Università degli Studi di Roma "La Sapienza"

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria del Software

Tesina per il corso di Metodi Formali nell' Ingegneria del Software

Verifica formale in Spin di WF-nets e Diagrammi delle Attività UML

Professore: Toni Mancini

Autore: Stefano Menotti

Anno Accademico 2006-2007

Indice

Introduzione

Obiettivo di questo lavoro è quello di sviluppare una metodologia per la verifica formale delle *Workflow Nets*, che costituiscono una classe particolare delle più generali *Reti di Petri* utilizzata per la descrizione dei *processi di workflow*.

Una workflow net che modella un processo di workflow corretto, che non presenta potenziali situazioni di deadlock e la cui terminazione è quindi garantita, deve necessariamente soddisfare alcune proprietà; tali proprietà possono essere verificate formalmente attraverso un *verification tool*; in questo lavoro si è scelto di utilizzare *Spin*.

Le workflow net non costituiscono l'unico formalismo adatto alla modellazione dei processi di workflow: i *Diagrammi delle Attività UML* ad esempio, possono essere utilizzati con lo stesso scopo. Le analogie che ci sono tra i due formalismi ci consentiranno di utilizzare le idee sviluppate per la verifica formale di una workflow net anche nel caso di un diagramma delle attività.

Il presente lavoro è così strutturato: nella sezione 1 vengono introdotte le *Reti di Petri*; nella sezione 2 vengono introdotte le *Workflow Nets* e le loro proprietà; nella sezione 3 viene descritta la metodologia sviluppata per la verifica formale in Spin di una workflow net; nella sezione 4 si applicano le idee sviluppate nella sezione 3 ai *Diagrammi delle Attività UML* e si estende la verifica ad ulteriori proprietà.

1 Reti di Petri

In questa sezione vengono introdotte le *Reti di Petri*, formalismo proposto nel 1962 da Carl Adam Petri come strumento per la descrizione di processi. In particolare nel paragrafo 1.1 vengono introdotti i concetti di base e gli elementi costituenti una rete di Petri; nel paragrafo 1.2 gli stessi concetti, che hanno delle basi teoriche ben definite, vengono presentati attraverso una descrizione formale.

1.1 Concetti di base

Una rete di Petri consiste di *places* e *transizioni*. Un place viene rappresentato graficamente mediante un cerchio, una transizione invece mediante un rettangolo. La Figura 1.1 mostra una semplice rete di Petri composta da tre place (*reclamo, valutazione, fine*) e da tre transizioni (*registra*, *paga*, *invia_lettera*). Questa rete modella il processo di gestione di un reclamo in una compagnia di assicurazioni: un reclamo viene prima registrato, successivamente viene effettuata una valutazione in base alla quale viene deciso se effettuare il pagamento oppure se inviare una lettera con le motivazioni del rifiuto del reclamo.
Place e transizioni sono collegati mediante archi diretti; ci sono due tipi di archi: archi che collegano un place ad una transizione ed archi che collegano una transizione ad un place. Archi da place a place e da transizione a transizione non sono permessi.
Un place *p* è un *input place* per una transizione *t* se e solo se esiste un arco diretto che collega *p* a *t*. Allo stesso modo, è possibile determinare gli *output place* di una transizione: un place *p* è un *output place* per una transizione *t* se e solo se esiste un arco diretto da *t* a *p*. Nella rete in Figura 1.1 ogni transizione ha esattamente un input place ed un output place.

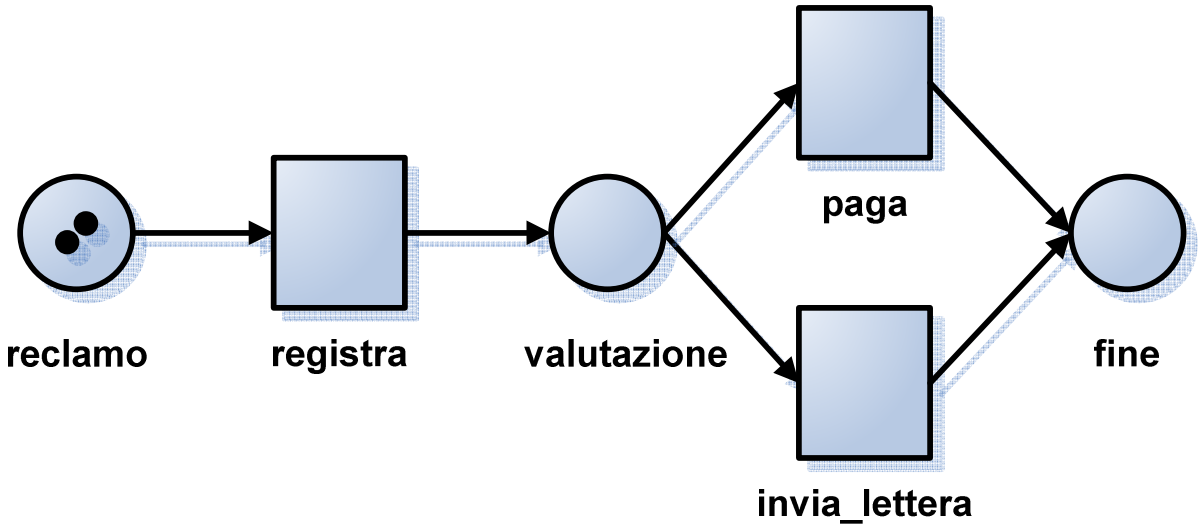


Figura 1.1 Rete di Petri per la gestione di un reclamo

Un place può contenere dei *token*, rappresentati graficamente da un cerchietto nero. In Figura 1.1 il place *reclamo* contiene due token. La struttura di una rete di Petri è statica, non cambia cioè durante l'esecuzione del processo che modella, tuttavia la distribuzione dei token nei place può cambiare. La transizione *registra* può "consumare" un token dall'input place *reclamo* ed inviarlo all'output place *valutazione*: si dice in questo caso che la transizione *registra* "scatta".

Lo stato di una rete di Petri è indicato dalla distribuzione dei token nei place della rete. Possiamo ad esempio descrivere lo stato illustrato in Figura 1.1 attraverso il vettore (2, 0, 0), che indica la presenza di due token nel place *reclamo* e di nessun token nei place *valutazione* e *fine*.

Una transizione può scattare solamente se essa è abilitata. Ciò avviene nel momento in cui è presente almeno un token in ciascuno dei suoi input place. In figura 1.1, *registra* è abilitata, le altre due transizioni invece non lo sono.

Nel momento in cui una transizione scatta, un token viene rimosso da ciascuno degli input place e viene aggiunto un token in ciascuno degli output place. In altre parole, nel momento in cui scatta, una transizione "consuma" un token da ciascuno degli input place e "produce" un token in ciascuno degli output place. La Figura 1.2 mostra cosa accade quando *registra* scatta: un token viene trasferito dal place *reclamo* al place *valutazione*; possiamo anche descrivere la nuova situazione attraverso il vettore (1, 1, 0).

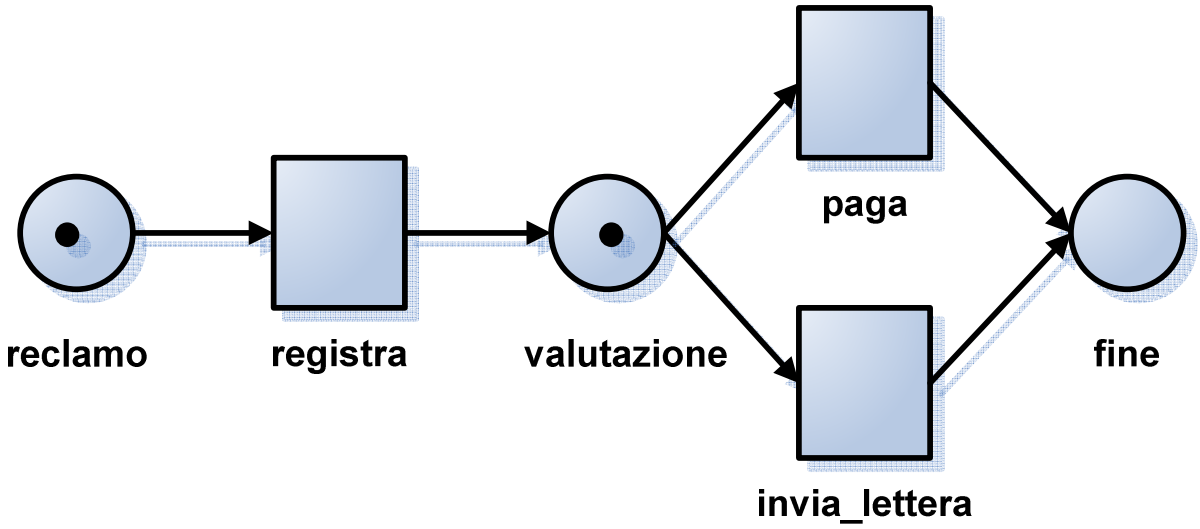


Figura 1.2 Stato successivo allo "scatto" di *registra*

Una volta che la transizione *registra* è scattata, le transizioni abilitate diventano tre: *registra* può scattare ancora perché è presente ancora un token in *reclamo*, *paga* e *invia_lettera* possono scattare perché è presente un token in *valutazione*. In questa situazione, non è possibile stabilire quale transizione scatterà per prima. Se assumiamo che sia *paga* a scattare, il nuovo stato sarà quello illustrato in Figura 1.3.

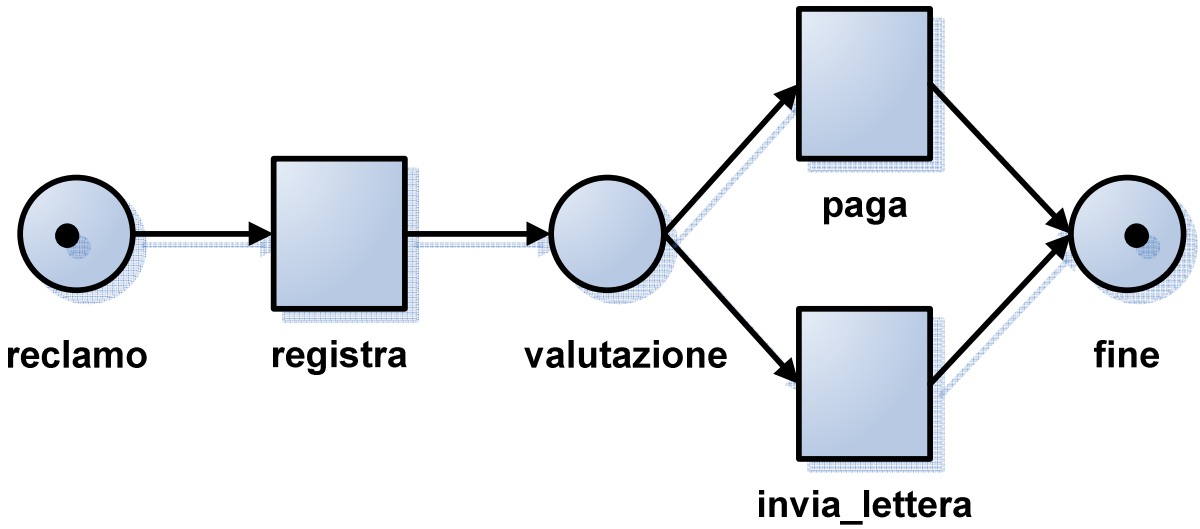


Figura 1.3 Stato successivo allo "scatto" di *paga*

Notiamo che *invia_lettera*, che era abilitata prima che *paga* scattasse, non lo è più; *registra* invece lo è ancora. Alla fine, dopo un totale di quattro scatti, verrà raggiunto lo stato (0, 0, 2) con due token nel place *fine*; in questo stato nessuna transizione potrà scattare.

Le transizioni sono le componenti attive di una rete di Petri; quando una transizione scatta, il processo modellato dalla rete passa da uno stato all'altro. Una transizione quindi spesso rappresenta un evento, una operazione oppure una trasformazione. I place invece rappresentano le componenti passive, nel senso che essi non possono cambiare lo stato della rete. Un place di solito rappresenta una fase od una condizione. I token spesso rappresentano oggetti, ma anche informazioni. Nella rete considerata precedentemente, ogni token rappresenta ad esempio un singolo reclamo.

1.2 Definizione Formale

Le reti di Petri hanno delle basi teoriche ben definite; il loro scopo infatti è quello di modellare sistemi e processi in modo preciso e non ambiguo. Vediamo quindi una descrizione formale dei concetti introdotti nel paragrafo 1.1.

Definizione 1 (Rete di Petri). Una rete di Petri è un tripla (*P, T, F*) :

- *P* è un insieme finito di place;
- *T* è un insieme finito di transizioni ($P \cap T = \emptyset$);
- $F \subseteq (P \times T) \cup (T \times P)$ è un insieme di archi (relazione di flusso).

Utilizziamo la notazione $\bullet t$ per riferirci all'insieme degli input place di una transizione *t*. Le notazioni $t\bullet$, $\bullet p$ e $p\bullet$ hanno significato simile, quindi $p\bullet$ ad esempio, è l'insieme delle transizioni che condividono *p* come input place.

In ogni istante un place contiene zero o più token; Uno stato *M* (marcatura), rappresenta la distribuzione di token nei place della rete. Con la notazione $1p_1+2p_2+1p_3+0p_4$ ad esempio, si indica lo stato in cui è presente un token nel place p_1, due token in p_2, un token in p_3 e nessun token in p_4. Per confrontare stati definiamo un ordine parziale; per ogni coppia di stati M_1 e M_2, $M_1 \leq M_2$ se e solo se per ogni $p \in P$ $M_1(p) \leq M_2(p)$, dove $M(p)$ denota il numero di token nel place *p* nello stato *M*.

Una transizione *t* è abilitata se e solo se ogni input place *p* di *t* contiene almeno un token.

Una transizione abilitata può scattare. Se la transizione *t* scatta, *t* consuma un token da ogni input place *p* di *t* e produce un token in ogni output place *p* di *t*.

Consideriamo una Rete di Petri (*P, T, F*) ed uno stato M_1, si usano le seguenti notazioni:

- $M_1 \xrightarrow{t} M_2$: la transizione *t* è abilitata nello stato M_1 e quando scatta la rete si porta nello stato M_2.
- $M_1 \rightarrow M_2$: esiste una transizione *t* tale che $M_1 \xrightarrow{t} M_2$.

- $M_1 \xrightarrow{\sigma} M_n$: la sequenza di scatti $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ porta dallo stato M_1 allo stato M_n attraverso un insieme (anche vuoto) di stati intermedi $M_2, \ldots, M_{n-1}$.

Uno stato *Mn* è chiamato raggiungibile da M_1 ($M_1 \xrightarrow{*} M_n$) se e solo se esiste una sequenza di scatti σ tale che $M_1 \xrightarrow{\sigma} M_n$.

(*PN*, *M*) indica una Rete di Petri con stato iniziale *M*. Uno stato *M'* è uno stato raggiungibile di (*PN*, *M*) se e solo se $M \xrightarrow{*} M'$.

Definizione 2 (Live). Una Rete di Petri (*PN*, *M*) è *live* se e solo se per ogni stato raggiungibile *M'* e per ogni transizione *t* esiste uno stato *M''* raggiungibile da *M'* che abilita *t*.

Definizione 3 (Bounded, Safe). Una Rete di Petri (*PN*, *M*) è *bounded* se e solo se per ogni place esiste un numero naturale *n* tale che per ogni stato raggiungibile il numero di token in *p* è al massimo *n*. La rete è *safe* se e solo se per ogni place il numero massimo di token è uno.

Definizione 4 (Cammino). Un cammino *C* dal nodo n_1 al nodo n_k è una sequenza $(n_1, n_2, \ldots, n_k)$ tale che $(n_i, n_{i+1}) \in F$ per $1 \leq i \leq k-1$.

2 Workflow Nets

In questa sezione vengono introdotte le *Workflow Nets*, che costituiscono una classe di reti di Petri utilizzata per la descrizione dei processi di workflow. Nel paragrafo 2.1 vengono introdotti i concetti di base; nel paragrafo 2.2 vengono descritte le proprietà che rendono una workflow net corretta; nel paragrafo 2.3 vengono presentati formalmente i concetti introdotti in 2.1 e 2.2.

2.1 Concetti di base

Le reti di Petri costituiscono un formalismo particolarmente adatto per la descrizione dei processi di workflow. Una rete di Petri che rappresenta un processo di workflow è nota come *Workflow net* (*WF-net*).

Lo scopo di un processo di workflow è quello di specificare i task (unità di lavoro atomiche) che devono essere eseguiti e l'ordine in cui tali task devono essere eseguiti; un processo di workflow ha senso solo se ha un inizio ed una fine ben definiti.

Appare subito evidente come sia possibile tradurre i concetti relativi ad un processo di workflow in una rete di Petri: le transizioni, che sono le componenti attive in una rete di Petri, rappresentano i task di un processo di workflow; i place, componenti passive di una rete di Petri, rappresentano invece le condizioni per l'attivazione di un task in un processo di workflow.

La rete dovrà avere inoltre un unico punto d'inizio (un place senza archi entranti) ed un unico punto finale (un place senza archi uscenti).

Esaminiamo un processo per la gestione dei reclami. Un reclamo è dapprima registrato, successivamente vengono contattati sia il cliente che ha effettuato il reclamo che il dipartimento interessato dal reclamo per ottenere maggiori informazioni; questi due task possono essere eseguiti in parallelo. Successivamente tutte le informazioni sono raccolte e valutate al fine di prendere una decisione. In base alla decisione viene inviato un risarcimento al cliente oppure una lettera con le motivazioni del mancato risarcimento. Infine il reclamo viene depositato.

La Figura 2.1 mostra come il processo appena descritto possa essere modellato attraverso una workflow net.

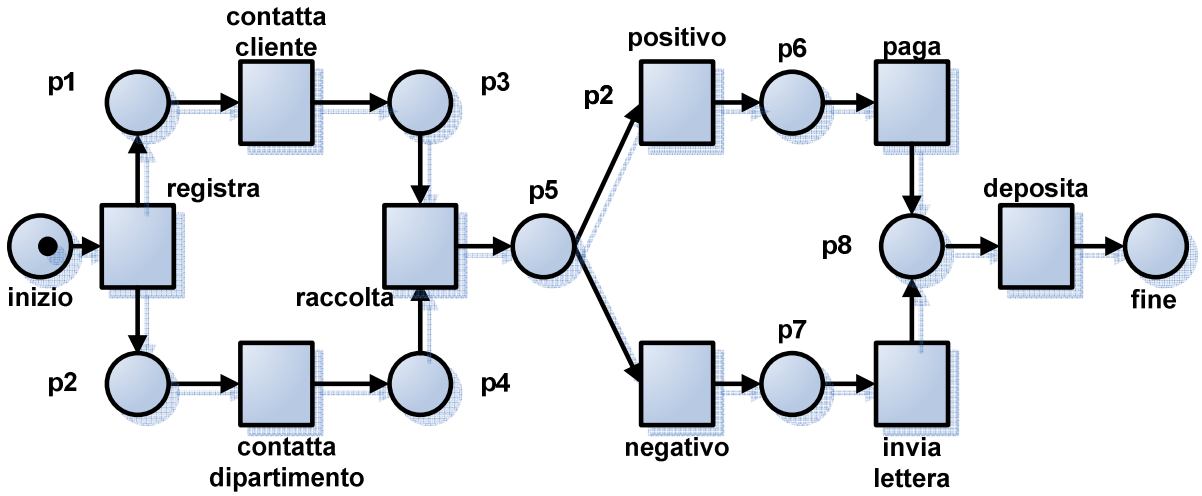


Figura 2.1 Workflow Net per il processo "gestione reclamo"

I task *registra*, *contatta cliente*, *contatta dipartimento*, *paga*, *invia lettera* e *deposita* sono modellati attraverso delle transizioni. La valutazione di un reclamo è modellata utilizzando due transizioni: *positivo* e *negativo*; la transizione *positivo* corrisponde ad una decisione positiva, la transizione *negativo* corrisponde ad una decisione negativa. I place *inizio* e *fine* corrispondono all'inizio ed alla fine del processo. Gli altri place corrispondono alle condizioni per l'esecuzione dei task e coprono due ruoli importanti: da un lato assicurano che i task procedano nell'ordine corretto e dall'altro permettono di stabilire lo stato in cui si trova il processo.

Ogni processo di workflow modellato attraverso una wf-net dovrebbe rispettare due requisiti. (1) deve essere sempre possibile raggiungere, eseguendo un certo numero di task, uno stato in cui c'è un token in *fine*; e (2) quando c'è un token in *fine*, tutti gli altri place devono essere vuoti. Questi due requisiti assicurano che ogni esecuzione del processo prima o poi termina e nel momento in cui termina (un token arriva in *fine*) non ci sono task ancora da eseguire.

In un processo di workflow i task possono essere opzionali, cioè possono esistere task che devono essere eseguiti solamente in certi casi. Anche l'ordine in cui i task sono eseguiti può variare da caso a caso. Ci sono quattro costrutti di base per definire il cosiddetto *routing* attraverso un processo di workflow, vediamo come ciascuno di essi viene modellato in una wf-net.

(a) Routing sequenziale. Quando i task sono eseguiti uno dopo l'altro si parla di *routing sequenziale*. Se due task devono essere eseguiti uno dopo l'altro, esiste di solito una chiara interdipendenza fra loro. Per esempio il risultato del primo task è richiesto per l'esecuzione del

secondo task. In una wf-net, questa forma di routing è modellata collegando due transizioni mediante un place. La Figura 2.2. mostra un esempio di routing sequenziale.

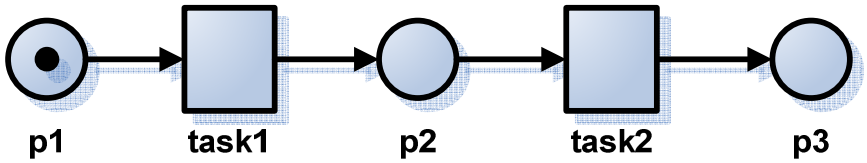


Figura 2.2 Routing sequenziale

Il task corrispondente alla transizione $task_2$ viene eseguito una volta che il task corrispondente alla transizione $task_1$ viene completato.

(b) Routing parallelo. Quando due più task possono essere eseguiti in parallelo, si parla di *routing parallelo*. Consideriamo solamente due task, $task_1$ e $task_2$, ci sono allora tre possibilità: entrambi i task possono essere eseguiti simultaneamente; $task_1$ può essere eseguito per primo, seguito da $task_2$, oppure è $task_2$ ad essere eseguito per primo seguito da $task_1$. La Figura 2.3 mostra come sia possibile modellare questa situazione attraverso una wf-net.

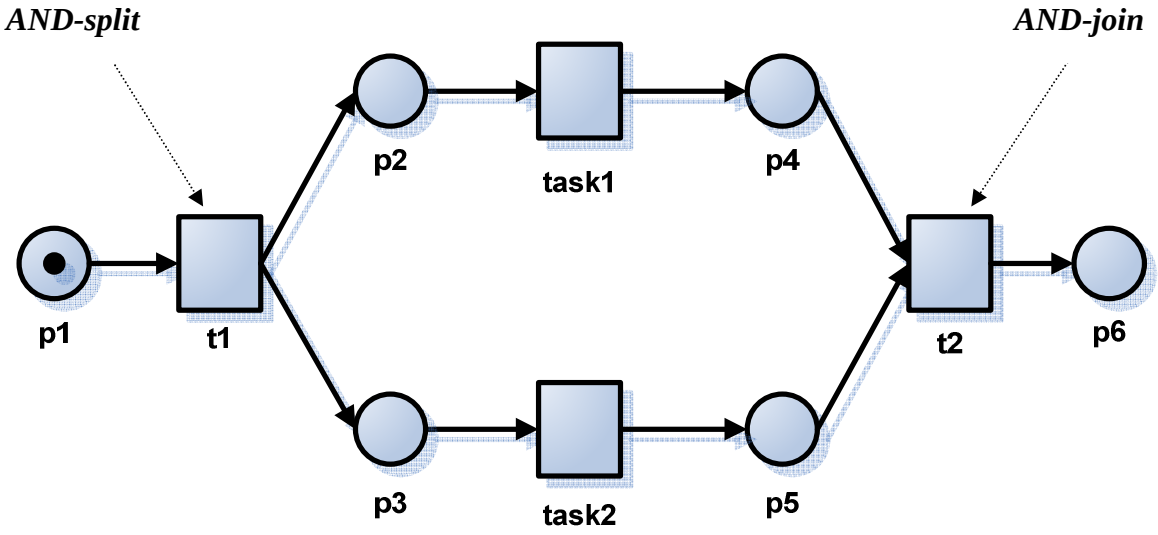


Figura 2.3 Routing parallelo

Per abilitare l'esecuzione parallela di $task_1$ e $task_2$, introduciamo il cosiddetto *AND-split*. Quest'ultimo è un task aggiuntivo che permette di gestire più task allo stesso tempo. In Figura 2.3, la transizione t_1 rappresenta un AND-split; t_1 scatta quando è presente un token in p_1 e produce un token sia in p_2 che in p_3, t_1 quindi abilita due task, $task_1$ e $task_2$ che possono pertanto essere eseguiti in parallelo; quando entrambi i task sono stati eseguiti, ci sarà un token in p_4 ed un token in p_5, solamente a questo punto t_2 può scattare e l'esecuzione del processo continuare; t_2 è l'equivalente di un *AND-join*: un task aggiuntivo per sincronizzare due o più rami paralleli.
In figura 2.3 sono stati inseriti due task, t_1 e t_2 per modellare rispettivamente un AND-split ed un AND-join; t_1 e t_2 sono task che non corrispondono ad una effettiva unità di lavoro; grazie ad essi $task_1$ e $task_2$ possono essere eseguiti in parallelo. E' comunque possibile far corrispondere ad unità di lavoro effettive task come t_1 e t_2; in Figura 2.1 per esempio il task *registra* corrisponde ad un AND-split, il task *raccolta* ad un AND-join.

(c) Routing selettivo. Esistono casi in cui deve essere effettuata una scelta tra l'esecuzione di due o più task; in questo caso si parla di *routing selettivo*. La Figura 2.4 mostra un esempio modellato con una rete di Petri.

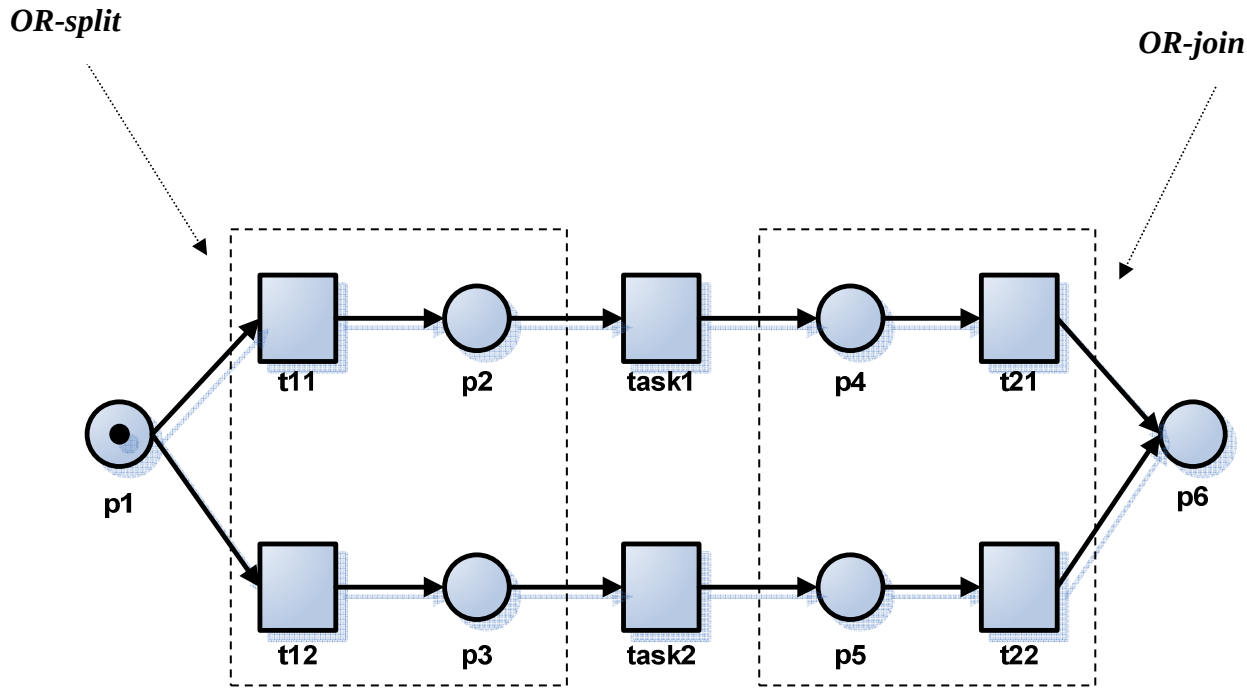


Figura 2.3 Routing selettivo (1)

Quando è presente un token in p_1, una sola tra t_{11} o t_{12} scatterà; se a scattare è t_{11}, $task_1$ verrà abilitata, se a scattare invece è t_{12} viene abilitata $task_2$; esiste quindi una scelta tra t_{11} e t_{12}. Chiamiamo la rete composta dalle transizioni t_{11} e t_{12} e dai place p_2 e p_3 un *OR-split*. Quando uno dei due task viene eseguito, l'*OR-join* assicura che un token comparirà in p_6. L'OR-join viene modellato usando una rete costituita da due place (p_4 e p_5) e due transizioni (t_{21} e t_{22}).
In Figura 2.3 l'OR-split e l'OR-join sono modellati esplicitamente; tuttavia è anche possibile modellarli implicitamente, come mostrato in Figura 2.4

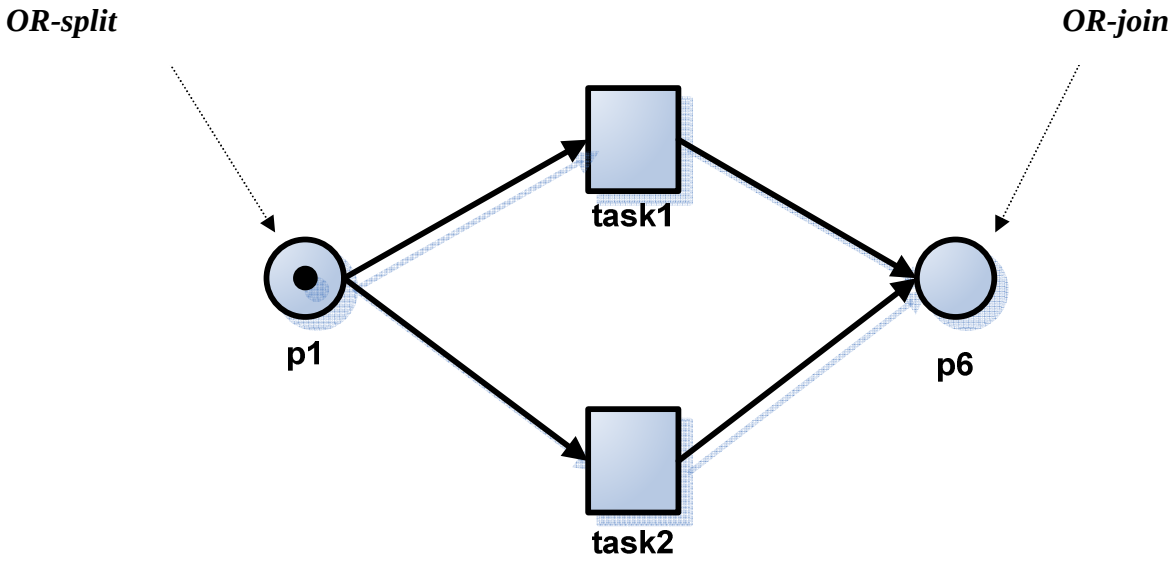


Figura 2.4 Routing selettivo (2)

Poiché nel modellare un processo di workflow costrutti come l'AND-split, l'AND-join, l'OR-split e l' OR-join vengono usati spesso, sono stati introdotti dei simboli speciali per riferirsi ad essi. I simboli sono mostrati in Figura 2.5.

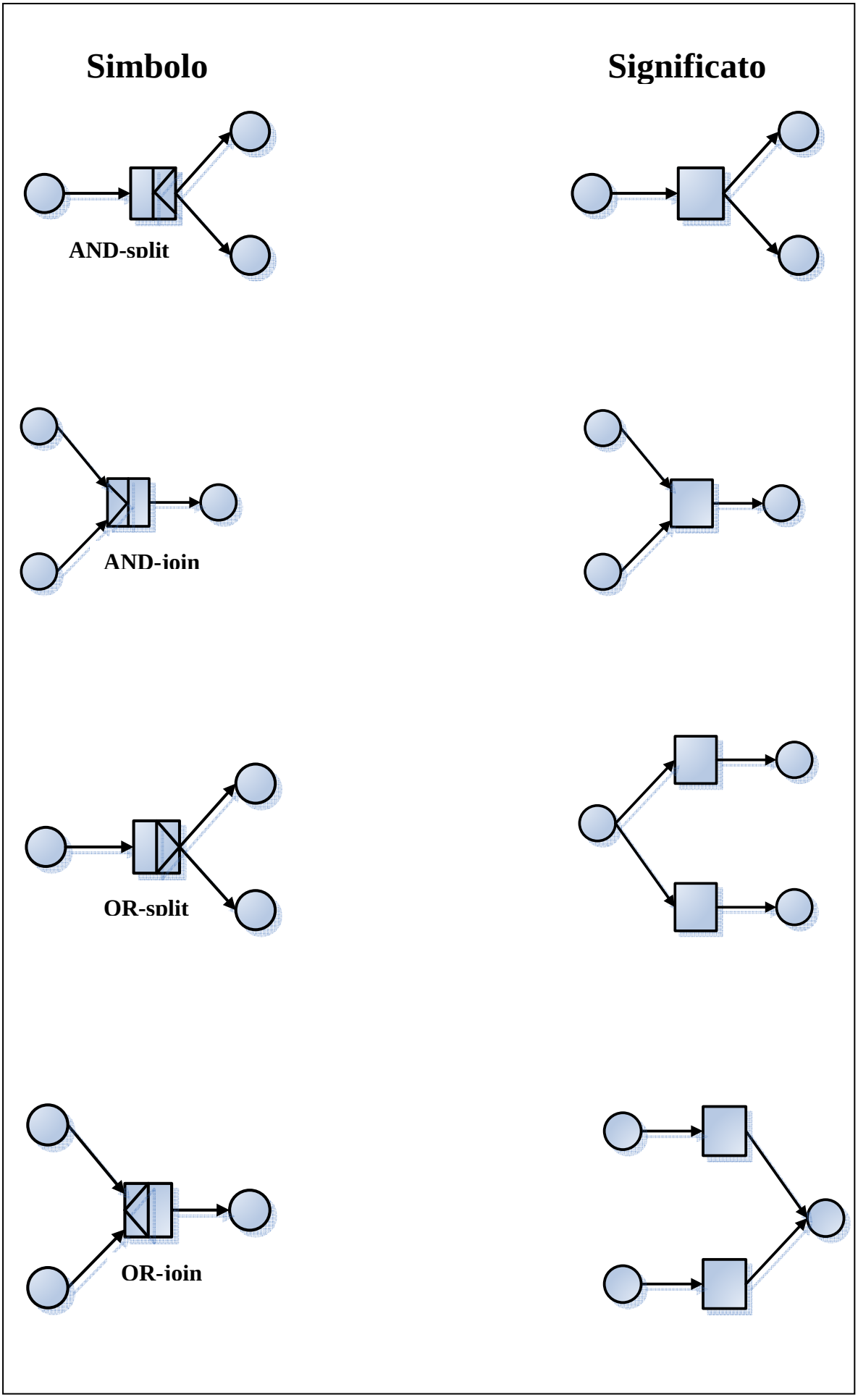


Figura 2.5 Simboli per i costrutti base

(d) Routing iterativo. L'ultima forma di routing consiste nella ripetizione di uno o più task; in alcune situazioni infatti, è necessario applicare il cosiddetto *routing iterativo*. La Figura 2.6 mostra come modellare il routing iterativo in una wf-net.

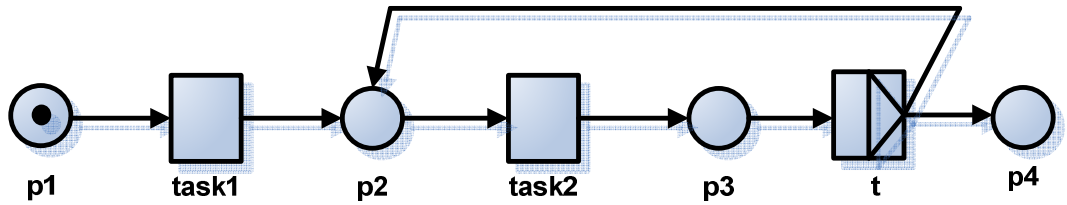


Figura 2.6 Routing iterativo (1)

Quando $task_2$ è stato eseguito, il token in p_3 abilita l'OR-split che determina se sia necessario eseguire di nuovo $task_2$ (un token viene prodotto in p_2) oppure no (un token viene prodotto in p_4).
In figura 2.6 si assume che $task_2$ venga eseguito almeno una volta. Se non è questo ciò che si vuole modellare, si può applicare il costrutto rappresentato in Figura 2.7

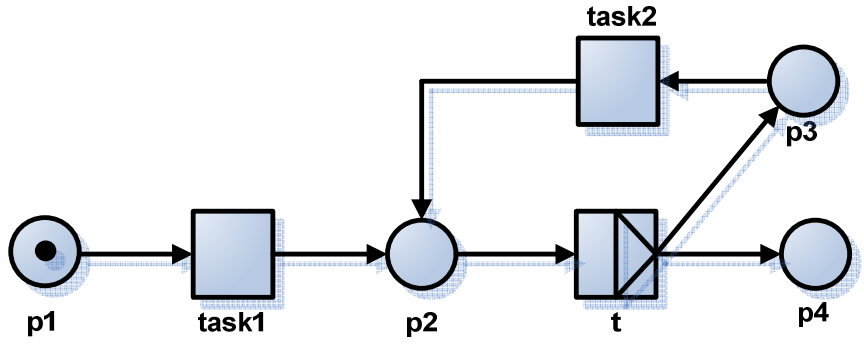


Figura 2.7 Routing iterativo (2)

2.2 Proprietà

Nella sezione 2.1 abbiamo detto che un processo di workflow ha senso solo se ha un unico punto iniziale ed un unico punto finale, una rete di Petri che descrive un processo di workflow, cioè una WF-net è caratterizzata dall'avere dunque un unico place iniziale *inizio*, un unico place finale *fine*. In una WF-net inoltre, per definizione ogni nodo della rete (transizione o place) si trova in un cammino diretto da *inizio* a *fine*. Grazie a questo requisito infatti, ogni task può essere raggiunto dal place *inizio*, ed il place *fine* è raggiungibile da ogni task.
Una WF-net quindi deve soddisfare i requisiti sintattici appena elencati; nonostante questo però, è comunque possibile che una WF-net abbia delle anomalie che potrebbero portare alla non terminazione del processo.

Oltre ai requisiti sintattici quindi, è opportuno introdurre ulteriori requisiti che una WF-net deve rispettare:

1. Per ogni token inserito nel place *inizio*, uno ed uno solo token prima o poi appare in *fine*.
2. Quando il token appare in *fine*, tutti gli altri place sono vuoti.
3. Per ogni transizione *t* (task) è possibile partire dallo stato iniziale con un solo token in *inizio* ed arrivare ad uno stato in cui *t* è abilitata.

Una WF-net che soddisfa questi requisiti è detta *sound*.

Il primo requisito indica che ogni esecuzione di un processo, prima o poi termina. Il secondo requisito stabilisce che una volta che l'esecuzione del processo termina, nessun riferimento ad essa rimane nel processo. I primi due requisiti combinati indicano che, partendo dallo stato iniziale con un unico token nel place *inizio*, esiste un unico stato finale con precisamente un token nel place *fine*. L'ultimo requisito esclude i cosiddetti *dead tasks*, task cioè che possono bloccare l'esecuzione del processo perché le condizioni per la loro esecuzione non sono mai soddisfatte.

2.3 Definizione Formale

Vediamo ora come i concetti introdotti nei paragrafi 2.1 e 2.2 possano essere formalizzati.
Formalizziamo prima i requisiti che una rete di Petri deve rispettare per poter essere definita una WF-net, successivamente formalizziamo i requisiti che una WF-net deve rispettare per poter essere definita *sound*.

Definizione 5 (WF-net). Una rete di Petri *PN=(P, T, F)* è una WF-net se e solo se:

- Esiste un unico place iniziale *inizio* $\in P$ tale che $\bullet inizio = \emptyset$;
- Esiste un unico place finale *fine* $\in P$ tale che $fine \bullet = \emptyset$;
- Ogni nodo $x \in P \cup T$ si trova in un cammino da *inizio* a *fine*.

Sia *i* lo stato iniziale con un solo token nel place *inizio* e sia *f* lo stato finale con un solo token nel place *fine*:

Definizione 6 (Sound). Una WF-net *PN=(P, T, F)* è *sound* se e solo se:

1. Per ogni stato *M* raggiungibile dallo stato *i*, esiste una sequenza di scatti che porta dallo stato *M* allo stato *f*. Formalmente:

   $$\forall M \quad (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} f)$$

2. Lo stato *f* è l'unico stato con un token nel place *fine* raggiungibile dallo stato *i*. Formalmente:

   $$\forall M \quad (i \xrightarrow{*} M \quad \wedge \quad M \geq f) \Rightarrow (M = f)$$

3. Non ci sono *dead transitions* in *(PN, i)*. Formalmente:

   $$\forall t \in T \quad \exists M, M' \ (i \xrightarrow{*} M \xrightarrow{t} M')$$

Il requisito 1 indica che partendo dallo stato con un token nel place *inizio* (stato *i*), è sempre possibile raggiungere lo stato con un token nel place *fine* (stato *f*).
Il requisito 2 indica che nel momento in cui un token arriva nel place *fine*, tutti gli altri place devono essere vuoti; infatti la formula ci dice che partendo dallo stato *i* con un solo token nel place *inizio*, se si raggiunge uno stato $M \geq f$, cioè in cui ogni place contiene un numero di token maggiore od uguale al numero di token che il place contiene nello stato *f*, allora tale stato è necessariamente *f*, quello cioè con un unico token nel place *fine* e nessun token negli altri place della rete.
L'ultimo requisito indica che non esistono transizioni che bloccano l'esecuzione del processo in quanto le condizioni per la loro esecuzione non sono mai verificate.

3 WF-Nets: verifica formale in Spin

Spin è uno strumento che supporta la verifica automatica utilizzando la logica LTL : i modelli in Spin vengono descritti mediante un linguaggio nominato *Promela* (PROcess Meta LAnguage) con caratteristiche che consentono di descrivere in particolar modo le interazioni tra processi concorrenti, sincrone o asincrone. Grazie a Spin è poi possibile specificare un insieme di proprietà formulate in logica LTL e verificarle con un'opportuna interfaccia di configurazione dei parametri di verifica (*XSpin*).
Scopo di questa sezione è descrivere come utilizzare Spin per poter eseguire simulazioni su una WF-net al fine di stabilire se la proprietà *sound* introdotta nei paragrafi 2.2 e 2.3 sia verificata e nel caso in cui non lo fosse individuare la presenza di "dead transitions" In primo luogo quindi, sarà necessario definire formalmente una WF-net nel linguaggio Promela; successivamente la proprietà *sound* verrà tradotta in opportune formule LTL che verranno infine valutate sul modello Promela.
Il paragrafo 3.1 descrive come una WF-net possa essere specificata in Promela; nel paragrafo 3.2 viene descritta la traduzione della proprietà *sound* in formule LTL; nel paragrafo 3.3 viene mostrato un esempio completo di verifica di una WF-net in Spin.

3.1 Traduzione di una WF-Net in Promela

Un place di una WF-net può contenere uno o più token e la distribuzione dei token attraverso i place cambia durante l'esecuzione del processo; per specificare questo comportamento in Promela, è stato scelto di tradurre un place di una WF-net in una variabile intera: il valore che assume la variabile associata al place rappresenta il numero di token presenti in quel particolare place. In Promela possiamo definire delle variabili, che chiamiamo appunto *Place*, di tipo intero, attraverso la seguente sintassi, che ricorda la definizione di macro nel linguaggio C :

```
# define Place int
```

Una transizione può "scattare" solamente se essa è abilitata, cioè se tutti gli input place della transizione contengono almeno un token. Quando scatta, la transizione "consuma" un token da ciascuno degli input place e "produce" un token in ciascuno degli output place. In Promela questo comportamento può essere specificato nel corpo di un processo; è stato scelto quindi di tradurre una

transizione di una WF-net in un processo Promela nel cui corpo viene definito il comportamento associato alla transizione.
In Promela, un processo viene definito attraverso la seguente sintassi:

```
proctype processo()
        {       /* corpo del processo*/     }
```

In generale il corpo di un processo Promela è composto essenzialmente da una sequenza di *statements*: ogni statement può essere eseguibile o bloccato. Se nello statement è espressa una condizione non verificata, esso è bloccato. Nel nostro caso possiamo utilizzare uno statement per definire la condizione che abilita una transizione: finché tale condizione non è verificata, la transizione è bloccata e non può scattare; nel momento in cui la condizione risulta verificata, vengono eseguite le istruzioni successive che nel nostro caso tradurranno il processo di consumazione e di produzione dei token, il tutto viene inserito in un ciclo indefinito *do*; lo statement sarà inoltre dichiarato come *atomic*, una parola chiave che in Promela permette di specificare sequenze di codice che vanno eseguite in modo appunto atomico al fine di evitare situazioni che potrebbero portare a *deadlock*; definiamo infine, per ogni transizione, una variabile booleana *transizione_eseguita* inizializzata a *false*:

```
bool transizione_eseguita=false;
```

che sarà posta a *true* nel momento in cui la transizione scatta; questa variabile risulterà utile in fase di verifica in quanto indicherà se la transizione è scattata almeno una volta e ci permetterà dunque di stabilire se si tratta di una “dead transition”, cioè una transizione per la quale non esiste una simulazione in cui essa “scatta” in quanto le condizioni per la sua abilitazione non sono mai verificate. Nella definizione di una WF-net in Promela quindi, il corpo di un processo (transizione) sarà costituito da un ciclo infinito *do* contenente uno statement per la verifica dell’abilitazione della transizione, seguito dalle istruzioni relative alla produzione e consumazione di token e dall’istruzione che pone a *true* la variabile *transizione_eseguita*; un processo corrispondente ad una transizione avrà cioè la seguente struttura:

```
Proctype transizione()
 {
do :: atomic {
```

```
TEST_ABILITAZIONE → CONSUMA_TOKEN; PRODUCI_TOKEN;transizione_eseguita=true;
                }
    od; }
```

In base al tipo di transizione, il codice relativo a TEST_ABILITAZIONE, CONSUMA_TOKEN e PRODUCI_TOKEN sarà diverso e dipenderà dal numero di input place e di output place della transizione e dal tipo di transizione (transizione semplice, AND-split, OR-split, AND-join, OR-join).

3.1.1 Transizione semplice

Consideriamo il caso più semplice, cioè quello di una transizione T con un unico input place p_i ed un unico output place p_o , come illustrato in Figura 3.1:

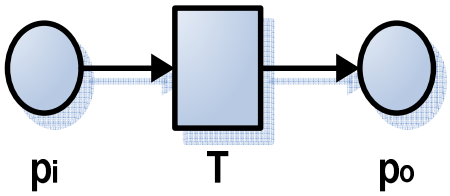


Figura 3.1 Transizione semplice

La transizione è abilitata quando il place p_i contiene almeno un token, in tal caso T può scattare e quindi consumare un token da p_i e produrne uno in p_o; il codice corrispondente a TEST_ABILITAZIONE dovrà quindi verificare che la variabile p_i sia maggiore di zero; il codice relativo a CONSUMA_TOKEN dovrà diminuire la varaibile p_i di uno, ed il codice relativo a PRODUCI_TOKEN dovrà incrementare di uno la variabile p_o.
Il codice Promela del processo corrispondente ad una transizione semplice sarà quindi il seguente:

```
proctype T () {
        do :: atomic  {
           (pi > 0) → pi= pi -1; po= po +1; T_eseguita = true;
                        }
        od      }
```

3.1.2 AND-split

Come mostrato in Figura 3.2, una transizione che realizza un AND-split ha un unico input place p_i ed n output place p_{o1}, p_{o2}, …, p_{on}.

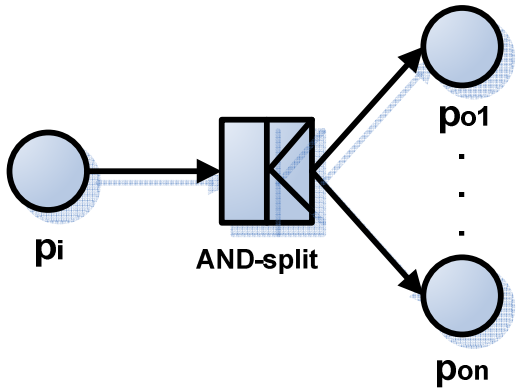


Figura 3.2 And-split

L'AND-split è quindi abilitato quando p_i contiene almeno un token; quando scatta consuma un token da p_i e ne produce uno in ciascuno degli n output place.
In questo caso quindi, il codice corrispondente a TEST_ABILITAZIONE dovrà verificare che la variabile p_i sia maggiore di zero, il codice corrispondente a CONSUMA_TOKEN dovrà decrementare p_i di uno e il codice corrispondente a PRODUCI_TOKEN dovrà incrementare di uno tutte le variabili corrispondenti agli output place p_{o1}, …, p_{on}.
Il codice Promela corrispondente ad un AND-split è pertanto il seguente:

```
proctype AND-split () {
        do :: atomic  {
           (pi > 0) → pi= pi -1; po1= po1+1; … ; pon =pon+1; AND-split_eseguita = true;
                        }
        od
              }
```

3.1.3 AND-join

Una transizione che realizza un AND-join ha *n* input place p_{i1}, p_{i2}, …, p_{in} ed un unico output place p_o, come mostrato in Figura 3.3:

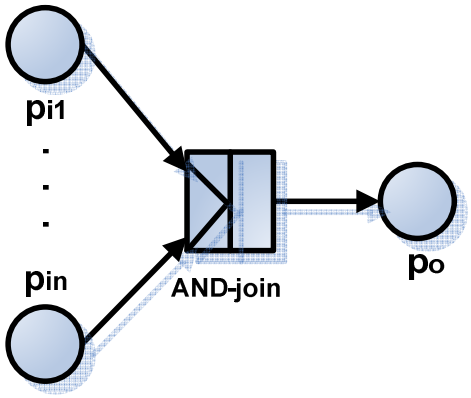


Figura 3.3 And-join

L'AND-join è quindi abilitato quando tutti gli *n* input place contengono almeno un token e quando scatta consuma un token da ciascuno degli input place e ne produce uno nell'output place p_o .
In questo caso dunque, il codice corrispondente a TEST_ABILITAZIONE dovrà verificare che tutte le variabili p_{i1}, p_{i2}, …, p_{in} siano maggiori di zero; il codice corrispondente a CONSUMA_TOKEN dovrà decrementare di uno tutte le variabili p_{i1}, p_{i2}, …, p_{in}; il codice corrispondente a PRODUCI_TOKEN dovrà invece incrementare di uno la variabile p_o.
Il codice Promela corrispondente ad un AND-join è pertanto il seguente:

```
proctype AND-join () {
do :: atomic  {
        ( (pi1 > 0) &&  … && (pin > 0) ) →
            pi1= pi1 -1; …; pin= pin -1; p0= p0 +1; AND-join_eseguita= true;
                }
    od
    }
```

3.1.4 OR-split

Consideriamo ora il caso di una transizione che realizza un OR-split come mostrato in Figura 3.4.

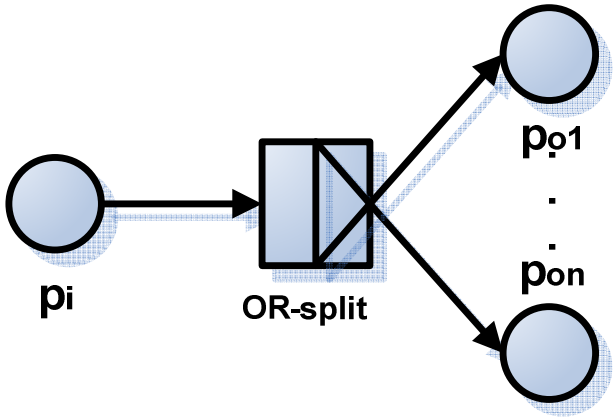


Figura 3.4 OR-split

L'OR-split è una transizione con unico input place p_i, ed n output place p_{o1}, p_{o2}, …, p_{on} ed è quindi abilitato quando l'input place contiene almeno un token; quando scatta produce un token in uno solo degli n output place, quello per il quale è verificata una certa condizione. Ai fini della nostra verifica, non interessa sapere quali siano le condizioni da verificare per la scelta di un place rispetto ad un altro, ciò che interessa è che un token verrà prodotto in uno ed uno solo degli output place; per specificare in Promela il comportamento di un OR-split è stato scelto di utilizzare il costrutto *if* che è così strutturato:

```
if
        :: choice1 → stat1.1;stat1.2; …;stat1.n;
        :: choice2 → stat2.1;stat2.2; …;stat2.n;
        :: ….
        :: choicen→ statn.1;statn.2; …;statn.n;
fi;
```

ogni "choice" è a sua volta uno statement e viene chiamato *guardia*. L'if statement è eseguibile solo se almeno una delle guardie è eseguibile: se questa esiste, il processo proseguirà sugli statements successivi alla freccia. In presenza di più guardie eseguibili, Spin sceglierà *non deterministicamente*

quella su cui proseguire. È stato scelto quindi di far corrispondere ad ogni output place una guardia *true* (cioè sempre eseguibile) seguita dal codice corrispondente a CONSUMA_TOKEN e PRODUCI_TOKEN relativo allo specifico output place. In tal modo l'intero costrutto *if* è composto da tutte guardie eseguibili, pertanto nella simulazione verrà scelto *non deterministicamente* un output place tra gli *n* disponibili.
In definitiva quindi, il codice Promela corrispondente ad un OR-split è il seguente:

```
proctype OR-split () {
        do :: atomic  {
           (p_i > 0) → if
                              :: true→ p_i= p_i -1; p_o1= p_01+1; OR-split_eseguita = true;
                              :: true → p_i= p_i -1; p_o2= p_02+1; OR-split_eseguita = true;
                                                    .
                                                    .
                                                    .
                              :: true → p_i= p_i -1; p_on= p_0n+1; OR-split_eseguita = true;
                        fi
                        }
        od
        }
```

Vediamo che per ogni output place è presente nel costrutto *if* una guardia *true* seguita dal codice per consumare un token nell'input place p_i, e produrre un token nell'unico output place selezionato.

3.1.5 OR-join

Una transizione che realizza un OR-join ha *n* input place p_{i1}, p_{i2}, …, p_{in} ed un unico output place p_o, come mostrato in Figura 3.5:

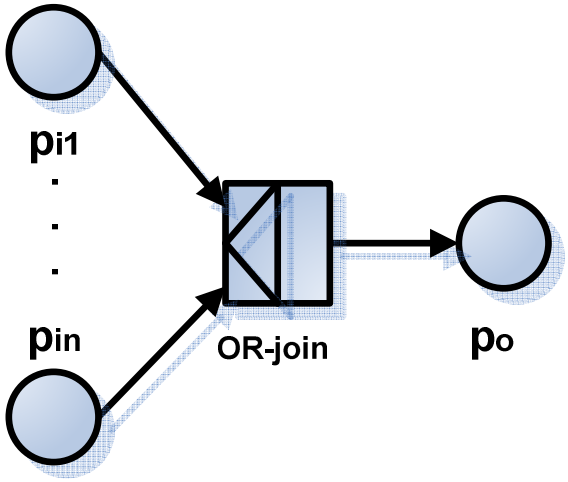


Figura 3.5 OR-join

Sulla base delle considerazioni fatte sulla definizione dell' OR-split, il codice che specifica in Promela un OR-join ha la seguente struttura:

```
proctype OR-join ()  {
        do :: atomic  {
                ( (pi1 > 0) || (pi2 > 0) || … || (pin > 0) ) →
                                if
                                :: (pi1 > 0) → pi1= pi1-1; po= po +1; OR-join_eseguita = true;
                                :: (pi2 > 0) → pi2= pi2-1; po= po +1; OR-join_eseguita = true;
                                                        .
                                                        .
                                                        .
                                :: (pin > 0) → pin= pin-1; po= po +1; OR-join_eseguita = true;
                                fi
                        }
        od      }
```

L’OR-join è abilitato quando uno degli *n* input place contiene almeno un token, quando la transizione scatta consuma un token dall’input place e ne produce uno nell’unico output place.
In questo caso il codice che deve occuparsi di verificare che la transizione sia abilitata dovrà quindi verificare che una delle variabili associate agli input place sia maggiore di zero:

$$((p_{i1} > 0) \,||\, (p_{i2} > 0) \,||\, \ldots \,||\, (p_{in} > 0))$$

Come nel caso dell’OR-split, anche in questo caso utilizziamo il costrutto *if*, questa volta le guardie però sono delle istruzioni per verificare quale sia stato effettivamente l’input place che ha abilitato la transizione; in base alla guardia che risulta vera, viene decrementata la variabile associata all’input place giusto.

3.1.6 Loop

Come ultimo caso consideriamo quello relativo ad un loop. Come abbiamo visto nella sezione 2.1 un loop viene realizzato in una WF-net utilizzando un OR-split; la traduzione in Promela pertanto è riconducibile alla traduzione di un OR-split. Consideriamo ad esempio il loop di Figura 3.6:

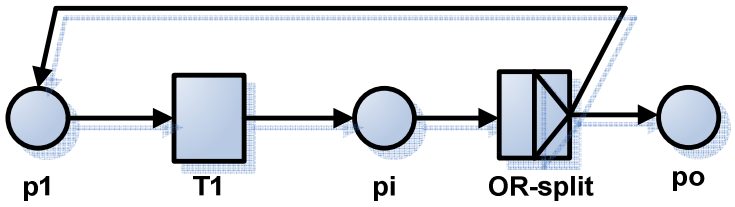


Figura 3.6 Loop

La transizione T_1 è una transizione semplice e la sua traduzione sarà pertanto del tipo di quella descritta in 3.1.1; il codice Promela relativo all'OR-split, sulla base della descrizione data in 3.1.4 sarà il seguente:

```
proctype OR-split () {
        do :: atomic  {
           (p_i > 0) → if
                                :: true→ p_i= p_i -1; p_o= p_0+1; OR-split_eseguita = true;
                                :: true → p_i= p_i -1; p_1= p_1+1; OR-split_eseguita = true;

                        fi
                        }
        od
        }
```

Se quindi verrà eseguita la prima guardia true, il ciclo termina; se viceversa è la seconda guardia ad essere eseguita, il ciclo verrà ripetuto.

3.1.7 Schema di traduzione

Sulla base delle considerazioni appena fatte quindi, una WF-net verrà specificata in Promela seguendo questo schema di base:

```
# define Place int                    //i Place vengono tradotti in variabili di tipo intero
                                      //che indicano il numero di token presenti nel place

bool transizione1_eseguita =false;    //per ogni transizione viene definita una variabile
.                                     //booleana inizializzata a false che verrà posta a
.                                     //true quando la transizione scatta.
.
bool transizionen_eseguita=false;


Place inizio,p1,p2,…,pn, fine;        //per ogni place viene definita una variabile di tipo Place


proctype transizione1()               //ad ogni transizione corrisponde un processo nel cui
{                                     //corpo è presente il codice specifico della transizione
}                                     //come descritto nel paragrafo 3.1
.
.
.
proctype transizionen()
{
}


init{                                 //inizializzazione
atomic{
inizio=1;                             //marcatura iniziale: un token nel place inizio e nessun
p1=0;                                 //token negli altri place
p2=0;
```

```
.
.
.
pn=0;
fine=0;

run transizione1();          //vengono lanciati tutti i processi corrispondenti alle transizioni
.                            //della rete
.
.
run transizionen();
}
}
```

3.2 Traduzione in LTL della proprietà *Sound*

Ricordiamo che una WF-net è *sound* se e solo se:

1. Per ogni token inserito nell'unico place iniziale *inizio*, uno ed uno solo token prima o poi appare nell'unico place finale *fine*.
2. Quando il token appare in *fine*, tutti gli altri place sono vuoti.
3. Per ogni transizione *t* è possibile partire dallo stato iniziale ed arrivare ad uno stato in cui *t* scatta.

Il nostro obiettivo è quello di tradurre i requisiti 1, 2 e 3 in formule LTL che dovranno poi essere valutate su un modello Promela corrispondente ad una WF-net, al fine di verificare se quest'ultima sia *sound*. Introduciamo a tal fine i seguenti predicati:

marcato(p) per indicare che il place *p* è marcato (contiene cioè un token);
abilitata(t) per indicare che la transizione *t* è abilitata (tutti gli input place di *t* contengono un token);
eseguita(t) per indicare che la transizione *t* è stata eseguita, è cioè "scattata".

A questo punto possiamo utilizzare i predicati appena elencati per tradurre i requisiti 1, 2 e 3 in formule LTL. Considereremo sempre che la marcatura iniziale della rete preveda un solo token nel place *inizio* e zero token in tutti gli altri place, come già è stato fatto notare nello schema di traduzione del paragrafo 3.1.7.

I requisiti 1 e 2 possono essere espressi in LTL attraverso la seguente formula:

$$F\ (marcato(fine) \wedge \neg marcato(inizio) \wedge \neg marcato(p_1) \wedge \neg marcato(p_2) \wedge \ldots \wedge \neg marcato(p_n))$$

Partendo dallo stato con un solo token nel place inizio, la formula è valida se prima o poi (*Future*) il place *fine* contiene un solo token e tutti gli altri zero.
Nel nostro modello Promela, un place contiene un token se la variabile ad esso associata è pari ad uno, la formula LTL da valutare in Spin diventa pertanto la seguente:

$$F\ (fine==1\ \&\&\ inizio==0\ \&\&\ p_1==0\ \&\&\ p_2==0\ \&\&\ \ldots\ \&\&\ p_n==0) \qquad (1)$$

Il requisito 3 può essere verificato valutando, per ogni transizione *t*, la non validità della seguente formula LTL:

$$G(\neg eseguita(t))$$

Se la formula risulta valida infatti, significa che la transizione in questione non sarà mai eseguita e quindi si tratta di una “dead transition”.

Nel nostro modello Promela, una transizione risulta eseguita quando la variabile *transizione_eseguita* è stata posta al valore *true*, la formula LTL da valutare in Spin diventa pertanto la seguente:

$$G (\neg (t\_eseguita = = true)) \quad (2)$$

Se tale formula risultasse valida, significherebbe che non esiste uno stato in cui la variabile *t_eseguita* viene posta a *true*, cioè non esiste uno stato in cui *t* scatta.

3.3 Esempio

Riportiamo ora un esempio completo; consideriamo la WF-net di Figura 3.7, definiremo prima la rete in Promela e successivamente verificheremo se si tratta di una WF-net *sound*.

La rete è composta da cinque transizioni *OR split*, *AND split*, T_1, *AND join* ed *OR join* e da otto place *inizio*, p_1, p_2, p_3, p_4, p_5, p_6, *fine*.

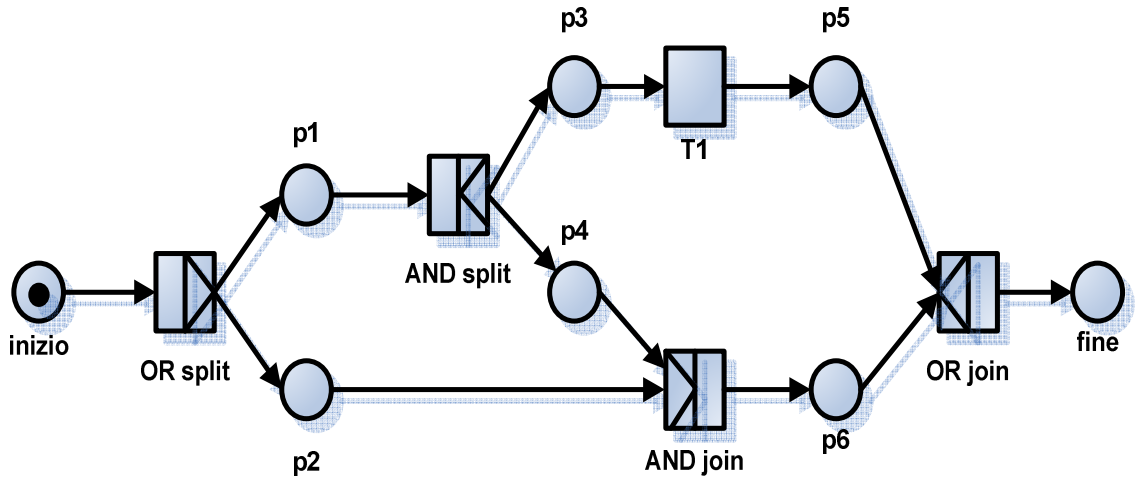


Figura 3.7 Esempio

Il codice Promela corrispondente alla rete è riportato nel file *Esempio.txt*.

Verifichiamo ora la proprietà *sound*; consideriamo prima di tutto la formula (1) definita nel paragrafo 3.2. Definiamo quindi in Spin la formula *formula1*:

#define formula1 (fine==1 && inizio==0 && p1==0 && p2==0 && p3==0 && p4==0 && p5==0 && p6==0)

e verifichiamo la formula LTL:

F(formula1)

Spin esegue una simulazione e la formula risulta non valida; in particolare viene fornito un contro esempio in cui quando un token raggiunge il place *fine (fine==1*) il place *p4* contiene ancora un token (*p4==1*). La WF-net di figura 3.7 non è pertanto *sound*. Il codice per Spin, relativo alla verifica della proprietà sound è riportato nel file *EsempioTerminazione.txt.ltl*.

Verificando ora la formula (2) definita nel paragrafo 3.2 per ogni transizione della rete, possiamo individuare quali transizioni corrispondano a delle "dead transition". Definiamo quindi in Spin le seguenti formule LTL:

```
#define formula2orsplit      (ORSPLIT_eseguita==true)
#define formula2andsplit     (ANDSPLIT_eseguita==true)
#define formula2orjoin       (ORJOIN_eseguita==true)
#define formula2andjoin      (ANDJOIN_eseguita==true)
#define formula2T1           (T1_eseguita==true)
```

E verifichiamo le formule:

```
G(!formula2orsplit)
G(!formula2andsplit)
G(!formula2orjoin)
G(!formula2andjoin)
G(!formula2T1)
```

L'unica formula che risulta valida è *G(!formula2andjoin),* quindi la transizione *andjoin* è una "dead transition". Notiamo infatti che la condizione per far scattare l'AND-join è che ci sia un token in p_2 ed un token in p_4, ma per come è costruita la rete ciò non potrà mai avvenire.
Le altre formule risultano non valide e Spin fornisce per ciascuna di esse un contro esempio nel quale la transizione viene eseguita.
Il codice relativo alla verifica delle formule è riportato nei file *EsempioOrSplit.txt.ltl*, *EsempioAndSplit.txt.ltl*, *EsempioOrJoin.txt.ltl*, *EsempioAndJoin.txt.ltl*, *EsempioT1.txt.ltl*.

4 Diagrammi delle Attività UML

In questa sezione introduciamo i *Diagrammi delle Attività UML* e vediamo come sia possibile, utilizzando le idee sviluppate per la verifica delle WF-nets, effettuare una verifica formale anche dei diagrammi delle attività. In particolare, la verifica prevede che un diagramma delle attività venga prima trasformato in una WF-net equivalente, successivamente su quest'ultima verranno valutate le formule LTL introdotte nella sezione 3; l'idea è che se la WF-net equivalente risulta *sound*, possiamo considerare il diagramma delle attività di partenza corretto e ben strutturato e garantire così la terminazione del processo modellato dal diagramma.
Il paragrafo 4.1 introduce i diagrammi delle attività ed in particolare gli elementi che utilizzeremo.
Il paragrafo 4.2 mostra come un diagramma delle attività possa essere trasformato in una WF-net equivalente. Il paragrafo 4.3 descrive la verifica formale in Spin dopo la trasformazione. Il paragrafo 4.4 infine, descrive come sia possibile verificare formalmente ulteriori proprietà oltre a quelle già introdotte nella sezione 3.

4.1 Concetti di base

I *Diagrammi delle Attività UML* possono essere visti come una estensione dei Diagrammi degli Stati e delle Transizioni in cui in particolare è stato aggiunto il supporto della concorrenza ed in cui tutti gli stati sono *activity state*, cioè stati in cui viene svolta un'attività.
Un diagramma delle attività quindi consiste in una sequenza di attività e supporta l'esecuzione parallela, quella condizionale e l'esecuzione di cicli, risulta particolarmente utile quindi per la descrizione un processo di workflow.
Nella nostra trattazione prenderemo in considerazione gli elementi elencati nel seguito:

- **Nodo Attività**. Rappresenta un'attività che deve essere svolta. Le attività sono collegate tra loro per mezzo di archi diretti che rappresentano le transizioni tra un'attività e l'altra. Graficamente viene utilizzato il simbolo di Figura 4.1.

Figura 4.1 Nodo Attività

- **Nodo Iniziale.** Indica il punto iniziale del diagramma. Il simbolo grafico utilizzato è rappresentato in Figura 4.2. Nella nostra trattazione assumeremo che un diagramma delle attività abbia un unico Nodo Iniziale.

Figura 4.2 Nodo Iniziale

- **Nodo Finale.** Indica il punto finale del diagramma. In Figura 4.3 è riportato il simbolo grafico utilizzato. Assumeremo che un diagramma delle attività abbia un unico Nodo Finale.

.

Figura 4.3 Nodo Finale

- **Nodo Fork.** Specifica l'esecuzione parallela di attività; viene rappresentato graficamente (Figura 4.4) attraverso una barra con un arco in ingresso e due o più archi in uscita: non appena viene completata l'attività in ingresso, il nodo attiva tutte le attività in uscita, in parallelo.

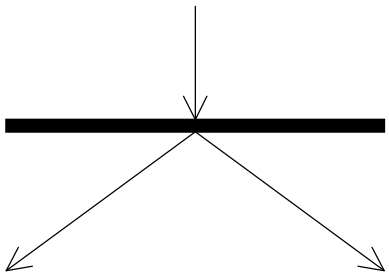


Figura 4.4 Nodo Fork

- **Nodo Join.** Permette la sincronizzazione di attività svolte in parallelo; graficamente viene rappresentato (Figura 4.5) attraverso una barra con due o più archi in ingresso ed un arco in uscita: quando tutte le attività in ingresso sono state completate, il nodo attiva l'attività in uscita.

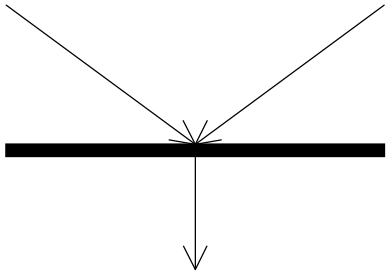


Figura 4.5 Nodo Join

- **Nodo Decisione.** Permette di specificare diramazioni del processo che devono essere eseguite solo se sono verificate determinate condizioni. Graficamente viene utilizzato un rombo con un arco in ingresso e due o più archi in uscita come illustrato in Figura 4.6; agli archi in uscita sono associate delle condizioni, che devono essere in mutua esclusione; verrà quindi attivata l'attività in uscita alla quale è associata una condizione vera, che deve esistere ed essere unica.

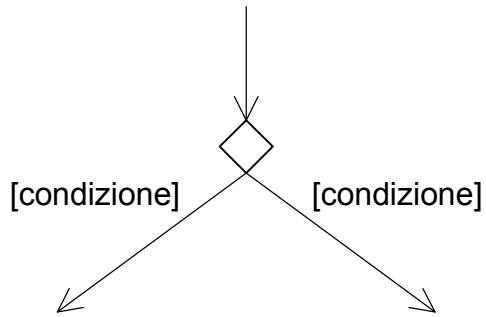


Figura 4.6 Nodo Decisione

- **Nodo Merge.** Viene rappresentato graficamente attraverso un rombo con due o più archi in ingresso ed un arco in uscita come illustrato in Figura 4.7: quando una delle *n* attività in ingresso è completata, il costrutto attiva l'attività in uscita.

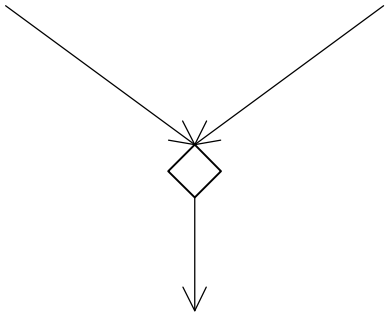


Figura 4.7 Nodo Merge

Per consentire ad un diagramma delle attività di essere interpretato con una semantica analoga a quella di un automa a stati finiti, è necessario che le attività siano correttamente annidate; a tal fine ad ogni Nodo Fork deve corrispondere un Nodo Join e ad ogni Nodo Decisione deve corrispondere un Nodo Merge.

Il diagramma di Figura 4.8 mostra il diagramma delle attività UML per la gestione di un ordine; una volta ricevuto l'ordine, le attività relative alla spedizione e quelle relative alla ricezione del pagamento possono essere svolte in parallelo, a tal fine viene utilizzato un nodo Fork; per sincronizzare le attività prima di chiudere l'ordine viene invece utilizzato un nodo Join. Per modellare la possibilità di scelta della modalità di spedizione viene infine utilizzato un nodo Decisione con opportune condizioni associate agli archi.

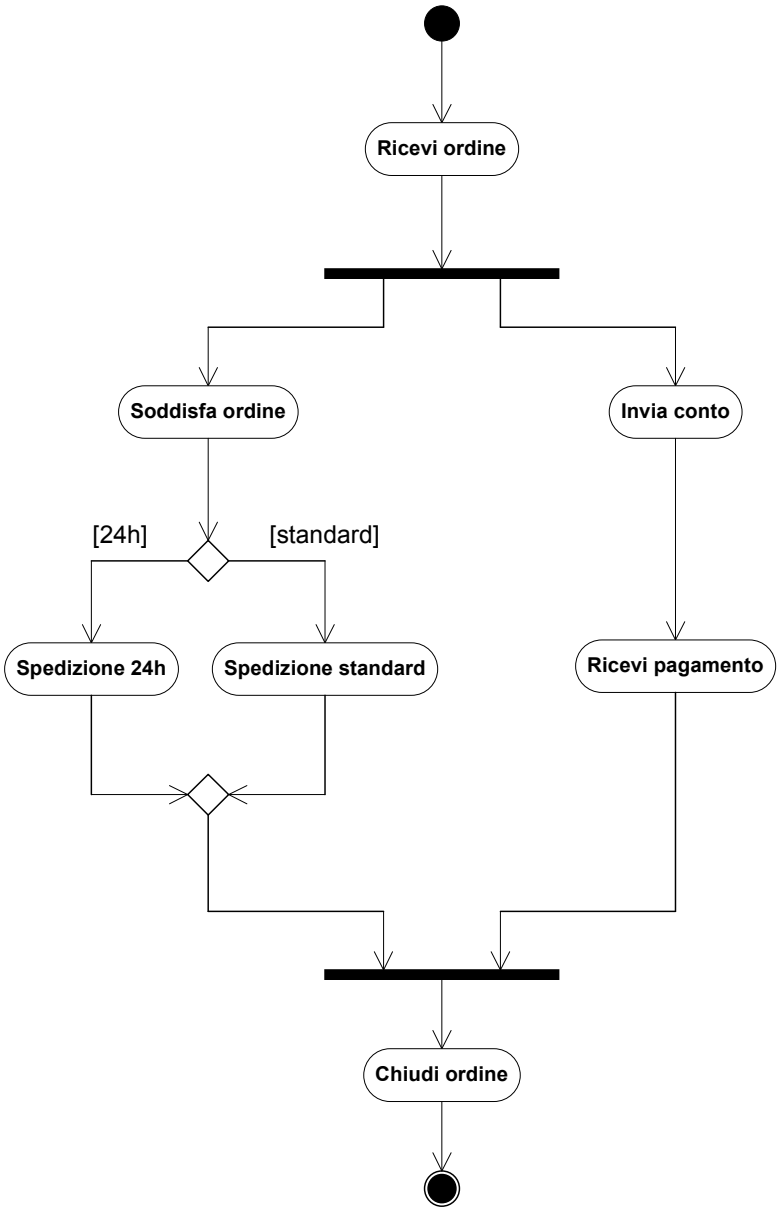


Figura 4.8 Diagramma delle Attività per la gestione di un ordine

4.2 Da Diagramma delle Attività UML a WF-net

Vediamo ora come sia possibile trasformare un Diagramma delle Attività UML in una WF-net equivalente; vediamo in particolare come ad ogni singolo elemento di un diagramma delle attività corrisponda un elemento di una WF-net.

4.2.1 Nodo Attività

Abbiamo visto che un Nodo Attività modella una attività che deve essere eseguita. In una WF-net le attività del processo di workflow che devono essere eseguite sono modellate dalle transizioni; appare evidente quindi la corrispondenza tra un Nodo Attività di un diagramma delle attività UML ed una transizione di una WF-net; nella trasformazione di un diagramma delle attività in una WF-net quindi, un Nodo Attività viene trasformato in una transizione. In Figura 4.1 viene mostrata la trasformazione.

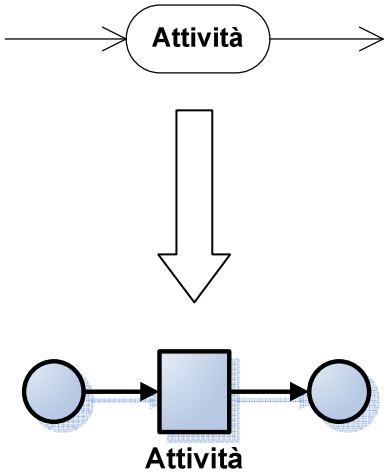


Figura 4.9 Trasformazione Nodo Attività-Transizione

4.2.2 Nodo Iniziale

Un Nodo Iniziale di un diagramma delle attività corrisponde in una WF-net ad un place senza archi entranti; abbiamo visto che una WF-net per definizione ha un unico place senza archi entranti che abbiamo chiamato place *inizio*; nel passaggio da diagramma delle attività a WF-net quindi, il Nodo Iniziale viene trasformato nel place *inizio* come mostrato in Figura 4.10.

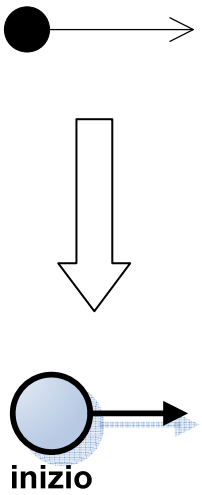


Figura 4.10 Trasformazione Nodo iniziale-Place *inizio*

4.2.3 Nodo Finale

Il Nodo Finale di un diagramma delle attività corrisponde ad un place senza archi uscenti in una WF-net; abbiamo visto che in una WF-net per definizione esiste un unico place senza archi uscenti che abbiamo denominato place *fine*; di conseguenza faremo corrispondere il Nodo Finale di un diagramma delle attività al place *fine* di una WF-net, come mostrato in Figura 4.11.

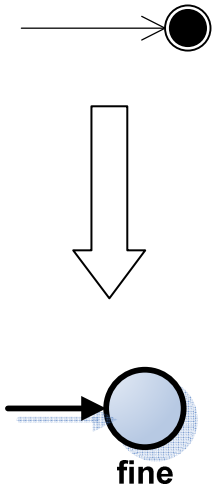


Figura 4.11 Trasformazione Nodo Finale-Place *fine*

4.2.4 Nodo Fork

L'elemento di una WF-net corrispondente ad un Nodo Fork di un diagramma delle attività è chiaramente l'AND-split; la semantica associata a questi due costrutti è infatti la stessa, pertanto nel passaggio da diagramma delle attività a WF-net un Nodo Fork viene trasformato in un AND-split come mostra la Figura 4.12.

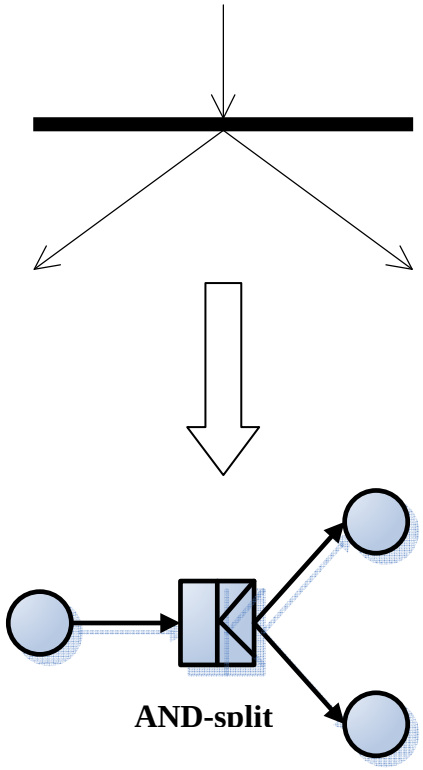


Figura 4.12 Trasformazione Nodo Fork-AND-split

4.2.5 Nodo Join

L'elemento di una WF-net corrispondente ad un Nodo Join di un diagramma delle attività è l'AND-join; la semantica associata a questi due costrutti è infatti la stessa, pertanto nel passaggio da diagramma delle attività a WF-net un Nodo Join viene trasformato in un AND-join, come mostra la Figura 4.13.

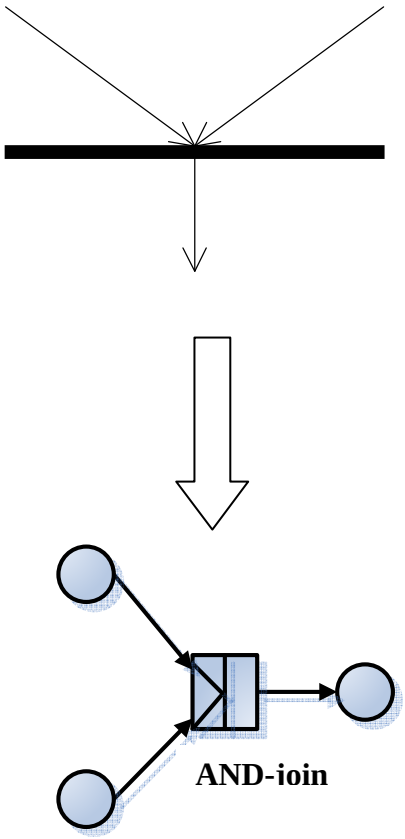


Figura 4.13 Trasformazione Nodo Join-AND join

4.2.6 Nodo Decisione

L'elemento di una WF-net corrispondente ad un Nodo Decisione di un diagramma delle attività è l'OR-split; la semantica associata a questi due costrutti è infatti la stessa, pertanto nel passaggio da Diagramma delle attività a WF-net un Nodo Decisione viene trasformato in un OR-split come mostra la Figura 4.14.

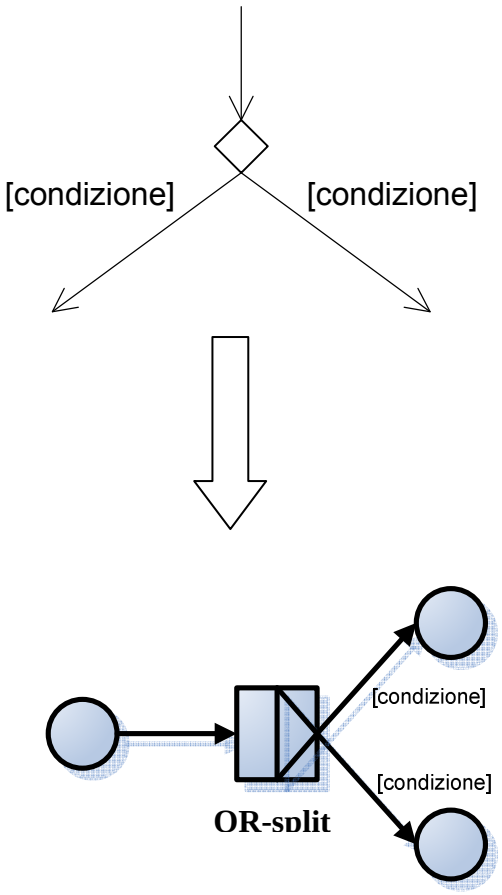


Figura 4.14 Trasformazione Nodo Decisione-OR split

4.2.7 Nodo Merge

L'elemento di una WF-net corrispondente ad un Nodo Merge di un diagramma delle attività è l'OR-join; la semantica associata a questi due costrutti è infatti la stessa, pertanto nel passaggio da diagramma delle attività a WF-net un Nodo Merge viene trasformato in un OR-join come mostra la Figura 4.15.

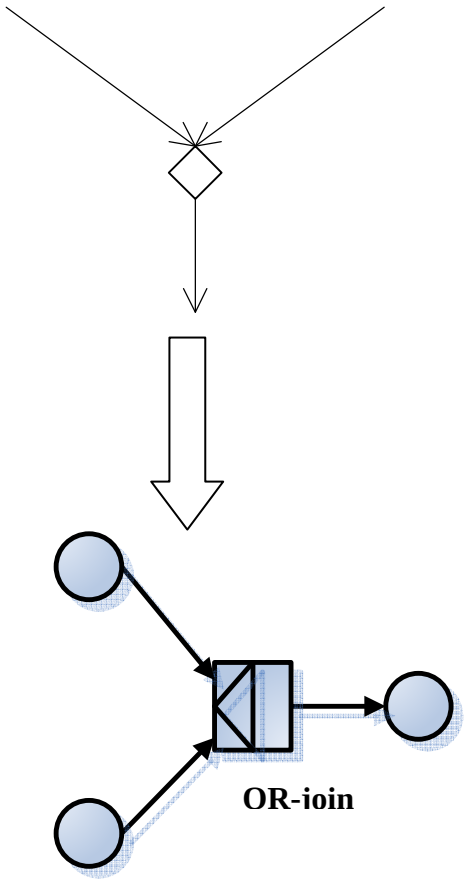


Figura 4.15 Trasformazione Nodo Merge-OR join

4.2.8 Esempio

Mostriamo ora un esempio completo di trasformazione di un diagramma delle attività UML in una WF-net. Il diagramma che andiamo a trasformare è quello di Figura 4.8. La WF-net che si ottiene seguendo lo schema di traduzione descritto nel paragrafo 4.2 è illustrata in Figura 4.16.

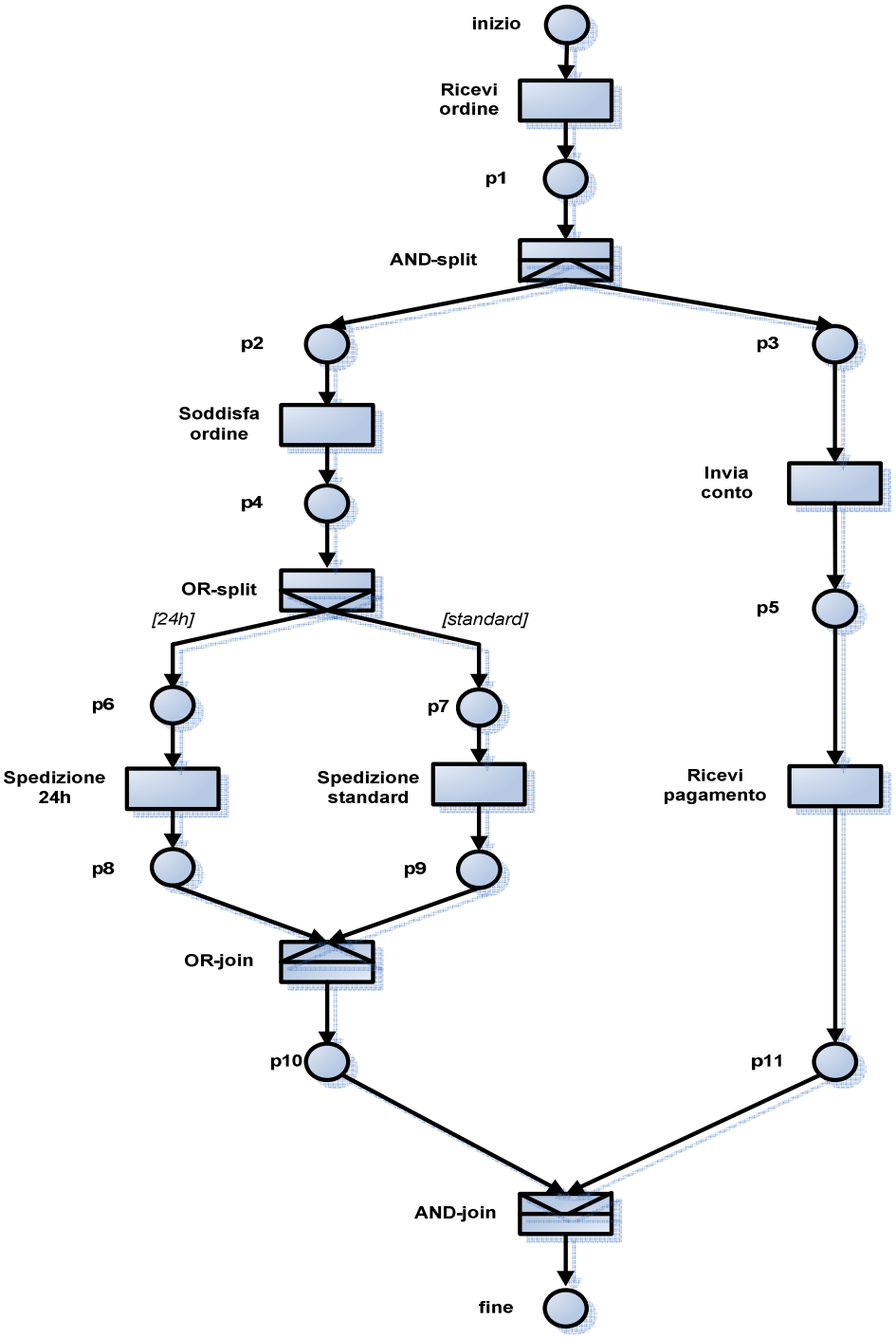


Figura 4.16 WF-net corrispondente al Diagramma delle Attività di Figura 4.8

4.3 Verifica formale dei Diagrammi delle Attività

Vediamo ora come poter verificare formalmente in Spin un Diagramma delle Attività utilizzando le idee sviluppate nella sezione 3 per la verifica formale delle WF-nets. Partendo da un diagramma delle attività, lo trasformeremo in una WF-net equivalente di cui effettueremo la verifica formale in Spin.

Il diagramma delle attività da verificare è illustrato in Figura 4.17 e si riferisce alle attività che un'agenzia di viaggi svolge per organizzare un viaggio. In primo luogo il cliente viene registrato, successivamente un operatore ricerca delle opportunità di viaggio per il cliente che vengono poi comunicate al cliente stesso telefonicamente o via e-mail. Successivamente il cliente viene contattato, a questo punto ci sono tre possibilità: il cliente richiede altre offerte, il cliente non è interessato e chiede di non essere più contattato, il cliente sceglie un'offerta di viaggio. Nel primo caso l'operatore continuerà a cercare offerte per il cliente, nel secondo caso il processo termina, nel terzo caso il viaggio selezionato viene prenotato; in parallelo inoltre vengono offerte al cliente due tipi di assicurazione, il cliente può decidere di non stipulare alcuna assicurazione, di stipularne soltanto una o di stipulare entrambe le assicurazioni. Successivamente la prenotazione viene o cancellata o confermata; nel primo caso la prenotazione viene cancellata ed il processo termina, nel secondo caso verranno inviati i documenti ed il processo terminerà quando il viaggio avrà avuto inizio.

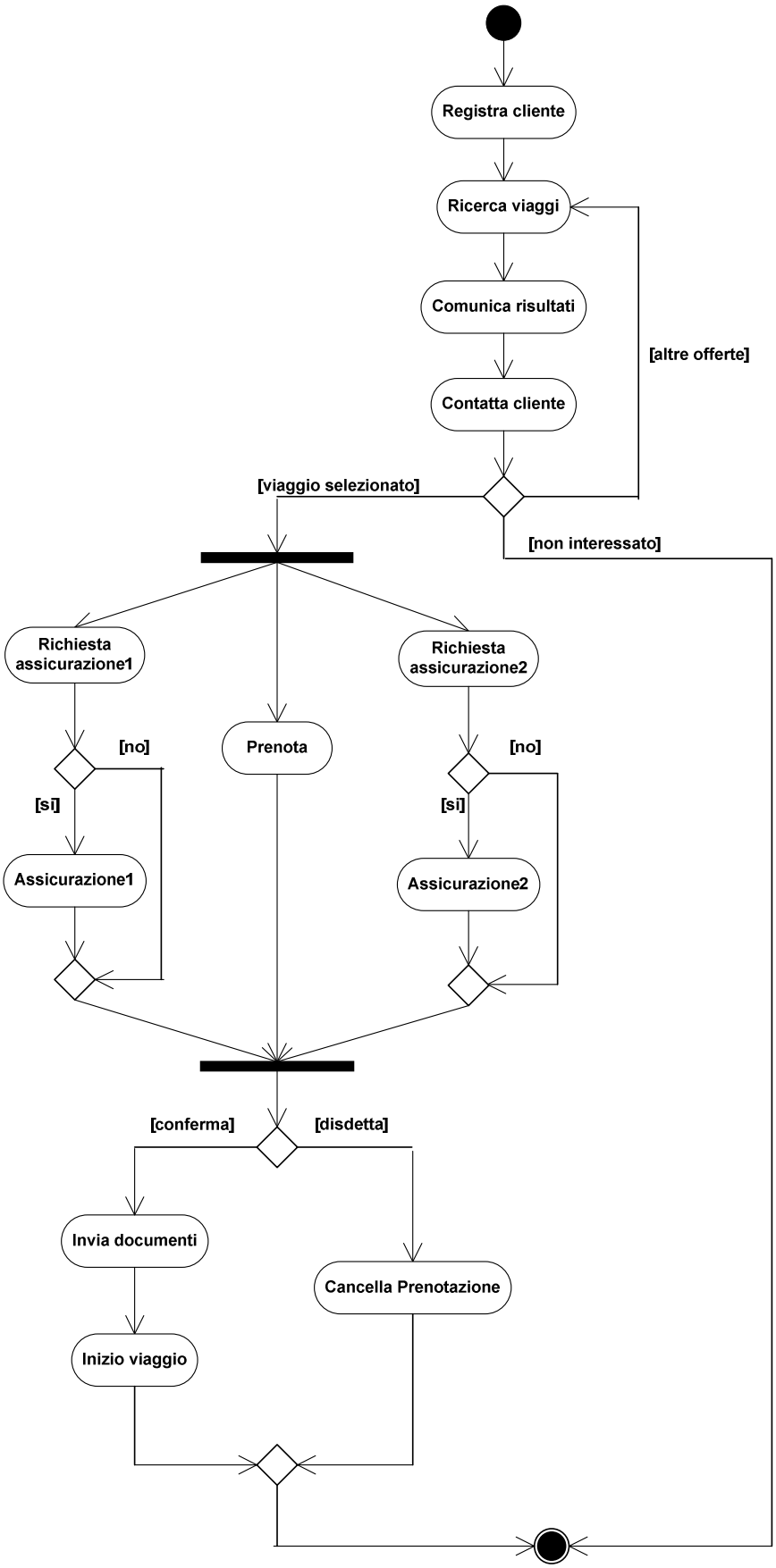


Figura 4.17 Diagramma delle Attività per un'agenzia di viaggi

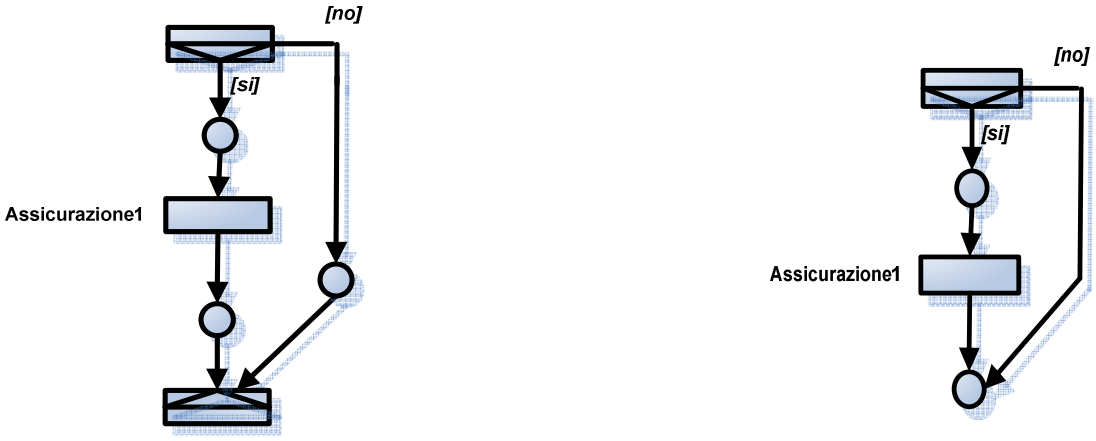
4.3.1 Trasformazione Diagramma delle Attività-WF net

Sulla base delle considerazioni svolte nel paragrafo 4.2, possiamo procedere alla trasformazione del diagramma delle attività in una WF-net equivalente. Al fine di ridurre il numero di elementi della WF-net, nel passaggio dal diagramma delle attività alla WF-net sono state effettuate le seguenti semplificazioni che non variano la semantica della rete ma ne migliorano la leggibilità:

- Nel diagramma delle attività, il Nodo Attività *Contatta cliente* è seguito da un Nodo Decisione; nel passaggio alla WF-net pertanto la trasformazione porterebbe ad una transizione *Contatta cliente* seguita da una transizione *OR-split*, come illustrato in Figura 4.18 (a); la Figura 4.18 (b) mostra però come possiamo associare la transizione *Contatta cliente* direttamente ad un OR-split è ridurre così il numero di transizioni senza modificare la semantica della rete. Allo stesso modo avvengono le semplificazioni per i Nodi Attività *Richiesta assicurazione1* e *Richiesta Assicurazione2*, entrambe seguite da un Nodo Decisione.

Figura 4.18 (a) Trasformazione. (b) Trasformazione con semplificazione

- Nel passaggio da diagramma delle attività a WF-net, la parte del diagramma relativa alla stipulazione dell'Assicurazione1 verrebbe trasformata nella rete di Figura 4.19(a); tuttavia, senza modificarne la semantica, la rete può essere semplificata come in Figura 4.19(b) utilizzando un OR-join implicito invece di uno esplicito e riducendo così sia il numero di place che di transizioni. La stessa cosa avviene per la parte relativa alla Assicurazione2.



(a) **(b)**

Figura 4.19 (a) Trasformazione. (b) Trasformazione con semplificazione

La WF-net completa corrispondente al diagramma delle attività di Figura 4.17 è riportata in Figura 4.20.

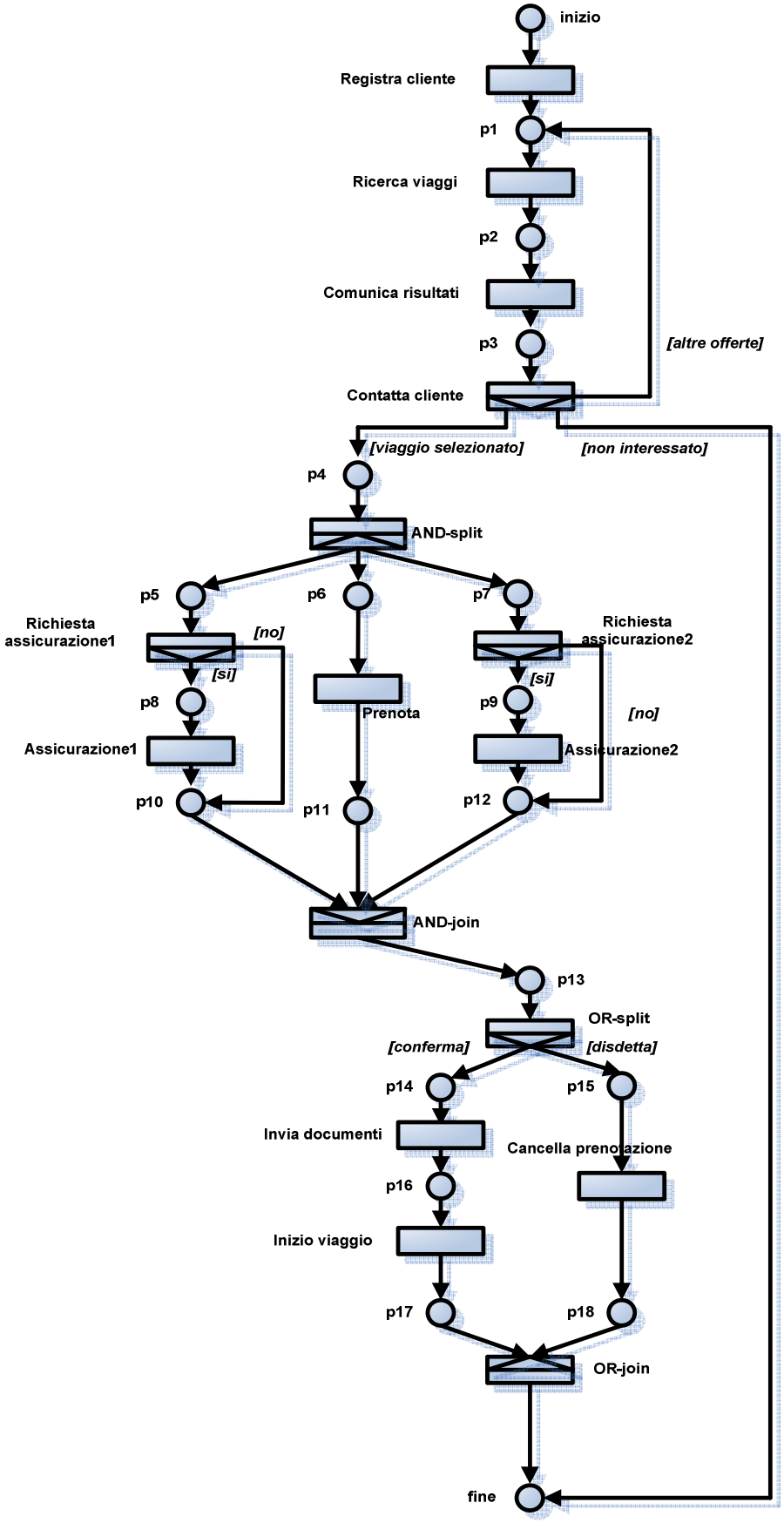


Figura 4.20 WF-net per il Diagramma delle Attività di Figura 4.17

4.3.2 Verifica

A questo punto la WF-net di Figura 4.20 può essere sottoposta a verifica. In particolare, dopo aver tradotto la rete nel modello Promela, possiamo verificarne le proprietà descritte in 3.3. Se la verifica dovesse risultare valida, la WF-net sarebbe allora *sound* e di conseguenza il diagramma delle attività di partenza rappresenterebbe un processo di workflow ben strutturato e corretto. Il modello Promela corrispondente alla WF-net di Figura 4.20 è riportato nel file *Agenzia.txt*.

Verifichiamo prima di tutto la formula (1) introdotta nel paragrafo 3.3. A tal fine verifichiamo in Spin la seguente formula LTL:

F terminazione

dove *terminazione* è definita nel seguente modo:

 # define terminazione (fine==1 && inizio==0 && p1==0 && p2==0 && p3==0 && p4==0 && p5==0 && p6==0 && p7==0 && p8==0 && p9==0 && p10==0 && p11==0 && p12==0 && p13==0 && p14==0 && p15==0 && p16==0 && p17==0 && p18==0)

La verifica in Spin ci dice che la formula è valida, la WF-net è cioè *sound* e di conseguenza il diagramma delle attività di Figura 4.17 rappresenta un processo ben strutturato la cui terminazione è garantita. Il codice per la verifica della proprietà *sound* della WF-net di figura 4.20 è riportato nel file *Agenzia.tx*t.

Notiamo che nella WF-net è presente un ciclo, quello costituito dalle transizioni *Ricerca viaggi*, *Comunica risultati*, *Contatta cliente* e dai place p_1, p_2 e p_3. Se tale ciclo venisse eseguito infinite volte, è chiaro che la terminazione non potrebbe mai essere verificata; per la verifica della terminazione quindi i cicli vengono considerati sempre finiti. A tale scopo, nel codice Promela è stata introdotta una variabile contatore che viene incrementata ogni volta che il ciclo si ripete, superato il valore 20, il ciclo non potrà più ripetersi.

4.4 Verifica di ulteriori proprietà

In questo paragrafo viene descritto come possano essere verificate in Spin ulteriori proprietà di diagrammi delle attività e in generale delle WF-net. In particolare ci riferiamo a proprietà specifiche del processo modellato dal diagramma. Come esempio consideriamo sempre il diagramma di Figura 4.17 e la WF-net corrispondente di Figura 4.20. Nel contesto modellato dal diagramma potrebbe essere utile verificare alcune proprietà relative alle possibilità che il processo offre; potremmo ad esempio voler verificare le seguenti proprietà:

(1) Un cliente può partire per un viaggio senza aver stipulato alcuna assicurazione.
(2) Un viaggio prenotato può essere cancellato.
(3) Se un viaggio viene cancellato, i documenti non vengono inviati.
(4) Una volta inviati i documenti, un viaggio non può più essere cancellato.

Le proprietà elencate possono essere tradotte in formule LTL e verificate sul modello Promela della WF-net.

La proprietà (1) può essere verificata attraverso una formula che valuti se il fatto che le transizioni *Assicurazione1* e *Assicurazione2* non vengano eseguite implicano che la transizione *Inizio viaggio* non venga eseguita. La formula è la seguente:

$$G\ (\ (!assicurazione1\ \&\&\ !assicurazione2) \rightarrow (!inizioviaggio)\ )$$

Dove *assicurazione1*, *assicurazione2* ed *inizioviaggio* sono così definiti:

```
#define assicurazione1 (ASSICURAZIONE1_eseguita==true)
#define assicurazione2 (ASSICURAZIONE2_eseguita==true)
#define inizioviaggio (INIZIOVIAGGIO_eseguita==true)
```

La formula risulta non valida e Spin mostra un contro esempio nel quale le due transizione *Assicurazione1* e *Assicurazione2* non vengono eseguite ma la transizione *Inizio viaggio* viene eseguita, mostrando così che nel processo modellato dal diagramma un cliente può partire per un viaggio senza aver stipulato alcuna assicurazione.

La proprietà (2) può essere verificata attraverso una formula che valuti se il fatto che venga eseguita la transizione *Prenota* implica che la transizione *Cancella prenotazione* non venga eseguita. La formula è pertanto la seguente:

G (prenota → !cancellaprenotazione)

Dove *prenota* e *cancellaprenotazione* sono così definiti:

```
#define prenota (PRENOTA_eseguita==true)
#define cancellaprenotazione (CANCELLAPRENOTAZIONE_eseguita==true)
```

Anche in questo caso la formula è non valida e Spin mostra un contro esempio in cui vengono eseguite entrambe le transizioni, mostrando quindi che nel processo modellato dal diagramma un viaggio prenotato può successivamente essere cancellato.

La proprietà (3) può essere verificata attraverso la seguente formula LTL:

G (cancellaprenotazione → !inviadocumenti)

Dove *inviadocumenti* e *cancellaprenotazione* sono così definiti:

```
#define inviadocumenti (INVIADOCUMENTI_eseguita==true)
#define cancellaprenotazione (CANCELLAPRENOTAZIONE_eseguita==true)
```

La formula verifica se ogni qualvolta viene eseguita la transizione *Cancella prenotazione*, questo implica che la transizione *Invia documenti* non venga eseguita.
La formula risulta valida in Spin, mostrando che nel processo modellato, se una prenotazione viene cancellata, i documenti relativi al viaggio correttamente non vengono inviati al cliente.

La proprietà (4) può essere verificata attraverso la seguente formula LTL:

G (inviadocumenti → !cancellaprenotazione)

Dove *inviadocumenti* e *cancellaprenotazione* sono definiti come per la proprietà (3);

La formula verifica se ogni qualvolta viene eseguita la transizione *Invia documenti*, questo implica che la transizione *Cancella prenotazione* non venga eseguita.
La formula risulta valida in Spin, mostrando che nel processo modellato, una volta che i documenti relativi al viaggio sono stati inviati al cliente, non è più possibile cancellare il viaggio.
Il codice relativo alla verifica delle proprietà (1), (2), (3) e (4) è presente nei seguenti file: *AgenziaProprieta1.txt.ltl*, *AgenziaProprieta2.txt.ltl*, *AgenziaProprieta3.txt.ltl*, *AgenziaProprieta4.txt.ltl*.

Riferimenti

[1] Will van der Aalst, Kees Max van Hee. *Workflow Management: Models, Methods, and Systems*.

[2] Theo C. Ruys. *SPIN Beginners' Tutorial*.

[3] Gerard Holzmann, Theo C. Ruys. *Advanced SPIN Tutorial*.

[4] *http://spinroot.com*