specifica in Prism

4.3.2 Generazione della specifica

Per eseguire questa fase, l'utente dovrà aver installato il model checker Prism e lanciarne l'esecuzione. Data un'istanza di Xprism, per caricare il modello sarà necessario selezionare con il mouse **Model** > **Open Model...** e selezionare il file **esempio.pm**. Dopodiché l'utente dovrà generare il modello selezionando **Model** > **Build Model**. Effettuata questa operazione l'utente otterrà una finestra come quella in Figura 4.2.

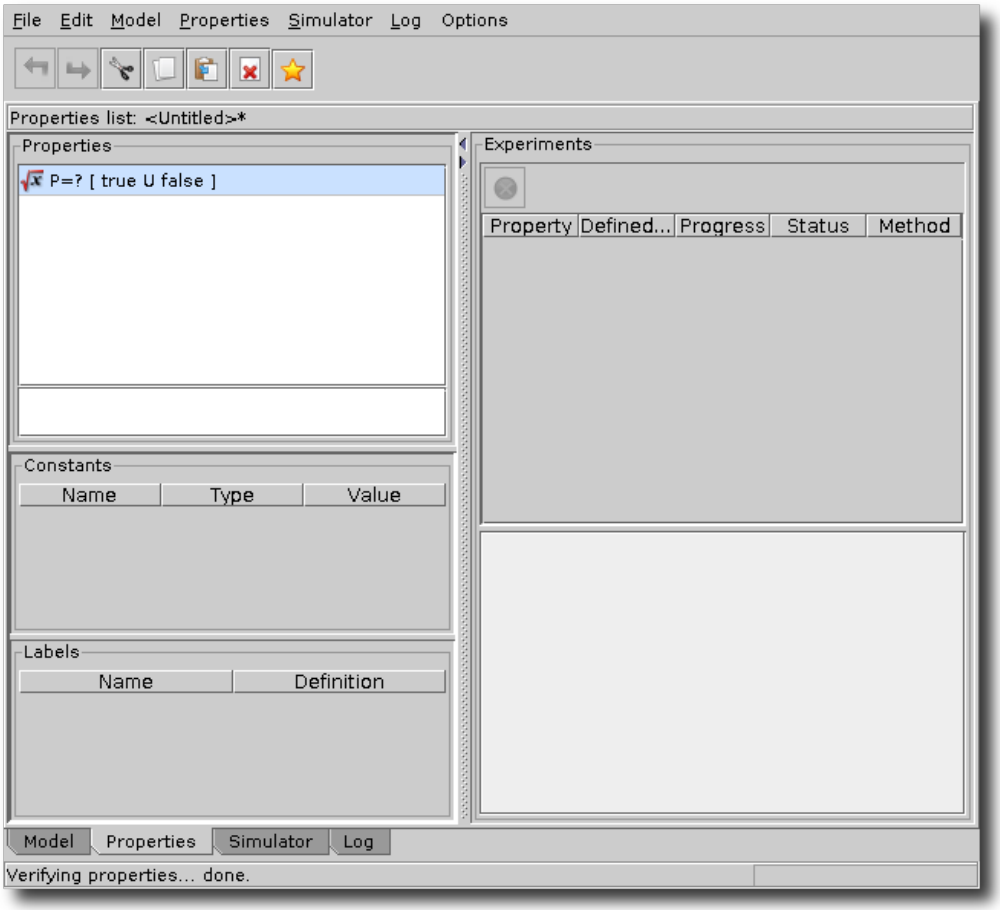


Figura 4.3: Finestra delle proprietà

4.3.3 Analisi del modello

Per editare e verificare delle formule dovremo cliccare con il mouse sul tab **Properties** in basso a sinistra nella finestra. Nella nuova finestra potremo aggiungere formule cliccando con il tasto destro del mouse nel box **Properties** e selezionando dal menù a tendina la voce **Add**. Nella maschera che comparirà potremo editare la formula.
Terminato l'inserimento, potremo testare una formula selezionandola e, quindi, cliccando su **Propeties** > **Verify**. Per un esempio di finestra con risultato delle formule si veda la Figura 4.3.

Per un utilizzo più avanzato delle funzionalità di verifica consigliamo al lettore di visionare il manuale di Prism [34].

Capitolo 5

Caso di studio: carta di credito virtuale

In questo capitolo illustreremo le traduzioni di servizi effettuata con B*lite*2Prism tramite l'esempio della carta di credito virtuale introdotto in Sezione 2.2.2.

Dopo una breve descrizione del funzionamento, sarà illustrata la traduzione e di seguito la verifica di alcune proprietà di interesse. L'esempio espliciterà i benifici della conversione automatica da B*lite* a Prism e di come sia semplificata la fase di analisi anche grazie alle *Label* create nella traduzione.

Le componenti di questa sezione simulano il comportamento di una carta prepagata e del relativo possessore. Questo caso di studio rappresenta la trasposizione della specifica B*lite* presentata nell'introduzione in B*lite*C.

5.1 Descrizione

Lo scenario si compone di due attori:

1. un fornitore di carte virtuali;
2. un client del servizio.

5.1.1 Virtual Card Provider

Il fornitore di carte virtuali è definito nel Listing 5.1. Il funzionamento del servizio è molto semplice: il servizio attende che gli sia trasmesso il numero di carta e la quantità di credito da caricare per la creazione della carta virtuale; dopo questa operazione attende richieste di prelievo fino all'esaurimento del credito. Il codice nel Listing 5.1 risulta molto simile a quello introdotto precedentemente in Tabella 2.1: le uniche evidenti differenze sono le annotazioni aggiunte per la verifica con Prism.

Listing 5.1: Specifica B*lite* del fornitore di carte prepagate

```
virtualcard ::= {s_vcard}{x_id};;

s_vcard ::=
[
  seq
    rcv <p_createcard,cb_createcard> o_newcard <x_id,x_cash>;

    @declare: cash~2
    @condition: "cash<2"
    if(x_cash>0)
    {
      seq
        x_resp:="Carta Virtuale n. ". x_id . " creata con successo";

        inv <cb_createcard> o_newcard <x_id,x_resp>;


        @condition: "cash<2"
        while(x_cash>0)
        {
          seq
            rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;

            @prob: WDR~100
            @update: "cash'=cash+1"
            if(x_cash>=x_wdr)
            {
              seq
                x_cash:=x_cash-x_wdr;
                x_resp := "Prelievo di ". x_wdr . "Euro accettato"
              qes
            }
            else
            {
               x_resp := "Prelievo di ". x_wdr . "Euro non accettato"
            };
            inv <x_clt> o_getcash <x_id,x_resp>
          qes
        }
      qes
    }
    else
    {
      seq
        x_resp:="Carta Virtuale n. ". x_id . " non creata. Credito Insufficiente";
        inv <cb_createcard> o_newcard <x_id,x_resp>
      qes
    }
  qes
];;
```

Commenteremo adesso, costrutto per costrutto, la traduzione della specifica del servizio.

Il primo costrutto incontrato nella traduzione è `seq`, dunque il traduttore genererà un modulo dove inserire tutti i costrutti contenuti nel corpo dell'operatore di sequenzializzazione.

Il costrutto successivo nella specifica del servizio B*lite* è una ricezione che permette l'attivazione del processo. Per trattare questo costrutto, B*lite*2Prism memorizzerà le informazioni sull'endpoint utilizzato e poi genererà il codice presente nel Listing 5.2.

Listing 5.2: Dettaglio della traduzione della prima receive del virtual card provider

```
// rec0 ----
[] ( virtualcardStatus = 0 & rec0 = 0 ) -> 1:(rec0' = 1);
```

Le informazioni sull'endpoint che sono state memorizzate saranno necessarie alla generazione del canale al termine della scansione di tutti i processi partecipanti al modello. Il comando nel Listing 5.2 attiva la ricezione di messaggi da parte dell'attività di ricezione.

Il costrutto incontrato successivamente è quello condizionale, come possiamo vedere nel Listing 5.3.

Listing 5.3: Dettaglio Virtual Card Provider

```
@declare: cash~2
@condition: "cash<2"
if(x_cash>0)
{
  seq
    x_resp:="Carta Virtuale n. ". x_id . " creata con successo";

    inv <cb_createcard> o_newcard <x_id,x_resp>;


    @condition: "cash<2"
    while(x_cash>0)
    {
      seq
        rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;

        @prob: WDR~100
        @update: "cash'=cash+1"
        if(x_cash>=x_wdr)
        {
          seq
            x_cash:=x_cash-x_wdr;
            x_resp := "Prelievo di ". x_wdr . "Euro accettato"
          qes
        }
        else
```

```
        {
            x_resp := "Prelievo di ". x_wdr . "Euro non accettato"
        };
        inv <x_clt> o_getcash <x_id,x_resp>
      qes
    }
  qes
 }
 else
 {
  seq
    x_resp:="Carta Virtuale n. ". x_id . " non creata. Credito Insufficiente";
    inv <cb_createcard> o_newcard <x_id,x_resp>
  qes
 }
```

La traduzione, riportata nel Listing 5.4, ci mostra che il costrutto non può essere valutato finché la receive non è terminata con successo (`rec0=3`). Quando la precedente condizione è soddisfatta, viene effettuata la scelta del ramo dell'attività condizionale con la quale il modello continuerà l'evoluzione. In questo caso la scelta non è effettuata attraverso un rate, ma attraverso una condizione su una variabile aggiuntiva.

Listing 5.4: Dettaglio della traduzione del costrutto condizionale del card provider

```
// con0 ----
[] ( virtualcardStatus = 0 & rec0 = 3 & cash < 2 & con0 = 0 ) -> 1:(con0' = 1);
[] ( virtualcardStatus = 0 & rec0 = 3 & !(cash < 2) & con0 = 0 ) -> 1:(con0' = 2);

// inv0 ----
[] ( virtualcardStatus = 0 & con0 = 1 & (end1Length < MAX_ENDPOINT_BUFFER_LENGHT &
     inv0MsgStatus < MAX_QUEUE_LENGTH) & inv0 = 0 ) -> MEDIUM_RELIABILITY:(
     inv0MsgStatus' = inv0MsgStatus + 1) & (inv0' = 1) + 100 - MEDIUM_RELIABILITY:(
     virtualcardStatus' = 2);
[] ( virtualcardStatus = 0 & con0 = 1 & (end1Length >= MAX_ENDPOINT_BUFFER_LENGHT |
     inv0MsgStatus >= MAX_QUEUE_LENGTH) & inv0 = 0 ) -> 1:(virtualcardStatus' = 2);

// ite0 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & inv1 = 1 ) -> 1:(ite0' = 0);
[] ( virtualcardStatus = 0 & con0 = 1 & inv0 = 1 & rec1 = 0 & con1 = 0 & inv1 = 0 &
     cash < 2 & ite0 = 0 ) -> 1:(ite0' = 1);
[] ( virtualcardStatus = 0 & con0 = 1 & inv0 = 1 & rec1 = 0 & con1 = 0 & inv1 = 0 &
     !(cash < 2) & ite0 = 0 ) -> 1:(ite0' = 2);

// rec1 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & rec1 = 0 ) -> 1:(rec1' = 1);
[] ( virtualcardStatus = 0 & ite0 = 0 & rec1 = 3 ) -> 1:(rec1' = 0);

// con1 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & rec1 = 3 & con1 = 0 ) -> WDR:(cash' = cash +
     1) & (con1' = 1) + 100 - WDR:(con1' = 2);
[] ( virtualcardStatus = 0 & ite0 = 0 & con1 != 0 ) -> 1:(con1' = 0);
```

```
// inv1 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & (con1 = 1 | con1 = 2) & (end3Length <
     MAX_ENDPOINT_BUFFER_LENGHT & inv1MsgStatus < MAX_QUEUE_LENGTH) & inv1 = 0 ) ->
     MEDIUM_RELIABILITY:(inv1MsgStatus' = inv1MsgStatus + 1) & (inv1' = 1) + 100 -
     MEDIUM_RELIABILITY:(virtualcardStatus' = 2);
[] ( virtualcardStatus = 0 & ite0 = 1 & (con1 = 1 | con1 = 2) & (end3Length >=
     MAX_ENDPOINT_BUFFER_LENGHT | inv1MsgStatus >= MAX_QUEUE_LENGTH) & inv1 = 0 ) ->
     1:(virtualcardStatus' = 2);
[] ( virtualcardStatus = 0 & ite0 = 0 & inv1 = 1 ) -> 1:(inv1' = 0);

// inv2 ----
[] ( virtualcardStatus = 0 & con0 = 2 & (end1Length < MAX_ENDPOINT_BUFFER_LENGHT &
     inv2MsgStatus < MAX_QUEUE_LENGTH) & inv2 = 0 ) -> MEDIUM_RELIABILITY:(
     inv2MsgStatus' = inv2MsgStatus + 1) & (inv2' = 1) + 100 - MEDIUM_RELIABILITY:(
     virtualcardStatus' = 2);
[] ( virtualcardStatus = 0 & con0 = 2 & (end1Length >= MAX_ENDPOINT_BUFFER_LENGHT |
     inv2MsgStatus >= MAX_QUEUE_LENGTH) & inv2 = 0 ) -> 1:(virtualcardStatus' = 2);
```

Potendo effettuare più scelte, le condizioni associate ai vari rami saranno differenti. Infatti la receive che si raggiunge quando la condizione è vera dipenderà dal controllo `con0=1`, invece l'altra scelta dipenderà dal controllo `con0=2`.

La traduzione dell'attività di invocazione nel ramo `then`, risulta molto semplice, infatti questa è composta da due comandi. Il primo, in ordine di introduzione, effettua l'invio del messaggio tenendo conto dell'affidabilità del mezzo, il secondo invece fa fallire il processo nel caso che il buffer dell'endpoint o dell'invocazione sia saturo. Come per la precedente receive le informazioni sull'endpoint sono memorizzate per la generazione del canale.

Il costrutto seguente è quello iterativo, si veda Listing 5.5

Listing 5.5: Dettaglio Virtual Card Provider

```
@condition: "cash<2"
while(x_cash>0)
{
  seq
    rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;

    @prob: WDR~100
    @update: "cash'=cash+1"
    if(x_cash>=x_wdr)
    {
      seq
        x_cash:=x_cash-x_wdr;
        x_resp := "Prelievo di ". x_wdr . "Euro accettato"
      qes
    }
    else
    {
```

```
            x_resp := "Prelievo di ". x_wdr . "Euro non accettato"
        };
        inv <x_clt> o_getcash <x_id,x_resp>
      qes
    }
```

La traduzione del comportamento del costrutto, seguendo i comandi riportati nel Listing 5.6, si ha unendo il comportamento del primo comando Prism, che sceglie se la condizione è vera o falsa e del secondo comando, che resetta la condizione della variabile del costrutto di iterazione per permettere la rivalutazione della condizione del ciclo. È importante notare che la valutazione della condizione a guardia del ciclo è effettuata solo quando tutti i costrutti nel corpo del ciclo sono non inizializzati, questo perché altrimenti si potrebbero avere comportamenti anomali come la valutazione della guardia quando ancora alcuni costrutti non sono stati reinizializzati.

Il comportamento dei costrutti nel corpo del ciclo è simile a quello della precedente trattazione, ad eccezione di alcune piccole differenze. La prima aggiunta apportata è il controllo sulla condizione del costrutto di iterazione (`ite0=1`), in modo da non eseguire il costrutto nel caso la condizione di iterazione non sia soddisfatta. L'altra aggiunta è un comando necessario alla reinizializzazione delle variabili di tutti i costrutti contenuti nel corpo del ciclo; ogni costrutto ha una comando Prism che controlla se il ciclo è in stato non inizializzato e, nel caso lo sia, resetta il valore del costrutto a non inizializzato (`0`).

Listing 5.6: Dettaglio della traduzione del costrutto di iterazione del card provider

```
// ite0 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & inv1 = 1 ) -> 1:(ite0' = 0);
[] ( virtualcardStatus = 0 & con0 = 1 & inv0 = 1 & rec1 = 0 & con1 = 0 & inv1 = 0 &
     cash < 2 & ite0 = 0 ) -> 1:(ite0' = 1);
[] ( virtualcardStatus = 0 & con0 = 1 & inv0 = 1 & rec1 = 0 & con1 = 0 & inv1 = 0 &
     !(cash < 2) & ite0 = 0 ) -> 1:(ite0' = 2);
```

La rimanente traduzione sfrutta i concetti prima esplicitati per generare il codice dei rimanenti costrutti.
La traduzione completa in Prism del servizio presentato nel Listing 5.1 è riportata nel Listing 5.7.

Listing 5.7: Traduzione in Prism dell'utilizzatore della carta virtuale

```
module virtualcard

  inv0 : [0..1] init 0;
  cash : [0..2] init 0;
  con1 : [0..2] init 0;
  inv1 : [0..1] init 0;
  ite0 : [0..2] init 0;
  inv2 : [0..1] init 0;
```

```
con0 : [0..2] init 0;

// virtualcard ----
[] ( virtualcardStatus = 0 & (inv2 = 1 | ite0 = 2) ) -> 1:(virtualcardStatus' = 1);
[] ( virtualcardStatus = 1 ) -> 1:(virtualcardStatus' = 1);
[] ( virtualcardStatus = 2 ) -> 1:(virtualcardStatus' = 2);

// rec0 ----
[] ( virtualcardStatus = 0 & rec0 = 0 ) -> 1:(rec0' = 1);

// con0 ----
[] ( virtualcardStatus = 0 & rec0 = 3 & cash < 2 & con0 = 0 ) -> 1:(con0' = 1);
[] ( virtualcardStatus = 0 & rec0 = 3 & !(cash < 2) & con0 = 0 ) -> 1:(con0' = 2);

// inv0 ----
[] ( virtualcardStatus = 0 & con0 = 1 & (end1Length < MAX_ENDPOINT_BUFFER_LENGHT &
     inv0MsgStatus < MAX_QUEUE_LENGTH) & inv0 = 0 ) -> MEDIUM_RELIABILITY:(
     inv0MsgStatus' = inv0MsgStatus + 1) & (inv0' = 1) + 100 - MEDIUM_RELIABILITY:(
     virtualcardStatus' = 2);
[] ( virtualcardStatus = 0 & con0 = 1 & (end1Length >= MAX_ENDPOINT_BUFFER_LENGHT |
     inv0MsgStatus >= MAX_QUEUE_LENGTH) & inv0 = 0 ) -> 1:(virtualcardStatus' = 2);

// ite0 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & inv1 = 1 ) -> 1:(ite0' = 0);
[] ( virtualcardStatus = 0 & con0 = 1 & inv0 = 1 & rec1 = 0 & con1 = 0 & inv1 = 0 &
     cash < 2 & ite0 = 0 ) -> 1:(ite0' = 1);
[] ( virtualcardStatus = 0 & con0 = 1 & inv0 = 1 & rec1 = 0 & con1 = 0 & inv1 = 0 &
     !(cash < 2) & ite0 = 0 ) -> 1:(ite0' = 2);

// rec1 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & rec1 = 0 ) -> 1:(rec1' = 1);
[] ( virtualcardStatus = 0 & ite0 = 0 & rec1 = 3 ) -> 1:(rec1' = 0);

// con1 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & rec1 = 3 & con1 = 0 ) -> WDR:(cash' = cash +
     1) & (con1' = 1) + 100 - WDR:(con1' = 2);
[] ( virtualcardStatus = 0 & ite0 = 0 & con1 != 0 ) -> 1:(con1' = 0);

// inv1 ----
[] ( virtualcardStatus = 0 & ite0 = 1 & (con1 = 1 | con1 = 2) & (end3Length <
     MAX_ENDPOINT_BUFFER_LENGHT & inv1MsgStatus < MAX_QUEUE_LENGTH) & inv1 = 0 ) ->
     MEDIUM_RELIABILITY:(inv1MsgStatus' = inv1MsgStatus + 1) & (inv1' = 1) + 100 -
     MEDIUM_RELIABILITY:(virtualcardStatus' = 2);
[] ( virtualcardStatus = 0 & ite0 = 1 & (con1 = 1 | con1 = 2) & (end3Length >=
     MAX_ENDPOINT_BUFFER_LENGHT | inv1MsgStatus >= MAX_QUEUE_LENGTH) & inv1 = 0 ) ->
     1:(virtualcardStatus' = 2);
[] ( virtualcardStatus = 0 & ite0 = 0 & inv1 = 1 ) -> 1:(inv1' = 0);

// inv2 ----
[] ( virtualcardStatus = 0 & con0 = 2 & (end1Length < MAX_ENDPOINT_BUFFER_LENGHT &
     inv2MsgStatus < MAX_QUEUE_LENGTH) & inv2 = 0 ) -> MEDIUM_RELIABILITY:(
     inv2MsgStatus' = inv2MsgStatus + 1) & (inv2' = 1) + 100 - MEDIUM_RELIABILITY:(
     virtualcardStatus' = 2);
```

```
  [] ( virtualcardStatus = 0 & con0 = 2 & (end1Length >= MAX_ENDPOINT_BUFFER_LENGHT |
       inv2MsgStatus >= MAX_QUEUE_LENGTH) & inv2 = 0 ) -> 1:(virtualcardStatus' = 2);

endmodule
```

5.1.2 Virtual Card Client

Il client è presentato nel Listing 5.8. Il servizio in esame invia un messaggio al fornitore di virtual card per creare la carta prepagata, poi richiede prelievi fino all'esaurimento del credito. Le differenze della specifica B*lite*C con quella B*lite* riportata in Tabella 2.2 sono più marcate rispetto al precedente servizio: infatti, oltre alle annotazioni necessarie alla verifica, l'utente dovrà specificare i valori delle variabili dell'istanza inserendoli tra i costrutti `let ... in` e non attraverso la forma $\{x_{id} \mapsto 42\}$. La rimanente specifica B*lite*C risulta molto vicina alla controparte B*lite*.

Listing 5.8: Specifica B*lite* del servizio client

```
vcard_owner ::= {s_owner}{x_id};;

s_owner ::=
  let
      x_id := 42;
      x_cash := 500;
      x_g := 120;
      x_wdr := 120;
  in
  [
      seq
        inv <p_createcard,cb_createcard> o_newcard <x_id,x_cash>;
        rcv <cb_createcard> o_newcard <x_id,x_cash>;

        @condition: "cash<2"
        while(x_g <= x_cash)
        {
          seq
            inv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;
            rcv <x_clt> o_getcash <x_id,x_wdr>;
            x_g := x_g + x_wdr
          qes
        }
      qes
  ];;
```

La traduzione del possessore della carta prepagata avviene in modo simile alla precedente, infatti sono usati gli stessi costrutti introdotti nella precedente sottosezione. La traduzione in Prism è riportata nel Listing 5.9.

Listing 5.9: Traduzione in Prism del servizio client

```
module vcard_owner

  inv3 : [0..1] init 0;
  inv4 : [0..1] init 0;
  ite1 : [0..2] init 0;

  // vcard_owner ----
  [] ( vcard_ownerStatus = 0 & ite1 = 2 ) -> 1:(vcard_ownerStatus' = 1);
  [] ( vcard_ownerStatus = 1 ) -> 1:(vcard_ownerStatus' = 1);
  [] ( vcard_ownerStatus = 2 ) -> 1:(vcard_ownerStatus' = 2);

  // inv3 ----
  [] ( vcard_ownerStatus = 0 & (end0Length < MAX_ENDPOINT_BUFFER_LENGHT & inv3MsgStatus
       < MAX_QUEUE_LENGTH) & inv3 = 0 ) -> MEDIUM_RELIABILITY:(inv3MsgStatus' =
      inv3MsgStatus + 1) & (inv3' = 1) + 100 - MEDIUM_RELIABILITY:(vcard_ownerStatus'
      = 2);
  [] ( vcard_ownerStatus = 0 & (end0Length >= MAX_ENDPOINT_BUFFER_LENGHT |
      inv3MsgStatus >= MAX_QUEUE_LENGTH) & inv3 = 0 ) -> 1:(vcard_ownerStatus' = 2);

  // rec2 ----
  [] ( vcard_ownerStatus = 0 & inv3 = 1 & rec2 = 0 ) -> 1:(rec2' = 1);

  // ite1 ----
  [] ( vcard_ownerStatus = 0 & ite1 = 1 & rec3 = 3 ) -> 1:(ite1' = 0);
  [] ( vcard_ownerStatus = 0 & rec2 = 3 & inv4 = 0 & rec3 = 0 & cash < 2 & ite1 = 0 )
      -> 1:(ite1' = 1);
  [] ( vcard_ownerStatus = 0 & rec2 = 3 & inv4 = 0 & rec3 = 0 & !(cash < 2) & ite1 = 0
      ) -> 1:(ite1' = 2);

  // inv4 ----
  [] ( vcard_ownerStatus = 0 & ite1 = 1 & (end2Length < MAX_ENDPOINT_BUFFER_LENGHT &
      inv4MsgStatus < MAX_QUEUE_LENGTH) & inv4 = 0 ) -> MEDIUM_RELIABILITY:(
      inv4MsgStatus' = inv4MsgStatus + 1) & (inv4' = 1) + 100 - MEDIUM_RELIABILITY:(
      vcard_ownerStatus' = 2);
  [] ( vcard_ownerStatus = 0 & ite1 = 1 & (end2Length >= MAX_ENDPOINT_BUFFER_LENGHT |
      inv4MsgStatus >= MAX_QUEUE_LENGTH) & inv4 = 0 ) -> 1:(vcard_ownerStatus' = 2);
  [] ( vcard_ownerStatus = 0 & ite1 = 0 & inv4 = 1 ) -> 1:(inv4' = 0);

  // rec3 ----
  [] ( vcard_ownerStatus = 0 & ite1 = 1 & inv4 = 1 & rec3 = 0 ) -> 1:(rec3' = 1);
  [] ( vcard_ownerStatus = 0 & ite1 = 0 & rec3 = 3 ) -> 1:(rec3' = 0);

endmodule
```

5.1.3 Canali

I canali usati dalla specifica Prism sono riportati nel Listing 5.10.

Listing 5.10: Rappresentazione in Prism dei canali

```
// ( Endpoints ) ----

module end3

  [] ( rec3 = 1 ) -> 1.0:(rec3' = 2);
  [] ( inv1MsgStatus > 0 & rec3 = 2 ) -> 1.0:(rec3' = 3) & (inv1MsgStatus' =
       inv1MsgStatus - 1);

endmodule

module end0

  [] ( rec0 = 1 ) -> 1.0:(rec0' = 2);
  [] ( inv3MsgStatus > 0 & rec0 = 2 ) -> 1.0:(rec0' = 3) & (inv3MsgStatus' =
       inv3MsgStatus - 1);

endmodule

module end2

  [] ( rec1 = 1 ) -> 1.0:(rec1' = 2);
  [] ( inv4MsgStatus > 0 & rec1 = 2 ) -> 1.0:(rec1' = 3) & (inv4MsgStatus' =
       inv4MsgStatus - 1);

endmodule

module end1

  [] ( rec2 = 1 ) -> 1.0:(rec2' = 2);
  [] ( inv2MsgStatus > 0 & inv0MsgStatus = 0 & rec2 = 2 ) -> 1.0:(rec2' = 3) & (
       inv2MsgStatus' = inv2MsgStatus - 1);
  [] ( inv0MsgStatus > 0 & inv2MsgStatus = 0 & rec2 = 2 ) -> 1.0:(rec2' = 3) & (
       inv0MsgStatus' = inv0MsgStatus - 1);
  [] ( inv0MsgStatus > 0 & inv2MsgStatus > 0 & rec2 = 2 ) -> 0.5:(rec2' = 3) & (
       inv0MsgStatus' = inv0MsgStatus - 1) + 0.5:(rec2' = 3) & (inv2MsgStatus' =
       inv2MsgStatus - 1);

endmodule
```

I canali sono generati attraverso le informazioni collezionate nella compilazione dei precedenti attori. Infatti dal Listing 5.10 possiamo notare come il modulo `end3` rappresenti l'endpoint di creazione della carta virtuale `<p_createcard, z_card> o_newcard`. Con questo canale interagiscono il costrutto di ricezione `rcv0` e il costrutto di invocazione `inv3`. I restanti canali sfruttano lo stesso meccanismo per rappresentare le interazioni tra i costrutti di comunicazione.

5.1.4 Label

Le *Label* generate da B*lite*2Prism sono mostrate nel Listing 5.11.

Listing 5.11: Label generate da B*lite*C

```
// ( Labels )----

// Labels of rec0 ----
label "rec0Success" = rec0 = 3;
label "rec0Available" = rec0 = 1;
label "rec0Selected" = rec0 = 2;
label "rec0NotInit" = rec0 = 0;

// Labels of con0 ----
label "con0True" = con0 = 1;
label "con0False" = con0 = 2;
label "con0NotInit" = con0 = 0;

// Labels of inv0 ----
label "inv0Success" = virtualcardStatus = 0 & inv0 = 1;
label "inv0Faulted" = virtualcardStatus = 2 & inv0 = 0;
label "inv0NotInit" = virtualcardStatus = 0 & inv0 = 0;

// Labels of ite0 ----
label "ite0False" = ite0 = 2;
label "ite0True" = ite0 = 1;
label "ite0NotInit" = ite0 = 0;

// Labels of rec1 ----
label "rec1Success" = rec1 = 3;
label "rec1Available" = rec1 = 1;
label "rec1Selected" = rec1 = 2;
label "rec1NotInit" = rec1 = 0;

// Labels of con1 ----
label "con1True" = con1 = 1;
label "con1False" = con1 = 2;
label "con1NotInit" = con1 = 0;

// Labels of inv1 ----
label "inv1Success" = virtualcardStatus = 0 & inv1 = 1;
label "inv1Faulted" = virtualcardStatus = 2 & inv1 = 0 & (con1 = 1 | con1 = 2);
label "inv1NotInit" = virtualcardStatus = 0 & inv1 = 0;

// Labels of inv2 ----
label "inv2Success" = virtualcardStatus = 0 & inv2 = 1;
label "inv2Faulted" = virtualcardStatus = 2 & inv2 = 0;
label "inv2NotInit" = virtualcardStatus = 0 & inv2 = 0;

// Labels of virtualcard ----
label "virtualcardRunning" = virtualcardStatus = 0;
label "virtualcardTerminated" = virtualcardStatus = 1;
```

```
label "virtualcardFaulted" = virtualcardStatus = 2;

// Labels of inv3 ----
label "inv3Success" = vcard_ownerStatus = 0 & inv3 = 1;
label "inv3Faulted" = vcard_ownerStatus = 2 & inv3 = 0;
label "inv3NotInit" = vcard_ownerStatus = 0 & inv3 = 0;

// Labels of rec2 ----
label "rec2Success" = rec2 = 3;
label "rec2Available" = rec2 = 1;
label "rec2Selected" = rec2 = 2;
label "rec2NotInit" = rec2 = 0;

// Labels of ite1 ----
label "ite1False" = ite1 = 2;
label "ite1True" = ite1 = 1;
label "ite1NotInit" = ite1 = 0;

// Labels of inv4 ----
label "inv4Success" = vcard_ownerStatus = 0 & inv4 = 1;
label "inv4Faulted" = vcard_ownerStatus = 2 & inv4 = 0;
label "inv4NotInit" = vcard_ownerStatus = 0 & inv4 = 0;

// Labels of rec3 ----
label "rec3Success" = rec3 = 3;
label "rec3Available" = rec3 = 1;
label "rec3Selected" = rec3 = 2;
label "rec3NotInit" = rec3 = 0;

// Labels of vcard_owner ----
label "vcard_ownerRunning" = vcard_ownerStatus = 0;
label "vcard_ownerTerminated" = vcard_ownerStatus = 1;
label "vcard_ownerFaulted" = vcard_ownerStatus = 2;
```

I nomi delle **label** permettono all'utente di scrivere le formule senza doversi ricordare i valori associati alla traduzione del costrutto.

Nel Listing 5.11 possiamo vedere che per ogni costrutto sono presenti delle **label** che evidenziano lo stato non inizializzato e lo stato di terminazione con successo, oltre ad altri stati di interesse.

5.2 Analisi

Mostreremo nel seguito alcune proprietà di interesse verificabili sul modello tradotto da B*lite*2Prism.

È da notare che il modello è stato annotato in modo da far dipendere sia il possessore della carta virtuale, sia il fornitore della carta dallo stesso credito, in modo tale da rendere

il modello più fedele alla realtà oltre a rendere minore il numero di stati del modello in Prism.

Proprietà 1: Non è possibile prelevare fino a quando la carta di credito non è creata

Questa proprietà verifica che non si possa eseguire un prelievo fino a quando non viene creata una carta prepagata da parte del servizio cliente.

La proprietà è resa in Prism attraverso due etichette generate da B*lite*2Prism, si veda il Listing 5.12.

Listing 5.12: Non è possibile prelevare finché la carta non è creata

```
P>=1 [ !"rec0Success"=>!"rec1Success" ]
```

La proprietà risulta verificata perché se non è mai eseguita la ricezione `rec1` prima della ricezione `rec0`, non è disponibile nessuna carta, dunque non è possibile effettuare nessun prelievo prima della creazione della carta.

Proprietà 2: Non è possibile prelevare dopo che il credito è esaurito

Questa proprietà verifica che dopo l'esaurimento del credito, e quindi la terminazione del modulo `vitualcard`, non sia possibile far variare lo stato dell'attività di ricezione `rec1`. Questa proprietà, visionabile nel Listing 5.13 è espressa in Prism attraverso l'ausilio delle etichette `rec1NotInit` e `ite0False`: la prima è usata perché al momento della scelta nel costrutto di iterazione `ite0` tutte attività contenute nel suo corpo devono essere reinizializzate, dunque, se l'attività di iterazione termina queste ultime rimarranno non inizializzate; l'altra è usata per evidenziare la valutazione falsa della condizione del costrutto di iterazione.

Listing 5.13: Non è possibile prelevare dopo che il credito è esaurito

```
P>=1 [ "ite0False"=>"rec1NotInit" ]
```

La proprietà risulta verificata perché dopo la terminazione dell'attività di iterazione il costrutto di ricezione non è più inizializzato, come desiderato.

Proprietà 3: Il client prima o poi termina

Questa proprietà controlla la terminazione dell'utente della carta virtuale. La proprietà risulta di semplice espressione grazie alla label `vcard_ownerTerminated`, infatti la proprietà necessita solo di questa condizione per essere espressa.

Listing 5.14: Il client prima o poi termina

```
P>=1 [ F "vcard_ownerTerminated" ]
```

In questo caso la terminazione dipende dall'affidabilità del mezzo, infatti nel caso questo sia 100 sicuramente il client terminerà, nel caso che invece sia minore di 100 la terminazione non è garantita in quanto si potrebbe perdere un messaggio e quindi il client potrebbe rimanere in attesa di risposta indeterminatamente.

Proprietà 4: La carta virtuale prima o poi termina

Come la proprietà precedente, anche questa proprietà è esprimibile attraverso la label `mod0Terminated`, come è possibile vedere nel Listing 5.15.

Listing 5.15: La carta virtuale prima o poi termina

```
P>=1 [ F "virtualcardTerminated" ]
```

Come la precedente anche questa risulta essere verificata in modo dipendente dall'affidabilità del mezzo.

Proprietà 5: Il client non effettua una richiesta di prelievo prima di aver creato la carta

La proprietà controlla che l'utente non effettui un'operazione di prelievo, prima di quella di creazione della carta. Anche in questo caso possono essere usate le label create da B*lite*2Prism infatti `inv3NotInit` e `inv4Success` sono le condizioni che dobbiamo testare. Nel Listing 5.16 è controllato se in nessuno stato si può trovare la richiesta di credito `inv4` terminata e la richiesta di creazione della carta `inv3` non inizializzata.

Listing 5.16: Il client non effettua una richiesta di prelievo prima di aver creato la carta

```
P>=1 [ G !("inv4Success"&"inv3NotInit") ]
```

La proprietà è ovviamente rispettata in quanto l'utente della carta virtuale è stato progettato nel modo corretto.

Proprietà 6: Calcolo della probabilità di prelievo dopo la creazione della carta

La proprietà che analizzeremo verifica che sia sempre possibile prelevare dopo aver creato una carta prepagata. Come si può vedere dal Listing 5.17 sono usate le label `rec0Success` e `rec1Available`, che indicano che rispettivamente il successo della ricezione per la creazione della carta e la disponibilità della ricezione per il prelievo di credito.

Listing 5.17: Probabilità di prelievo dopo aver creato la carta

```
P=? [ F( "rec0Success" & "rec1Available") ]
```

La proprietà restituisce probabilità minore di 1.0 se l'affidabilità del mezzo è minore di 100: questo perché a causa del mezzo inaffidabile, alcuni messaggi potrebbero andare persi non permettendo l'evoluzione del sistema.

Proprietà 7: Calcolo della probabilità che il client si sincronizzi con il fornitore di carte virtuali

Con questa verifica si cercherà di calcolare la probabilità che avvenga una sincronizzazione tra il client ed il fornitore di carte virtuali. Nel Listing 5.18 possiamo notare l'utilizzo della label `rec0Success` che evidenzia l'avvenuta sincronizzazione tra l'attività di ricezione `rec0` e l'attività di invocazione `inv3`.

Listing 5.18: Probabilità che il client si sincronizzi con il fornitore delle carte virtuali

```
P=? [ F ("rec0Success") ]
```

La verifica della proprietà restituisce come risultato che la sincronizzazione avviene un numero di volte pari all'affidabilità del mezzo. Il valore è verosimile all'ambiente in cui lavora il sistema perché l'attività di ricezione e invio della richiesta sono la prima sincronizzazione effettuata dal modello, dunque nessun'altra attività potrà condizionare questo valore dell'affidabilità del mezzo.

Proprietà 8: Calcolo della probabilità di poter prelevare dopo aver già fatto un prelievo

Questa proprietà controlla la probabilità di effettuare un prelievo dopo averne effettuato uno in precedenza. La proprietà è mostrata nel Listing 5.19. La proprietà risulta più complessa delle precedenti a causa dell'utilizzo dell'operatore **F** annidato.

Listing 5.19: Probabilità di prelevare dopo un altro prelievo

```
P=? [ F ("rec1Success"&(F (!"rec1Success"&(X "rec1Success")))) ]
```

La formula è esprimibile in modo semplice con la frase :"Prima o poi si effettua un prelievo, dopo il quale prima o poi si effettuerà un'altra sincronizzazione". L'espressione `(!''rec1Succ'' & (X ''rec1Succ''))` è necessaria perché dopo il primo prelievo la variabile di sincronizzazione deve essere resettata e reimpostata.

Il risultato della verifica è strettamente legato all'affidabilità del mezzo, infatti se il mezzo è completamente affidabile la risposta è probabilità 1.0, se invece si fa diminuire l'affidabilità il valore della probabilità scende velocemente. Infatti per un'affidabilità dell'80% abbiamo una probabilità di circa 33%.

Proprietà 9: Calcolo della disponibilità di prelievo dopo la creazione della carta (esperimento al variare del tempo)

La proprietà nel Listing 5.20 verifica la disponibilità di poter prelevare al variare del tempo.

Listing 5.20: Disponibilità di prelievo dopo la creazione della carta

```
P=? [ F<=TIME ("rec0Success"&"ite0True"&"rec1Available") ]
```

Questa proprietà fa uso degli esperimenti di Prism e di una costante non vincolata `TIME` per verificare il comportamento del modello tra l'unità di tempo e 0 l'unità di tempo 30. Infatti a differenza delle precedenti formule, per questa viene verificata la probabilità di arrivare in uno stato che abbia effettuato la creazione della carta virtuale e che abbia l'attività di ricezione pronta alla comunicazione entro i prime 30 unità di tempo in cui passa il sistema.

Il grafico in Figura 5.1 mostra che nei prime 30 unità di tempo, per vari valori dell'affidabilità del mezzo di trasmissione. È facile vedere che per una piccola variazione dell'affidabilità c'è una grande variazione della probabilità di poter effettuare più volte il prelievo.

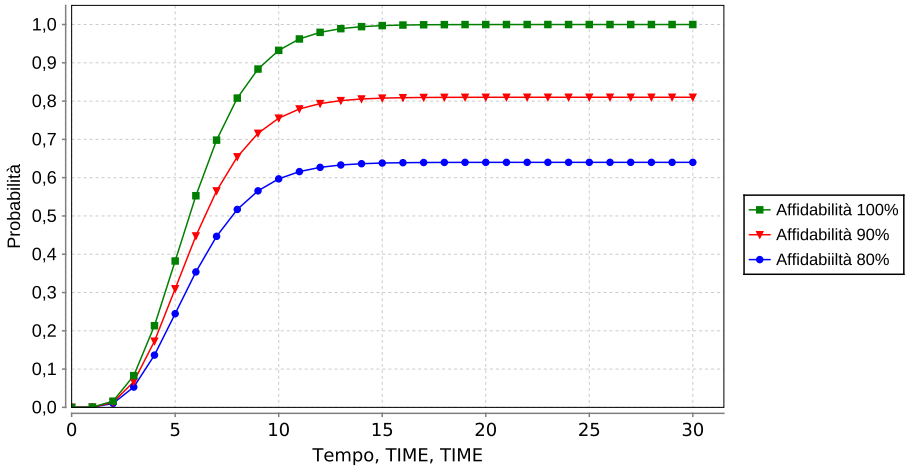


Figura 5.1: Disponibilità di prelievo

5.3 Compilazione WS-BPEL

Di seguito commenteremo brevemente la compilazione del servizio in esame per far notare la semplicità con cui l'utente può passare dalla fase di analisi a quella di esecuzione del processo su di un engine WS-BPEL. L'utente non dovrà fare altro che aggiungere una semplice parte di configurazione per predisporre il sistema alla compilazione.

Infatti specifica mostrata nel Listing 5.1 è compilabile per poter essere eseguita su di un engine WS-BPEL. Fornendo in ingresso al modulo per la compilazione di B*lite*C su Apache ODE, la specifica con l'aggiunta di una breve parte di configurazione(Listing 5.21), l'utente sarà in grado di eseguire il processo sull'engine.

Nel caso di Apache ODE, la compilazione eseguita sulla specifica genererà due pacchetti eseguibili dall'engine: uno relativo al client della carta ricaricabile e l'altro al servizio fornitore delle carte virtuali. Ogni pacchetto è composto di tre file: il file WS-BPEL associato al processo, il file *deploy.txt* contenente le informazione necessarie all'engine ed infine il file WSDL relativo al processo.

Listing 5.21: B*lite*C - Configurazione virtual card

```
<?blm

c1@virtualcard
::
ADDRESSES
{
  myns   => "http://example";
  myaddress => "http://localhost:8080/active-bpel/vcard";
}

VARIABLES
{
  <x_id,x_cash>  => gen:type1,
                    <part1, part2>,
                  <xsd:int, xsd:int>;

  <x_id,x_wdr> => gen:type2,
                  <part1,part2>,
                  <xsd:int,xsd:int>;

  <x_id,x_resp> => gen:type3,
                <part1,part2>,
                <xsd:int,xsd:string>;
}
::
?>
```

Fornitore delle carte virtuali

Il processo WS-BPEL associato alla specifica B*lite* del Listing 5.1 è riportato nel Listing 5.22.

Listing 5.22: Codice Bpel relativo al Fornitore delle schede

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by blitec-core84ec141c9e4b15570bf26ef1f2af39b9-->
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable" xmlns:bpel="http://docs.oasis-open.org/
      wsbpel/2.0/process/executable" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:mwl="http://example/
      virtualcard.wsdl" suppressJoinFailure="yes" name="virtualcardProcess" targetNamespace="http://example/
      virtualcard.bpel" queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0" expressionLanguage="urn:oasis
      :names:tc:wsbpel:2.0:sublang:xpath2.0">
  <import location="virtualcard.wsdl" namespace="http://example/virtualcard.wsdl" importType="http://schemas.xmlsoap.
        org/wsdl/" />
  <partnerLinks>
    <partnerLink name="cardPL" partnerLinkType="mwl:cardPLT" myRole="p_createcard" />
    <partnerLink name="cltPL" partnerLinkType="mwl:cltPLT" myRole="p_vcard" />
  </partnerLinks>
  <variables>
    <variable name="var1" messageType="mwl:type2" />
    <variable name="var2" messageType="mwl:type3" />
    <variable name="var0" messageType="mwl:type1" />
  </variables>
  <correlationSets>
```

```
  <correlationSet name="x_idCorr" properties="mwl:x_idProp" />
</correlationSets>
<faultHandlers>
  <catchAll>
    <sequence>
      <compensate />
      <empty />
    </sequence>
  </catchAll>
</faultHandlers>
<sequence>
  <receive partnerLink="cardPL" operation="o_newcard" variable="var0" createInstance="yes">
    <correlations>
      <correlation set="x_idCorr" initiate="yes" />
    </correlations>
  </receive>
  <if>
    <condition>$var0/payload/mwl:type1EL/mwl:x_cashEL &gt; 0</condition>
    <sequence>
      <sequence>
        <assign>
          <copy>
            <from>
              <literal>
                <type3EL xmlns="http://example/virtualcard.wsdl">
                  <part1EL />
                  <part2EL />
                </type3EL>
              </literal>
            </from>
            <to variable="var2" part="payload" />
          </copy>
        </assign>
        <assign>
          <copy>
            <from>concat('Carta Virtuale n. ',string($var0/payload/mwl:type1EL/mwl:x_idEL),' creata con successo')</
                  from>
            <to variable="var2" part="payload">
              <query>//mwl:type3EL/mwl:x_respEL</query>
            </to>
          </copy>
        </assign>
      </sequence>
      <sequence>
        <assign>
          <copy>
            <from variable="var0" part="payload">
              <query>//mwl:type1EL/mwl:x_idEL</query>
            </from>
            <to variable="var2" part="payload">
              <query>//mwl:type3EL/mwl:x_idEL</query>
            </to>
          </copy>
        </assign>
        <reply operation="o_newcard" partnerLink="cardPL" variable="var2">
          <correlations>
            <correlation set="x_idCorr" initiate="no" />
          </correlations>
        </reply>
      </sequence>
    </sequence>
    <else>
      <sequence>
        <assign>
          <copy>
            <from>concat('Carta Virtuale n. ',string($var0/payload/mwl:type1EL/mwl:x_idEL),' non creata. Credito
                  Insufficiente')</from>
            <to variable="var2" part="payload">
              <query>//mwl:type3EL/mwl:x_respEL</query>
            </to>
          </copy>
```

```
          </assign>
          <sequence>
            <assign>
              <copy>
                <from variable="var0" part="payload">
                  <query>//mwl:type1EL/mwl:x_idEL</query>
                </from>
                <to variable="var2" part="payload">
                  <query>//mwl:type3EL/mwl:x_idEL</query>
                </to>
              </copy>
            </assign>
            <reply operation="o_newcard" partnerLink="cardPL" variable="var2">
              <correlations>
                <correlation set="x_idCorr" initiate="no" />
              </correlations>
            </reply>
          </sequence>
        </sequence>
      </else>
    </if>
    <while>
      <condition>$var0/payload/mwl:type1EL/mwl:x_cashEL &gt; 0</condition>
      <sequence>
        <receive partnerLink="cltPL" operation="o_getcash" variable="var1">
          <correlations>
            <correlation set="x_idCorr" initiate="no" />
          </correlations>
        </receive>
        <if>
          <condition>$var0/payload/mwl:type1EL/mwl:x_cashEL &gt;= $var1/payload/mwl:type2EL/mwl:x_wdrEL</condition>
          <sequence>
            <assign>
              <copy>
                <from>$var0/payload/mwl:type1EL/mwl:x_cashEL - $var1/payload/mwl:type2EL/mwl:x_wdrEL</from>
                <to variable="var0" part="payload">
                  <query>//mwl:type1EL/mwl:x_cashEL</query>
                </to>
              </copy>
            </assign>
            <assign>
              <copy>
                <from>concat('Prelievo di ',string($var1/payload/mwl:type2EL/mwl:x_wdrEL),'Euro accettato')</from>
                <to variable="var2" part="payload">
                  <query>//mwl:type3EL/mwl:x_respEL</query>
                </to>
              </copy>
            </assign>
          </sequence>
          <else>
            <assign>
              <copy>
                <from>concat('Prelievo di ',string($var1/payload/mwl:type2EL/mwl:x_wdrEL),'Euro non accettato')</from>
                <to variable="var2" part="payload">
                  <query>//mwl:type3EL/mwl:x_respEL</query>
                </to>
              </copy>
            </assign>
          </else>
        </if>
        <sequence>
          <assign>
            <copy>
              <from variable="var1" part="payload">
                <query>//mwl:type2EL/mwl:x_idEL</query>
              </from>
              <to variable="var2" part="payload">
                <query>//mwl:type3EL/mwl:x_idEL</query>
              </to>
            </copy>
          </assign>
```

```
          <reply operation="o_getcash" partnerLink="cltPL" variable="var2">
            <correlations>
              <correlation set="x_idCorr" initiate="no" />
            </correlations>
          </reply>
        </sequence>
      </sequence>
    </while>
  </sequence>
</process>
```

Il WSDL associato alla specifica WS-BPEL del processo analizzato è quella nel Listing 5.23. Questo documento è molto importante perché fondamentale per permettere la comunicazione con il processo.

Listing 5.23: Specifica WSDL associata al processo del fornitore delle schede

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by blitec-core56b2a36dbbe3237357a9fc9746c86aed-->
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://example/virtualcard.wsdl" xmlns:
      xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:plnk="http://
      docs.oasis-open.org/wsbpel/2.0/plnktype" xmlns:prop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
      targetNamespace="http://example/virtualcard.wsdl">
  <wsdl:types>
    <xsd:schema targetNamespace="http://example/virtualcard.wsdl" elementFormDefault="qualified">
      <xsd:element name="type2EL" type="tns:type2ELCT" />
      <xsd:complexType name="type2ELCT">
        <xsd:sequence>
          <xsd:element name="x_idEL" type="xsd:int" />
          <xsd:element name="x_wdrEL" type="xsd:int" />
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="type3EL" type="tns:type3ELCT" />
      <xsd:complexType name="type3ELCT">
        <xsd:sequence>
          <xsd:element name="x_idEL" type="xsd:int" />
          <xsd:element name="x_respEL" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="type1EL" type="tns:type1ELCT" />
      <xsd:complexType name="type1ELCT">
        <xsd:sequence>
          <xsd:element name="x_idEL" type="xsd:int" />
          <xsd:element name="x_cashEL" type="xsd:int" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="type1">
    <wsdl:part name="payload" element="tns:type1EL" />
  </wsdl:message>
  <wsdl:message name="type3">
    <wsdl:part name="payload" element="tns:type3EL" />
  </wsdl:message>
  <wsdl:message name="type2">
    <wsdl:part name="payload" element="tns:type2EL" />
  </wsdl:message>
  <wsdl:portType name="p_vcardPT">
    <wsdl:operation name="o_getcash">
      <wsdl:input message="tns:type2" />
      <wsdl:output message="tns:type3" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:portType name="p_createcardPT">
    <wsdl:operation name="o_newcard">
      <wsdl:input message="tns:type1" />
      <wsdl:output message="tns:type3" />
    </wsdl:operation>
```

```
  </wsdl:portType>
  <wsdl:binding name="p_vcardBinding" type="tns:p_vcardPT">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="o_getcash">
      <soap:operation soapAction="http://example/virtualcard.wsdl/o_getcash" style="document" />
      <wsdl:input>
        <soap:body use="literal" namespace="http://example/virtualcard.wsdl" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" namespace="http://example/virtualcard.wsdl" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="p_createcardBinding" type="tns:p_createcardPT">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="o_newcard">
      <soap:operation soapAction="http://example/virtualcard.wsdl/o_newcard" style="document" />
      <wsdl:input>
        <soap:body use="literal" namespace="http://example/virtualcard.wsdl" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" namespace="http://example/virtualcard.wsdl" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="p_vcardService">
    <wsdl:port name="p_vcardPort" binding="tns:p_vcardBinding">
      <soap:address location="http://localhost:8080/active-bpel/vcard/ode/processes/p_vcardService" />
    </wsdl:port>
  </wsdl:service>
  <wsdl:service name="p_createcardService">
    <wsdl:port name="p_createcardPort" binding="tns:p_createcardBinding">
      <soap:address location="http://localhost:8080/active-bpel/vcard/ode/processes/p_createcardService" />
    </wsdl:port>
  </wsdl:service>
  <plnk:partnerLinkType name="cltPLT">
    <plnk:role name="p_vcard" portType="tns:p_vcardPT" />
  </plnk:partnerLinkType>
  <plnk:partnerLinkType name="cardPLT">
    <plnk:role name="p_createcard" portType="tns:p_createcardPT" />
  </plnk:partnerLinkType>
  <prop:property name="x_idProp" type="xsd:int" />
  <prop:propertyAlias propertyName="tns:x_idProp" messageType="tns:type3" part="payload">
    <prop:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0">//tns:type3EL/tns:x_idEL</prop:query>
  </prop:propertyAlias>
  <prop:propertyAlias propertyName="tns:x_idProp" messageType="tns:type2" part="payload">
    <prop:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0">//tns:type2EL/tns:x_idEL</prop:query>
  </prop:propertyAlias>
  <prop:propertyAlias propertyName="tns:x_idProp" messageType="tns:type1" part="payload">
    <prop:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0">//tns:type1EL/tns:x_idEL</prop:query>
  </prop:propertyAlias>
</wsdl:definitions>
```

Il file di configurazione del deploy generato da B*lite*C è riportato nel Listing 5.24.

Listing 5.24: File di configurazione necessario a Apache ODE per effettuare il deploy del processo

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by blitec-core19e7033969e2100c44de438fd09ce81c-->
<deploy xmlns="http://www.apache.org/ode/schemas/dd/2007/03" xmlns:bpelns="http://example/virtualcard.bpel" xmlns:
      mwl="http://example/virtualcard.wsdl">
  <process name="bpelns:virtualcardProcess">
    <active>true</active>
    <provide partnerLink="cltPL">
      <service name="mwl:p_vcardService" port="p_vcardPort" />
    </provide>
    <provide partnerLink="cardPL">
      <service name="mwl:p_createcardService" port="p_createcardPort" />
    </provide>
  </process>
```

```
</deploy>
```

Come è possibile vedere, un processo piuttosto semplice come quello in esame è tradotto in un insieme specifiche XML molto prolisse e di non facile comprensione. Questo semplice esempio evidenzia in modo chiaro l'utilità di B*lite*C come strumento di automazione del processo di deploy.

Utilizzatore della carta

L'utilizzatore della carta essendo un'istanza, non è ancora supportato in questa versione di B*lite*C. Ma nelle prossime versioni sarà possibile compilarlo come un normale processo WS-BPEL corredato di un inizializzatore che l'utente potrà utilizzare per inizializzare la comunicazione tra i due partner.

Capitolo 6

Conclusioni

Il contributo di questa tesi è quello di abilitare l'applicazione di tecniche di verifica formale a sistemi di servizi web, per individuare e risolvere eventuali problemi in fase di progettazione e realizzazione del sistema e semplificare la fase di test su di un engine WS-BPEL. Ciò perché alcune proprietà verificabili in modo semplice su di un modello potrebbero altrimenti non essere facilmente verificabili.

Lo scopo di questa tesi è quindi quello di fornire uno strumento che, in modo semplice e intuitivo, faciliti la fase di analisi e permetta all'utente di concentrarsi sulla scrittura della specifica del sistema e sulle varie proprietà che il sistema dovrà garantire. Lo strumento proposto è un modulo di B*lite*C, B*lite*2Prism, che accetta in ingresso specifiche B*lite* e fornisce in uscita specifiche accettate dal model checker probabilistico Prism. La scelta di un model checker probabilistico è stata dettata dalla necessità di esprimere proprietà quantititative, oltre alle usuali proprietà qulitative.

Le specifiche generate permettono l'analisi del comportamento generale e del comportamento in caso di fallimento e sono fornite da B*lite*2Prism in un formato di semplice lettura e comprensione.

Le proprietà verificabili sui modelli sono a discrezione dell'utente, infatti si potranno verificare sia proprietà qualitative (funzionali), sia proprietà quantitative (non funzionali). Dunque l'utente potrà verificare sia proprietà più classiche come *deadlock* e *liveness*, sia proprietà più specifiche come quelle di accessibilità del servizio o di ordine di esecuzione dei costrutti componenti un processo WS-BPEL.

La traduzione dei costrutti è stata realizzata in modo modulare, cosicché il cambiamento del comportamento di un costrutto non compromette la traduzione degli altri.

Inoltre in fase di specifica di un sistema nel linguaggio B*lite*, l'utente potrà personalizzare il comportamento del processo attraverso alcune annotazione (in stile Java). Le

annotazioni permettono all'utilizzarore di cambiare il nome che un costrutto avrà nella traduzione in Prism, variare il comportamento del modello modificando i rate di default delle transizioni, aggiungere condizioni accessorie o in caso di necessità permettere l'esecuzione multipla di un processo. In questo modo si abilita l'analisi del sistema da più punti di vista, infatti con l'aggiunta o la rimozione di una annotazione si può rendere un sistema più o meno astratto.

Nell'implementazione si è cercato di mantenere la modularizzazione in modo che una semplice modifica ad un costrutto non comporti una pesante riscrittura di B*lite*2Prism. Inoltre questo permetterà a future modifiche di essere effettuate senza compromettere il lavoro già svolto.

Le estensioni al lavoro effettuato possono essere molteplici. Quelle che prevediamo di realizzare in una prossima versione di B*lite*C sono le seguenti:

1. aggiunta di dati primitivi alle operazioni di comunicazione;
2. gestione di istanze multiple di uno stesso processo;
3. implementazione di un tool di supporto alla scrittura e verifica delle proprietà.

La prima estensione aggiunge la possibilità di trasmettere dati semplici tra una invoke ed una receive, abilitando oltre all'analisi del comportamento anche quello dei dati. Questa estensione risulta molto interessante, ma necessita di una traduzione efficiente perché l'introduzione di variabili associate ai possibili valori ricevibili o inviabili da un processo potrebbe facilmente portare ad una esplosione di stati, rendendo così l'analisi del modello del tutto irrealizzabile.

La seconda estensione permetterebbe la verifica del comportamento del servizio in caso di istanze multiple, abilitando la ricerca di problemi dovuti alle interazioni di più istanze dei vari servizi componenti il sistema. Questa estensione, come la precedente, deve essere effettuata attraverso una trasfomazione molto efficiente perché l'introduzione di molte istanze per ogni servizio può portare ad una esplosione di stati anche per sistemi di servizi molto semplici.

Infine, l'ultima estensione correderebbe B*lite*C con un linguaggio e relativo traduttore per la stesura di proprietà da testare sul modello generato. Questa aggiunta renderebbe più semplice il flusso delle operazioni da effettuare per predisporsi all'analisi della specifica generata perché l'utente potrebbe utilizzare nomi e costrutti presenti nella specifica B*lite* e sfruttare il tool per tradurre automaticamente le formule in proprietà accettate dal model checker.

Bibliografia

[1] Active Endpoints. ActiveVOS Designer, November 2011. http://www.active-endpoints.com.

[2] Active Endpoints 9.0. ActiveVOS, November 2011. http://www.active-endpoints.com.

[3] Alexandre Alves et al. Web Services Business Process Execution Language. Technical report, OASIS, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[4] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[5] Apache Foundation. Apache ODE 1.3.5, February 2011. http://ode.apache.org.

[6] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert K. Brayton. Verifying continuous time markov chains. In *CAV'96*, pages 269–276, 1996.

[7] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuousinuous-time markov chains. In *Proc. 10th Internationalonal Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 10th46–161, 1999.

[8] Luca Cesari. Progetto e implementazione di uno strumento per il supporto allo sviluppo di applicazioni WS-BPEL. Tesi di Laurea, Università di Firenze, 2009. http://rap.dsi.unifi.it/cows/theses/cesari_thesis_It.pdf.

[9] J. Clark and S. DeRose. XML Path Language (XPath) 1.0. Technical report, W3C, November 1999. http://www.w3.org/TR/xpath.

[10] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL'83*, pages 117–126, 1983.

[11] Eclipse Foundation. Eclipse Bpel Designer, December 2011. http://eclipse.org/bpel/.

[12] Stefan Edelkamp. Promela planning. In *In Proceedings of SPIN-03*, pages 197–212, 2003.

[13] Eviware. soapui 4.0, July 2011. http://www.soapui.org/.

[14] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University Of California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[15] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes Implementationn Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.

[16] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspect of Computing 6*, 4:512–535, 1994.

[17] C. A. R. Hoare. Communicating sequential processes, 2004.

[18] IBM. Websphere, December 2011. http://www-01.ibm.com/software/websphere/.

[19] IETF. FTP: File Transfer Protocol. Technical report, IETF, 1985. http://tools.ietf.org/html/rfc959.

[20] IETF. Hypertext Transfer Protocol – HTTP/1.0. Technical report, IETF, 1999. http://tools.ietf.org/html/rfc1945.

[21] IETF. SMTP: Simple Mail Transfer Protocol. Technical report, IETF, 2001. http://tools.ietf.org/html/rfc2821.

[22] IETF. Extensible Messaging and Presence Protocol (XMPP): Core. Technical report, IETF, 2004. http://www.ietf.org/rfc/rfc3920.txt.

[23] Intalio. Intalio Designer, April 2011. http://www.intalio.com/.

[24] Jax-RPC Project. Jax-rpc. Technical report, Jax-RPC Project, 2004. https://jax-rpc.dev.java.net/.

[25] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[26] A. Lapadula, R. Pugliese, and F. Tiezzi. Using formal methods to develop WS-BPEL applications. *Science of Computer Programming*, 2011. In corso di pubblicazione.

[27] Daniele Nucci. Sperimentazione e confronto di vari engine BPEL. Tesi di Laurea, Univeristà di Firenze, 2007. http://rap.dsi.unifi.it/cows/theses/nucci_thesis_It.pdf.

[28] OASIS. Uddi. Technical report, OASIS, October 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[29] OASIS. Web Services Reliable Messaging TC2 WS-Reliability 1.1. Technical report, OASIS, November 2004. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf.

[30] OASIS. Web Services Security: SOAP Message Security 1.1. Technical report, OASIS, February 2006. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[31] Oracle. Oracle BPEL Process Manager, August 2007. http://www.oracle.com/lang/it/appserver/bpel_home.html.

[32] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[33] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9:223–252, 1977.

[34] University of Oxford - Department of Computer Science. *Prism Manual*. http://www.prismmodelchecker.org/manual/.

[35] UserLand Software. Xml-rpc. Technical report, UserLand Software, 2003. http://www.xmlrpc.com/spec.

[36] W3C. WSDL 1.1. Technical report, W3C, 2001. http://www.w3.org/TR/wsdl/.

[37] W3C. Web Services Choreography Description Language Version 1.0. Technical report, W3C, December 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[38] W3C. XML: eXtensible Markup Language. Technical report, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.