Un linguaggio per esprimere transizioni

Paolo Bottoni Maria De Marsico Paolo Di Tommaso Stefano Levialdi
Domenico Ventriglia
Dipartimento di Informatica – Università di Roma "La Sapienza"

***Abstract*—**
I linguaggi di specifica del comportamento di agenti, e più in generale di sistemi reattivi, spesso utilizzano notazioni grafiche per esprimere le configurazioni significative dello stato dell'agente e le trasformazioni ammissibili di tali configurazioni. L'espressione dei due aspetti può essere delegata a diversi tipi di diagrammi, o venire incorporata in un unico diagramma. Inoltre, si possono sviluppare diagrammi comportamentali per diverse componenti del sistema, eventualmente impiegando diagrammi di tipo diverso per ogni sottosistema. A un livello astratto, tutti questi diagrammi esprimono qualche forma di trasformazione del sistema, che può essere caratterizzata esprimendone le pre- e post-condizioni, e una politica di esecuzione delle trasformazioni. In questo lavoro, proponiamo un approccio uniforme alla gestione di transizioni, indipendente dalla notazione diagrammatica adottata, e che può essere usata per stabilire corrispondenze tra modifiche della rappresentazione visuale e trasformazioni del modello sottostante.

I. Introduzione

I linguaggi di specifica del comportamento di agenti, e più in generale di sistemi reattivi, spesso utilizzano notazioni grafiche per esprimere le configurazioni significative dello stato dell'agente e le trasformazioni ammissibili di tali configurazioni. La scelta del modello da usare per la specifica del comportamento è spesso legata alle caratteristiche del problema, o alle notazioni in uso presso una comunità di utenti. In genere, però, si ha a che fare con specifiche a eventi discreti, che sono spesso il modo più naturale per esprimere comportamenti.

I modelli di sistemi complessi richiedono la composizione di sottosistemi specificati in modi differenti, per esempio a diversi livelli di astrazione. Così, un sistema produttore-consumatore può essere modellato mediante una rete di Petri, se ci si vuole focalizzare sulla relazione tra le due attività, mentre gli specifici comportamenti dei sistemi che agiscono come produttori o consumatori potrebbero essere definiti in modi diversi fra loro. In questi casi, si è spesso costretti a ricorrere a un unico formalismo (per esempio esprimendo tutti i comportamenti con reti di Petri), oppure i diversi comportamenti vanno connessi fra loro a un livello astratto.

La nostra proposta è di usare una nozione astratta di *transizione*, che si trova alla base delle più comuni specifiche di sistemi a eventi discreti espresse mediante linguaggi visuali. I diversi sistemi, o loro componenti, possono quindi venire modellati in termini delle transizioni che essi possono eseguire, indipendentemente dalla notazione utilizzata per specificarle. Inoltre, è possibile differenziare il *contenuto* di una trasformazione dalla *politica applicativa*, p.e. sincrona, asincrona, esaustiva, sequenziale.

Supportato parzialmente dal Ministero dell'Università e Ricerca e dall'ESPRIT Working Group SEGRAVIS

Le transizioni sono viste astrattamente come produzione e consumo di risorse, mentre la comunicazione fra componenti è realizzata attraverso interfacce di *import* e *export*, che definiscono quali risorse possono venire introdotte in un sistema o da questo estratte. Le singole transizioni vengono applicate in modo dipendente dalla semantica del modello.

Proponiamo qui il linguaggio WIPPOG (dalle iniziali di WHEN, IF, PRODUCES, PROCESSES, OUTS e GETS) e il suo modello computazionale, che opera su una rappresentazione astratta della transizione e mantiene una connessione con la configurazione corrente del sistema. In questo modo, possiamo disaccoppiare la notazione in cui viene espressa una specifica dal suo significato in termini di transizioni, nonché disaccoppiare la specifica di una transizione dal meccanismo di applicazione che definisce un passo di trasformazione.

Nella Sezione II presentiamo i concetti principali di WIPPOG nel contesto della letteratura sulla programmazione in logica lineare, e discutiamo alcuni modelli generali per l'espressione di comportamenti sviluppati nel campo dei linguaggi visuali. Le Sezioni III e IV sono dedicate rispettivamente alla presentazione del linguaggio e modello computazionale di WIPPOG e della *WIPPOG machine*. La sezione V presenta un'applicazione alla modellazione di sistemi gerarchici. La sezione VII illustra alcuni possibili sviluppi, e alcune conclusioni sono tratte in Sezione VIII.

II. Analisi della letteratura

Il modello computazionale di WIPPOG deriva da quello delle *Interaction Machines* [ACP93], a sua volta una visione astratta del modello di *LO* (Linear Objects) [AP91]. Questi modelli sono basati sul cosiddetto frammento moltiplicativo della logica lineare [Gir95], [And92], in cui un passo di computazione è essenzialmente costituito della produzione e del consumo di insiemi finiti di risorse in un *resource pool*, mentre alcune risorse possono essere trasmesse in modo *broadcast* ad altri agenti. A differenza del modello LO, in WIPPOG non permettiamo la creazione di nuovi agenti, e consideriamo un meccanismo di registrazione mediante interfacce di *import* e *export*. Un simile meccanismo è stato usato nell'ambiente *Forumtalk* [And95], dove le risorse possono venire trasmesse sia ad altri agenti nella stessa locazione virtuale, sia ad agenti le cui locazioni sono state registrate per l'import. In Forumtalk, come in LO, le risorse ricevute vengono internalizzate all'agente, senza distinguerle da quelle prodotte internamente.

Molti modelli definiscono sistemi reattivi o concorrenti in modo indipendente dalle realizzazioni specifiche, ma tenendo in considerazione le componenti generali di una trasformazione. Un esempio di questo approccio è il formalismo

visuale delle Δ- grammars [KLG93], in cui le trasformazioni sono definite da un insieme di Δ-produzioni, ognuna rappresentata da un triangolo e consistente di cinque componenti:

1) **retraction**: parte rimossa in una riscrittura;
2) **insertion**: parte creata in una riscrittura;
3) **context**: parte controllata dalla produzione ma non rimossa;
4) **restriction**: parte che non deve esistere affinché la produzione sia applicabile;
5) **guard**: condizione booleana che deve essere soddisfatta per poter applicare la condizione.

Solo la parte corrispondente alla **retraction** viene rimossa, e più applicazioni possono occorrere simultaneamente.

WIPPOG tiene conto sia delle trasformazioni interne, sia della comunicazione con altri sistemi. Esso va messo in relazione con gli approcci di metalivello, che offrono dei quadri di riferimento per la creazione e definizione di sintassi e semantiche per specifici linguaggi.

Un esempio è Moses [EJ01], che si basa su una descrizione di sentenze visuali in una sintassi astratta espressa con grafi attribuiti, secondo l'approccio in [Erw98]. Da questa specifica, risulta definito un controllore sintattico. Moses fornisce un ambiente in cui un utente può costruire una sentenza, mentre il controllore sintattico opera sullo sfondo, cosìche l'utente può creare frasi temporaneamente scorrette. La semantica viene gestita creando interpreti specifici nella forma di *Abstract State Machines* [BCR00] per ogni linguaggio visuale. Nel nostro approccio, WIPPOG è usato sia per esprimere azioni legali in un ambiente di costruzione di frasi guidato dalla sintassi, generato dalla specifica del linguaggio, sia per esprimere la semantica di una sentenza in termini di comportamento, in quanto entrambe possono essere espresse come sistemi a eventi discreti.

Il problema della connessione fra componenti (per la simulazione di sistemi) espresse con diversi formalismi visuali è affrontato nell'ambiente AToM^3 [dLV02]. I formalismi visuali sono espressi con meta-metamodelli. Una volta adottato un formalismo, si possono produrre metamodelli che descrivono il tipo di sistemi da simulare. Infine, un modello definisce una specifica istanza di sistema. Si possono esprimere trasformazioni fra modelli mediante grammatiche di grafi, potendo quindi trasformare una specifica da un formalismo a un altro. Il problema della connessione fra le componenti è risolto proiettando le diverse componenti su un unico formalismo. Nel nostro approccio, questo problema non si pone, in quanto la loro espressione in WIPPOG è indipendente dal formalismo visuale originale. WIPPOG definisce in termini *meta* le caratteristiche semantiche delle transizioni, e quindi potrebbe beneficiare di una definizione di metalivello delle notazioni grafiche, in quanto, al momento attuale, la traduzione della semantica operazionale dal formalismo visuale a WIPPOG va codificata esplicitamente per ogni linguaggio.

Mentre WIPPOG è basato sulla riscrittura di multinsiemi, la riscrittura di grafi è spesso usata per esprimere in modo uniforme sintassi e semantica di linguaggi visuali [BMST99]. Gli strumenti basati su riscrittura di grafi sono però legati a una specifica politica di applicazione, e devono ricorrere a forme ibride, che usano anche espressioni testuali, per guidare l'applicazione delle regole nel modo desiderato.

III. Presentazione di WIPPOG

WIPPOG è basato su una nozione astratta di *transition* eseguita da un agente, e descrive diversi aspetti delle sue pre- e post-condizioni. In questo contesto la parola "agente" non si riferisce a uno specifico modello di agente software, ma ad una qualsiasi entità modellata come sistema computazionale.

Un agente possiede un insieme di regole che definiscono l'insieme dei suoi possibili comportamenti, e un insieme (*pool*) di risorse che descrivono il suo stato a un dato momento. Una *risorsa* è un elemento tipato, descritto dalla sintassi[1]:

```
Item = TypeName '(' ['id' = Identifier ';']
                    CSLOFAVP ')'
```

dove `TypeName` è una stringa da un insieme TN e identifica il *tipo* della risorsa da un insieme T, `Identifier` è una stringa che identifica univocamente l'elemento (useremo la convenzione di formare identificatori concatenando le iniziali del nome del tipo con un numero progressivo), `CSLOFAVP` è una lista separata da virgole di coppie attributo-valore. Il tipo della risorsa determina univocamente i nomi degli attributi da un insieme Nms, mentre i valori sono presi da un dominio associato al nome dell'attributo. L'insieme di tutti i domini è detto D. Nessun attributo può apparire due volte in un elemento. I requisiti di cui sopra sono espressi formalmente come: $\forall t \in T$, $\exists atts(t) \subset Nms$ e $\forall nm \in Nms$, $\exists d_{nm} \in D$. Nell'implementazione attuale, i valori possono essere presi dai domini dei tipi base types, `int`, `string`, `boolean`, `Id` e da liste di questi tipi. Un tipo `Object` può essere specializzato per tipi definiti dall'utente. L'insieme di tutte le risorse definibili su T è detto $W(T)$.

Una transizione è modellata come il consumo e la produzione di risorse sotto certe condizioni. A questo scopo, WIPPOG distingue le risorse interne, che devono essere presenti nell'agente, da quelle esterne, che l'agente ha ricevuto da altri sistemi, o, in generale, dal suo ambiente (incluse le risorse che rappresentano ingressi da un utente). Analogamente le risorse possono essere prodotte per rimanere all'interno dell'agente o per essere diffuse nell'ambiente, possibilmente per essere utilizzate da altri agenti.

Le precondizioni alle transizioni possono includere controlli sui valori degli attributi di risorse interne od esterne da consumare. I valori per gli attributi delle risorse da produrre sono calcolati in una componente dedicata della transizione. Una regola può riferirsi al valore di un attributo per mezzo di una variabile. Le variabili nelle precondizioni vengono unificate con i valori per le risorse esistenti. Le variabili con lo stesso nome assumono lo stesso valore.

Una transizione WIPPOG è perciò definita da sei componenti:

- **WHEN:** risorse che devono essere internamente disponibili all'agente. Gli attributi possono venire menzionati con un valore costante o con un nome di variabile.
- **GETS:** risorse prodotte esternamente che devono essere disponibili all'agente. Gli attributi possono venire menzionati con un valore costante o con un nome di variabile.

[1] L'effettiva sintassi di WIPPOG è in XML ed è descritta da un DTD. Usiamo qui una versione semplificata.

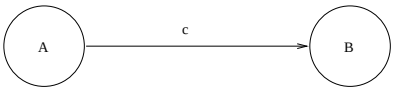

Fig. 1. Una transizione in una macchina a stati finiti.

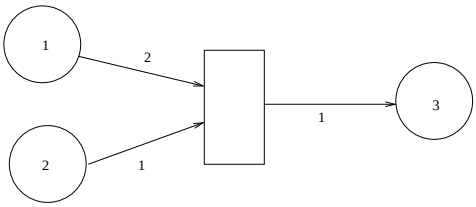

Fig. 2. Una transizione in una rete di Petri PT.

- **IF:** condizioni sulle variabili nelle componenti WHEN o GETS.
- **PROCESSES:** attività computazionali associate alle transizioni. Queste hanno sempre successo. In particolare, si possono qui specificare degli assegnamenti per i valori degli attributi di risorse da creare. Si possono usare nelle espressioni delle variabili introdotte nelle componenti WHEN o GETS.
- **PRODUCES:** risorse che devono essere create internamente come risultato della transizione. Se appaiono delle variabili, i loro nomi devono apparire o nelle componenti WHEN o GETS, o nel lato sinistro di un assegnamento nella componente PROCESSES.
- **OUTS:** risorse che sono rese esternamente disponibili come risultato della transizione. Se appaiono delle variabili, i loro nomi devono apparire o nelle componenti WHEN o GETS, o nel lato sinistro di un assegnamento nella componente PROCESSES.

L'applicazione di una transizione WIPPOG: a) rimuove dal pool di risorse (interne o esterne) disponibili a un agente quelle menzionate nelle componenti WHEN e GETS se le condizioni nella componente IF sono soddisfatte; b) esegue le attività specificate nella componente PROCESSES; c) produce (per uso interno o esterno) le risorse specificate nelle componenti PRODUCES e OUTS.

Ad esempio, una transizione in una macchina a stati finiti, da uno stato A a uno stato B alla ricezione dell'ingresso c, come mostrato in Figura 1, viene codificata dalla regola:

WHEN: current(id = A)
GETS: input(value = c)
PRODUCES: current(id = B)

Una transizione in una rete di Petri Posti/Transizioni, che rimuove due token da un posto $pl1$, un token da un posto $pl2$, e inserisce un token in un posto $pl3$, come specificato in Figura 2, è codificata dalla regola:

WHEN: place(id = 1; tokens = X), place(id = 2; tokens = Y), place(id = 3; tokens = Z)
IF: X $\geq$ 2, Y $\geq$ 1
PROCESSES: X1 := X -2, Y1 := Y -1, Z1 := Z +1
PRODUCES: place(id = 1; tokens = X1), place(id = 2; tokens = Y1), place(id = 3; tokens = Z1)

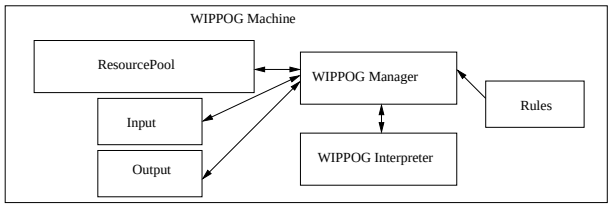

Fig. 3. L'architettura di una WIPPOG machine

IV. WIPPOG MACHINES

Una *WIPPOG machine* (WM) fornisce un ambiente computazionale per eseguire regole WIPPOG. Essa consiste, come mostrato in Figura 3, di un WIPPOG *Interpreter*, che applica una regola WIPPOG alla volta fra quelle presenti nella base di regole *Rule*, coerentemente col ruolo delle diverse componenti e secondo una politica di attivazione gestita dal WIPPOG *Manager*. Il processo agisce sia sul *Resource Pool* che contiene le risorse che descrivono lo stato dell'agente, sia sui compartimenti *Input* e *Output* della WM, che rappresentano le connessioni dell'agente con gli altri agenti. Le prossime sottosezioni dettagliano l'architettura della WM, il cui schema generale è riportato in Figura 3.

Nel seguito useremo la seguente terminologia: $P = \{p_1, \ldots, p_n\}$ indica l'insieme di regole WIPPOG disponibili nella WM. RS indica il multinsieme, costruito su $W(T_P)$, di risorse che descrivono il pool attuale della WM (includendo l'effettivo *Resource Pool* e il compartimento di *Input*), dove T_P è l'insieme dei tipi per le regole in P.

A. L'interprete

L'interprete WIPPOG è responsabile dell'esecuzione delle regole WIPPOG sullo stato corrente del pool di risorse. Esso deve quindi: 1) valutare le pre-condizioni, specificate dalla terna WHEN, GETS e IF; 2) eseguire le computazioni specificate nella componente PROCESSES; 3) valutare le post-condizioni, ovvero produrre le risorse come indicato nelle componenti PRODUCES e OUTS.

L'interprete è implementato da una classe `Interpreter` in cui sono definiti due metodi principali. Il metodo `Result run(Transition tr, ResourcePool pl, ResourcePool in)` tenta di applicare una transizione *tr* ai pool di risorse specificati e restituisce una descrizione del risultato del tentativo di applicazione. Il metodo `boolean applicable(Transition tr, ResourcePool pl, ResourcePool in)` permette di saggiare l'applicabilità di una regola senza modificare i pool di risorse, cioè solo valutando le precondizioni.

Il funzionamento di questi due metodi si basa su un meccanismo di indicizzazione del contenuto della Resource Pool in base al tipo delle risorse. Detto X un generico insieme di risorse, $\forall t \in T$ sia $Rt(X,t)$ l'insieme di tutti gli elementi di X di tipo t. Sia $p \in P$ una generica regola, e siano $TWHEN(p) = \{t_1, \ldots, t_m\} \subset T_P$ l'insieme dei tipi utilizzati nella componente WHEN e $TGETS(r) = \{t_1, \ldots, t_n\}$ l'insieme dei tipi utilizzati nella componente GETS di p. Inoltre $kWHEN(t,p)$ fornisce la cardinalità di occorrenze del tipo t nella componente

WHEN della regola r, e analogamente fa $kGETS(t,p)$ per la componente $GETS$.

Sia RS un pool di risorse (composto dalle risorse interne e da quelle ricevute in ingresso) e $C(RS,k)$ l'insieme delle possibili combinazioni delle risorse in RS a gruppi di k-elementi. Allora, il valore $H_{t,p} = C(Rt(RS_{internal},t), kWHEN(t,p))$ fornisce, $\forall p \in P$, $\forall t \in TWHEN(r)$, le possibili combinazioni di risorse del tipo t presenti nella componente interna della Resource Pool, che possono fornire un match per la parte WHEN della regola p. Analogamente $G_{t,p}$ $=C(Rt(RS_{input},t), kGETS(t,p))$ produrrà le possibili combinazioni di elementi di tipo t da verificare nella componente di input della Resource Pool per trovare un match per la componente GETS della regola p. Il numero di test da effettuare per individuare un match per la regola p, o per verificarne la non applicabilità risulta quindi al più di $H_{t_1,p} \times \ldots \times H_{t_m,p} \times G_{t_1,p} \times \ldots \times G_{t_n,p}$.

Un interprete tenta l'attivazione di una regola ogni volta che gli viene richiesto di farlo da un attivatore. Il linguaggio WIPPOG è stato esteso con una piccola capacità riflessiva, per tenere conto della possibilità da parte dell'interprete di rilevare l'applicabilità di una regola senza effettivamente applicarla. A questo scopo si utilizza una nominazione delle regole nell'insieme P. L'operazione `applicable(ruleName)` può quindi venire utilizzata nella componente IF di una transizione. Si possono quindi definire regole che, verificata l'impossibilità di effettuare la transizione corrispondente a *ruleName*, mettano in atto qualche politica di recupero, ad esempio facendo partire delle azioni di richiesta verso l'utente, o esportando verso l'esterno qualche segnalazione, o addirittura creando delle risorse appropriate.

B. Politiche di attivazione

Il WIPPOG Manager è responsabile dell'attivazione dell'interprete, implementando le politiche di attivazione e realizzando quindi la nozione di *step*. Si possono definire tre politiche di attivazione fondamentali: *sequenziale*, *concorrente* e *massimamente concorrente*.

Per descrivere queste tre politiche consideriamo, $\forall p_i \in P$, l'insieme $pre(p_i) \subset H_{1,p_i} \times \ldots \times H_{m,p_i} \times G_{1,p_i} \times \ldots \times G_{n,p_i}$ di tutte le combinazioni di risorse che soddisfano le precondizioni (componenti WHEN, GETS e IF) di p_i. Sia $RS(P) = \{pre(p_1), \ldots, pre(p_n)\}$ l'insieme di tutti gli insiemi così formati per le regole in P sul resource pool RS. Sia $\mathcal{RS}(P)$ l'insieme di tutti i possibili insiemi $RS(P)$.

Le possibili politiche di applicazione possono essere definite sfruttando le seguenti funzioni:

- **Sequenziale:** $seq : \mathcal{RS}(P) \rightarrow \mathcal{P}(W(T_P))$, dove $seq(RS(P)) = Sel$, con $Sel = rs$, e $\exists i \in \{1,\ldots,n\}$ tale che $rs \in pre(p_i)$. Vale a dire: la politica sequenziale seleziona solo un insieme di risorse che soddisfano le precondizioni di almeno una regola.
- **Concorrenza:** $conc : \mathcal{RS}(P) \rightarrow \mathcal{P}(\mathcal{P}(W(T_P)))$, dove $conc(RS(P)) = Sel$ e $Sel = \{rs_1, \ldots, rs_m\}$, tale che $\forall i \in \{1,\ldots,m\}$ $\exists j \in \{1,\ldots,n\}$ con $rs_i \in pre(p_j)$, e tale che $\forall i,k \in \{1,\ldots,m\}$ $i \neq k \Rightarrow rs_i \cap rs_k = \emptyset$. Vale a dire: la politica concorrente seleziona un insieme di regole non in conflitto che soddisfino le precondizioni di qualche regola.
- **Concorrenza Massimale:** $maxconc : \mathcal{RS}(P) \rightarrow \mathcal{P}(\mathcal{P}(W(T_P)))$, con $Sel = maxconc(RS(P))$, e Sel soddisfa le condizioni per la concorrenza e il vincolo addizionale: $\forall rs \in Sel$, $\forall rc \in (\bigcup_{i=1,\ldots,n} pre(p_i)) \setminus Sel$, $rs \cap rc \neq \emptyset$. Vale a dire: la politica di concorrenza massimale applica tutte le regole possibilmente concorrenti, in maniera tale che qualsiasi altra applicazione entrerebbe in conflitto con le istanze selezionate.

Sia Sel l'insieme degli insiemi di risorse selezionate per l'applicazione dalla politica corrente e $Rls = \{p_1,\ldots,p_m\}$ l'insieme di regole tali che elementi da $pre(p_i)$ appaiono in Sel. Sia $Appl = \{post(p_1),\ldots,post(p_m)\}$ l'insieme degli insiemi di risorse tali che $\forall p \in Rls$ esiste una biiezione tr tra $pre(p)$ e $post(p)$ tale che $\forall rs \in pre(p), tr(rs) \in post(p)$ rappresenta l'istanziazione delle componenti PRODUCES e OUTS di p quando rs è un'istanziazione delle componenti WHEN e GETS di p. Il nuovo pool di risorse NRS costruito su $W(T_P)$ sarà definito da $NRS = (RS \setminus (\bigcup_{p \in Rls} rs \in pre(p))) \cup \bigcup_{p \in Rls} ps \in post(p))$. In altre parole, tutte le risorse che appaiono nell'insieme delle precondizioni per una regola applicata vengono rimosse dal pool, mentre tutte le risorse negli insiemi di postcondizioni da tali regole vengono aggiunte al pool.

L'interprete identifica una possibile applicazione di una regola WIPPOG nel pool di risorse attuale e trasforma il pool secondo la specifica della regola, producendo un nuovo pool di risorse. La politica di attivazione può forzare l'interprete a continuare a lavorare su una versione di *Resource Pool*, in cui la sola modifica ammessa è la rimozione delle risorse consumate, fino a che un passo è considerato completato. A questo punto si aggiungono le risorse prodotte dalle regole applicate e il pool di risorse risultante sostituisce quello precedente. Il completamento di un passo può far scattare qualche specifica attività di altre entità nel sistema complessivo. In particolare, esso attiva lo spostamento dall'*Export* di una macchina al compartimento di *Import* di un'altra. Inoltre, eventuali osservatori esterni sono abilitati a fare riferimento al nuovo contenuto del *Resource Pool*.

Una schematizzazione del comportamento del *WIPPOG Manager* è mostrata nell'algoritmo seguente, in codice pseudo Java, che riassume le caratteristiche delle tre politiche considerate sopra. La classe `WIPPOGManager` mantiene un riferimento a un oggetto `interpreter` di tipo `Interpreter`. L'interprete viene fatto lavorare su una copia locale del *Resource Pool*, così che la concorrenza fra regole non in conflitto è realizzata permettendo l'uso solo delle risorse che non sono state consumate da regole preecedenti. Dopo ogni passo, viene aggiornato il contenuto del vero *Resource Pool* nella *WM*.

```
applyPolicy(Policy pl, ResPool rp, RuleSet rs) {
    boolean stepSucc, applSucc; Result result;
    Transition tr; ResPool rsp, acc; Iterator it; RuleSet failed, usable;
    tr: while (true) {
        rsp = localCopy(rp); usable = localCopy(rs);
        failed = emptySet(); acc = emptySet();
        applSucc = stepSucc = false;
```

```
    stp: while(!stepSucc) {
        usable.removeAll(failed); it = usable.iterator();
        rls: while (it.hasNext() && !stepSucc) {
            tr = (Transition)it.next();
            appl: while(!stepSucc) {
                result = interpreter.run(tr,rsp);
                if (applSucc = result.success()) {
                    rsp.removeAll(result.removed());
                    acc.addAll(result.added());
                    switch(pl) {
                        case Policy.SEQ: {
                            stepSucc = true; continue appl;
                        }
                        case Policy.CNC: {
                            stepSucc = rndBool();
                            if (rndBool()) continue appl;
                            break; }
                        case Policy.MXC: {
                            if (rndBool()) continue appl; break;
                        }
                    }       fine dello switch
                } else failed.add(rl); // fine dell'if
                break;
                // non richiesta nuova applicazione della regola
            } // fine del ciclo appl
        } // end of rls cycle
        stepSucc = !applSucc;
        // nessun'altra applicazione possibile
    } // fine del ciclo stp
    rsp.addAll(acc);
    rp = rsp;
    notifyStepCompletion();
    } // fine del ciclo tr
}
```

Per semplicità non abbiamo distinto fra le componenti *internal* e *input* del *Resource Pool* effettivo, da cui vengono rimosse le risorse menzionate nelle parti WHEN e GETS delle transizioni che hanno avuto successo, né fra le componenti che vengono aggiunte al pool o al compartimento di *Output*. La gestione relativa può facilmente essere ricostruita dal lettore.

Nell'implementazione corrente sono definite tre sottoclassi di `WIPPOG Manager`, una per ogni politica descritta. Esse manterranno quindi solo la parte di `applyPolicy` relativa alla politica da implementare, e non avranno quindi bisogno di una variabile di tipo `Policy`.

C. La comunicazione tra macchine

WM diverse possono comunicare fra loro, formando un sistema di macchine. A questo scopo si utilizzano dichiarazioni di *Import* ed *Export*. Una specifica WIPPOG può dichiarare come `Exports` un tipo di risorse. Quando prodotta nella componente OUTS di una regola, una risorsa di questo tipo sarà inviata alle macchine governate da specifiche che dichiarano quel tipo di risorsa come `Imports`. Tali macchine sono considerate registrate presso la macchina che esporta, e vengono notificate della creazione di tale risorsa.

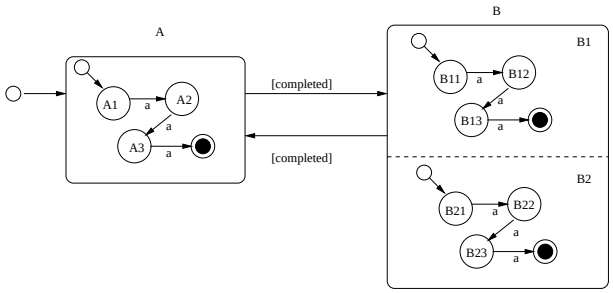

Fig. 4. Un esempio di macchina a stati gerarchica

Formalmente, siano T_1 e T_2 gli insiemi dei tipi per le regole negli insiemi P_1 e P_2 associati con due macchine WM_1 e WM_2. Una macchina WM_i può dichiarare un insieme $I_i \subset T_i$ di tipi come `Import` e un insieme $E_i \subset T_i$ come `Export`. Per $i, j \in \{1, 2\}$, ogni risorsa di tipo $t \in I_i \cap E_j$ dichiarata nella componente OUTS di una regola e prodotta durante una transizione sarà posta nel compartimento *Output* di WM_j. La semantica del meccanismo di comunicazione è tale che essa sarà quindi rimossa e una sua copia collocata nel compartimento *Input* di WM_i. Una regola in P_i che dichiari una risorsa di tipo t nella sua componente GETS sarà in grado di utilizzare tale risorsa, rimuovendola dalla componente *Input*.

Sono state definite due modalità di comunicazione: *broadcast* e *multicast*. Nella modalità broadcast, di default, la semplice dichiarazione di import/export è sufficiente a determinare l'insieme di macchine che devono ricevere la notifica. Nella modalità multicast perché una macchina riceva notifiche, essa deve registrarsi esplicitamente presso le macchine da cui deve ricevere risorse. In questo caso si considerano le dichiarazioni import/export, come nel caso precedente, per attivare le notifiche, ma restringendole alle sole macchine registrate. In ogni momento è possibile variare la modalità di comunicazione.

V. Modellazione di sistemi gerarchici in WIPPOG

L'architettura descritta permette l'associazione di una WIPPOG machine con ogni sistema in un insieme di sistemi cooperativi o concorrenti. In particolare, essa permette la descrizione di macchine a stati gerarchiche, associando una WM a ogni stato composto e vedendo la connessione tra un macrostato e le sue sottomacchine interne come una forma di comunicazione.

Come esempio, consideriamo la macchina a stati descritta in Figura 4. Qui, due macrostati, A e B, definiscono il sistema al livello più elevato. Per semplicità, tutte le sottomacchine sono descritte da automi identici, ognuno composto da tre stati (più gli pseudostati iniziale e finale).

Questo diagramma può essere trasformato in quattro specifiche WIPPOG. Per brevità non diamo i nomi degli attributi e consideriamo solo gli identificatori. La specifica della macchina più esterna è data dall'insieme di regole:

Rule 1

GETS: msg(start)
PRODUCES: current(A)
OUTS: start(A)

Rule 2

WHEN: current(A)
GETS: completed(A)
PRODUCES: current(B)
OUTS: start(B1), start(B2)

Rule 3

WHEN: current(B)
GETS: completed(B1), completed(B2)
PRODUCES: current(A)
OUTS: start(A)

Le tre sottomacchine condividono l'insieme di regole:

Rule 1

WHEN: current(initial)
GETS: input(a)
PRODUCES: current(1)

Rule 2

WHEN: current(1)
GETS: input(a)
PRODUCES: current(2)

Rule 3

WHEN: current(2)
GETS: input(a)
PRODUCES: current(3)

Rule 4

WHEN: current(3)
GETS: input(a)
PRODUCES: current(final)

e differiscono per una coppia di regole che seguono lo schema:

Rule 1

GETS: start(SubMachineName)
PRODUCES: current(initial)

Rule 2

WHEN: current(final)
OUTS: completed(SubMachineName)

Di fatto, la traduzione di Figura 4 in questa specifica segue lo stile della traduzione per automi a stati finiti in Sezione III, con l'aggiunta della produzione e consumo delle risorse esterne $start$ e $completed$ per le sottomacchine.

La specifica per la macchina più elevata dichiarerà i tipi di risorsa msg e $completed$ nella sua sezione di *Import* e il tipo di risorsa $start$ nella sua sezione di *Export*. Le specifiche per le sottomacchine dichiareranno $start$ e $input$ come *Import* e $completed$ come *Export*.

In questo modo, le macchine sono indipendenti dalla loro posizione nella gerarchia, che è invece definita dinamicamente dalle relazioni Import/Export tra le macchine. Questa soluzione evita il collasso della gerarchia in una macchina di "forma normale" [GPP98] e la memorizzazione di connessioni esplicite in parametri esterni come negli Extended Hierarchical Automata [LMM99]. Inoltre, si evita la costruzione di uno specifico interprete per ogni possibile diagramma [BCR00]. Questo approccio dovrebbe migliorare la modularità delle traduzioni e la flessibilità dell'esecuzione.

Senza dare una semantica completa di Statecharts, questo esempio cattura uno dei concetti fondamentali di WIPPOG: quello di separare gli elementi di base di una transizione da aspetti contestuali che riguardino la sua attivazione. In particolare, il completamento della sottomacchina B dipende dal completamento delle sottomacchine $B1$ e $B2$, ma non impone nessuna forma di sincronizzazione fra loro. Una macchina di alto livello non ha accesso privilegiato alle sue sottomacchine, ma comunica con loro attraverso le interfaccie $start$ e $completed$.

VI. Il WIPPOG Browser

L'ambiente di esecuzione di WIPPOG é completamente implementato secondo le specifiche discusse finora. Un suo primo utilizzo é nell'ambito di un sistema di generazione di sentenze visive secondo un approccio guidato dalla sintassi [BCM99]. In questo caso, due macchine WIPPOG sono messe in comunicazione. Una possiede le regole di transizione per un automa che governa l'interazione dell'utente. Lo stato dell'automa determina le interazioni possibili. Le risorse poste in OUTS forniscono a una seconda macchina, incaricata di fare evolvere lo stato della sentenza visiva, un'indicazione su quali regole fare scattare, per produrre nuove entitá grafiche.

Indipendentemente dall'ambito di utilizzo, é stato anche definito un *WIPPOG Browser* che permette di osservare l'evoluzione dei pool di risorse di una macchina WIPPOG, le regole disponibili, le risorse utilizzate per la comunicazione.

Figura 5 mostra la struttura del Browser. La finestra di sinistra riporta la descrizione della specifica della macchina. Il pool delle risorse é vuoto, mentre il pool di input contiene la risorsa *start*. La finestra `Explore Transitions`, in alto a destra mostra la codifica WIPPOG di una regola simile a quella di Figura 1, mentre la finestra in basso a sinistra puó mostrare il log dell'attivitá o le eccezioni sollevate durante l'esecuzione.

Figura 6 mostra la situazione dopo l'avvenuta inizializzazione e nel momento in cui é stata inserita in input una risorsa che descrive l'ingresso "a". La finestra di log mostra che é stata eseguita l'istruzione di inizializzazione e nella finestra di sinistra si osserva che l'automa si trova adesso nello stato 1.

Infine, Figura 7 mostra l'avvenuta esecuzione della regola. Lo stato con identificatore 1 é stato rimpiazzato dallo stato con identificatore 4.

VII. Sviluppi futuri

Nella loro forma attuale, il linguaggio e l'ambiente supportano un insieme sufficientemente vasto di tipi di transizione e di politiche di esecuzione. Si possono prevedere parecchie estensioni. Menzioniamo qui quelle che richiedono solo semplici estensioni dell'interprete.

Transizioni contestuali possono venire realizzate aggiungendo una componente CONTEXT, che elenchi gli elementi che devono esistere perché la transizione occorra, ma che non sono modificati da essa. Gli elementi menzionati in CONTEXT devono esistere, i valori dei loro attributi possono essere usati nelle componenti IF e PROCESSES, ma non possono essere modificati. Quindi gli elementi contestuali non possono apparire in PRODUCES. Durante l'esecuzione, gli elementi in CONTEXT non sono consumati da una regola che li usa nella componente CONTEXT, ma possono essere modificati da regole che li usano in altre componenti. Quindi, una regola che usi CONTEXT definisce uno spazio di applicazione locale, con una copia locale delle risorse contestuali.

Si può fornire supporto alle transazioni aggiungendo una componente TRANS che può contenere assegnamenti o chiamate a metodi. Il comportamento transazionale è responsabilità del metodo chiamato. Solo una transazione può essere associata a una regola. La transazione deve avere successo perché la regola sia applicabile. La transazione viene tentata solo se le componenti WHEN, GETS e IF sono state elaborate con successo.

La lettura di risorse esterne senza il loro consumo può essere specificata con una componente READS. Questa avrebbe lo stesso ruolo di GETS, così che le risorse specificate in READS vanno trovate nel compartimento di Import della WIPPOG machine. Esse non verrebbero però consumate, a differenza del caso GETS, che rende la risorsa non disponibile per altre regole concorrenti.

Queste estensioni rafforzerebbero le possibilità di concorrenza e permetterebbero l'espressione di parallelismo, come negli automi cellulari, in cui ad ogni passo ogni cella è trasformata, fornendo allo stesso tempo contesto per le altre celle.

Un altro importante sviluppo potrebbe essere un pieno supporto all'espressione di condizioni di applicazione negative. Attualmente, queste possono essere gestite parzialmente nel compartimento IF, per specificare valori inaccettabili per attributi di risorse esistenti. Un trattamento più completo richiede un compartimento NEGATIVE che dichiari risorse che non devono essere presenti con i valori descritti in un ulteriore compartimento SUCH_THAT.

Tutte queste modifiche non altererebbero il meccanismo di base e la semantica del nucleo di WIPPOG, basato su rimozione e inserimento di risorse, e possono essere realizzate da estensioni modulari dell'interprete WIPPOG.

Una possibile estensione delle politiche di attivazione è invece rappresentata dall'introduzione di un linguaggio di controllo dell'attivazione delle regole, attraverso l'uso di *espressioni su regole*. In questo caso si tratterebbe di utilizzare la nominazione delle regole, già utilizzata nell'estensione riflessiva del linguaggio vista in precedenza, affidando ad uno specifico tipo di attivatore l'interpretazione di espressioni del tipo **if** `applicable(tr1)` **then** `tr1` **else** `tr2` **end**. Si potrebbero quindi definire delle *unità di trasformazione* sul tipo di quelle introdotte per la trasformazione di grafi [DKKK00]. In una unità di trasformazione si considera un comportamento transazionale, così che le trasformazioni sul pool di risorse sono considerate valide solo se l'intera unità ha avuto successo.

VIII. Conclusioni

Abbiamo presentato il linguaggio eseguibile WIPPOG per la specifica di sistemi a eventi discreti. Esso permette un trattamento uniforme di diversi formalismi di specifica, facilitando l'interconnessione di componenti eterogenee. La comunicazione fra sistemi avviene attraverso la trasmissione di risorse, in forma broadcast, basandosi su un meccanismo di registrazione. Un'implementazione in Java è in funzione presso il Dipartimento di Informatica dell'Università "La Sapienza" di Roma, ed è impiegata per gestire la connessione fra il livello visuale e quello logico di uno strumento interattivo di produzione di sentenze visuali [BCM99].

References

[ACP93] M. Andreoli, P. Ciancarini, and R. Pareschi., *Interaction abstract machines*, Research Directions in Concurrent Object Oriented Programming (G. Agha, A. Yonezawa, and P. Wegner, eds.), MIT Press, Cambridge, Ma, U.S.A., 1993, pp. 257–280.

[And92] J.-M. Andreoli, *Logic programming with focusing proofs in linear logic*, Journal of Logic and Computation **2** (1992), no. 3, 297–347.

[And95] J-M. Andreoli, *Programming in forumtalk*, Tech. Report CT– 003, Rank Xerox Research Centre, Grenoble Lab., France, 1995.

[AP91] J.-M. Andreoli and R. Pareschi, *Communication as Fair Distribution of Knowledge*, Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications, 1991, pp. 212–229.

[BCM99] P. Bottoni, M. F. Costabile, and P. Mussio, *Specification and dialogue control of visual interaction through visual rewriting systems*, ACM TOPLAS **21** (1999), no. 6, 1077–1136.

[BCR00] E. Börger, A. Cavarra, and E. Riccobene, *Modeling the dynamics of UML state machine*, Abstract State Machines: Theory and Applications (Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds.), LNCS, vol. 1912, Springer, 2000, pp. 223–241.

[BMST99] R. Bardohl, M. Minas, A. Schurr, and G. Taentzer, *Application of graph transformation to visual languages*, Handbook of Graph Grammars and Computing by Graph Transformation. Volume II: Applications, Languages and Tools (H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds.), World Scientific, 1999, pp. 105–180.

[DKKK00] F. Drewes, P. Knirsch, H.-J. Kreowski, and S. Kuske, *Graph transformation modules and their composition*, Proc. AGTIVE 1999 (M. Nagl, A. Schürr, and M. Münch, eds.), 2000, pp. 15–30.

[dLV02] J. de Lara and H. L. Vangheluwe, AToM^3*: A tool for multi-formalism and meta-modelling.*, European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE), LNCS, no. 2306, Springer-Verlag, 2002, pp. 174–188.

[EJ01] R. Esser and J. W. Janneck, *Moses - a tool suite for visual modelling of discrete-event systems*, Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, HCC01, 2001, pp. 272–279.

[Erw98] M. Erwig, *Abstract syntax and semantics of visual languages*, Journal of Visual Languages and Computing **9** (1998), no. 5, 461–483.

[Gir95] J.-Y. Girard, *Linear logic: Its syntax and semantics*, Advances in Linear Logic (J.-Y. Girard, Y. Lafont, and L. Regnier, eds.), Cambridge University Press, 1995, Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993, pp. 1–42.

[GPP98] M. Gogolla and F. Parisi-Presicce, *State diagrams in UML: A formal semantics using graph transformations*, Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques (Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, eds.), Technische Universität München, TUM-I9803, 1998.

[KLG93] S. M. Kaplan, J. P. Loyall, and S. K. Goering, *Specifying concurrent languages and systems with delta-grammars*, Research Directions in Concurrent Object-Oriented Programming (G. Agha, P. Wegner, and A. Yonezawa, eds.), MIT Press, London, 1993, pp. 235–256.

[LMM99] D. Latella, I. Majzik, and M. Massink, *Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker*, Formal Aspects of Computing. The International Journal of Formal Methods **11** (1999), no. 6, 637–664.

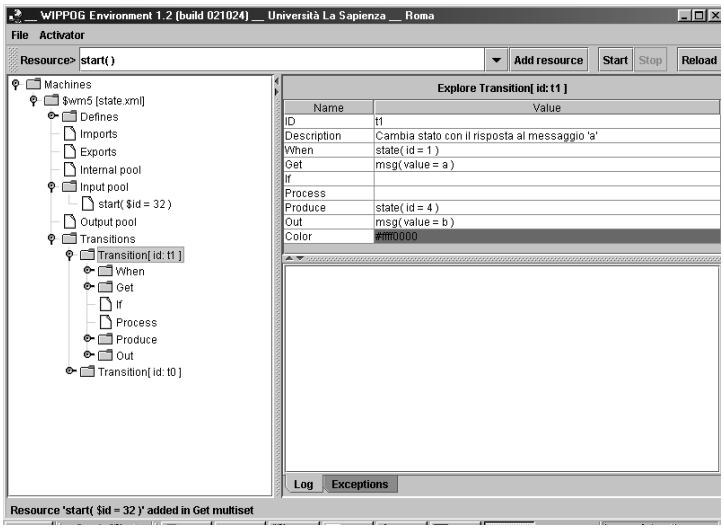

Fig. 5. Il WIPPOG Browser all'inizializzazione dell'automa.

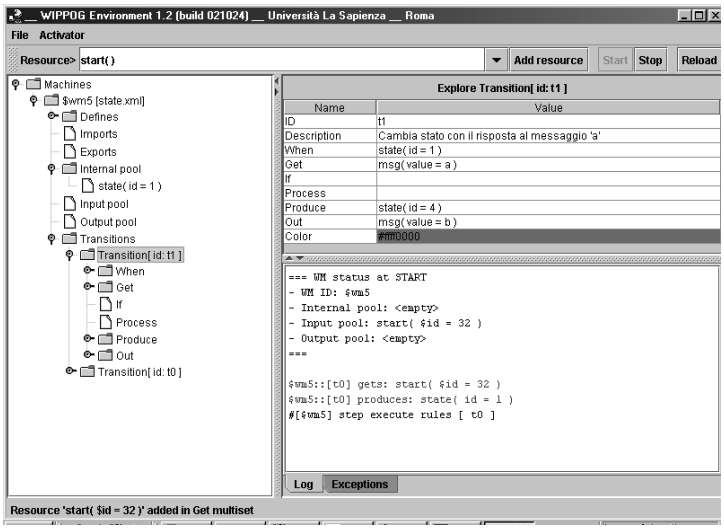

Fig. 6. Prima dell'esecuzione di una regola.

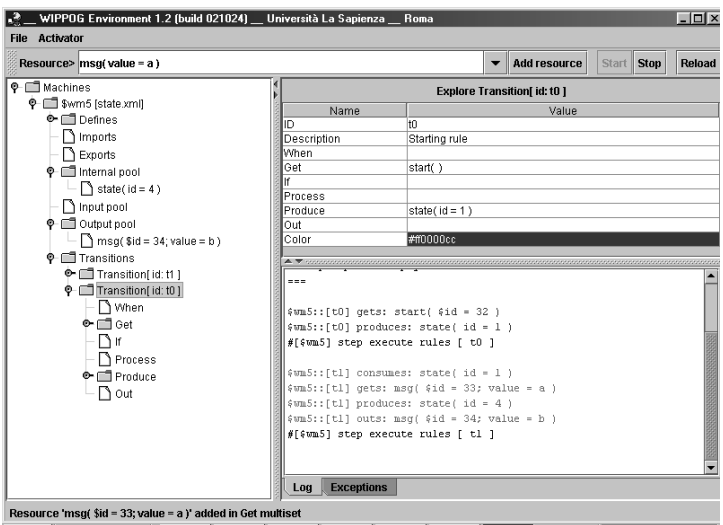

Fig. 7. La nuova configurazione