UNIVERSITA' DI PISA

FACOLTA' DI SCIENZE MATEMATICHE FISICHE E NATURALI

Corso di Laurea Specialistica in Tecnologie Informatiche

Dipartimento di Informatica

TESI DI LAUREA

PROGETTAZIONE E REALIZZAZIONE DI UN PROTOTIPO DI SIMULATORE DI SPECIFICHE DI SISTEMI P2P

Relatori:

Prof: **Antonio BROGI**

Dott: **Razvan POPESCU**

Candidato:

Tommaso CIAMPALINI

ANNO ACCADEMICO 2006-2007

Indice

Capitolo1 Introduzione

1.1 Contesto

I sistemi Peer to Peer (P2P) sono architetture in cui ogni elemento della rete può agire, allo stesso tempo, sia da "utente di servizi" (*client*) che da "fornitore di servizi" (*service provider*). In aggiunta, la maggior parte delle comunicazioni P2P sono basate su infrastrutture non pre-esistenti ma piuttosto su reti dinamiche *ad-hoc* tra i vari peer [10].

I sistemi *Embedded Peer to Peer* (EP2P) [4] rappresentano una nuova svolta nello sviluppo di software per i sistemi distribuiti: questi sistemi sono composti da piccole/grandi unità indipendenti (autonome) eterogenee, magari aventi basso costo e bassi consumi (ad esempio sensori o PDAs, Personal Data Assistant), che comunicano tra loro scambiandosi dati e informazioni tramite l'uso di canali wireless. Inoltre, data la varietà e l'autonomia di ogni singolo elemento costituente l'architettura, si può pensare ad un sistema avente un alto grado di decentralizzazione con una topologia in continua evoluzione nel tempo: questi aspetti rendono possibile l'applicazione di sistemi EP2P in una vasta area di applicazioni che variano dalla telefonia, ai sistemi domestici passando per i sistemi di monitoraggio ambientale [11].

Chiaramente questi aspetti dell'architettura sollevano diversi problemi: l'autonomia dei vari componenti del sistema permette ad ogni singolo elemento di entrare ed uscire dalla rete in modo indipendente, causando frequenti "ri-organizzazioni" del sistema; unità differenti comunicano in modalità differenti (bande differenti con differenti potenze, differenti supporti di comunicazioni - onde radio, infrarossi,…-, differenti protocolli); le applicazioni distribuite possono essere eseguite su unità aventi differenti capacità computazionale (e quindi differenti tempi di completamento).

Il vantaggio di questi sistemi P2P è la possibilità di astrarre da tutti questi problemi usando un appropriato middleware: senza l'uso di questo ultimo (con l'ausilio di strumenti appropriati e metodologie adatte) la definizione, la creazione e il mantenimento di applicazioni distribuite è dispendioso (sia a livello economico che temporale) ed è soggetto

ad errori (che dovrebbero essere affrontati nuovamente da zero in ogni sviluppo di nuove applicazioni distribuite)
Un middleware appropriato dovrebbe nascondere la complessità delle strutture sottostanti e dei relativi problemi (sicurezza, mobilità, ricerca ed uso di risorse) e nello stesso tempo offrire delle interfacce a terze parti per lo sviluppo di applicazioni.

L'obbiettivo del progetto "*Secure Middleware for Embedded Peer-To-Peer Systems* (SMEPP)[1]" [1] è proprio quello di sviluppare un middleware con le caratteristiche descritte sopra, ovvero che sia sicuro, generico e altamente personalizzabile in base ai vari peer (telefoni cellulari, sensori, PDAs) o in base alle varie attività (dal monitoraggio di sistemi critici all'home entertainment o alle comunicazioni). Uno degli obbiettivi di SMEPP è quello di fornire un linguaggio ad alto livello per far interagire le varie unità dell'architettura, astraendo dalle caratteristiche peculiari che ognuna di esse possiede. Inoltre, l'interfaccia SMEPP dovrebbe poter esporre le funzionalità di queste unità hardware a terze parti (altre unità o applicazioni vere e proprie) come servizi/peer SMEPP.

Le primitive offerte da SMEPP (che in sostanza hanno la funzione di aggregare i peer in gruppi al fine di pubblicare/invocare servizi, garantendo anche un certo livello di sicurezza nelle operazioni svolte) vengono strutturate attraverso l'uso del linguaggio di modellazione SMoL (*SMEPP Modelling Language*): il linguaggio SMoL può essere usato per una prima specifica del peer e/o del servizio, la quale può essere sia tradotta in codice eseguibile (per esempio Java), sia ulteriormente migliorata per definire comportamenti più dettagliati. Inoltre questo linguaggio, data la sua semplicità, permette anche l'analisi del comportamento dei peer e dei servizi.

1.2 Problema considerato e obiettivo della tesi

Come in tutti i sistemi, da quelli più semplici a quelli più complessi, durante la progettazione e lo sviluppo delle singole componenti, si rende necessario l'analisi dei comportamenti dei singoli elementi, ovvero controllare che le operazioni svolte dai singoli peer nell'interagire con l'ambiente esterno siano corrette e (soprattutto) siano eseguite nell'ordine corretto. Avendo a che fare con gruppi eterogenei composti da molti peer aventi magari

[1] Per ulteriori informazioni, visitare il sito: http://www.smepp.org .

caratteristiche, servizi e comportamenti molto differenti, si è reso necessario dotarsi di uno strumento, di un tool in grado di analizzare il comportamento di ogni singola unità in modo da poter essere confrontato con gli altri in un determinato contesto per cercare di capirne il funzionamento ed eventualmente i problemi.
Per capire il comportamento di singola unità P2P, l'applicazione dovrebbe poter simulare tutti i possibili *workflow*, sia in assenza che in presenza di fault: solo in questo modo si potrebbe analizzare in maniera soddisfacente il comportamento complessivo dell'entità (peer oppure servizio). Come è possibile intuire, la mole di dati computati dal programma potrebbe essere non indifferente quindi è naturale pensare ad un prototipo in grado di poter salvare questi dati in un formato standard (come, per esempio, *xml*) in modo da poterli riutilizzare (analizzare) in un secondo momento, anche con strumenti software differenti dal tool in questione (cioè con altri programmi con funzionalità totalmente differenti).
Inoltre, per poter usare lo strumento in maniera semplice, dovrebbe essere dotato di una interfaccia grafica snella, in grado di offrire naturalmente un accesso a tutte le funzionalità e in grado di visualizzare, istante per istante, lo *stato* corrente dell'applicazione in maniera chiara, istantanea.
In questa ottica è stato sviluppato un simulatore di *workflows* per il linguaggio di modellazione SMoL: con questa applicazione è possibile simulare tutte le possibili tracce (e quindi i comportamenti) descritte da un file SMoL affinché sia possibile, ancora prima di passare alla parte implementativa vera e propria (risparmiando così una discreta quantità di tempo), accertare la validità di un servizio o meno in presenza di fault, di comportamenti anomali o semplicemente durante una esecuzione normale ed eventualmente correggere queste anomalie.

1.3 Contenuto della tesi

Il simulatore sviluppato, prende in ingresso i sorgenti SMoL (descritti in *xml*) e genera le tracce in base alla semantica del linguaggio. Le modalità di funzionamento sono sostanzialmente due:

- *Manuale* - la traccia viene eseguita passo-passo interagendo con l'utente, il quale indica al simulatore, ogni volta che è necessario, il percorso da seguire (se sollevare o meno fault, quale scelta fare di fronte ad un comando *if*, ecc...);

- *Automatica* - il simulatore procede a calcolare tute le possibili tracce ottenibili dal sorgente in maniera autonoma, senza interagire con l'utente, restituendo alla fine della computazione un resoconto delle operazioni svolte.

Il vantaggio della prima modalità è che un utente è in grado di verificare autonomamente una particolare traccia, di poterla anche modificare (tramite gli appositi comandi), di poterla salvare per poi analizzarla in seguito. Più precisamente, su di una singola traccia, il simulatore è in grado di:

- visualizzare, istante per istante, tutte le istruzioni eseguibili (che compongono lo *stato* corrente della simulazione) e di poter far scegliere all'utente quale eseguire;
- visualizzare tutte le istruzioni eseguite fino ad un preciso istante;
- eseguire una istruzione alla volta, scelta dall'utente nel set di istruzioni eseguibili (*step-by-step*, l'utente indica al programma ogni singola operazione da compiere);
- inserire/disinserire la modalità di esecuzione *Play* in un qualsiasi momento (anche a simulazione già iniziata), che comporta l'esecuzione della singola traccia in automatico (l'interazione con l'utente è ridotta al minimo);
- annullare l'esecuzione delle (*K*) ultime istruzioni, eseguendo una specie di *rollback* della simulazione;
- sollevare fault in qualsiasi momento (compatibilmente con le operazioni eseguibili selezionate);
- caricare e salvare simulazioni in un formato standard (*xml*);
- visualizzare tutte le operazioni svolte dal simulatore in una speciale console (il cui contenuto può essere salvato dentro un file di log);
- visualizzare, oltre alla versione semplificata delle primitive SMoL, anche il codice sorgente *xml* delle stesse (in questo modo è possibile un confronto tra le due versioni);

La seconda modalità invece permette all'utente di analizzare tutti i possibili comportamenti che un'entità può avere, simulando tutte le possibili combinazioni che si possono ottenere da un programma SMoL di partenza (sia in presenza che in assenza di eccezioni): in questo modo viene tolto l'onere all'analizzatore di pensare e di simulare "a mano" le possibili tracce fonti di problemi (si generano infatti anche tutte le tracce ottenibili dai fault).

Tutti i *workflows* simulati in questa seconda modalità (che possono essere anche in grande quantità rispetto a programmi di media/piccola dimensione), vengono salvati in modo che l'utente possa poi, se vuole, analizzarle singolarmente: inoltre, in fase di salvataggio vengono creati dei file *xml* che descrivono le varie tracce in maniera da poter far usare i risultati ottenuti da ulteriori applicazioni (tutte queste opzioni sono settabili all'interno dell'applicazione).

E' importante notare come le due modalità di esecuzione facciano uso di una ben precisa logica di base comune, utilizzino cioè gli stessi identici algoritmi (sebbene in maniera leggermente diversa) che descrivono le operazioni da compiere per simulare la vera e propria esecuzione delle istruzioni del linguaggio: queste procedure (che risultano essere l'essenza vera e propria del simulatore) sono state progettate e realizzate in base alle regole semantiche del linguaggio SMoL, cercando di ottimizzare sia gli stessi algoritmi che le strutture dati usate nel tentativo di offrire un'applicazione compatta ed efficiente.

1.4 Organizzazione del documento

Nei prossimi capitoli verranno descritte le caratteristiche del prototipo, le funzionalità, le problematiche riscontrate (sia a livello di progettazione che di implementazione) con le relative soluzioni adottate per cercare di risolverle, le potenzialità e i limiti del progetto.
Nel secondo capitolo verrà proposta una panoramica di insieme sui linguaggi di riferimento, attraverso la descrizione delle primitive SMEPP e del linguaggio SMoL. Nel terzo capitolo verranno presentate le scelte progettuali e la loro relativa implementazione, con un occhio di riguardo agli algoritmi di *visita* e di *esecuzione* dei comandi operanti sull'albero sintattico SMoL (i pilastri su cui si basa l'intera applicazione). Nel quarto capitolo verranno mostrati alcuni *screenshots* del prototipo al lavoro, nel tentativo prendere familiarità con l'applicazione con esempi pratici; infine, nel quinto capitolo verranno tratte e discusse alcune conclusioni, i limiti e le prospettive future di sviluppo dell'intero lavoro.

Capitolo2 Il linguaggio SMoL

Nel capitolo verranno descritte le primitive del linguaggio SMEPP e la sintassi del suo linguaggio di modellazione SMoL: nella sezione 2.1 introdurremo i motivi che hanno portato allo sviluppo di SMEPP e nella sezione successiva verranno descritte le primitive astratte del linguaggio; nella sezione 2.3 verrà dettagliatamente analizzato il linguaggio SMoL di cui verranno proposti alcuni esempi nella sezione 2.4.

2.1 Breve introduzione a SMEPP

L'analisi dello stato dell'arte dei sistemi EP2P (per esempio [2, 3, 5-8]) rivela il fatto che esistono già molti *frameworks* per lo sviluppo di applicazioni P2P ma che:

- non provvedono ad implementare un modello ad alto livello per rappresentare le interazioni tra servizi che sia semplice e sufficientemente astratto, tale da poter semplificare lo sviluppo di applicazioni P2P,
- non gestiscono in modo ottimale né la sicurezza del sistema, né gli eventi o i fault sollevati dalle unità dello stesso,
- non offrono un linguaggio formale, astratto che possa essere utilizzato per progettare, simulare e verificare il comportamento, le operazioni eseguite da peer e servizi e le loro modalità di interazione.

Per cercare di risolvere queste limitazioni è stato definito il linguaggio SMEPP che è fornito di un insieme di primitive astratte le quali possono essere usate per definire applicazioni P2P in modo semplice ed ad alto livello. L'obbiettivo è quello di delineare una serie di operazioni eseguibili per poi fornirne una implementazione più a basso livello (ovvero delle API - *Application Programming Interface*- dipendenti dal linguaggio di programmazione usato - C#, Java,…-) con le quali realizzare, in maniera tangibile, le vere e proprie applicazioni eseguibili.

Inoltre, il modello SMEPP proposto si basa sui concetti applicati ai moderni Web Service come, per esempio, le sintassi basate su Xml (*eXtensible Markup Language*), le signature

delle operazioni definite dai file WSDL [12] (*Web Server Description Language*), i comportamenti definiti da BPEL [9] (*Business Process Execution Language*) o le annotazioni ontologiche di OWL [13] (*Web Ontology Language*).

2.2 Descrizione informale delle primitive SMEPP

Un programma SMEPP può definire sia il comportamento di un peer che di un servizio. La differenza tra le due entità sta nella possibilità di eseguire o meno alcune operazioni: i peer, infatti, a differenza dei servizi, sono in grado di creare, cercare, unirsi e lasciare dei gruppi di peer e anche di pubblicare servizi (operazioni che ai servizi sono vietate). Conseguentemente, le primitive SMEPP sono di due gruppi: quelle usabili solo dai peer e quelle usabili da entrambe. Del primo gruppo fanno parte: **`newPeer`**, **`createGroup`**, **`joinGroup`**, **`leaveGroup`**, **`publish`**, **`unpublish`**, e **`getIncludingGroups`**. Il resto delle primitive possono essere usate da entrambe le entità.
All'interno del middleware, ogni entità è identificata in modo univoco da un ID.
Di seguito verranno elencate le primitive e ne sarà brevemente descritta la funzionalità: per un maggior dettaglio sui parametri, sui *fault* e sul funzionamento in generale, si rimanda a [16].

2.2.1 NewPeer

`peerId newPeer(credentials) throws exception`

```
Exceptions:
     - invalidCall
```

Crea un nuovo peer SMEPP: **`credentials`** serve per autenticare il nuovo peer.

2.2.2 GetPeerId

`peerId getPeerId(id?)`[2] **`throws exception`**

```
Exceptions:
     - invalidId
```

Restituisce l'identificatore del peer che offre il servizio avente ID **`id`** (se specificato), altrimenti restituisce l'ID del peer chiamante.

2.2.3 GetPeers

[2] Con "**`?`**" si vuole indicare i parametri opzionali.

peerId[] getPeers(groupId) throws exception

Exceptions:
- invalidGroupId
- CallerNotInGroup

Restituisce gli ID dei peer facenti parte del gruppo identificato da **groupId**.

2.2.4 CreateGroup

groupId createGroup(groupDescription) throws exception

Exceptions:
- invalidCall

Crea un gruppo di peer avente le caratteristiche descritte da **groupDescription**: i gruppi SMEPP sono associazioni logiche di peer.

2.2.5 GetGroups

groupId[] getGroups(groupDescription?)

Ottiene una lista di gruppi (esistenti) che soddisfano le caratteristiche descritte da **groupDescription** (se il parametro viene passato), altrimenti restituisce tutti i gruppi.

2.2.6 GetGroupDescription

groupDescription getGroupDescription(groupId) throws exception

Exceptions:
- invalidGroupId

Restituisce le informazioni caratteriali del gruppo di ID **groupId**.

2.2.7 JoinGroup

void joinGroup(groupId, credentials) throws exception

Exceptions:
- accessDenied
- invalidGroupId
- invalidCall

Unisce il peer chiamante al gruppo identificato da **groupId** (se ha in **credentials** i requisiti necessari per essere inserito).

2.2.8 LeaveGroup

void leaveGroup(groupId) throws exception

Exceptions:
- peerNotInGroup
- invalidGroupId
- invalidCall

Permette al peer di uscire dal gruppo **groupId**.

2.2.9 GetIncludingGroups

groupId[] getIncludingGroups() throws exception

Exceptions:
- invalidCall

Restituisce una lista con gli ID dei gruppi di cui il peer chiamante fa parte.

2.2.10 GetPublishingGroups

groupId getPublishingGroup(id?) throws exception

Exceptions:
- invalidCall
- invalidId

In caso di parametro **id** assente, restituisce l'identificatore del gruppo in cui il servizio chiamante è stato pubblicato altrimenti restituisce il gruppo in cui il servizio di identificatore **id** è stato pubblicato.

2.2.11 Publish

<groupServiceId, peerServiceId> publish(groupId, serviceContract) throws exception

Exceptions:
- invalidCall
- invalidService
- invalidGroupId
- invalidId

Il peer chiamante pubblica un servizio che offre all'interno del gruppo, affinché altri peer possano usufruirne;

2.2.12 UnPublish

void unpublish(peerServiceId) throws exception

Exceptions:
- invalidServiceId
- peerNotServiceOwner
- invalidCall

Invocato da un peer quando vuole togliere la possibilità ad altri peer di usufruire del servizio **peerServiceId**, servizio pubblicato in precedenza (tramite la primitiva **publish**).

2.2.13 GetServices

<groupId, groupServiceId, peerServiceId>[] getServices(groupId?, peerId?, serviceContract?, maxResults?, credentials) throws exception

Exceptions:
- invalidGroupId
- invalidPeerId
- invalidService

Restituisce una lista di servizi che soddisfano alcune caratteristiche particolari che vengono delineate in base ai parametri passati alla primitiva.

2.2.14 GetServiceContract

serviceContract getServiceContract(id) throws exception

Exceptions:
- invalidId

Restituisce il contratto del servizio identificato dall'identificatore **id**.

2.2.15 StartSession

sessionId startSession(serviceId) throws exception

Exceptions:
- invalidServiceId
- accessDenied
- cannotStartSession

Le entità chiamano la primitiva per avviare una nuova sessione (paragonabile ad un canale di comunicazione virtuale, con il relativo stato personale): in questo modo è possibile far comunicare i clienti con il servizio in modalità *molti-a-uno* (i clienti *condividono* una unica sessione con il servizio) o *uno-a-uno* (i clienti comunicano *singolamente* con una istanza di un servizio e possono, nello stesso tempo, comunicare con più istanze del solito servizio).

2.2.16 Invoke

output? invoke(entityId, operationName, input?) throws exception

Exceptions:
- invalidServiceId
- invalidPeerId
- invalidOperation
- concurrentRequest
- invalidInputParameter
- invalidOutputParameter
- accessDenied

La primitiva serve per chiamare (invocare) una operazione di una entità identificata da **entityId** (sia chiamante che chiamato devono appartenere al solito gruppo). L'istruzione

può essere di due tipi (in base al suo comportamento): se è di tipo *one-way*, il chiamante non si blocca al momento dell'invocazione dell'operazione mentre se è di tipo *request-response*, il chiamante è bloccato fino a che non riceve il messaggio di risposta (vedi **reply** in seguito).

2.2.17 ReceiveMessage

<callerId, input?> receiveMessage(groupId?, operationName) throws exception

```
Exceptions:
      - invalidGroupId
      - invalidOperation
      - invalidInputParameter
      - callerNotInGroup
```

Permette di ricevere un messaggio da un peer o da un servizio che fanno parte di un gruppo di cui il peer chiamante fa parte. La ricezione di un messaggio può dare il via all'esecuzione di operazioni di tipo *one-way* o *request-response*, in maniera analoga a quanto detto prima per **`invoke`**.

2.2.18 Reply

void reply(callerId, operationName, output?, faultName?) throws exception

```
Exceptions:
      - invalidPeerId
      - invalidOperation
      - missingReceiveMessage
      - invalidServiceId
```

La primitiva segna la terminazione di una operazione che ritorna un risultato all'entità remota che aveva richiesto il servizio.

2.2.19 Event

void event(groupId?, eventName, input?) throws exception

```
Exceptions:
      - invalidGroupId
      - callerNotInGroup
      - invalidEvent
```

Solleva un evento all'interno del gruppo identificato da **`groupId`**.

2.2.20 ReceiveEvent

<callerId, input?> receiveEvent(groupId?, eventName) throws exception

```
Exceptions:
      - invalidGroupId
      - callerNotInGroup
      - invalidInputParameter
```

La primitiva è simile a **`ReceiveMessage`**, riceve un evento sollevato all'interno di un gruppo.

2.2.21 Subscribe

`void subscribe(eventName?, groupId?) throws exception`

```
Exceptions:
      - invalidGroupId
      - callerNotInGroup
```

Le entità usano la primitiva per "iscriversi" alla ricezione dell'evento **`eventName`** (se specificato, altrimenti a tutti gli eventi) all'interno del gruppo **`groupId`** (se specificato, altrimenti in tutti i gruppi di cui il peer fa parte).

2.2.22 UnSubscribe

`void unsubscribe(eventName?, groupId?) throws exception`

```
Exceptions:
      - invalidGroupId
      - callerNotInGroup
      - notSubscribed
```

La primitiva cancella (totalmente o parzialmente) le iscrizioni effettuate in precedenza ad eventi (eseguite con la primitiva **`subscribe`**).

2.3 Descrizione sintassi SMoL

Il linguaggio di modellazione delle primitive SMEPP (*SMEPP modeling language*, SMoL) definisce il comportamento dei peer e dei servizi: è fornito di *comandi elementari* (i quali sono composti sia dalle primitive SMEPP sia da altre primitive definite dal linguaggio stesso) e di *comandi strutturati*, i quali servono ad "orchestrare", a strutturare i comandi elementari prima citati.

Se da una parte il linguaggio può essere usato per definire (parziali) comportamenti dei servizi in maniera astratta, dall'altra la semplicità della semantica del linguaggio dà la possibilità di analizzare il comportamento sia di peer che di servizi ad alto livello.

Nelle seguenti sezioni verranno descritti sia i comandi elementari "aggiunti" che i comandi strutturati di SMoL.

2.3.1 Comandi Elementari

Un *comando elementare* è sia una primitiva SMEPP, sia uno dei comandi definiti di seguito.

2.3.1.1 Empty

`void empty()`

L'esecuzione del comando equivale ad non eseguire nessuna operazione: è utile, per esempio, nei rami di comandi **`if`** oppure per sopprimere dei fault (vedere **`FaultHandler`** in seguito).

2.3.1.2 Wait

`void wait(for?`[3]**`, until?, repeatEvery?)`**

I programmi chiamano **`wait`** per ritardare la loro esecuzione sia di un intervallo di tempo (attraverso il parametro **`for`**), sia fino al raggiungimento di un preciso istante temporale (con il parametro **`until`**). E' importante notare come il comando prenda i parametri **`for`** e **`until`** singolarmente e non insieme. Per esempio, **`wait("2008-06-30")`** significa "aspetta fino al 30 Giugno 2008" mentre **`wait(P3DT10H)`** sta per "aspetta per 3 giorni e 10 ore". Nel caso in cui il parametro **`for`** abbia un valore negativo o uguale a zero, oppure **`until`** faccia riferimento ad una data già passata, il comando **`wait`** termina istantaneamente.

Il parametro **`repeatEvery`** può essere usato solo nel contesto di un **`InformationHandler`** (vedi la definizione di **`InformationHandler`** in seguito) e può essere affiancato da uno degli altri parametri: in caso venga passato senza gli altri parametri esegue il comando ripetutamente nel tempo (ad intervalli temporali costanti) mentre, a seconda del parametro ausiliare passato, aspetta lo scadere del parametro ausiliario (con **`for`** aspetta *per* un determinato periodo, con **`until`** aspetta *fino* a un preciso istante) e poi esegue ripetutamente nel tempo il comando ad intervalli regolari fino a che il corpo principale del comando strutturato non termina.

2.3.1.3 Throw

[3] Con "**?**" si vuole indicare i parametri opzionali.

`void throw(faultName, faultVariable?)`

Il comando **`throw`** solleva esplicitamente il fault **`faultName`** all'interno del programma chiamante. Il campo **`faultVariable`** esprime i dati (opzionali) aggiuntivi del fault. Tali fault sono catturati e gestiti con il comando **`FaultHandler`** (vedi la descrizione successiva).

2.3.1.4 Catch/CatchAll

`faultVariable catch(faultName)`
`<faultName, faultVariable?> catchAll()`

`Catch` serve per catturare e gestire i fault sollevati all'interno del programma chiamante. Può essere usato solo nel contesto di un **`FaultHandler`** (vedere la definizione di seguito): il parametro **`faultName`** indica il nome del fault da gestire e l'output **`faultVariable`** contiene i dati associati al fault catturato. Il comando **`catchAll`** cattura tutti i fault sollevabili: come output restituisce il nome del fault catturato e i dati (opzionali) associati.

2.3.1.5 Exit

`void exit()`

Il comando termina l'esecuzione del programma chiamante. Tutti i comandi che sono in fase di esecuzione vengono immediatamente interrotti.

2.3.2 Comandi Strutturati

I *comandi strutturati* includono uno o più comandi (sia elementari che strutturati). Essi sono descritti di seguito.

2.3.2.1 Assign

```
Assign
   Copy
      From
      To
   End Copy
   . . .
   Copy
      From
      To
   End Copy
End Assign
```

`Assign` definisce uno o più comandi **`Copy`** in cui il valore della variabile, espressione o risultato di una chiamata di primitiva **`From`** è copiato nella variabile di destinazione **`To`**. Il

campo **From** può avere come valore la parola chiave *opaque*: in questo caso si decide di *non* rendere pubblico il valore contenuto in **To**.

Da notare che gli assegnamenti incompatibili sollevano l'eccezioni **invalidAssignment**.

Per semplicità, con la notazione **To = From** verrà inteso:

```
Assign
   Copy
      From
      To
   End Copy
End Assign
```

2.1.2.2 Sequence

```
Sequence
   command [4]
   . . .
   command
End Sequence
```

Sequence implementa un semplice control-flow sequenziale, ovvero esegue le operazioni in ordine lessicale. Quando viene avviata la sequenza, viene eseguito il primo comando della lista, poi il secondo e così via; conseguentemente, **Sequence** termina con l'esecuzione della ultima istruzione della lista.

2.3.2.3 Flow

```
Flow
   command
   . . .
   command
End Flow
```

Il comando **Flow** esegue i comandi contenuti al suo interno in parallelo, mettendoli in concorrenza tra di loro. Il comando strutturato termina quando tutti i comandi al suo interno sono terminati.

2.3.2.4 While

```
While boolCond
   command
End While
```

Il comando **While** esegue i comandi contenuti al suo interno fino a che la condizione booleana **boolCond** rimane vera. Il comando valuta la condizione booleana all'*inizio* di ogni

[4] Con il termine "**command**" si indicano sia *comandi elementari* che *comandi strutturati*.

ciclo. **`While`** termina quando la condizione booleana diventa falsa prima dell'inizio dell'esecuzione del figlio **`command`**.

2.3.2.5 RepeatUntil

```
RepeatUntil boolCond
   command
End RepeatUntil
```

`RepeatUntil` è simile al comando **`While`** in quanto esegue comandi contenuti al suo interno fino a che la condizione booleana **`boolCond`** rimane falsa (e non vera come fa il **`While`**). Inoltre, la condizione booleana viene verificata dal comando alla *fine* di ogni ciclo. **`RepeatUntil`** termina quando la condizione booleana diventa vera dopo la fine dell'esecuzione del figlio **`command`**.

2.3.2.6 If

```
If boolCond
   command
Else
   command
End If
```

Il comando **`If`** esegue una scelta deterministica tra due rami che lui stesso definisce, il ramo **`then`** e il ramo **`else`** (i quali entrambi hanno associati comandi). Il ramo **`else`** è opzionale.

`If` esegue il **`command`** associato al ramo **`then`** se la guardia booleana **`boolCond`** risulta essere soddisfatta, altrimenti esegue il **`command`** associato al ramo **`else`** e termina quando il comando associato al ramo scelto termina.

2.3.2.7 Pick

```
Pick
   <callerId, input?> = receiveMessage(groupId?, operationName)
      command
   . . .
   <callerId, input?> = receiveEvent(groupId?, eventName)
      command
   . . .
   wait(for?, until?)
      command
   . . .
End Pick
```

`Pick` esegue una scelta non deterministica all'interno del programma. L'ambiente (per esempio un cliente che manda un messaggio al chiamante del **`Pick`** oppure un evento di notifica del middleware) decide il **`command`** da eseguire. Il comando strutturato definisce una

serie di **receive** (di messaggi o di eventi) e di **wait** (allarmi): all'interno deve essere presente almeno una guardia. L'esecuzione di **Pick** inizia con la ricezione di un evento o di un messaggio o con lo scadere di un timer: il primo tra questi ultimi a verificarsi determina il comando da eseguire. L'esecuzione complessiva del comando termina quando finisce l'esecuzione del **command** prescelto.

2.3.2.8 InformationHandler

```
InformationHandler
   command

   <callerId, input?> = receiveMessage(groupId?, operationName)
      command
   . . .
   <callerId, input?> = receiveEvent(groupId?, eventName)
      command
   . . .
   wait(for?, until?, repeatEvery?)
      command
   . . .
End InformationHandler
```

L'**InformationHandler** riceve e processa messaggi, eventi ed allarmi per tutta la durata di esecuzione del *comando associato* (con "*comando associato*" viene indicato il comando posto al primo posto all'interno del comando strutturato, prima delle varie **receive** o **wait**): ha un comportamento simile al comando **Pick** ma differisce da questo nel fatto che è in grado di ricevere e processare più messaggi e/o eventi e/o allarmi. Infatti, l'**InformationHandler** è abilitato quando parte l'esecuzione del *comando associato* e finché questo ultimo non è terminato (quindi l'**InformationHandler** disabilitato) è in grado di ricevere e processare più messaggi/eventi/allarmi. Chiaramente, se il *comando associato* è terminato ma esistono dei rami associati alle guardie in esecuzione (che devono essere ancora completati), si deve permettere a questi rami di portare a termine la loro esecuzione.

Una ulteriore differenza con il comando **Pick** è la possibilità di eseguire i comandi guardia **wait** aventi come parametro **repeatEvery** (per ulteriori informazioni vedere la precedente descrizione del comando **wait**).

2.3.2.9 FaultHandler

```
FaultHandler
   command

   faultVariable0? = catch0(faultName)
```

```
      command
   . . .
   faultVariable_N ? = catch_N (faultName)
      command
   . . .
   <faultName, faultVariable?> = catchAll()
      command
   . . .
End FaultHandler
```

In maniera analoga all'**InformationHandler**, il **FaultHandler** ha associato un comando (che risulta essere quello al primo posto al suo interno). Inoltre, il comando strutturato definisce anche dei **catch**es i quali processano i fault (eccezioni) che possono essere sollevati durante l'esecuzione del *comando associato*. Quando un fault è sollevato (in maniera esplicita -con **throw**- oppure implicita -quando l'esecuzione di qualche primitiva non va a buon fine-), l'esecuzione del comando associato è interrotta e i **catch**es vengono valutati (alla ricerca del **catch** appropriato) nell'ordine in cui vengono letti: successivamente viene eseguito il comando associato al primo **catch** che risulta essere adatto a gestire il fault sollevato. Una volta terminata l'esecuzione del ramo **catch** (in caso di fault sollevato) o del *comando associato* (in caso di assenza di fault) al **FaultHandler**, viene eseguito il comando subito successivo all'intero blocco **FaultHandler**. La scelta se un **catch** è adatto o meno a gestire un fault viene basata sul parametro **faultName** (che è obbligatorio). L'output **faultVariable** è opzionale nei **catch**es in quanto permette di avere delle ulteriori informazioni relative al fault sollevato (ma non è detto che i dati siano presenti).

Il **catchAll** cattura tutti i fault sollevabili: come output restituisce il nome del fault catturato ed (eventualmente) **FaultVariable** contenente dei dati associati al fault. Se un **FaultHandler** non è in grado di gestire l'eccezione, allora questa ultima viene sollevata a sua volta "ancora più in alto", alla ricerca di un **FaultHandler** più esterno (ogni programma SMoL ha per definizione un **FaultHandler** di default che contiene tutto il codice, il quale, in caso di fault non gestito, lo propaga nell'ambiente).

2.4 Esempi di programmi SMoL

Verranno mostrati alcuni esempi in cui è usato SMoL per modellare, sia il comportamento dei peer, sia il funzionamento di servizi: l'obbiettivo di questi esempi è quello di dare

un’idea su un possibile uso del linguaggio per modellare le possibili interazioni tra entità (peer/servizi) e non quello di fornire una soluzione ottimale per gli scenari proposti.
Gli esempi vengono riportati sia usando la notazione semplificata dei comandi SMoL (più facile da leggere e da comprendere), sia in codice sorgente xml (il sorgente allegato è stato creato e validato sul file xml schema *SMoL.xsd*, il cui contenuto è riportato nell’Allegato A del presente documento) Inoltre, il codice sorgente inserito è quello strettamente necessario per definire i vari comandi SMoL usati (anche questa scelta è stata fatta per agevolare la lettura).

2.4.1 Monitoraggio della temperatura (I)

Un peer **`TempReaderPeer`** crea un gruppo **`TempReaderGroup`** nel quale pubblica un servizio **`TempReaderService`**: questo servizio definisce una operazione **`getTemp`** senza parametri la quale misura e restituisce la temperatura dell’ambiente. Un altro peer, **`InvokerPeer`**, si unisce al gruppo **`TempReaderGroup`** e invoca l’operazione **`getTemp`** del **`TempReaderService`**. La seguente figura rappresenta schematicamente lo scenario proposto.

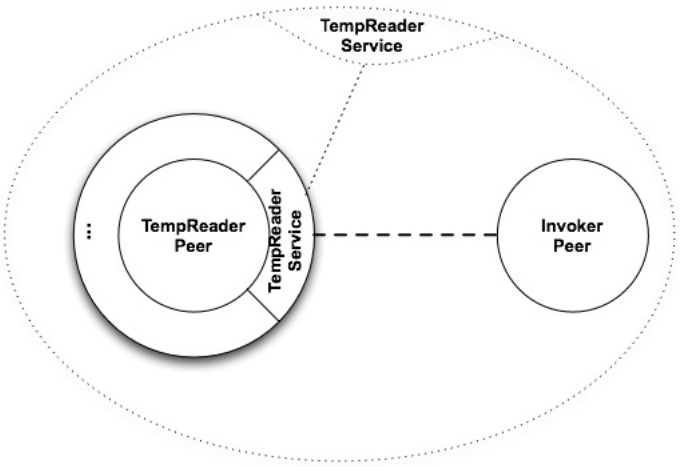

Figura 2.1: Monitoraggio della temperatura (I)

In particolare, questo esempio di interazione tra peer e servizi vuole descrivere:

- come creare peer e gruppi, pubblicare servizi, unirsi a dei gruppi già esistenti;
- come invocare direttamente i servizi offerti dai peer e come funzionano le operazioni che restituiscono dei risultati;

Di seguito verranno descritte le tre entità coinvolte nell'esempio.

Peer **TempReaderPeer** (Descrizione SMoL):

```
Sequence
   myCredentials = opaque
   newPeer(myCredentials)
   myGroupDescription = opaque
   tempGroupId = createGroup(myGroupDescription)
   tempReaderServiceContract = opaque
   publish(tempGroupId, tempReaderServiceContract)
   wait("PT1H") // aspetta un'ora[5]
End Sequence
```

TempReaderPeer per prima cosa si auto-registra come un nuovo peer SMEPP e poi crea il gruppo **TempReaderGroup**. Successivamente, pubblica il servizio **TempReaderService** all'interno del gruppo appena creato e aspetta un'ora prima di terminare. E' fondamentale notare come la terminazione dell'ultima istruzione della lista (in questo caso **wait**) comporti la terminazione del codice del peer e, implicitamente, renda indisponibile il servizio **TempReaderService** pubblicato per ulteriori invocazioni (il middleware penserà in automatico a rimuovere del gruppo la pubblicazione associata al servizio).

Peer **TempReaderPeer** (Codice Sorgente XML):

```
<sequence>

   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="myCredentials"></to>
      </copy>
   </assign>

   <newPeer>
      <input>
         <credentials variable="myCredentials"></credentials>
      </input>
   </newPeer>

   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="myGroupDescription"></to>
      </copy>
   </assign>

   <createGroup>
      <input>
         <groupDescription variable="myGroupDescription"></groupDescription>
      </input>
      <output>
         <groupId variable="tempGroupId"></groupId>
      </output>
   </createGroup>
```

[5] I commenti vengono rappresentati usando "//", una notazione C-like.

```
   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="tempReaderServiceContract"></to>
      </copy>
   </assign>

   <publish>
      <input>
         <groupId variable="tempGroupId"></groupId>
         <serviceContract variable="tempReaderServiceContract"></serviceContract>
      </input>
   </publish>

   <wait>
      <for>
         <literal>PT1H</literal> // aspetta un'ora
      </for>
   </wait>

</sequence>
```

Servizio **`TempReaderService`** (Signature SMoL):

La signature riportata è stata semplificata rispetto a quella che il servizio avrebbe nella realtà per renderla più leggibile (ai fini dell'esempio la semplificazione non influisce):

```
serviceName = "TempReaderService"
serviceType = state-less
request-response operation = temp getTemp()
```

Servizio **`TempReaderService`** (Descrizione SMoL):

```
Sequence
   groupId = getPublishingGroup()
   callerId = receiveMessage(groupId, "getTemp")
   temp = opaque // misurazione temperatura ambientale
   reply(callerId, "getTemp", temp)
End Sequence
```

Il servizio rimane in attesa dell'invocazione dell'operazione **`getTemp`**: successivamente misura la temperatura ambientale e la restituisce al peer che ha invocato il servizio. L'esecuzione del servizio termina subito dopo l'esecuzione dell'istruzione **`reply`**.

Servizio **`TempReaderService`** (Codice Sorgente XML):

```
<sequence>

   <getPublishingGroup>
      <output>
         <groupId variable="groupId"></groupId>
      </output>
   </getPublishingGroup>

   <receiveMessage>
      <input>
         <groupId variable="groupId"></groupId>
         <operationName>
            <literal>getTemp</literal>
```

```
            </operationName>
         </input>
         <output>
            <callerId variable="callerId"></callerId>
         </output>
      </receiveMessage>

      <assign>  // misurazione della temperatura ambientale
         <copy>
            <from opaque="yes"></from>
            <to variable="temp"></to>
         </copy>
      </assign>

      <reply>
         <input>
            <callerId variable="callerId"></callerId>
            <operationName>
               <literal>getTemp</literal>
            </operationName>
            <output variable="temp"></output>
         </input>
      </reply>

</sequence>
```

Peer **`InvokerPeer`** (Descrizione SMoL):

```
Sequence
   myCredentials = opaque
   newPeer(myCredentials)
   desiredGroupDescription = ”TempReaderGroup”
   gid[] = getGroups(desiredGroupDescription)
   tempReaderServiceContract = opaque
   <groupId, tempReaderServiceGroupServiceId,
      tempReaderServicePeerServiceId>[] = getServices(gid[0],
      tempReaderServiceContract, myCredentials)
   joinGroup(groupId[0], myCredentials)
   temp = invoke(tempReaderServicePeerServiceId[0], “getTemp”)
End Sequence
```

Il peer che invoca il servizio per prima cosa si auto-registra come peer SMEPP e successivamente va alla ricerca sia del gruppo **`TempReaderGroup`** (che trova, e ne inserisce un riferimento in **`gid[0]`**) sia del servizio **`TempReaderService`** (che si presuppone sia già stato pubblicato dal peer **`TempReaderPeer`** all'interno del gruppo -anche di questo ultimo inserisce un riferimento in **`tempServicePeerServiceId[0]`**[6]-). Da notare che per semplicità assumiamo che il peer chiamante conosca il contratto (segnature) del servizio. A questo punto, l' **`InvokerPeer`** si unisce al gruppo e in seguito invoca l'operazione **`getTemp`** del servizio: il chiamante viene bloccato finché l'operazione non restituisce il risultato (*invocazione sincrona*). **`InvokerPeer`** termina dopo aver ricevuto il risultato della **`invoke`**.

[6] Con **`tempServicePeerServiceId[0]`** si intende il valore di **`TempServicePeerServiceId`** in **`<groupId, tempServiceGroupServiceId, tempServicePeerServiceId, serviceContract>[0]`**.

Peer **`InvokerPeer`** (Codice Sorgente XML):

```
<sequence>

   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="myCredentials"></to>
      </copy>
   </assign>

   <newPeer>
      <input>
         <credentials variable="myCredentials"></credentials>
      </input>
   </newPeer>

   <assign>
      <copy>
         <from>
            <literal>TempReaderGroup</literal>
         </from>
         <to variable="desiredGroupDescription"></to>
      </copy>
   </assign>

   <getGroups>
      <input>
         <groupDescription variable="desiredGroupDescription"></groupDescription>
      </input>
      <output>
         <groupIdArray variable="gid"></groupIdArray>
      </output>
   </getGroups>

   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="tempReaderServiceContract"></to>
      </copy>
   </assign>

   <getServices>
      <input>
         <groupId variable="gid[0]"></groupId>
         <serviceContract variable="tempReaderServiceContract"></serviceContract>
         <credentials variable="myCredentials"></credentials>
      </input>
      <output>
         <groupId variable="groupId"></groupId>
         <groupServiceId
            variable="tempReaderServiceGroupServiceId"></groupServiceId>
         <peerServiceId variable="tempReaderServicePeerServiceId"></peerServiceId>
      </output>
   </getServices>

   <joinGroup>
      <input>
         <groupId variable="groupId[0]"></groupId>
         <credentials variable="myCredentials"></credentials>
      </input>
   </joinGroup>

   <invoke>
      <input>
         <entityId variable="tempReaderServicePeerServiceId[0]"></entityId>
         <operationName>
```

```
            <literal>getTemp</literal>
         </operationName>
      </input>
      <output>
         <output variable="temp"></output>
      </output>
   </invoke>

</sequence>
```

2.4.2 Monitoraggio della temperatura (II)

I questo esempio verrà considerato un peer **`TempReaderPeer`** che, oltre a creare un gruppo **`TempReaderGroup`** (in maniera analoga a quanto visto nel precedente esempio), periodicamente solleva due eventi - **`temp5s`** e **`temp10s`** - ad intervalli di 5-10 secondi rispettivamente. Oltre a questa unità, verranno considerati altri due peer: **`ClientPeer`**$_1$, che si unisce al gruppo e si mette in ascolto degli eventi **`temp5s`** e **`ClientPeer`**$_2$ il quale, oltre ad unirsi anch'esso al gruppo, pubblica un servizio **`ClientService`**$_2$ (servizio che rimane in ascolto di eventi **`temp10s`**). Entrambi i clienti monitorano la temperatura per un'ora dopo di che essi si rimuovono dal gruppo e terminano. La figura successiva illustra lo scenario proposto.

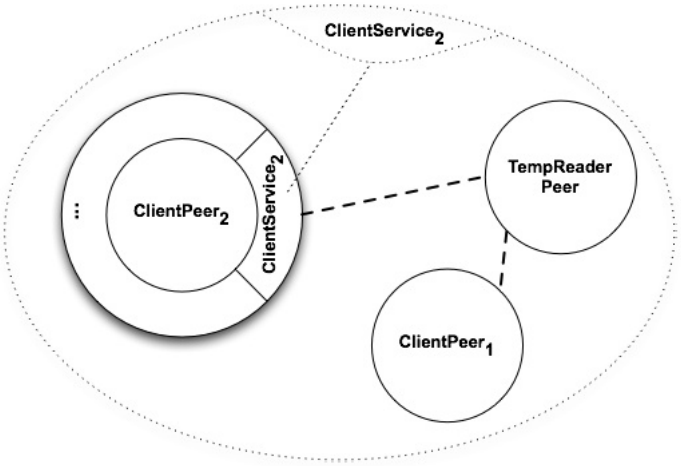

Figura 2.2: Monitoraggio della temperatura (II)

Questo esempio vuole focalizzare l'attenzione su:

- come creare peer e gruppi, pubblicare servizi, unirsi a gruppi già esistenti;
- come sottoscriversi ad eventi, come lanciare e catturare i eventi.

Di seguito verranno descritti i comportamenti delle quattro entità presenti nello scenario.

Peer **TempReaderPeer** (Descrizione SMoL):

```
Sequence
   myCredentials = opaque
   newPeer(myCredentials)
   myGroupDescription = opaque
   tempGroupId = createGroup(myGroupDescription)

   Flow
      While true
         Sequence
            temp5s = opaque
            event(tempGroupId, “temp5s”, temp5s)
            wait(“PT5S”)
         End Sequence
      End While

      While true
         Sequence
            temp10s = opaque
            event(tempGroupId, “temp10s”, temp10s)
            wait(“PT10S”)
         End Sequence
      End While
   End Flow
End Sequence
```

Il peer **TempReaderPeer** per prima cosa registra se stesso come un nuovo peer SMEPP e subito dopo crea un nuovo gruppo **TempReaderGroup**. In seguito, esegue in parallelo i due cicli **while** (che vanno avanti potenzialmente all’infinito) il primo dei quali, dopo aver letto la temperatura, solleva un evento **temp5s** e aspetta 5 secondi mentre il secondo esegue le stesse operazioni del primo con l’unica differenza che solleva un evento **temp10s** e dopo aspetta 10s (infatti, come è già stato anticipato in precedenza, il peer solleva due tipi di eventi, uno ogni 5 secondi e uno ogni 10).

Peer **TempReaderPeer** (Codice Sorgente XML):

```
<sequence>

   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="myCredentials"></to>
      </copy>
   </assign>

   <newPeer>
      <input>
         <credentials variable="myCredentials"></credentials>
      </input>
   </newPeer>
```

```
<assign>
   <copy>
      <from opaque="yes"></from>
      <to variable="myGroupDescription"></to>
   </copy>
</assign>

<createGroup>
   <input>
      <groupDescription variable="myGroupDescription"></groupDescription>
   </input>
   <output>
      <groupId variable="tempGroupId"></groupId>
   </output>
</createGroup>

<flow>

   <while>
      <condition>true</condition>
      <sequence>

         <assign>
            <copy>
               <from opaque="yes"></from>
               <to variable="temp5s"></to>
            </copy>
         </assign>

         <event>
            <input>
               <groupId variable="tempGroupId"></groupId>
               <eventName>
                  <literal>temp5s</literal>
               </eventName>
               <input variable="temp5s"></input>
            </input>
         </event>

         <wait>
            <for>
               <literal>PT5S</literal>
            </for>
         </wait>

      </sequence>

   </while>

   <while>
      <condition>true</condition>
      <sequence>

         <assign>
            <copy>
               <from opaque="yes"></from>
               <to variable="temp10s"></to>
            </copy>
         </assign>

         <event>
            <input>
               <groupId variable="tempGroupId"></groupId>
               <eventName>
                  <literal>temp10s</literal>
               </eventName>
               <input variable="temp10s"></input>
            </input>
         </event>
```

```
            <wait>
               <for>
                  <literal>PT10S</literal>
               </for>
            </wait>

         </sequence>

      </while>

   </flow>

</sequence>
```

Peer **`ClientPeer`** N° 1 (Descrizione SMoL):

```
Sequence
   myCredentials = opaque
   newPeer(myCredentials)
   desiredGroupDescription = ”TempReaderGroup”
   gid[] = getGroups(desiredGroupDescription)
   join(gid[0], myCredentials)
   subscribe(“temp5s”, gid[0])
   InformationHandler
      Sequence
         wait(“PT1H”)
         unsubscribe()
      End Sequence

      <cId, temp> = receiveEvent(gid[0], “temp5s”)
         // Usa la temperatura “temp” per qualcosa
   End InformationHandler
End Sequence
```

`ClientPeer`$_1$ registra se stesso come un nuovo peer SMEPP dopo di che ottiene l’identificatore del gruppo **`TempReaderGroup`** e si unisce al gruppo. Una volta unitosi si sottoscrive all’evento **`temp5s`** che viene sollevato all’interno del gruppo. Infine si mette in attesa per un’ora dopo la quale toglie la propria sottoscrizione da un qualsiasi evento del gruppo (al quale si era precedentemente sottoscritto). Durante questa ora di attesa, il peer riceve gli eventi **`temp5s`** ed è in grado di elaborarli in qualche modo.

Peer **`ClientPeer`** N° 1 (Codice Sorgente XML):

```
<sequence>

   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="myCredentials"></to>
      </copy>
   </assign>

   <newPeer>
      <input>
         <credentials variable="myCredentials"></credentials>
      </input>
   </newPeer>
```

```
    <assign>
       <copy>
          <from>
             <literal>TempReaderGroup</literal>
          </from>
          <to variable="desiredGroupDescription"></to>
       </copy>
    </assign>

    <getGroups>
       <input>
          <groupDescription variable="desiredGroupDescription"></groupDescription>
       </input>
       <output>
          <groupIdArray variable="gid"></groupIdArray>
       </output>
    </getGroups>

    <joinGroup>
       <input>
          <groupId variable="gid[0]"></groupId>
          <credentials variable="myCredentials"></credentials>
       </input>
    </joinGroup>

    <subscribe>
       <input>
          <eventName>
             <literal>temp5s</literal>
          </eventName>
          <groupId variable="gid[0]"></groupId>
       </input>
    </subscribe>

    <informationHandler>

       <sequence>

          <wait>
             <for>
                <literal>PT1H</literal>
             </for>
          </wait>

          <unsubscribe></unsubscribe>

       </sequence>

       <onEvent>
          <sequence>
             // usa la temperatura “temp” per qualcosa
          </sequence>
          <receiveEvent>
             <input>
                <groupId variable="gid[0]"></groupId>
                <eventName variable="temp5s"></eventName>
             </input>
             <output>
                <callerId variable="cId"></callerId>
                <input variable="temp"></input>
             </output>
          </receiveEvent>
       </onEvent>

    </informationHandler>

 </sequence>
```

Peer **ClientPeer** N° 2 (Descrizione SMoL):

```
Sequence
   myCredentials = opaque
   newPeer(myCredentials)
   desiredGroupDescription = ”TempReaderGroup”
   gid[] = getGroups(desiredGroupDescription)
   join(gid[0], myCredentials)
   clientService2Contract = opaque
   <gsid, psid> = publish(gid[0], clientService2Contract)
   invoke(psid, “monitor”, gid[0])
EndSequence
```

In maniera analoga a quanto fatto dal **ClientPeer**$_1$, **ClientPeer**$_2$ si auto-registra come nuovo peer SMEPP e ottiene l'identificativo del gruppo **TempReaderGroup** al quale si unisce. Successivamente pubblica il servizio **ClientService**$_2$ (vedi la descrizione in seguito): il **ClientPeer**$_2$ avvia lui stesso l'esecuzione del servizio (passandogli l'identificativo del gruppo **TempReaderGroup** ed invocando l'operazione **monitor** del servizio stesso) e, prima di terminare, aspetta che il servizio concluda le operazioni che deve svolgere .

Peer **ClientPeer** N° 2 (Codice Sorgente XML):

```
<sequence>

   <assign>
      <copy>
         <from opaque="yes"></from>
         <to variable="myCredentials"></to>
      </copy>
   </assign>

   <newPeer>
      <input>
         <credentials variable="myCredentials"></credentials>
      </input>
   </newPeer>

   <assign>
      <copy>
         <from>
            <literal>TempReaderGroup</literal>
         </from>
         <to variable="desiredGroupDescription"></to>
      </copy>
   </assign>

   <getGroups>
      <input>
         <groupDescription variable="desiredGroupDescription"></groupDescription>
      </input>
      <output>
         <groupIdArray variable="gid"></groupIdArray>
      </output>
   </getGroups>

   <joinGroup>
      <input>
```

```
            <groupId variable="gid[0]"></groupId>
            <credentials variable="myCredentials"></credentials>
         </input>
      </joinGroup>

      <assign>
         <copy>
            <from opaque="yes"></from>
            <to variable="clientService2Contract"></to>
         </copy>
      </assign>

      <publish>
         <input>
            <groupId variable="gid[0]"></groupId>
            <serviceContract variable="clientService2Contract"></serviceContract>
         </input>
         <output>
            <groupServiceId variable="gsid"></groupServiceId>
            <peerServiceId variable="psid"></peerServiceId>
         </output>
      </publish>

      <invoke>
         <input>
            <entityId variable="psid"></entityId>
            <operationName>
               <literal>monitor</literal>
            </operationName>
            <input variable="gid[0]"></input>
         </input>
      </invoke>

   </sequence>
```

Servizio **`ClientService`** N° 2 (Signature SMoL):

```
serviceName = “ClientService₂”
serviceType = session-less
request-response operation = monitor(groupId)
```

Servizio **`ClientService`** N° 2 (Descrizione SMoL):

```
Sequence
   <ownerPeer, gid> = receiveMessage(“monitor”)
     // alternativamente avremmo potuto usare getPublishingGroup()
     // invece di passare il parametro gid al livello dell’applicazioni
   subscribe(“temp10s”, gid)
   InformationHandler
      Sequence
         wait(“PT1H”)
         unsubscribe(“temp10s”, gid)
      End Sequence

      <cId, temp> = receiveEvent(gid, “temp10s”)
       // Usa la temperatura “temp” per qualcosa
   End InformationHandler
   reply(ownerPeer, “monitor”)
End Sequence
```

`ClientService`$_2$ aspetta per prima cosa di ricevere l’identificatore del **`TempReaderGroup`**: dopo di che sottoscrive alla ricezione degli eventi **`temp10s`** sollevati

nel gruppo. Poi, in modo analogo al **`ClientPeer`**$_1$ (da notare che ora è il *servizio* ad effettuare determinate operazioni e non più il *peer*), aspetta un'ora durante la quale rimane in attesa di eventuali eventi **`temp10s`**. Alla fine, dopo aver rimosso la propria sottoscrizione (inserita in precedenza), segnala la terminazione del periodo di monitoraggio al peer **`ClientPeer`**$_2$.

Servizio **`ClientService`** N° 2 (Codice Sorgente XML):

```
<sequence>

   <receiveMessage>
      <input>
         <operationName>
            <literal>monitor</literal>
         </operationName>
      </input>
      <output>
         <callerId variable="ownerPeer"></callerId>
         <input variable="gid"></input>
      </output>
   </receiveMessage>

   <subscribe>
      <input>
         <eventName>
            <literal>temp10s</literal>
         </eventName>
         <groupId variable="gid"></groupId>
      </input>
   </subscribe>

   <informationHandler>

      <sequence>

         <wait>
            <for>
               <literal>PT1H</literal>
            </for>
         </wait>

         <unsubscribe>
            <input>
               <eventName>
                  <literal>temp10s</literal>
               </eventName>
               <groupId variable="gid"></groupId>
            </input>
         </unsubscribe>

      </sequence>

      <onEvent>
         <sequence>
            // usa la temperatura “temp” per qualcosa
         </sequence>
         <receiveEvent>
            <input>
               <groupId variable="gid[0]"></groupId>
               <eventName variable="temp10s"></eventName>
            </input>
            <output>
               <callerId variable="cId"></callerId>
```

```
                <input variable="temp"></input>
             </output>
          </receiveEvent>
       </onEvent>

    </informationHandler>

    <reply>
       <input>
          <callerId variable="ownerPeer"></callerId>
          <operationName>
             <literal>monitor</literal>
          </operationName>
       </input>
    </reply>

</sequence>
```

Capitolo3 Progettazione e realizzazione del simulatore

Nel capitolo verrà illustrata tutta la parte di progettazione e di implementazione del prototipo di simulatore e verranno descritte tutte le soluzioni adottate in fase di realizzazione, evidenziando, per ognuna di esse, gli aspetti chiave: nella sezione 3.1 verranno descritti gli obiettivi per cui si è voluto implementare il prototipo; nella sezione 3.2 verranno illustrate le scelte fatte in fase di progettazione mentre, nella sezione 3.3, quelle fatte in fase di implementazione (compresa la descrizione dettagliata degli algoritmi di *visita* e di *esecuzione* delle primitive dei programmi SMoL).

3.1 Obiettivi del simulatore

Dovendo progettare e realizzare un simulatore, si è resa necessaria una attenta analisi dei requisiti che questo ultimo doveva avere, per cercare di delineare le caratteristiche fondamentali dell'applicazione: si è cercato infatti di sviluppare uno strumento il più completo e flessibile possibile, chiaramente cercando di dare maggiore enfasi agli aspetti che si sono ritenuti più importanti e più utili nella operazione di simulazione complessiva di tutte le tracce possibili.
Senza dubbio, in primo luogo, si è cercato di facilitare il più possibile l'utilizzo dello strumento, semplificando al massimo l'aspetto grafico e fornendo l'accesso alle varie funzionalità offerte in maniera semplice ed immediata: inoltre, per agevolare l'utente nel processo di analisi, si è voluto dotare l'interfaccia grafica della possibilità di visualizzare, durante l'esecuzione di una traccia[7], tutte le operazioni svolte dall'applicativo e lo stato complessivo della simulazione.
In secondo luogo, si è voluto dare all'utente la possibilità di modificare a suo piacimento la traccia simulata ovvero far sì che l'utilizzatore possa

- eseguire una traccia in automatico oppure passo per passo,

[7] *Comportamento*, *traccia* e *simulazione* vengono intesi come sinonimi.

- poter annullare le ultime *K* istruzioni eseguite,
- poter scegliere le modalità di esecuzioni di un *if* o di un ciclo *while*/*repeatUntil*,
- sollevare in un qualsiasi momento un fault,
- salvare/caricare una singola simulazione

oppure possa direttamente far eseguire tutte le tracce possibili al simulatore in automatico, aspettandosi come *output* un resoconto alla fine dell'operazione.

3.2 Progettazione del simulatore

Durante la fase di progettazione, è venuto naturale pensare fin da subito all'applicazione come composizione di tre moduli fondamentali, nel tentativo di separare le altrettante macro-operazioni svolte dal sistema (come mostrato in figura 3.1): un *parser* (in grado di estrarre da un testo xml una struttura dati adatta alle necessità), un *engine* (che esegua materialmente le simulazioni) e un'*interfaccia grafica* (che sia il *trait d'union* tra l'utente e il simulatore).

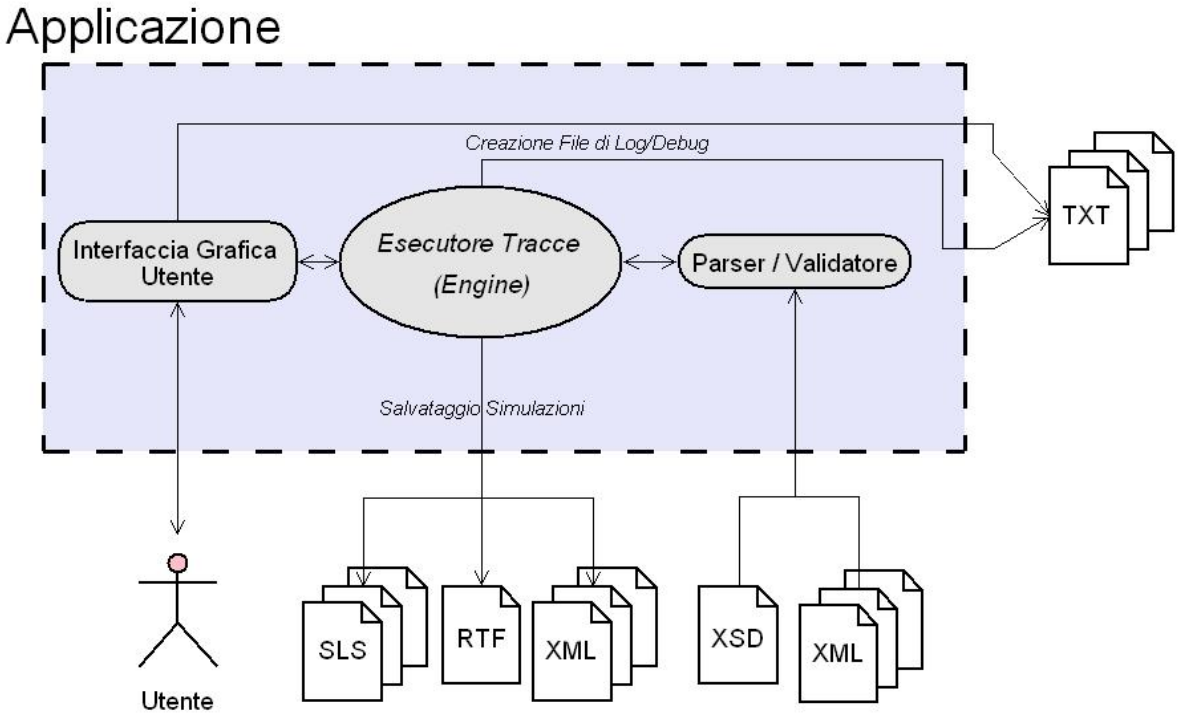

Figura 3.1: Schema Generale Applicazione

Lo schema sovrastante serve per meglio focalizzare sia la modalità di funzionamento, sia la modularità dell'applicazione. Questa ultima caratteristica si è resa necessaria, oltre che per una maggiore pulizia del codice e per una facile manutenzione, dal fatto che si deve essere in grado di apportare in futuro modifiche al codice nel momento in cui venga modificato la sintassi del linguaggio SMoL (ad esempio, una modifica sintattica come l'aggiunta di un comando semplice o la rimozione di un parametro ad una primitiva) o, in generale, apportare modifiche parziali al sistema in caso di bisogno: questo tipo di approccio permette di modificare solo le parti necessarie lasciando intatte le altre (chiaramente dipende dalla modifica da apportare, se si va a modificare *semanticamente* -e non solo *sintatticamente*- il comportamento di un comando strutturato, oltre al *parser* anche l'*esecutore* dovrà essere modificato).
Per ognuno di questi moduli[8] (implementati in C#, usando il FrameWork 2.0 e Microsoft Visual Studio .NET [14]), in fase di compilazione verrà creata una dll (*Dynamic Link Library*) a sé stante (tranne, chiaramente, per il progetto di *StartUp* per il quale verrà creato un file eseguibile - .EXE -). Questa scelta è dovuta in parte al concetto di modularità sopra descritto (poter sostituire singole parti con altre simili ma più aggiornate), in parte dal fatto di voler permettere il riuso del codice anche da programmi scritti non in C# (infatti, essendo codice compilato, tramite le opportune API -*Application Programming Interface*-, può essere usato da programmi scritti in linguaggi differenti).

3.2.1 Il Parser

Il *parser* è la componente del prototipo che si fa carico di leggere e di validare i file xml SMoL in base al file XML Schema *SMoL.xsd* (riportato nell'Allegato A del presente documento) che definisce il linguaggio. Una volta letto e validato, il *parser* crea una struttura dati, e più precisamente un albero sintattico, rappresentante il programma SMoL analizzato: questa struttura, che viene successivamente passata all'*esecutore tracce*, risulta essere la struttura base su cui poi simulare tutte le tracce possibili. Nel caso in cui il file che si vuole caricare risulti essere non valido (per vari motivi, errori sintattici, comandi non riconosciuti, file non SMoL,…), il *parser* solleverà delle eccezioni che segnaleranno i problemi riscontrati.

[8] *Componente*, *modulo* e *progetto* vengono usati come sinonimi.

L'albero sintattico creato dal *parser* a partire dal file *xml* letto è composto da istanze della classe *Nodo*: ogni istanza della classe contiene le informazioni necessarie a rappresentare ogni singolo comando di SMoL. La modalità con cui vengono inserite le istanze all'interno dell'albero (e conseguentemente la forma risultante dell'albero stesso) risulta essere dettata dalla semantica del linguaggio: ogni comando *strutturato* viene rappresentato da una particolare combinazione di nodi, posti in relazione tra loro in un modo ben preciso (l'argomento verrà ripreso in seguito) mentre le *primitive* sono rappresentate sulle foglie dell'albero.
Il perché si usi la classe *Nodo* e non direttamente la classe *XMLNode* che il C# fornisce è presto detto: la classe *XMLNode* fornisce una descrizione sintattica fin troppo precisa dell'albero SMoL e la fornisce ad un livello di astrazione troppo basso; invece la classe *Nodo* è più "astratta", più semplice rispetto alla classe prima citata, è composta da informazioni accessibili in maniera diretta che rendono gli algoritmi di visita e di esecuzione più semplici e efficienti possibile, nasconde le informazioni non utili ai fini realizzativi dell'applicazione. Inoltre, ogni istanza *Nodo* contiene un riferimento al *XMLNode* a cui si riferisce (in maniera da poter accedere direttamente al codice, se necessario).

Nella implementazione del *parser* si è fatto uso di package generati in automatico -in base al file schema *SMoL.xsd*- dal programma Altova XMLSpy 2008 [15], i quali permettono un parsing un po' più snello e veloce, togliendo al programmatore parte del lavoro "ripetitivo" che lo sviluppo di un *parser* comporterebbe. Questi package sono inseriti all'interno del progetto e vengono compilati, insieme alle altre due classi, nella libreria *ParserSMoL.dll*.

3.2.2 L'Esecutore Tracce (Engine)

L'*esecutore tracce* (o *engine*) è la parte fondamentale dell'intera applicazione: sull'albero sintattico ricevuto dal *parser* è in grado di simulare tutti i possibili comportamenti del peer/servizio (comportamento basato sulla semantica del linguaggio SMoL ed in particolare dei suoi comandi *strutturati*).

Il funzionamento del componente è molto semplice: quando simula una esecuzione di una entità, la traccia che ottiene è vista come un'insieme di *stati* S_0, S_1,…, S_K ognuno dei quali è composto da un gruppo di istruzioni (*primitive*) potenzialmente eseguibili, tranne l'ultimo

stato S_K che non contiene nessuna istruzione (questa è la condizione che segnala la terminazione di una traccia).

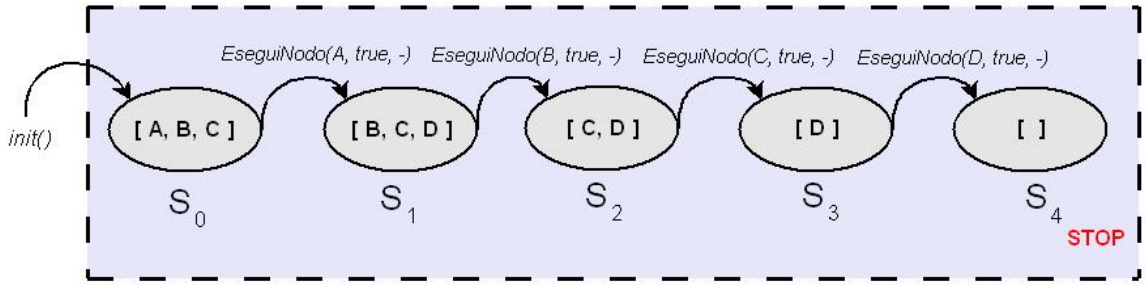

Figura 3.2: Esempio di esecuzione di una traccia

Il passaggio da uno *stato* all'altro è eseguito da una funzione di transizione, chiamata "*EseguiNodo*", che preso in ingresso lo stato corrente S_J, una istruzione $I \in S_J$, un booleano indicante se I deve o no sollevare un fault ed (opzionalmente) una stringa contenente il nome del fault da sollevare, restituisce un nuovo stato S_{J+1}: solamente il primo stato S_0 non è ottenuto come risultato della funzione di transizione ma come risultato di una operazione di inizializzazione chiamata "*init*" (vedi l'esempio mostrato in Figura 3.2).

La funzione *init* (che inizializza lo stato S_0) esegue una visita "*in profondità*" dell'albero sintattico: tutte le foglie (*primitive*) che raggiunge sono le istruzioni che vanno a comporre lo stato iniziale. La visita, nonostante si sviluppi dalla radice alle foglie, non è classificabile semplicemente come DFS (*Depth First Search*) in quanto, ogni nodo interno all'albero (che rappresenta un comando *strutturato* SMoL) ha delle modalità di visita ben precise, che differiscono tra nodo interno e nodo interno e che verranno descritte nel paragrafo 3.3.4.

La funzione di transizione *EseguiNodo* esegue, *in linea di massima*, due operazioni: da principio segna come eseguita la *foglia* che rappresenta il comando selezionato per l'esecuzione; successivamente si sposta sul *padre* per vedere se questo nodo ha dei figli direttamente connessi non ancora eseguiti: in caso affermativo, effettua una visita (se i nodi rappresentano comandi *strutturati*) su questi figli rimanenti (lo stesso tipo di visita fatta dalla funzione *init*) altrimenti si sposta ancora sul *padre* (sul *nonno* della foglia eseguita) e ripropone lo stesso controllo. La descrizione appena proposta rappresenta l'idea sostanziale che sta dietro all'algoritmo vero e proprio: chiaramente, le operazioni compiute da

EseguiNodo variano in base ai nodi interni su cui si va ad operare: per esempio, quando abbiamo a che fare con un nodo che rappresenta un *FaultHandler*, nel momento in cui ci si sposta sul nodo perché il primo dei suoi figli (che contiene il blocco istruzioni associato al comando *strutturato*) risulta essere stato eseguito in maniera corretta (cioè senza aver sollevato fault), i nodi rimanenti (che rappresentano i vari *catch* associati) del *FaultHandler* non vengono nemmeno considerati; un altro esempio può essere l'*InformationHandler* il quale, a seconda dei figli eseguiti, può portare alla duplicazioni di interi rami dell'albero.
Tutte queste possibilità verranno descritte approfonditamente nella descrizione dell'algoritmo riportata nella sezione 3.3.4

L'*engine* mantiene traccia di tutte le primitive SMoL che simula salvandole in uno *stack*: in particolare, sullo *stack* vengono salvate triple di dati che rappresentano i parametri passati alla funzione di transizione ogni volta che è invocata (ovvero l'*istruzione*, l'*esito* e il *nome del fault*). E' importante notare che nei dati salvati sullo *stack* non compaia lo stato S_J da cui partire per calcolare il nuovo S_{J+1}: si è fatta questa scelta perché sarebbe inutile tenersi copia di tutti gli stati (e oltre tutto potrebbe richiedere anche molto spazio fisico in memoria) in quanto il generico stato S_{J+1} può essere visto come il risultato di *J* esecuzioni consecutive della funzione di transizione a partire dallo stato iniziale S_0.
Per esempio, uno stato S_2 generico può essere visto come

$$S_2 = EseguiNodo(EseguiNodo(init(), I_0, esito_0, faultName_0), I_1, esito_1, faultName_1)$$

Inoltre, mantenere uno *stack* composto da strutture dati semplici, agevola la fase di salvataggio e di caricamento delle simulazioni (che verranno descritte in seguito).

Per cercare di spiegare al meglio il funzionamento degli algoritmi, nelle seguenti figure viene riportato un' applicazione pratica: l'albero riportato rappresenta un programma SMoL generico che non esegue nessuna operazione in particolare, in cui le primitive vengono rappresentate con le foglie tonde, senza precisare quale primitiva sia in quanto ai fini descrittivi sarebbero informazioni inutili: al suo interno sono presenti gran parte dei comandi strutturati che compongono il linguaggio.

Durante la simulazione di esempio verranno usate diverse convenzioni grafiche: le foglie (o comandi *strutturati*) non ancora visitate saranno colorate di *grigio*, le foglie eseguibili appartenenti ai vari *stati* saranno evidenziate con il *verde*, i nodi eseguiti correttamente

marcati con il *rosa* e le primitive la cui esecuzione genera un fault con il *rosso*. In tutti quei casi in cui compare una *croce* sul nodo significa che il nodo o non è stato visitato e, arrivati ad un certo punto della simulazione, non ha più senso visitarlo (e quindi viene scartato) o che è stato visitato ma è stato scartato ugualmente senza essere eseguito. Infine, con una *freccia* rossa viene indicata l'istruzione scelta per la successiva esecuzione.

Inoltre, per motivi di spazio all'interno dello schema, verranno adottate delle abbreviazioni: *IH* per *InformationHandler*, *FH* per *FaultHandler*, *Seq* per *Sequence* e *RepUnt* per *RepeatUntil* (queste abbreviazioni verranno usate anche in altri parti del capitolo).

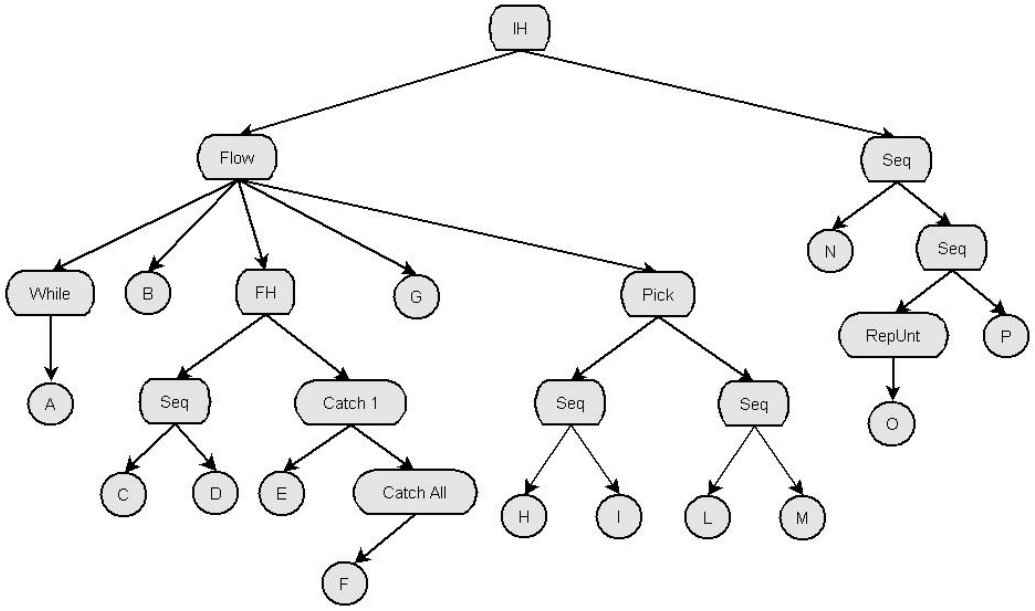

Figura 3.3a: Albero non inizializzato

In figura 3.3a viene mostrato un generico albero SMoL, prima ancora di non essere inizializzato. Su questo albero viene invocata la funzione *init* (che in pratica esegue una *visita* della struttura dati).

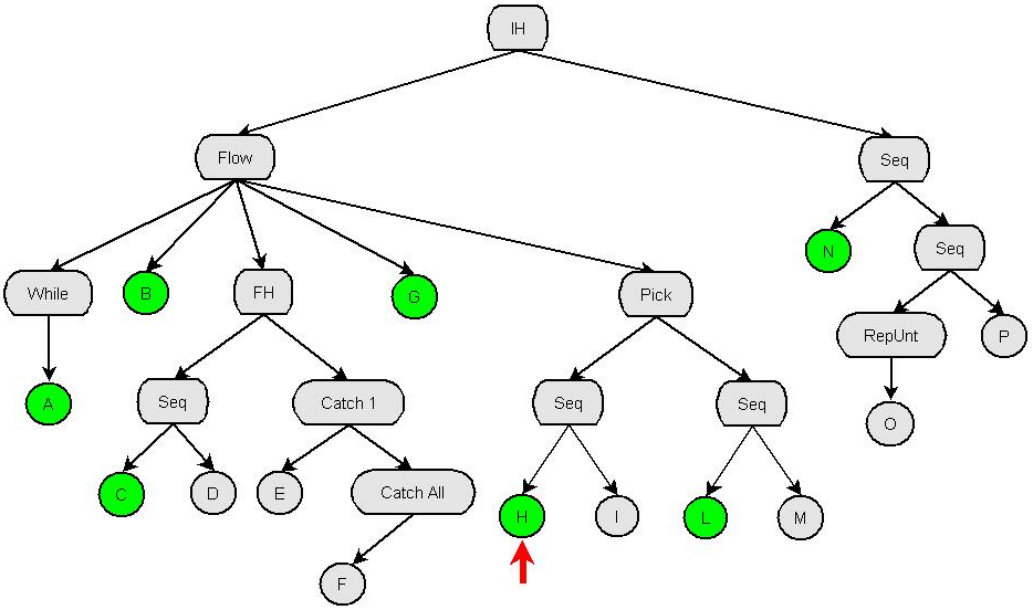

Figura 3.3b: Albero inizializzato (Stato S_0)

Il risultato di *init* (più precisamente della *visita* fatta dalla procedura di *init*) è lo stato S_0 = { A, B, C, G, H, L, N } (sono le foglie verdi), in cui viene selezionata per l'esecuzione la primitiva H[9] (scelta evidenziata in figura 3.3b con la freccia rossa). A questo punto viene invocata la funzione *EseguiNodo(H, true,-)* nello stato S_0.

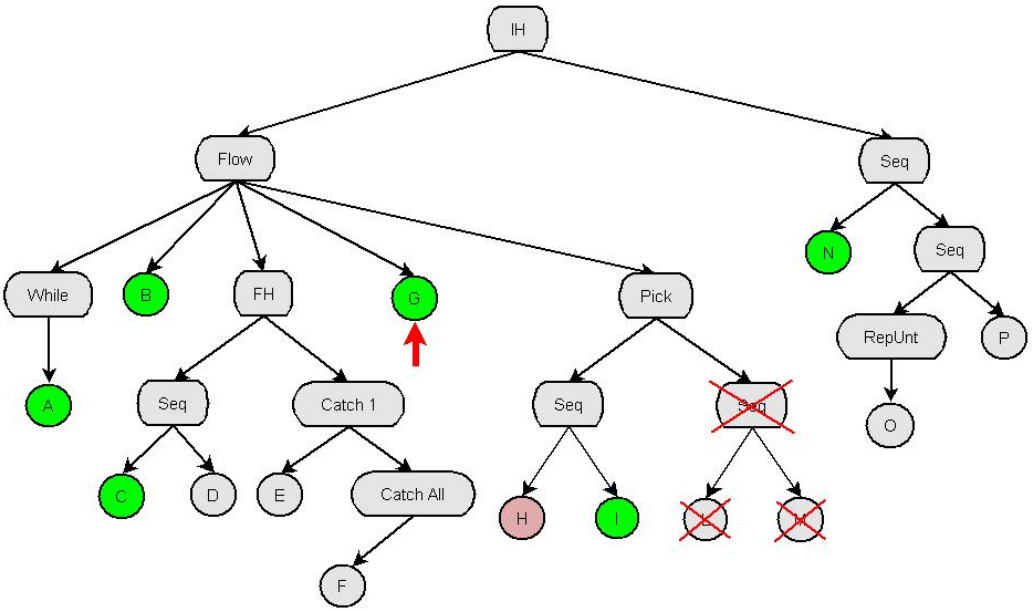

Figura 3.3c: Stato S_1 risultato di *EseguiNodo(H, true, -)* nello stato S_0

[9] Si sarebbe potuto scegliere benissimo un'altra istruzione a caso, la scelta è caduta sulla primitiva H.

Come si può vedere nella figura 3.3c, l'esecuzione della primitiva H (il cui colore è passato da verde a rosa) comporta la rimozione dallo stato non solo di H ma anche della primitiva L (in generale, vengono rimosse tutte quelle primitive eseguibili che vengono raggiunte da quel ramo *Pick* che non è stato scelto -si ricorda che *Pick* esegue una scelta non deterministica tra più rami ed esegue solo le primitive del ramo scelto-). Lo stato ottenuto è quindi S_1 = { A, B, C, G, I, N }, su cui si sceglie di eseguire la primitiva G.

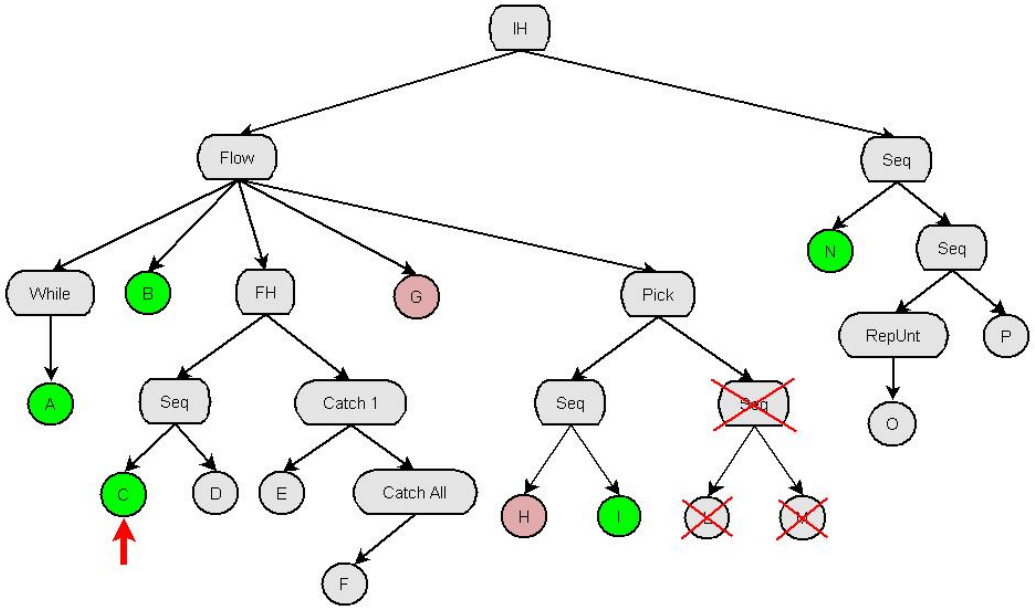

Figura 3.3d: Stato S_2 risultato di *EseguiNodo(G, true, -)* nello stato S_1

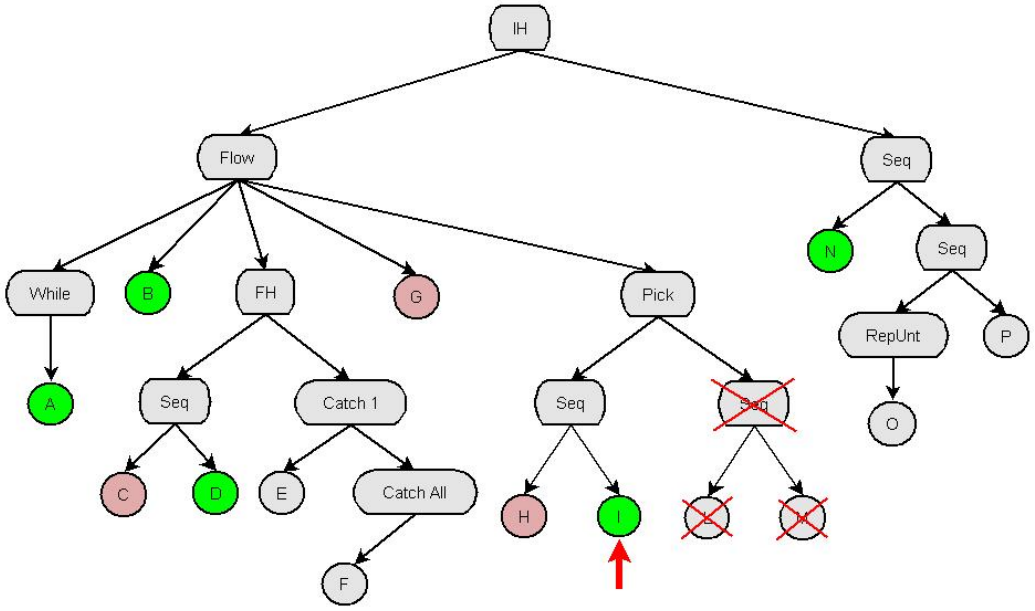

Figura 3.3e: Stato S_3 risultato di *EseguiNodo(C, true, -)* nello stato S_2

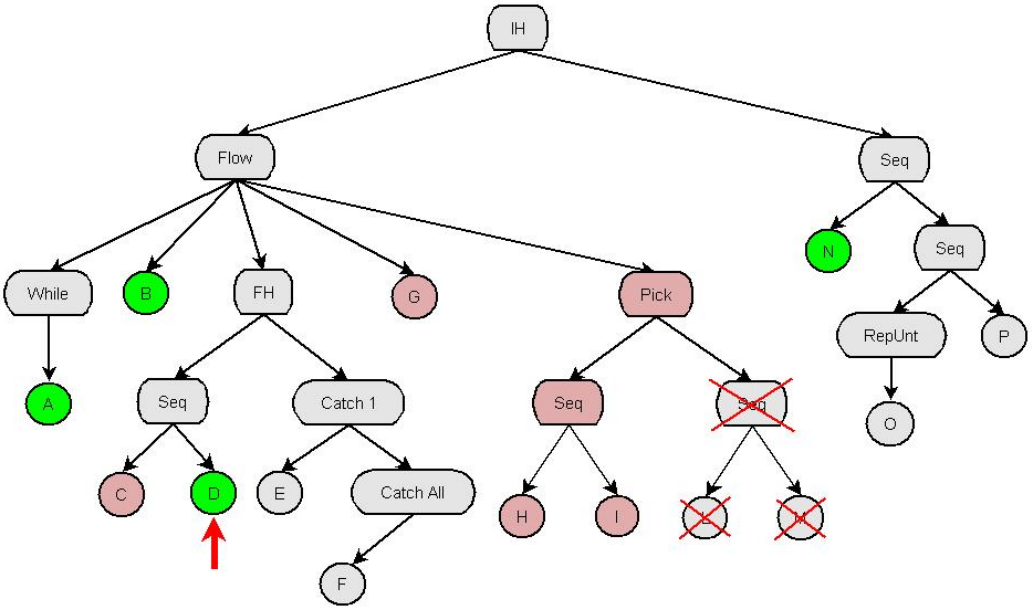

Figura 3.3f: Stato S_4 risultato di *EseguiNodo(I, true, -)* nello stato S_3

L'esecuzione dell'istruzione I comporta anche l'esecuzione di tutto il comando *Pick* (come mostrato nella figura 3.3f, tutto il comando *strutturato* è colorato di rosa): lo stato risultante è S_4 = { A, B, D, N }. A questo punto si decide di far sollevare alla istruzione D (selezionata per l'esecuzione) un fault.

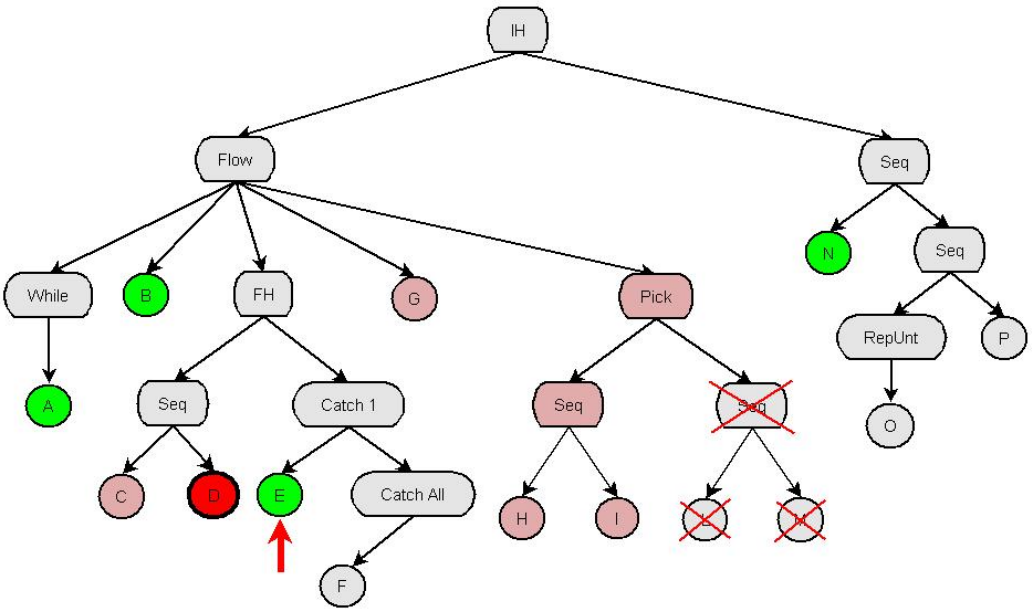

Figura 3.3g: Stato S_5 risultato di *EseguiNodo(D, false, "faultName")* nello stato S_4

L'istruzione fallita D (in rosso nella figura 3.3g) fa visitare all'algoritmo *EseguiNodo* il ramo *catch* del *FaultHandler*, ramo che, con esecuzione corretta di C e di D, non verrebbe considerato (si presuppone che il fault sollevato sia gestibile dal primo *catch*, quindi il *catchAll* continua a non essere considerato). Come si può notare, lo stato risultate di questa applicazione della funzione di transizione si è arricchito della primitiva E, risultando essere S_5 = { A, B, E, N }. Come prossima istruzione da eseguire è stata scelta la primitiva E (si vuole quindi gestire il fault sollevato).

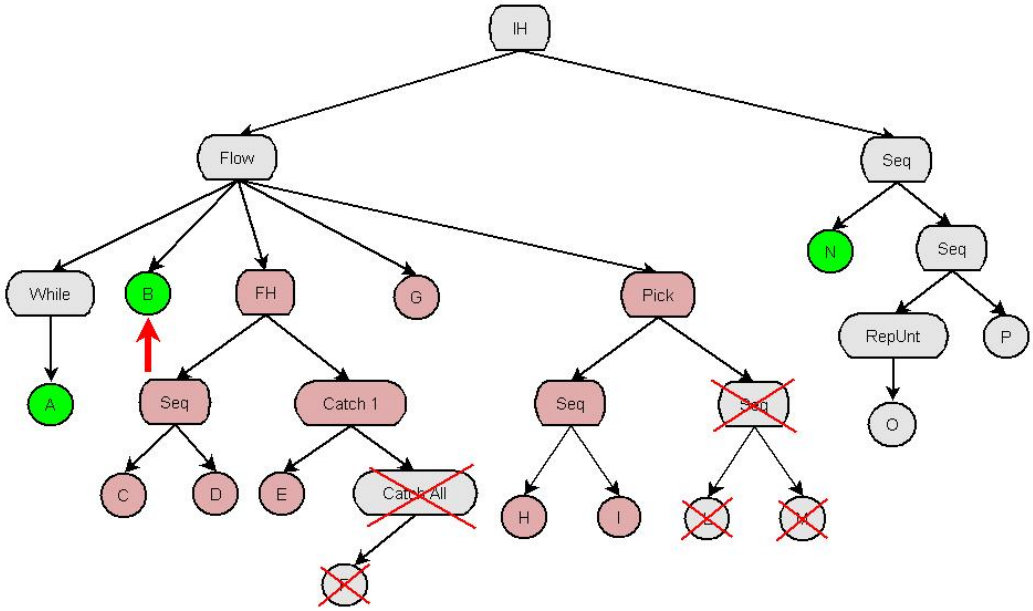

Figura 3.3h: Stato S_6 risultato di *EseguiNodo(E, true, -)* nello stato S_5

Con l'esecuzione della primitiva E si completa l'esecuzione dell'intero *FaultHandler* quindi l'intero sotto albero avente radice in FH viene etichettato come eseguito e colorato di rosa (ad esclusione dei nodi *CatchAll* ed F che, come già accennato, non vengono nemmeno visitati).

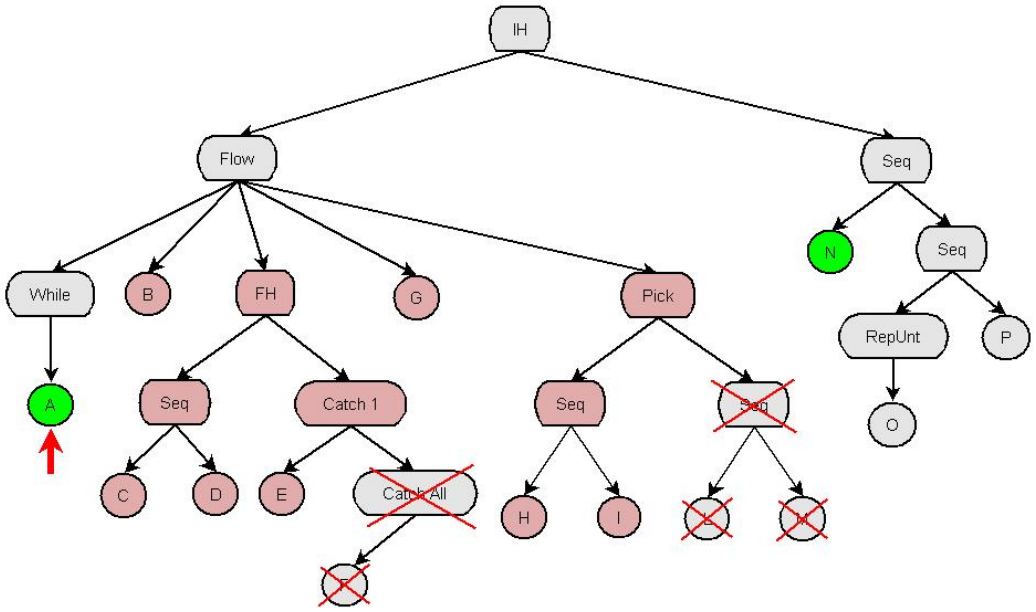

Figura 3.3i: Stato S_7 risultato di *EseguiNodo(B, true, -)* nello stato S_6

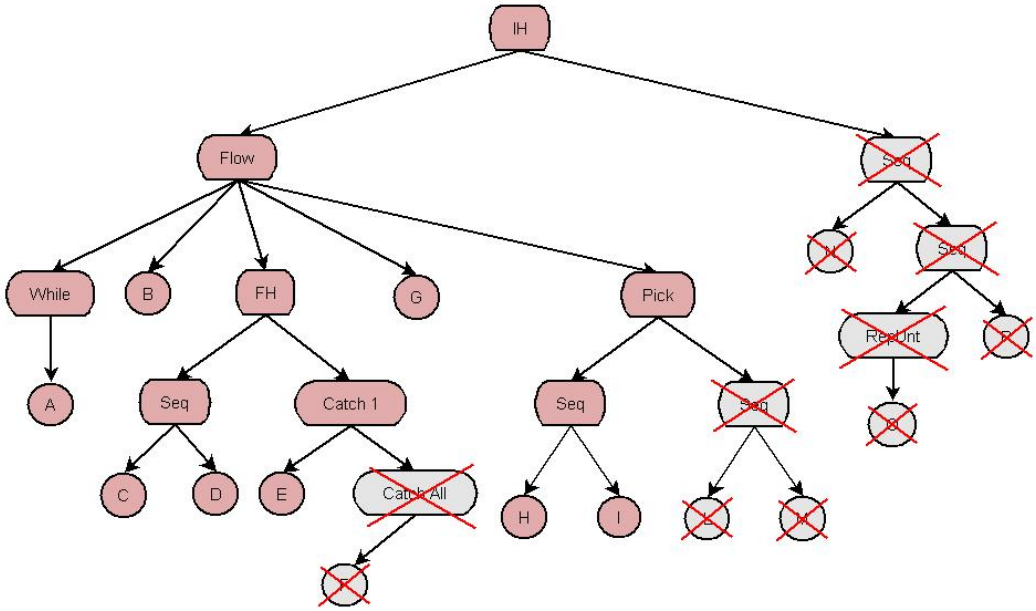

Figura 3.3l: Stato S_8 risultato di *EseguiNodo(A, true, -)* nello stato S_7

Con l'esecuzione della primitiva A viene rimosso dallo stato risultante S_8 anche la primitiva guardia N: questo perché il corpo dell'*InformationHandler* viene terminato quindi non è più possibile ricevere nessun ulteriore *input* dall'ambiente (come è spiegato nel capitolo 2).

Siccome lo stato risultante S_8 risulta essere vuoto, la simulazione è terminata, dando come risultato la traccia rappresentata nella figura 3.4 sottostante.

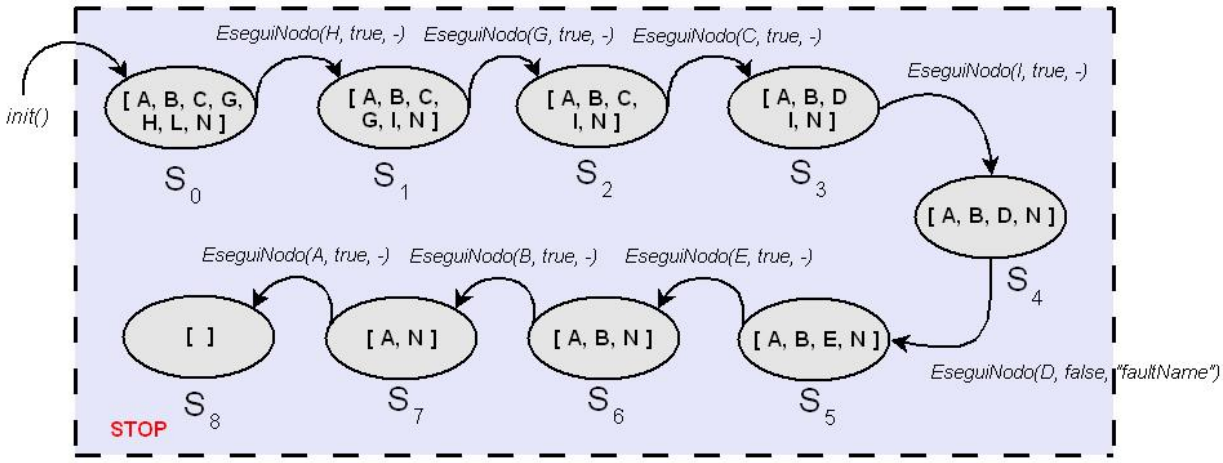

Figura 3.4: Insieme degli stati della simulazione

Sugli algoritmi di *visitaNodo* ed *EseguiNodo* e sulle strutture dati logiche/fisiche appena descritte (il concetto di *stato*/lo *stack* delle operazioni eseguite) si basano tutte le funzionalità offerte dal simulatore. Non solo, è proprio all'interno dello stesso *engine* che vengono fisicamente implementate tute le funzioni di "*modifica esecuzione delle tracce*" di cui il simulatore è dotato. In particolare, l'*engine* è in grado di offrire la possibilità di

- eseguire passo per passo tutte le primitive di un programma SMoL,
- far fallire una o più istruzioni sollevando dei fault,
- annullare le ultime *K* primitive eseguite,
- salvare/caricare una singola simulazione,
- simulare in automatico tutte le simulazioni possibili

Chiaramente, per implementare tutte queste funzioni sono necessarie ulteriori strutture dati: gran parte di queste ultime non verranno illustrate in quando non godono di particolari proprietà mentre quelle necessarie alla comprensione del sistema verranno ampliamente spiegate nel prossimo paragrafo.

3.2.3 L'Interfaccia Grafica

L’*interfaccia grafica* permette all’utente di interagire con il simulatore fornendo, in ogni momento, una visione generale delle operazioni compiute dal prototipo e, soprattutto, offrendo la possibilità di comprendere lo stato in cui si trova una simulazione (in modo da poterla modificare, se ritenuto necessario).
Lo sviluppo del modulo è stato portato avanti con l’idea di avere un’interfaccia minimale che fosse in grado di offrire all’utente tutte le funzionalità del prototipo in maniera semplice, immediata: inoltre si è pensato di implementare anche un semplice visualizzatore di codice xml affinché fosse possibile un confronto istantaneo tra il programma caricato e il codice sorgente di partenza (senza il bisogno di ricorrere ad ulteriori applicazioni).
Le caratteristiche dell’interfaccia grafica verranno meglio illustrate nel Capitolo 4 quando verranno mostrati degli *screenshots* del prototipo al lavoro.

3.3 Realizzazione del prototipo

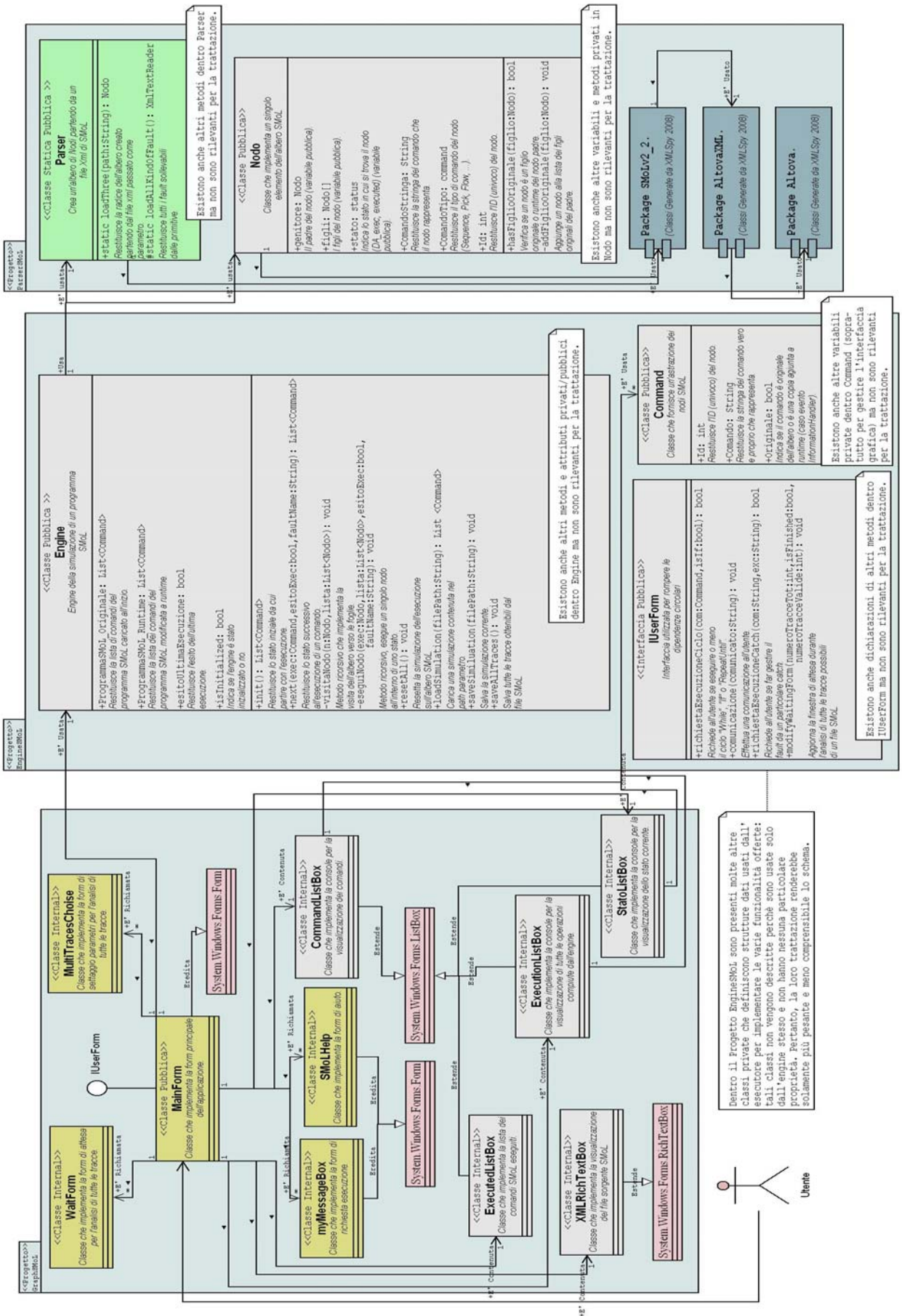

Figura 3.5: Schema UML Applicazione

Lo schema della figura 3.5 rappresenta la struttura delle classi della soluzione .NET *SolutionSMoL*: all'interno dello schema (e nella descrizioni che seguiranno) alcune classi non verranno descritte in quanto il loro contributo alla comprensione del sistema sarebbe poco significativo. Come si può notare viene mantenuta la struttura modulare accennata all'inizio del capitolo: i tre progetti *GraphSMoL*, *EngineSMoL* e *ParserSMoL* corrispondono rispettivamente ai tre moduli *interfaccia grafica*, *engine* e *parser* (come esemplifica la figura 3.6 in cui è anche possibile notare le strutture dati usate dai moduli per comunicare tra loro, - il loro uso verrà illustrato successivamente).

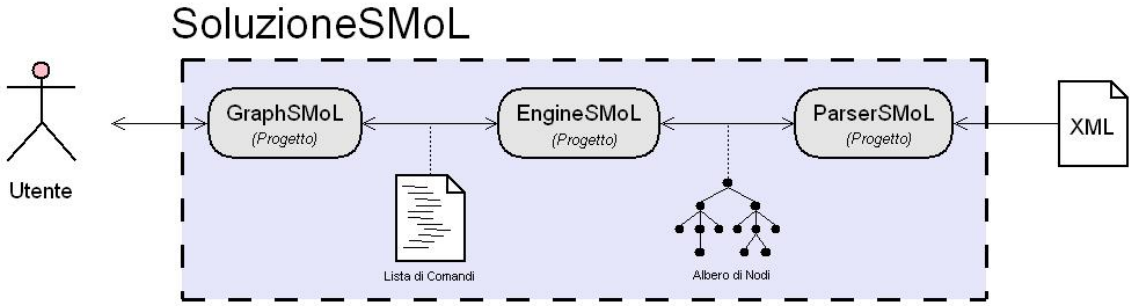

Figura 3.6: Schema Soluzione SMoL

Nel seguito verrà data particolare enfasi alla descrizione delle modalità di traduzione dei comandi *strutturati* SMoL in strutture di istanze di *Nodo* all'interno dell'albero sintattico (traduzione a carico del *parser*) e alla descrizione degli algoritmi *visitaNodo* (che corrisponde anche alla procedura *init*) ed *eseguiNodo* tramite l'uso di uno pseudo codice (algoritmi eseguiti dall'*engine*).

3.3.1 L'albero sintattico SMoL

L'albero sintattico generato dal *parser* e passato successivamente all'*esecutore tracce* è una struttura dati fondamentale al nostro simulatore: all'interno della struttura, le primitive del linguaggio sono le foglie mentre i nodi interni sono i comandi *strutturati*. Entrambe le tipologie sono descritte dalla classe *Nodo* (di cui viene fornita una descrizione in uno pseudo-codice):

```
Classe Nodo
{
   puntatore al Nodo padre (genitore);
   lista di puntatori ai Nodi figli (figli);
```

```
   identificatore univoco (id);
   stringa del comando associato al nodo (tipoDiComando);
   stato del nodo (stato = {DA_exec, executed});
}
```

Chiaramente, all'interno della classe sono presenti molte altre variabili e funzioni -utili a livello implementativo-, ma ai fini della spiegazione, le loro descrizioni sono superflue.
Ogni nodo è la rappresentazione di un comando SMoL ben preciso ed ha sia un riferimento al genitore (tranne nel caso del nodo *ROOT* che ha come genitore il valore *null*), sia riferimenti ad eventuali figli (ad esclusione delle primitive che sono le foglie dell'albero). Inoltre, ogni nodo è identificato in maniera univoca da un *id*: questo identificatore permette di capire, in un qualsiasi momento (sia *staticamente* che a *runtime*), a quale istanza di comando ci riferiamo (molto utile nel caso di *InformationHandler*, in cui si "duplica" il codice).
Lo stato del nodo indica se è stato già eseguito o meno: le modalità con cui viene cambiato di stato un nodo sono strettamente legate alla semantica del linguaggio (e di cui verrà riportato uno schema).

Di seguito vengono riportate le strutture usate nell'albero per rappresentare i comandi strutturati SMoL: all'interno delle figure, con il termine "*Comando*", si intende rappresentare sia una primitiva semplice (foglia), sia uno qualunque dei comandi *strutturati* illustrati. Nei casi in cui sia esplicitamente usato il temine "*Primitiva*", si intende che attraverso quel collegamento è raggiungibile solamente una primitiva.
Per avere ben chiara la modalità di rappresentazione usata per tutti i comandi si rimanda alla descrizione fornita nel Capitolo 2 (per i comandi *strutturati* e le primitive SMoL) e a [16] (per maggior dettaglio sulle primitive SMEPP).

Assign

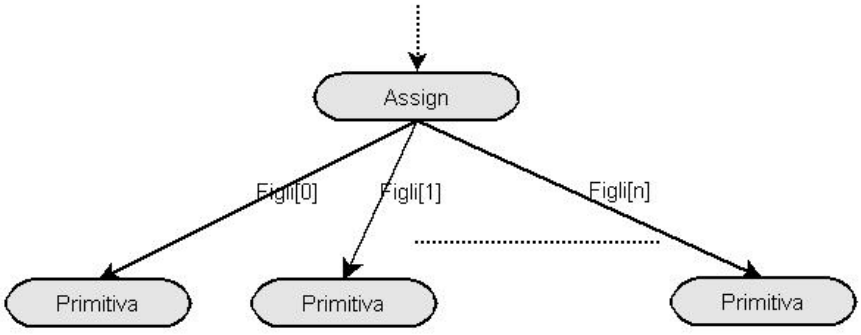

Figura 3.7a: Schema *Assign*

Al nodo *Assign* vengono connesse direttamente delle foglie e in particolare delle primitive nella forma "`TO = FROM`". Il nodo *Assign* è marcato come eseguito quando tutte le primitive direttamente connesse risultano essere eseguite.

Sequence/Flow

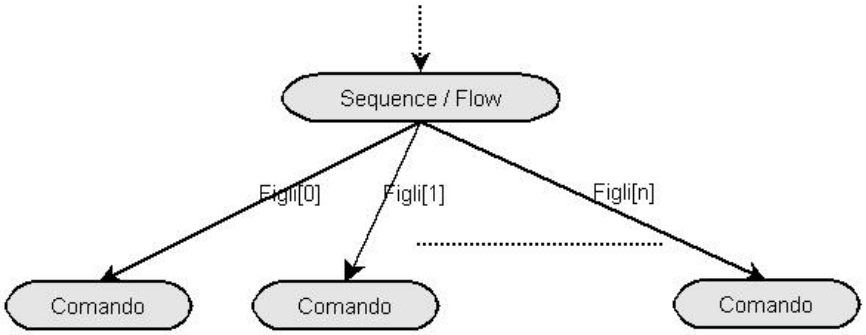

Figura 3.7b: Schema *Sequence*/*Flow*

I nodi che rappresentano i comandi *Flow* e *Sequence* possono avere n figli direttamente connessi e sono marcati come eseguiti quando tutte i figli direttamente connessi risultano essere eseguiti.

RepeatUntil / While

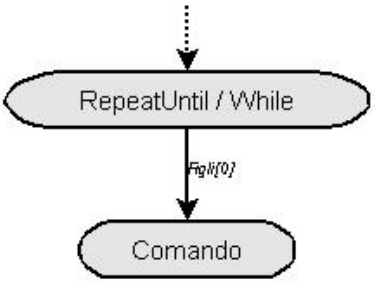

Figura 3.7c: Schema *While*/*RepeatUntil*

I cicli *RepeatUntil* e *While* hanno solamente un figlio, il quale può essere una primitiva oppure un altro comando strutturato. Ad ogni inizio di esecuzione del ciclo (*While*) oppure

alla fine di esecuzione del ciclo (*RepeatUntil*), viene valutata la possibilità di etichettare il nodo come eseguito[10].

If

Il nodo rappresentante il comando *if* ha, a seconda dei casi, uno o due figli: figli[0] punta al primo comando (primitiva o strutturato) che si dovrebbe eseguire nel caso di guardia verificata; figli[1] punta al primo comando del corpo dell'*else* (figli[1] è opzionale, mentre figli[0] è necessario).

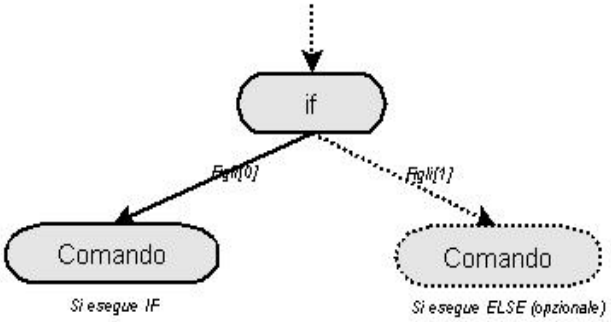

Figura 3.7d: Schema *If*

Il comando è etichettato come eseguito quando il figlio scelto è stato a sua volta marcato come eseguito.

Pick

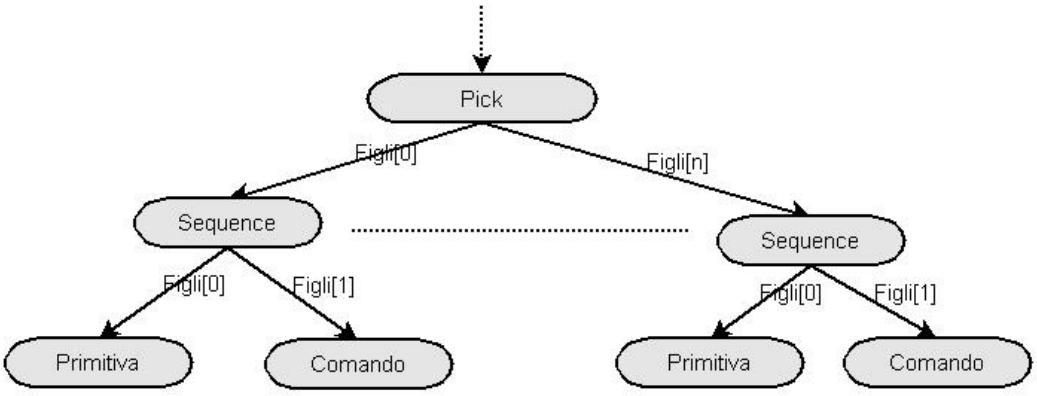

Figura 3.7e: Schema *Pick*

[10] La valutazione consiste nel chiedere all'utente se terminare con l'esecuzione del ciclo (nel caso di esecuzione *step-by-step*) oppure nel valutare alcune condizioni che il simulatore ha al suo interno (in caso di simulazione *automatica*).

Il comando *Pick* è rappresentato con nodi che possono avere fino a n figli di tipo *Sequence*: ognuno di questi ha, a sua volta, due figli: in posizione figli[0] ha una primitiva "guardia" (per esempio, *Wait, ReceiveMessage* o *ReceiveEvent*) e in posizione figli[1] ha un comando.
Quando viene scelto di eseguire un ramo *j* (perché si esegue la "guardia" *j*-esima associata), tutti i rami aventi come radice i figli di *Pick* differenti dal figlio *j*-esimo vengono segnati come eseguiti.
Il comando è marcato come eseguito quando il figlio *Sequence*, radice del ramo scelto per l'esecuzione, è stato etichettato come eseguito.

FaultHandler

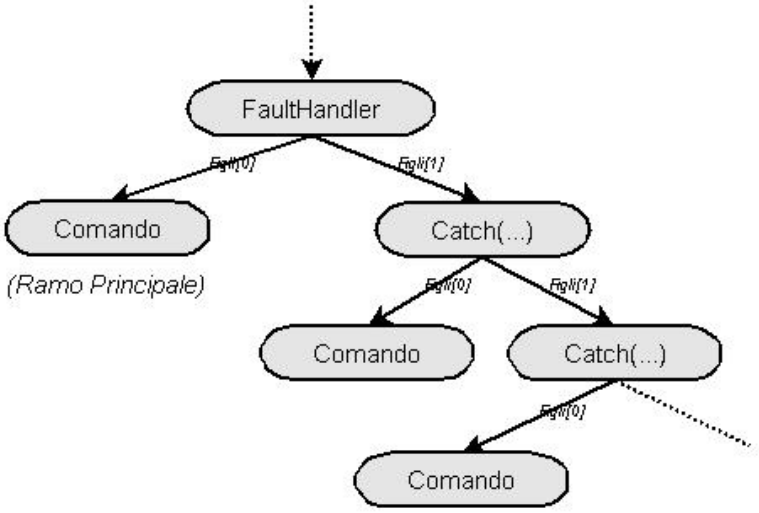

Figura 3.7f: Schema *FaultHandler*

In posizione figli[0], il nodo ha un puntatore al primo comando (primitiva o strutturato) del blocco da eseguire all'interno del *FaultHandler*. Il secondo figlio (figli[1]) punta a un nodo "*Catch(…)*" il quale effettua un confronto tra l'eccezione riscontrata e quella gestita (quella gestita è indicata tra le parentesi del comando *catch*): se il confronto da esito positivo, allora si passa ad eseguire i comandi relativi (che si trovano su figli[0] del nodo), altrimenti si passa al successivo nodo "*Catch(…)*" seguendo il puntatore figli[1] del nodo stesso. Qualora non si trovi nessun *catch* adatto, il fault viene propagato al padre del nodo *FaulHandler*. Da notare che:

- in caso di corretta esecuzione del *Ramo Principale*, tutta la parte relativa ai *catch* non viene nemmeno visitata;

- Il "*CatchAll()*" viene pensato come posto in ultima posizione della catena in quanto eventuali *catch* posti successivamente non vengono presi in considerazione.

Lo stato del nodo assume valore "*eseguito*" quando viene marcato come eseguito o il figlio[0] (in assenza di fault sollevati) oppure il figlio[1] (in presenza di fault gestiti da *Catch*/*CatchAll*).

InformationHandler

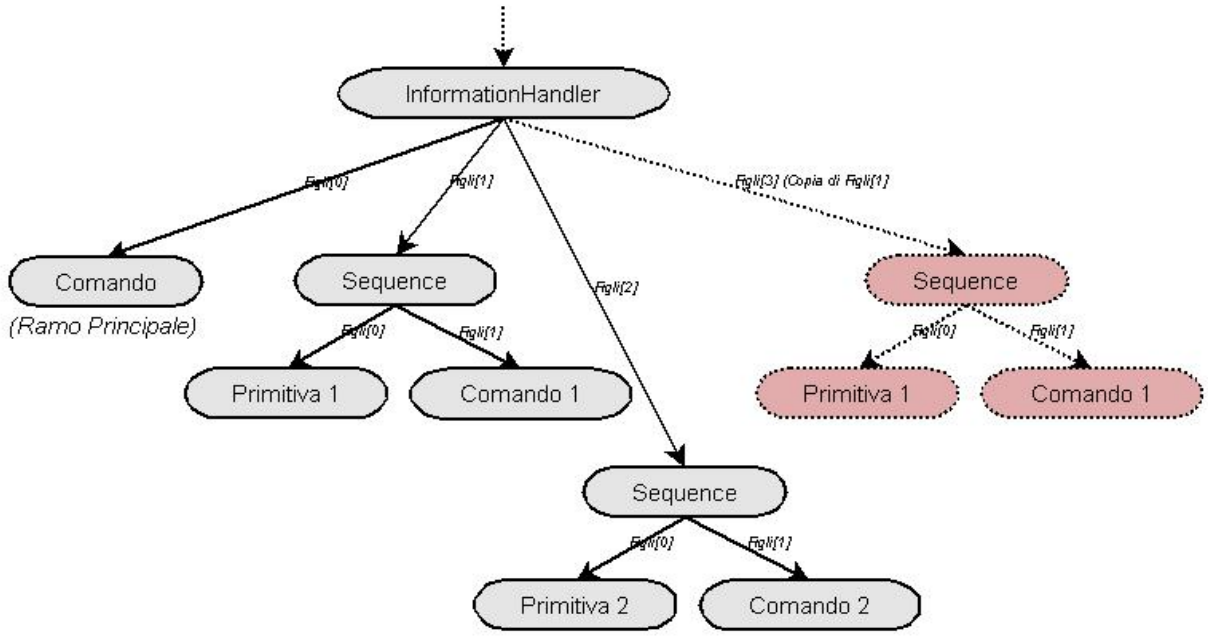

Figura 3.7g: Schema *InformationHandler*

Il nodo *InformationHandler*, ha il puntatore figli[0] che punta al primo comando (primitiva o strutturato) del blocco da eseguire, esecuzione durante la quale è richiesta attenzione a eventi che si possono manifestare.

Nei figli successivi (figli[*j*] con $j > 0$), vengono puntati dei nodi *Sequence* i quali hanno le guardie (che attendono eventi -*ReceiveEvent*-, messaggi -*ReceiveMessage*- o lo scadere di timeout -*Wait*-) nella parte sinistra (in figli[0]) e i relativi comandi da eseguire nel caso una guardia si verifichi nella parte destra (figli[1]). In genere, questi sotto-alberi non sono mai realmente eseguiti in quanto, quando si verifica una guardia, l'intero sotto-albero associato viene copiato e aggiunto come *n-esimo* figlio al nodo *InformationHandler* (e viene eseguita la copia, evidenziata in rosso nella precedente figura): in questo modo si permette di ricevere e di gestire più eventi dello stesso tipo durante l'esecuzione del *Ramo Principale*. Solamente

i sotto-alberi associati ad una guardia *Wait* senza il parametro *repeatEvery* non vengono copiati ma eseguiti direttamente in quanto possono essere processati solamente una volta (il parametro *repeatEvery* permette la ri-esecuzione periodica ad intervalli di tempo regolare[11])
Al termine dell'esecuzione della gestione dell'evento, il sotto-albero copia viene rimosso. Nel caso in cui una copia del ramo evento dell'albero fallisca (in quanto si è generato un fault che non ha il corrispettivo *catch* all'interno del sotto-albero stesso), tutto l'*InformationHandler* termina la sua esecuzione e il fault viene propagato al padre del nodo *IH*.
Infine, il nodo è etichettato come eseguito quando il figlio[0] è stato eseguito e sono stati completati tutti i rami di eventi il cui inizio è cronologicamente precedente alla fine del *Ramo Principale*.

Alla luce di quanto detto fino ad ora, se per esempio volessimo estrarre l'albero sintattico dal codice del peer `TempReaderPeer` dell'esempio *Monitoraggio della temperatura (II)* proposto nel capitolo 2, avremmo:

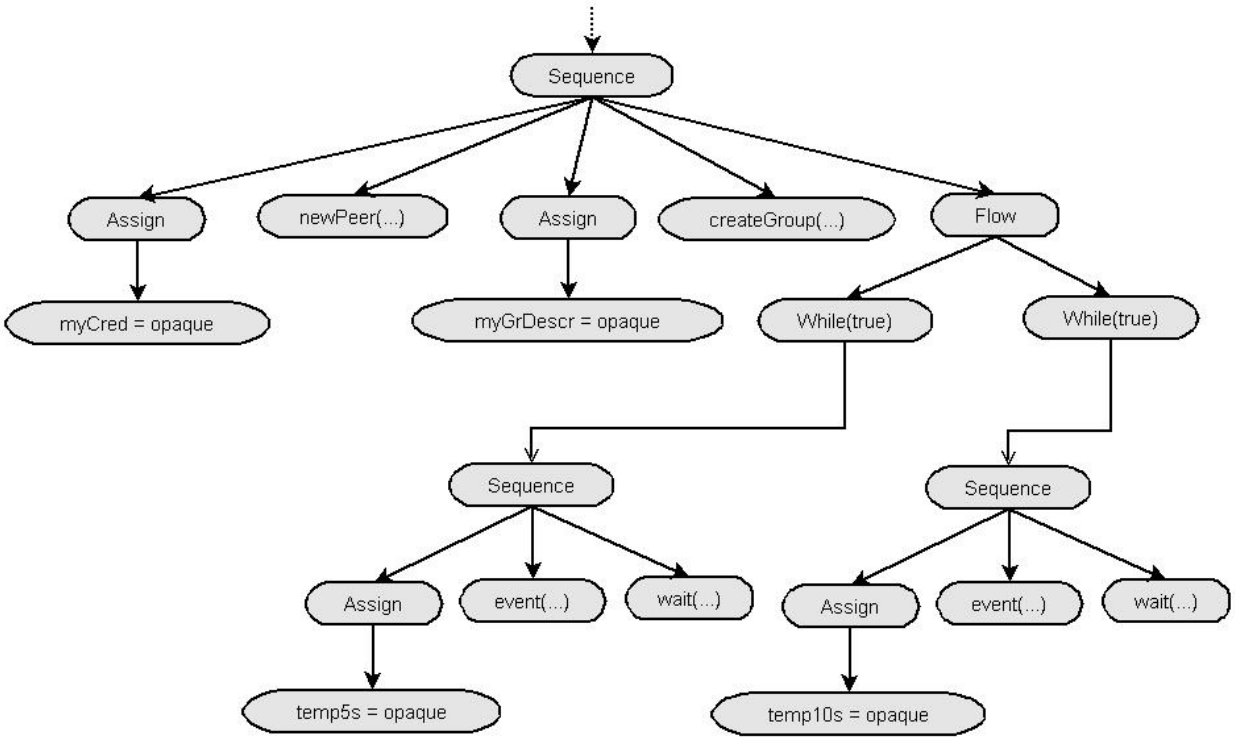

Figura 3.7h: Albero Sintattico di "Monitoraggio della temperatura (II)"

[11] Per maggiori informazioni guardare la descrizione della primitiva SMoL *Wait* nel Capitolo 2.

3.3.2 Implementazione del Parser

Il *parser* (implementato nel progetto *ParserSMoL* della solution .NET) è la componente alla base dell'intera soluzione. Il modulo è composto da due classi: la classe statica *Parser* (*Parser.cs*) e la classe Nodo (*ParserDataStructure.cs*). La prima offre sostanzialmente due metodi, *loadTree(String filePath)* per creare un albero sintattico dal file passato come parametro e *loadAllKindOfFault()* che permette di caricare tutti i fault sollevabili dai comandi SMoL (i fault vengono letti dal file *IstructionFault.xml* che l'applicazione, se non lo trova, genera in automatico: in questo modo e possibile editare il file xml per rimuovere o aggiungere nuovi fault).

Il parsing avviene in due fasi: da principio si verifica che il file passato come parametro sia un file xml nel formato SMoL e che non sia vuoto; in secondo luogo, si cominciano a scorrere i vari nodi xml dai quali si ricavano le istanze di *Nodo* che poi entreranno a far parte dell'albero sintattico.
Un aspetto da sottolineare è il modo in cui la classe *Parser* e la classe *Nodo* lavorino insieme e siano strettamente necessarie l'una con l'altra: se da un lato è la classe *Parser* che costruisce l'albero di nodi *Nodo* partendo dal file sorgente xml, implementando, per ogni comando strutturato, le strutture descritte nel paragrafo seguente (strutture ricavate dalla semantica del linguaggio SMoL), dall'altro è la classe *Nodo* che parsa le singole primitive e ne verifica la correttezza.
Come già accennato in precedenza, è stata preferita la classe *Nodo* alla classe *XMLNode* fornita dal C# in quanto è più semplice e fornisce le informazioni utili agli algoritmi in maniera diretta (rendendoli ancora più efficienti). Inoltre, se necessario, offre anche un riferimento al codice xml di base. Per capire meglio il vantaggio, prendiamo in considerazione il semplice assegnamento

```
x = 7
```

e il suo sorgente SMoL

```
<assign>
   <copy>
      <from>
         <literal>7</literal>
      </from>
      <to variable="x">
         <documentation>Variable x blah, blah, blah,...</documentation>
      </to>
   </copy>
```

```
</assign>
```

Il comando non fa che assegnare alla variabile *x* il valore *7*: inoltre al suo interno è anche presente della documentazione (banale) sulla variabile in questione (anche se ai fini del parsing/esecuzione, il tag *documentation* non influisce, nell'albero sintattico xml va preso in considerazione).
Alla luce di ciò, l'assegnamento preso in analisi da origine ai seguenti alberi (in rosso sono segnati i riferimenti delle istanze *Nodo* alle istanze *XMLNode*):

Albero XML (*XMLNode*): **Albero di istanze *Nodo*:**

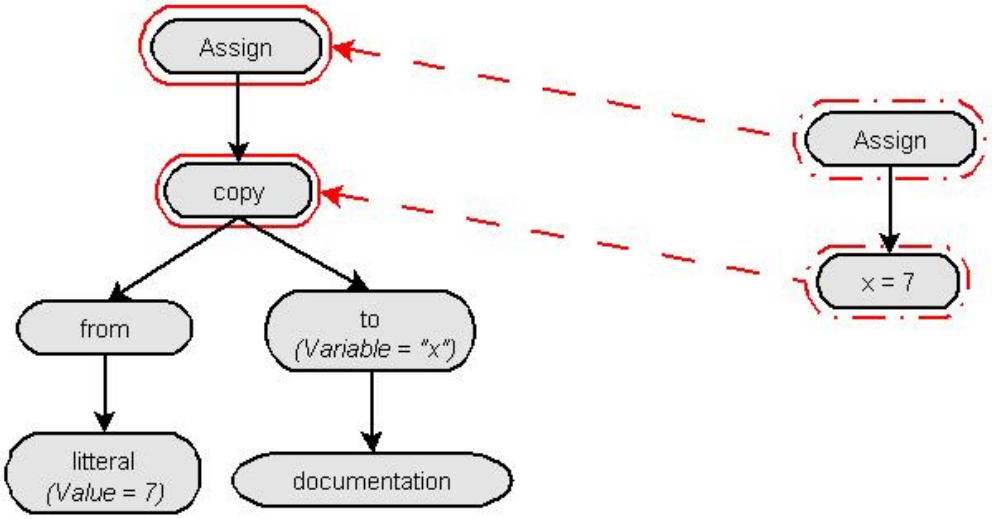

Figura 3.8: Differenze tra Alberi XML e Alberi di *Nodi*

Come si può notare, anche su un comando semplice come un assegnamento, l'albero di nodi *Nodo* risulta essere molto più compatto e snello di quello che risulterebbe essere un albero di nodi *XMLNode*.

Per la procedura di parsing, sia la classe *Parser* che la classe Nodo usano dei package generati in automatico -in base al file schema *SMoL.xsd*- dal programma Altova XMLSpy 2008.. Questi package sono inseriti all'interno del progetto e vengono compilati, insieme alle altre due classi, nella libreria *ParserSMoL.dll*.

3.3.3 Implementazione dell'Engine

L'*engine* (o *esecutore tracce*) è la parte del prototipo che "anima" un programma SMoL, che ne simula il comportamento, offrendo la possibilità all'utente di interagire con l'esecuzione stessa: implementata nel progetto *EngineSMoL* della solution .NET (e compilata nella libreria *EngineSMoL.dll*), riceve dal *parser* la rappresentazione del programma da simulare sotto forma di albero di nodi *Nodo* e su di essa esegue i flussi di controllo (*workflows*).
Inoltre, mantiene nascosta la struttura dati su cui lavora al modulo grafico, impedendogli un accesso diretto e quindi negandogli la possibilità di modificare l'albero: alla parte grafica, viene fornita una rappresentazione del programma SMoL sotto forma di lista di istanze della classe *Comando*, una classe che contiene le sole informazioni utili alla visualizzazione della simulazione (la traduzione del comando da xml alla sintassi SMoL, informazioni sulla modalità di visualizzazione, ecc...) (vedi figura 3.6 e lo schema UML in figura 3.5).
Tutte le funzionalità offerte dal simulatore vengono implementate all'interno del modulo: è importante notare come nella implementazione di tutte le operazioni eseguibili sull'albero si faccia uso (oltre a strutture dati private, create *ad-hoc* per l'operazione) di due algoritmi, quelli di *visita* e di *esecuzione* (che verranno descritti in seguito).
Di seguito verranno illustrate le scelte progettuali fatte per ognuna delle funzionalità affinché si possa avere una visuale completa sulla componente.

3.3.3.1 Next / Fault

Le funzionalità *Next* e *Fault* permettono l'esecuzione di una singola primitiva: entrambe le funzionalità possono essere ricondotte ad una stessa funzione di transizione *EseguiNodo* con la differenza che *Next* esegue una istruzione in maniera corretta mentre *Fault* la esegue facendogli sollevare un fault.
In fase di inizializzazione, l'albero sintattico SMoL viene *visitato* per vedere quali foglie sono raggiungibili e quali no: le fogli raggiungibili formano il primo stato S_0, su cui poi applicare la funzione di transizione *EseguiNodo* per passare allo S_1 e così via.
EseguiNodo esegue sostanzialmente due semplici operazioni sull'albero: per prima cosa, marca la foglia rappresentante il comando che si è scelto di eseguire come "*executed*" -eseguita- e successivamente verifica lo stato di completamento di quella parte dell'albero e se esistono le condizioni per effettuare ulteriori visite in altri sotto-rami oppure no.
"*Verificare lo stato di completamento di una parte dell'albero*" significa, abbastanza grossolanamente, verificare se il padre, nonno o altri avi della foglia eseguita hanno ulteriori

figli *foglie* da eseguire o comandi strutturati: in questo ultimo caso, per raggiungere i figli (discendenti) foglie si rende necessaria una visita.
Nelle visite compiute dall'algoritmo *visitaNodo* (richiamate anche da *init* e da *EseguiNodo*) si tende sempre, da un nodo radice o interno (che rappresentano i vari comandi strutturati), ad arrivare alle primitive, rappresentate dalle foglie: durante questa fase di visita, quando si arriva in presenza di un ciclo *while* o *repeatUntil* o di un comando *if* (comandi condizionano la visita in quanto escludono o meno interi sotto-alberi), la procedura chiede sempre cosa fare all'utente (questa fase di richiesta può essere chiaramente automatizzata).
L'esecuzione della funzione di transizione *EseguiNodo* viene applicata fino a che non si raggiunge uno *stato* vuoto, senza ulteriori istruzioni da eseguire. Raggiungere questo ultimo *stato* può significare due cose: che le istruzioni eseguibili sono state eseguite tutte e l'esecuzione complessiva risulta essere andata a buon fine oppure che è stato sollevato un fault che non è stato gestito. Quando viene sollevato un fault durante l'esecuzione di una primitiva, viene propagato nell'albero dalla foglia "fallita" verso la radice alla ricerca del primo *FaultHandler* disponibile (NB: tutti i nodi che vengono coinvolti nella fase di ricerca vengono marcati come eseguiti): se questo tipo di nodo viene trovato, si passa con l'esecuzione del *catch* appropriato (se esiste) altrimenti si continua a procedere a ritroso fino alla radice. Se, arrivati alla radice, non si è trovato un *FaultHandler* oppure lo si è trovato ma senza *catch* appropriato (quindi non ci sono istruzioni da eseguire), l'esecuzione termina segnalando all'utente il fallimento della stessa istruzione e della traccia in generale.

3.3.3.2 Play

L'esecuzione automatica di una singola traccia deriva direttamente dalla funzione *Next* prima descritta: *Play* non fa altro che inizializzare lo stato ed eseguire la funzione di transizione ad oltranza fino a che non ottiene uno stato vuoto. Nel caso in cui uno stato intermedio della traccia sia composto da più istruzioni eseguibili, *Play* prende di default la prima ed esegue quella.

3.3.3.3 Previous / Rollback to

Con le funzionalità *Previous* e *Rollback to* si ha la possibilità di "ri-avvolgere" l'esecuzione di una simulazione fino ad un punto arbitrario. Durante l'esecuzione del programma SMoL (che, come già detto, consiste in una sequenza di esecuzioni della funzione di transizione *EseguiNodo* da uno stato S_K ad uno S_{K+1}), il simulatore si "ricorda" le primitive eseguite,

impilandole su di uno *stack*: in particolare, sullo *stack* vengono salvate triple di tipo `<ID_primitiva, esito, faultName>` che descrivono in modo conciso l'esecuzione di un comando.
Quando si decide di tornare indietro nell'esecuzione fino ad una istruzione *j*-esima arbitraria, l'*engine* non fa altro che copiarsi lo *stack*, azzerare l'intera esecuzione (praticamente segnare tutte le foglie dell'albero come "*DA_exec*" -da eseguire-, rimuovere parti dell'albero aggiunte a *runtime* e ripulire eventuali strutture dati usate), rimuovere dallo *stack* le ultime *j* istruzioni eseguite e rieseguire da capo la simulazione, basandosi sulla copia dello *stack* modificata.

La scelta di rieseguire l'intera simulazione da capo è stata attentamente valutata in fase di progettazione e verificata in fase di implementazione: si sarebbe potuto infatti decidere di salvarsi tutti gli stati raggiunti, ovvero non solo salvare tutta la lista delle istruzioni eseguibili per ogni stato ma anche, più in generale, "fotografare" lo stato totale del simulatore, salvarsi le foglie/comandi strutturati dell'albero eseguiti e non, tutte le parti aggiunte all'albero a *runtime* ed eventuali strutture dati ausiliarie usate.

Le due soluzioni citate hanno caratteristiche molto differenti. La prima delle due (la soluzione adottata), a fronte di un netto risparmio di memoria occupata, penalizza la velocità di esecuzione dell'operazione di *Previous / Rollback to* man mano che il numero di istruzioni eseguite aumenta: nel caso di *N* istruzioni eseguite (con $N>>0$), cancellare l'ultima primitiva eseguita comporta la riesecuzione delle ultime *N-1* istruzioni. La seconda invece, "fotografando" lo stato complessivo del simulatore in un preciso istante, avrebbe un tempo di esecuzione costante anche nel caso di simulazioni aventi *N* istruzioni (con $N>>0$) però occuperebbe una enorme quantità di memoria.
Lo svantaggio che ha la soluzione adottata è, però, in gran parte compensato dalla semplicità e dalla velocità di esecuzione della funzione di transizione *EseguiNodo*: inoltre, il risparmio di memoria ottenuto è vantaggioso sia a livello computazionale (una programma che non alloca troppa memoria per la propria esecuzione risulta essere più interattivo nella esecuzione, non tende a rallentare il computer su cui è in esecuzione) sia al livello di salvataggio delle tracce (in quanto si vanno a salvare le tracce in file "leggeri", che occupano poco spazio).

3.3.3.4 Save / Load

L'*engine* offre la possibilità di salvare e di caricare delle simulazioni. Le tracce vengono salvate in due formati: in `*.sls` (*SMoL Simulation*, formato dati studiato *ad-hoc* per l'applicazione) e in formato `*.xml` (*eXtensible Markup Language*) per permettere, in futuro, che i risultati ottenuti dal simulatore possano essere usati da altre applicazioni o per altri scopi.
Nella fase di salvataggio, il modulo scrive su disco lo *stack* delle primitive eseguite fino a quel momento: nel file `sls` viene salvato proprio la struttura dati nel suo complesso (ovvero, un oggetto di tipo `List<ID_primitiva, esito, faultName>`) mentre nel file `xml` viene salvata una rappresentazione "leggibile" delle tracce.
In fase di caricamento di una soluzione, viene letta la struttura dati dal file `sls` e viene eseguita fedelmente per ottenere, alla fine, esattamente lo stesso *stato* in cui era stata salvata in precedenza la simulazione.
Il simulatore, nonostante salvi le tracce sia in formato `*.sls` che in `*.xml`, è in grado di aprire solamente i file nel primo formato. Questa scelta è stata appositamente voluta per motivi di consistenza tra il file sorgente SMoL e la simulazione salvata: infatti il modulo, al momento di salvare la traccia, prende il testo del file sorgente SMoL (su cui è stato creato l'albero), ne calcola un valore *hash* (*MD5 Hash*) e lo scrive come prima cosa nell'*header* del file `sls` e solo dopo inizia a scrivere la vera e propria struttura dati. In questo modo, alla riapertura della simulazione, il modulo è in grado di capire se la simulazione salvata è valida per il file SMoL aperto, confrontando l'*hash* del sorgente con quello scritto nel file `sls`. Inoltre, il formato `sls`, non potendo essere aperto con un editor di testo (come invece è possibile fare con un file `xml`), è garantito contro una qualsiasi modifica che possa essere apportata all'esterno del simulatore stesso: in questo modo è garantita che una simulazione salvata possa essere caricata solamente sul file su cui è stata creata.

3.3.3.5 Save all traces (salvataggio di tutte le possibili tracce)

La funzionalità più complessa dell'*engine* è, senza ombra di dubbio, quella che offre la possibilità di simulare tutte le tracce di un programma SMoL in automatico, senza interagire minimamente con l'utente.
Questa funzione basa la sua esecuzione sulla funzione *Next* (quindi sugli algoritmi di *visita* e di *esecuzione*) descritta in precedenza, però deve essere in grado di fornire tutti quegli *input* in automatico che altrimenti *Next* chiederebbe all'utente e, soprattutto, deve essere in grado

di memorizzare quale percorsi/tracce ha già eseguito e quali no (gli *input* che fornisce a *Next* si basano chiaramente su questa conoscenza).

Il meccanismo di funzionamento è semplice: si inizializza uno stato S_0 (come avviene per ogni simulazione) e poi si esegue una prima traccia portandola a termine, memorizzandosi, in strutture dati apposite, gli *insiemi delle istruzioni che in ogni stato erano eseguibili* (insiemi memorizzati in ordine cronologico). Una volta terminata l'esecuzione, si guardano gli insiemi delle istruzioni partendo da quello inserito per ultimo (che per semplicità, supporremo che si trovi in posizione *K*): questo avrà un'insieme composto da *N* istruzioni in cui risulterà che una è stata eseguita e *N-1* non ancora eseguite. A questo punto si riparte con l'esecuzione "da capo", eseguendo le stesse primitive che si erano eseguite nella traccia precedente fino all'insieme *K*: a questo punto si esegue una delle *N-1* istruzioni non ancora eseguite. Questa esecuzione può dar luogo alla creazione di altri insiemi di istruzioni eseguibili (uno per ogni stato raggiunto): anche questi ultimi verranno memorizzati insieme agli altri (sempre in ordine cronologico). Alla fine della traccia, si riesegue lo stesso controllo e via dicendo.

Quando, in fase di analisi degli insiemi delle istruzioni eseguibili, uno di questi risulta avere tutte le istruzioni eseguite, viene rimosso dalla lista e si passa all'analisi del precedente (da notare che il controllo sugli insiemi termina al primo insieme che si trova, partendo dal fondo, che non ha tutte le istruzioni eseguite in quanto la scelta di eseguire una istruzione non ancora eseguita darebbe vita ad una nuova traccia).

Cerchiamo di capire il funzionamento con un esempio. Supponiamo di avere il seguente (banale) programma SMoL (i numeri alla base di ogni comando sono gli ID associate alle primitive):

```
Flow
   createGroup(gd_1)3
   createGroup(gd_2)4
   createGroup(gd_3)5
End Flow
```

Il frammento di codice crea tre differenti gruppi di peer, aventi ciascuno caratteristiche differenti: il comando `Flow` permette l'esecuzione parallela delle istruzioni contenute al suo interno, quindi possono essere eseguite in un ordine qualsiasi. Perciò, le possibili tracce ottenibili sono (le istruzioni vengono riferite tramite il loro ID):

T_1 = { 3, 4, 5 } T_2 = { 3, 5, 4 } T_3 = { 4, 3, 5 }

T_4 = { 4, 5, 3 } T_5 = { 5, 3, 4 } T_6 = { 5, 4, 3 }

La prima traccia T_1 = { 3, 4, 5 } darà luogo ad uno gruppo di tre insiemi di istruzioni eseguibili (mostrati nello schema in figura 3.9a):

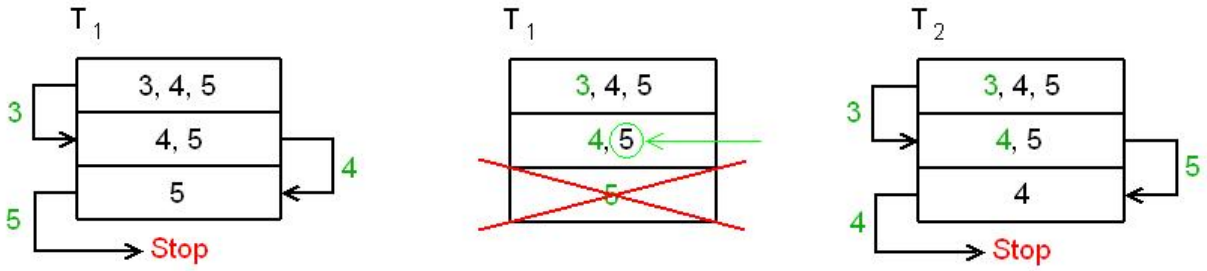

Figura 3.9 (a, b, c): Passaggio dalla traccia T_1 alla traccia T_2

Gli insiemi sono rappresentati come rettangoli nelle figura 3.9a: nel primo insieme sono presenti tutte e tre le istruzioni e si decide di eseguire la numero 3 (l'esecuzione è rappresentata con la freccia esterna che va dal primo al secondo rettangolo); nel secondo insieme sono presenti le istruzioni rimanenti al completamento della traccia, le numero 4 e 5 e si decide di eseguire la numero 4; infine nell'ultimo è presente solo la 5 e viene anch'essa eseguita, terminando così l'esecuzione della traccia. A questo punto vengono analizzati gli insiemi delle istruzioni eseguibili, partendo da quello inserito per ultimo (fare riferimento alla figura 3.9b): questo insieme contiene solo la istruzione numero 5 che è già stata eseguita (è colorata di verde) quindi l'insieme è rimosso. L'insieme precedente contiene l'istruzione 4 e 5: la 4 è già stata eseguita mentre la 5 no. Quindi verrà scelta per l'esecuzione proprio questa ultima istruzione. Il controllo termina in questo istante, in quanto è stata trovata una variante all'ultima traccia eseguita.

La nuova traccia T_2 = { 3, 5, 4 } avrà gli insiemi di istruzioni eseguibili descritti nella figura 3.9c.

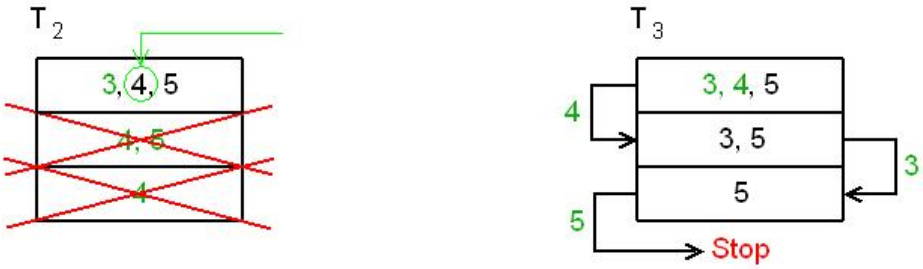

Figura 3.9 (d, e): Passaggio dalla traccia T_2 alla traccia T_3

A questo punto si esegue nuovamente l'analisi degli insiemi (schematizzata nella figura 3.9d) delle operazioni eseguibili (come fatto in precedenza). Analogamente a quanto successo prima, l'ultimo insieme risulta essere composto dalla sola istruzione 4 eseguita e quindi verrà rimosso. Ora però, anche l'insieme precedente risulta avere due istruzioni, la 4 e la 5 già eseguite: anche questo insieme verrà rimosso. L'unico insieme che a questo punto è rimasto è il primo che ha 3 istruzioni, di cui una eseguita e le altre due non ancora: verrà scelta per l'esecuzione la prima tra le non eseguite, ovvero la 4 (eseguendo così la traccia T_3 = { 4, 3, 5 }, si veda la figura 3.9e).
Il procedimento viene ripetuto ad oltranza, fino a quando non esistono più insiemi di operazioni eseguibili aventi istruzione non ancora eseguite.

Settando alcune impostazioni è possibile anche far sollevare tutti i fault possibili alle varie istruzioni in modo da simulare tutti i comportamenti anche in presenza di eccezioni: in questo caso, nei vari insiemi di istruzioni eseguibili saranno presenti più istanze della stessa istruzione e ognuna di esse solleverà un fault particolare.

Figura 3.9 (f, g): Simulazione della traccia T_1 `Fault` e relativa analisi

Per esempio, supponendo che l'istruzione con *ID = 4* sollevi i fault F_1 e F_2 (cosa non vera in realtà in quanto si tratta di un comando `createGroup` che può sollevare solamente un tipo di fault, `invalidCall`), gli insiemi della traccia T_1 = { 3, 4, 5 } sarebbero composti come mostrato nella figura 3.9f. Nel primo e nel secondo gruppo di istruzioni eseguibili è presente un sotto-insieme composto dalle varianti di esecuzione dell'istruzione 4 (4 *senza* fault, 4 *con* fault F_1 e 4 *con* fault F_2): appena una di queste varianti viene scelta tutto il sottoinsieme viene rimosso nell'insieme successivo (in quanto ogni istruzione può essere eseguita una sola volta).

Chiaramente però, quando si fa ad analizzare gli insiemi alla ricerca della variante che generi una nuova traccia (nella maniera descritta in precedenza) si avrà che il secondo insieme sarà composto da 4 istruzioni, una eseguita (la 4) e tre non ancora eseguite ($4F_1$, $4F_2$, 5) tra cui verrà selezionata la $4F_1$ per la successiva esecuzione. Per cui, la successiva traccia sarà: T_{2F} = { 3, $4F_1$ } (vedi figura 3.9g).

Nonostante l'esempio riportato faccia riferimento ad un caso particolare, il procedimento usato può essere astratto ed applicato a tutti i programmi SMoL possibili: c'è da dire però che il procedimento descritto necessita di piccole modifiche o di piccole precisazioni se ci si trova ad aver a che fare con *FaultHandler* o *InformationHandler*. Le modifiche/precisazioni da apportare al procedimento sono dovute alla semantica dei due comandi strutturati.

Per quanto riguarda il *FaultHandler*, "*calcolare tutte le tracce possibili*" significa dover simulare che un fault sollevato possa essere gestito da tutti i catch che soddisfano determinate caratteristiche (matching sintattico, catch con definito *faultName* variabile/opaque, ecc…).
Supponiamo, per esempio, di avere il seguente codice SMoL:

```
FaultHandler1
   Sequence
      pId = newPeer(myCred)3
   End Sequence

   catch(myExc)4
      empty()5
   cd = catch("invalidCall")6
      empty()7
   catchAll()8
      empty()9
End FaultHandler
```

In questo caso, le tracce ottenibili dal sorgente di esempio sono (ricordiamo che il comando `newPeer` può sollevare solo fault di tipo `invalidCall` -abbreviato con F_1-) (le istruzioni vengono riferite in base al loro ID):

T_{1F} = { 3 } T_{2F} = { 3 con "invalidCall", 5 }
T_{3F} = { 3 con "invalidCall", 7 }

Come è possibile notare fin da subito, le tracce possibili sono solo 3 in quanto il fault sollevabile da `newPeer` non verrà mai gestito dal `catchAll` finale (sarà sempre gestito dal catch precedente). Per permettere l'esecuzione di tutte le tracce, gli insiemi dei gruppi delle istruzioni eseguibili verranno arricchiti con l'*insieme dei catch eseguibili*: il funzionamento di questa nuova struttura è similare a quella descritta fino ad ora per le istruzioni eseguibili.

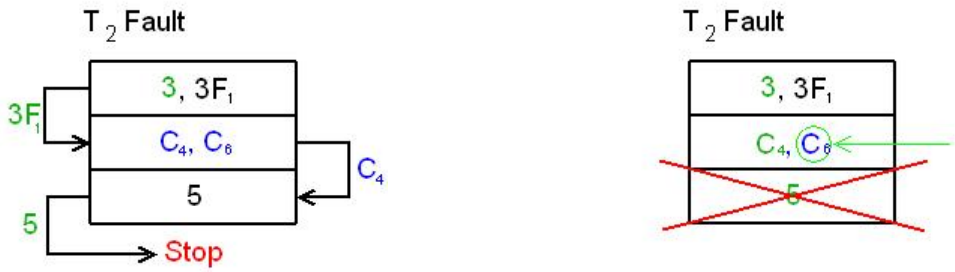

Figura 3.9 (h, i): Simulazione della traccia T_2 `Fault` e relativa analisi

Nel nostro caso, nella traccia T_{2F} avremmo l'insieme dei gruppi mostrato nella figura 3.9h: nel secondo stato vengono inseriti i catch che possono essere interessati nel gestire il fault sollevato (in cui non figura il `catchAll` -il modulo è in grado di riconoscere quali catch considerare o meno in automatico-) e subito dopo le istruzioni associate al catch eseguito. In modo analogo a quanto detto prima, durante l'analisi degli insiemi, il gruppo dei catch eseguibili viene rimosso quando non ci sono più catch eseguibili.

Per l'*InformationHandler* il discorso è un po' più complesso. Infatti, durante l'esecuzione del comando del *Ramo Principale* dell'*IH*[12], è possibile che vengano ricevuti un numero arbitrario di eventi, messaggi o timer dando luogo così a una lista potenzialmente infinita di tracce. Il simulatore quindi potrebbe ritrovarsi ad eseguire un numero infinito di tracce e questo chiaramente non può essere accettato: si è scelto, perciò, di simulare la ricezione di al massimo di un solo *input* esterno per ogni primitiva di guardia (cioè, di eseguire al massimo una volta, per ogni traccia, le guardie del comando strutturato *IH*).
Per esempio, il seguente (anche esso banale) codice SMoL

```
InformationHandler
   Sequence
      pId = newPeer(myCred)4
   End Sequence
```

[12] Con *IH* verrà abbreviato il comando *InformationHandler*.

```
   caller = receiveMessage(myMsg)6
      empty()7
   caller = receiveEvent(myEvent)9
      empty()10
End InformationHandler
```

(il programma non ha senso di per sé, ma è utile alla spiegazione perché è estremamente semplice) dà origine alle seguenti tracce (in grassetto sono evidenziate le guardie eseguite e con la scrittura "ID(*id*)" si indica "ID_istruzioneCopiata(*id_istruzioneOriginale*)")

T_1 = { **9**, 4, 13(*10*) }
T_2 = { **9**, 13(*10*), 4 }
T_3 = { 4 }
T_4 = { **6**, 4, 13(*7*) }
T_5 = { **6**, 13(*7*), 4 }
T_6 = { **9**, **6**, 4, 13(*10*), 16(*7*) }
T_7 = { **9**, **6**, 4, 16(*7*), 13(*10*) }
T_8 = { **9**, **6**, 13(*10*), 4, 16(*7*) }
T_9 = { **9**, **6**, 13(*10*), 16(*7*), 4 }
T_{10} = { **9**, **6**, 16(*7*), 4, 13(*10*) }
T_{11} = { **9**, **6**, 16(*7*), 13(*10*), 4 }
T_{12} = { **9**, 13(*10*), **6**, 4, 16(*7*) }
T_{13} = { **9**, 13(*10*), **6**, 16(*7*), 4 }
T_{14} = { **6**, **9**, 4, 13(*7*), 16(*10*) }
T_{15} = { **6**, **9**, 4, 16(*10*), 13(*7*) }
T_{16} = { **6**, **9**, 13(*7*), 4, 16(*10*) }
T_{17} = { **6**, **9**, 13(*7*), 16(*10*), 4 }
T_{18} = { **6**, **9**, 16(*10*), 4, 13(*7*) }
T_{19} = { **6**, **9**, 16(*10*), 13(*7*), 4 }
T_{20} = { **6**, 13(*7*), **9**, 4, 16(*10*) }
T_{21} = { **6**, 13(*7*), **9**, 16(*10*), 4 }

in cui non compare mai più di una volta l'esecuzione di una stessa guardia. Una volta fatta questa precisazione, il procedimento per eseguire tutte le tracce è quello descritto in precedenza.

3.3.4 Algoritmi di Visita e di Esecuzione

3.3.4.1 Algoritmo di Visita

L'algoritmo riportato è quello che viene implementato nel metodo *init()* della classe *Engine* (che risulta essere il metodo di inizializzazione dell'albero sintattico -descritto nel paragrafo precedente- il quale ha come risultato lo stato S_0).

La signature dell'algoritmo non ricalca precisamente quella del metodo: questo è dovuto alla scelta implementativa per la quale si è optato di nascondere completamente la struttura ad albero all'utente. Per cui l'intera struttura albero sarà contenuta all'interno di una istanza di *Engine* e sarà premura di questa ultima invocare il metodo *visitaNodo* con i giusti

parametri. Inoltre, per lo stesso motivo di "trasparenza", il metodo *init()* restituirà liste di *Comandi* invece che di *Nodi*.

Nonostante tutto, lo pseudo-codice che segue descrive in modo rigoroso le operazioni eseguite da *init()*.

```
// metodo pubblico, crea lo stato iniziale a partire dall'albero
inizializza(nodo radice ROOT)
{
  crea una nuova lista STATO;
  visitaNodo(ROOT, STATO);
  return STATO;
}

// metodo ricorsivo privato di visita
visitaNodo(nodo da visitare N, lista di foglie LISTA)
{
   if(N è una foglia) aggiungi N a LISTA;
   else
      switch N.tipoDiComando

         "Assign": foreach (Figlio in N.figli)
                   {
                      if(Figlio non è stato eseguito[13])
                      {
                         // i figli sono tutte foglie (FROM=TO)
                         visitaNodo(Figlio, LISTA);
                         return;
                       }
                   }
                   break;

         "Sequence": foreach (Figlio in N.figli)
                   {
                      if(Figlio non è stato eseguito)
                      {
                         visitaNodo(Figlio, LISTA);
                         return;
                       }
                   }
                   break;

         "Pick" or "Flow":
                   foreach (Figlio in N.figli)
                   {
                      visitaNodo(Figlio, LISTA);
                   }
                   break;

         "InformationHandler":
                   foreach (Figlio in N.figli)
                   {
                      visitaNodo(Figlio, LISTA);
                   }
                   break;
```

[13] Il controllo si basa sulla variabile *stato* del nodo: "*Nodo N non è stato eseguito*" equivale a N.*stato* ≠ *executed*.

```
"FaultHandler":
            visitaNodo(N.figli[0], LISTA);
            break;

"While"; if(!richiediEsecuzioneComando(N)[14])
         {
            Segno N come eseguito;
            if(padre è un "Sequence") {eseguiNodo(N.genitore,
               LISTA)[15];}
         }
         else { visitaNodo(N.figli[0], LISTA); }
         break;

"RepeatUntil":
         visitaNodo(N.figli[0], LISTA);
         break;

"If"; if(richiediEsecuzioneComando(N))
         {
            visitaNodo(N.figli[0], LISTA);
         }
         else
         {
            if(esiste ramo ELSE[16])
            {
               visitaNodo(N.figli[1], LISTA);
            }
            else
            {
               Segno N come eseguito;
               if(padre è un "Sequence")
                  { eseguiNodo(N.genitore, LISTA)[17]; }
               return;
            }
         }
         break;
```

[14] Funzione booleana che permette all'utente di scegliere le modalità di esecuzione dei comandi *While*, *RepeatUntil*, *If*.

[15] A differenza degli altri comandi, *While* e *If* al momento di essere visitati devono essere indirizzati in qualche modo dall'utente in quanto chiedono esplicitamente la modalità di esecuzione (se eseguire o meno i due comandi e, conseguentemente, visitarli o meno); la scelta dell'utente influenza la totale esecuzione del programma, avendo ripercussioni anche sull'esecuzione del padre: nel seguente esempio

```
InformationHandler
   Sequence
      msg = opaque
      While(…)
         Sequence
            callerId = receiveMessage(…)
            replay(callerId, "op", msg)
         End Sequence
      End Sequence

   ReceiveEvent(…)
      // fa qualcosa…
End InformationHandler
```

supponendo che il *While* venga eseguito completamente, se si tornasse a visitare -e basta- il padre *Sequence*, esso non verrebbe eseguito -ma solo visitato- (quindi, nonostante *Sequence* non abbia più figli marcati "*DA_exec*", non viene etichettato "*executed*") e conseguentemente l'intero *InformationHandler* rimarrebbe in sospeso in quanto il *Ramo Principale* non terminerebbe.

[16] Per come è strutturato il nodo *if*, se esiste ramo *Else* esiste N.*figli*[1].

[17] Vedi nota N°15.

```
}
```

3.3.4.2 Algoritmo di Esecuzione (Funzione di Transizione - Next)

Analogo discorso per quanto riguarda il metodo *next()* della classe *Engine*: anche esso, per motivi di "trasparenza" all'utente, restituirà liste di *Comandi* e non di *Nodi* ma la procedura è fedele allo pseudo-codice riportato.

```
// metodo pubblico, esegue la transizione da uno stato all'altro
next (nodo da eseguire EXEC, lista di foglie STATO, esito esecuzione
        ESITOEXEC, fault riscontrato FAULTNAME)
{
   verifico che EXEC faccia parte di un ramo "Pick": in caso affermativo,
   rimuovo tutti i rami non scelti[18].

   verifico che EXEC sia una guardia/evento di un InformationHandler: in
   questo caso copio il sotto albero ed eseguo la copia, lasciando intatto
   il ramo originale[19].

   eseguiNodo(EXEC, STATO, ESITOEXEC, FAULTNAME);

   return STATO;
}

// metodo privato ricorsivo, esegue un singolo comando alla volta
eseguiNodo (nodo da eseguire EXEC, lista di foglie LISTA, esito
        esecuzione ESITOEXEC, fault riscontrato FAULTNAME)
{
   // sono sul padre della radice, ho terminato con l'esecuzione
   if(EXEC = null)
   {
      if(!ESITOEXEC) {si è verificato un fault che nessuno ha gestito}

      pulisco la lista rimuovendo tutti gli elementi.

      return;
   }

   // sto tornando indietro a causa di un fault e il nodo che vado ad eseguire non è un FaultHandler
   if((!ESITOEXEC)&&(EXEC.tipoDiComando ≠ "FaultHandler"))
   {
      Segno EXEC come eseguito;
```

[18] Il controllo viene eseguito sul "nonno" del nodo EXEC: dal momento che la guardia di un ramo di *Pick* occupa la posizione NONNO.*figli[k].figli[0]*, si verifica che EXEC occupi proprio questa posizione rispetto al primo nodo *Pick* che si incontra percorrendo il ramo a ritroso, da EXEC a radice. In caso affermativo, tutti i rami di *Pick* che non siano quello a cui EXEC appartiene vengono marcati come "*executed*" ed eventuali primitive, presenti nello stato, rimosse. La funzione è implementata nel metodo privato *checkPickGuard*.

[19] La verifica consiste nel visitare il NONNO del nodo EXEC: se questo ultimo è un nodo *IH*, verifico se EXEC è nella posizione NONNO.*figli[j].figli[0]* con $j > 0$. In tal caso EXEC è una guardia di un evento (tipo *Wait*, *ReceiveMessage* o *ReceiveEvent*): se la primitiva *non* è un *Wait* senza il parametro *repeatEvery* devo copiare il sotto-albero a cui appartiene (a partire dal padre di EXEC, ovvero NONNO.*figli[j]* con $j > 0$) ed eseguire la copia (e lasciare intatta l'originale), altrimenti (se siamo di fronte ad una primitiva tipo *Wait(for)* o *Wait(until)*) eseguo direttamente il ramo originale. NB: Nel caso di evento verificatosi e di albero duplicato, il nodo che viene eseguito non è più EXEC (scelto dall'utente) ma EXEC_copia (la cui esecuzione ricalca fedelmente quella che avremmo avuto eseguendo EXEC) La funzione è implementata in *checkIHGuard*.

```
    // eseguo il padre
    eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULNAME);
    return;
}

switch EXEC.tipoDiComando

    "Assign":

          foreach (Figlio in EXEC.figli)
          {
             if(Figlio non è stato eseguito)
             {
                visitaNodo(Figlio, LISTA);  // i figli sono tutte foglie (TO=FROM)
                return;

             }
          }

          // ho finito i figli da eseguire/visitare
          Segno EXEC come eseguito;
          // eseguo il padre
          eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);

          break;

    "Sequence":

          foreach (Figlio in EXEC.figli)
          {
             if(Figlio non è stato eseguito)
             {
                // il primo figlio non eseguito che trovo, lo devo visitare
                visitaNodo(Figlio, LISTA);
                return;
             }
          }

          // ho finito i figli da eseguire/visitare
          Segno EXEC come eseguito;
          // eseguo il padre
          eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);

          break;

    "Flow":

          foreach (Figlio in EXEC.figli)
          {
             // se esiste un figlio non ancora eseguito, devo aspettare che termini
             if(Figlio deve ancora essere eseguito) { return; }
          }

          // ho finito i figli da eseguire/visitare
          Segno EXEC come eseguito;
          // eseguo il padre
          eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);

          break;

    "Pick":
```

```
            // ho finito i figli da eseguire/visitare (qui basta che ne sia stato eseguito uno solo)
            Segno EXEC come eseguito;
            // eseguo il padre
            eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);

            break;

      "InformationHandler":

            if(è stato eseguito il ramo principale[20])
            {
               rimuovo tutti i nodi "guardia-evento" dalla LISTA

               // nel nodo InformationHandler, possono esserci sotto-alberi originali e non in
               esecuzione, lo devo capire
               if(ci sono dei sotto-alberi "evento" già iniziati[21])
               {
                  // non faccio niente, devo aspettare la loro terminazione
                  // ma ci possono essere rami evento terminati (quello per cui sto ritornando)
                  cerco il ramo j che è stato appena eseguito in
                  EXEC.figli[22]

                  se EXEC.figli[j] ≠ originale, rimuovo il ramo
                  EXEC.figli[j] dai figli del nodo InformationHandler

                  return;
               }
               else
               {
                  // non ci sono rami iniziati ma ci possono essere rami evento terminati
                  // (per esempio, quello per cui sto lavorando)
                  cerco il ramo j che è stato appena eseguito in
                  EXEC.figli[23]
```

[20] Verifica che EXEC.*figli[0]* = *executed*. Da notare che in questo ramo *if* ci entriamo anche nel caso di ritorno dall'esecuzione di rami eventi iniziati prima che finisse l'esecuzione del *Ramo Principale* ma finiti dopo questa ultima.

[21] Il confronto avviene su EXEC.*figli[j]* con $j > 0$: ogni nodo ha un metodo *hasFiglioOriginale* il quale è in grado di dire se un nodo-figlio è stato inserito in fase di creazione dell'albero nel nodo-padre oppure se è stato inserito a *runtime*. Attraverso l'uso di questo metodo sui figli di EXEC, si può arrivare ad avere due possibilità: se il *j*-esimo figlio risulta essere stato inserito a *runtime*, si verifica se è marcato "*DA_exec*" oppure "*executed*" (il primo caso significa che il *j*-esimo ramo è stato avviato ma non ancora terminato, il secondo che il ramo è stato eseguito completamente ed è pronto per essere rimosso). Altrimenti, se risulta essere un nodo aggiunto in fase di creazione dell'albero -quindi, risulta essere un nodo originale- si verifica che la guardia associata sia una primitiva *Wait* ed abbia o meno il parametro *repeatUntil*: in caso negativo (quindi la primitiva ha come parametro *for* o *until*) si verifica lo stato sia del comando *Wait* che del padre *Sequence* (direttamente sul ramo/figlio originale di EXEC):

- se Wait.*stato* = *DA_exec* & Sequence.*stato* = *DA_exec* → il sotto-albero non è stato avviato;
- se Wait.*stato* = *executed* & Sequence.*stato* = *DA_exec* → il sotto-albero è stato avviato ma non terminato;
- se Wait.*stato* = *executed* & Sequence.*stato* = *executed* → il sotto-albero è stato completamente eseguito.

[22] Si rende necessaria la ricerca di una eventuale copia completamente eseguita e la sua rimozione in quanto si raggiunge questa parte di algoritmo anche quando si ha il *Ramo Principale* eseguito, un ramo-evento da eseguire e un ramo-evento di cui si sta terminando l'esecuzione (per il quale sto eseguendo il percorso a ritroso dell'albero).

[23] Se si sta eseguendo *l'InformationHandler* di ritorno da un ramo evento (l'unico del nodo ad essere stato avviato), con il *Ramo Principale* già eseguito, devo rimuoverlo in quanto copia (nel caso sia stato eseguito un ramo originale non viene compiuta nessuna operazione sull'albero).

```
                se EXEC.figli[j] ≠ originale, rimuovo il ramo
                EXEC.figli[j] dai figli del nodo InformationHandler

                // ho finito con l' esecuzione/visita del nodo
                Segno EXEC come eseguito;
                // eseguo il padre
                eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);
            }
        }
        else // ramo non principale eseguito, ho eseguito sicuramente un ramo evento
        {
            cerco il ramo j che è stato appena eseguito in EXEC.figli

            se EXEC.figli[j] ≠ originale, rimuovo il ramo EXEC.figli[j]
            dai figli del nodo InformationHandler
        }

        break;

    "FaultHandler":
        // l'esecuzione è andata bene, torno sul padre (NB: in tutti gli altri comandi
        controllavo se l'esecuzione era fallita, qui controllo il contrario)
        if(ESITOEXEC)
        {
            Segno EXEC come eseguito;
            // eseguo il padre
            eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);

            return;
        }

        // si sono verificati dei problemi e devo capirne la provenienza

        if(!fault proviene dal ramo principale[24])
        {
            // è fallito un catch e passo tutto al padre

            Segno EXEC come eseguito;
            // eseguo il padre
            eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);
        }
        else
        {
            // è fallito il ramo principale, cerco di gestirlo

            rimuovo tutti i comandi ragiungibili da EXEC.figli[0] che
            sono presenti nella LISTA[25];

            if(esiste "nodoCatch" appropriato[26])
            {
                // visito il sotto-albero relativo al catch
                visitaNodo(nodoCatch, LISTA);
            }
```

[24] Quando si arriva ad eseguire questo controllo, significa che sicuramente il *Ramo Principale* è fallito (altrimenti l'esecuzione si sarebbe fermata sul precedente *if* in quanto ESITOEXEC = *true*) e che quindi EXEC.*figli[0]* = *executed*. Può darsi anche di star eseguendo il nodo in fase di ritorno dovuto ad un fault verificatosi in un catch: in questo caso, EXEC.*figli[1]* (FH ha solo due figli) è marcato come "*executed*" ed ESITOEXEC è *false*: in questo caso non c'è da fare niente se non da spostarsi sul padre.

[25] Operazione eseguita dal metodo privato *rimuoviFoglieRaggiunte*.

[26] L'esistenza di un catch appropriata viene eseguita dalla funzione ricorsiva *controlloEsistenzaCatch*.

```
            else
            {
                // il catch appropriato non esiste, allora passo ad eseguire il padre

                Segno EXEC come eseguito;
                // eseguo il padre
                eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);
            }
        }

        break;

    "Catch"[27];
        // qui non importa fare i controlli sulla corretta esecuzione o sulla presenza di fault
        Segno EXEC come eseguito;
        // eseguo il padre
        eseguiNodo(EXEC.genitore, LISTA,ESITOEXEC, FAULTNAME);

        break;

    "CatchAll";
        // qui non importa fare i controlli sulla corretta esecuzione o sulla presenza di fault
        Segno EXEC come eseguito;
        // eseguo il padre
        eseguiNodo(EXEC.genitore, LISTA,ESITOEXEC, FAULTNAME);

        break;

    "While" or "RepeatUntil";

        // devo scegliere se rieseguire il ciclo oppure no

        if(!richiediEsecuzioneComando(EXEC))
        {
            Segno EXEC come eseguito;
            // eseguo il padre
            eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);
        }
        else
        {
            marco tutto il sotto-albero del ciclo "While" come
            "DA_exec"[28]

            visitaNodo(EXEC.figli[0], LISTA);
        }

        break;

    "If";
        // qui non importa fare i controlli sulla corretta esecuzione o sulla presenza di fault
```

[27] A differenza con la procedura di visita, qui i nodi *Catch* e *CatchAll* vengono eseguiti in maniera specifica: questo è dovuto dal fatto che quando viene visitato un *FaultHandler*, i vari catch vengono ignorati in quanto si spera *(!)* che l'esecuzione del *Ramo Principale* non sollevi nessun fault (anche in caso di fault sollevato, viene visitato, se esiste, solo il ramo catch in grado di catturare il fault, mentre gli altri continuano ad essere ignorati - scelta implementata nella funzione *controlloEsistenzaCatch* -); durante l'esecuzione, invece, dovendo andare a ritroso sull'albero, ci si potrebbe imbattere su di un nodo *Catch*/*CatchAll* (nel caso di terminazione dell'esecuzione di un ramo catch) ed in questo caso bisogna poter continuare il percorso semplicemente spostandosi sul padre del nodo stesso.

[28] La marcatura di tutto il sotto-albero del ciclo viene eseguita tramite l'invocazione del metodo *setResetTree*.

```
        Segno EXEC come eseguito;
        // eseguo il padre
        eseguiNodo(EXEC.genitore, LISTA,ESITOEXEC, FAULTNAME);

        break;

default:

        // qui va inserita la corretta esecuzione delle primitive

        Segno EXEC come eseguito;
        Rimuovo la foglia/primitiva EXEC dallo stato LISTA
        // eseguo il padre
        eseguiNodo(EXEC.genitore, LISTA, ESITOEXEC, FAULTNAME);

        break;
```

Capitolo4 Il prototipo al lavoro

In questo capitolo verranno mostrati alcuni *screenshots* dell'applicazione, con relativi ingrandimenti sulle singole parti che compongono la schermata, affinché sia descritto il più chiaramente possibile il comportamento del programma.

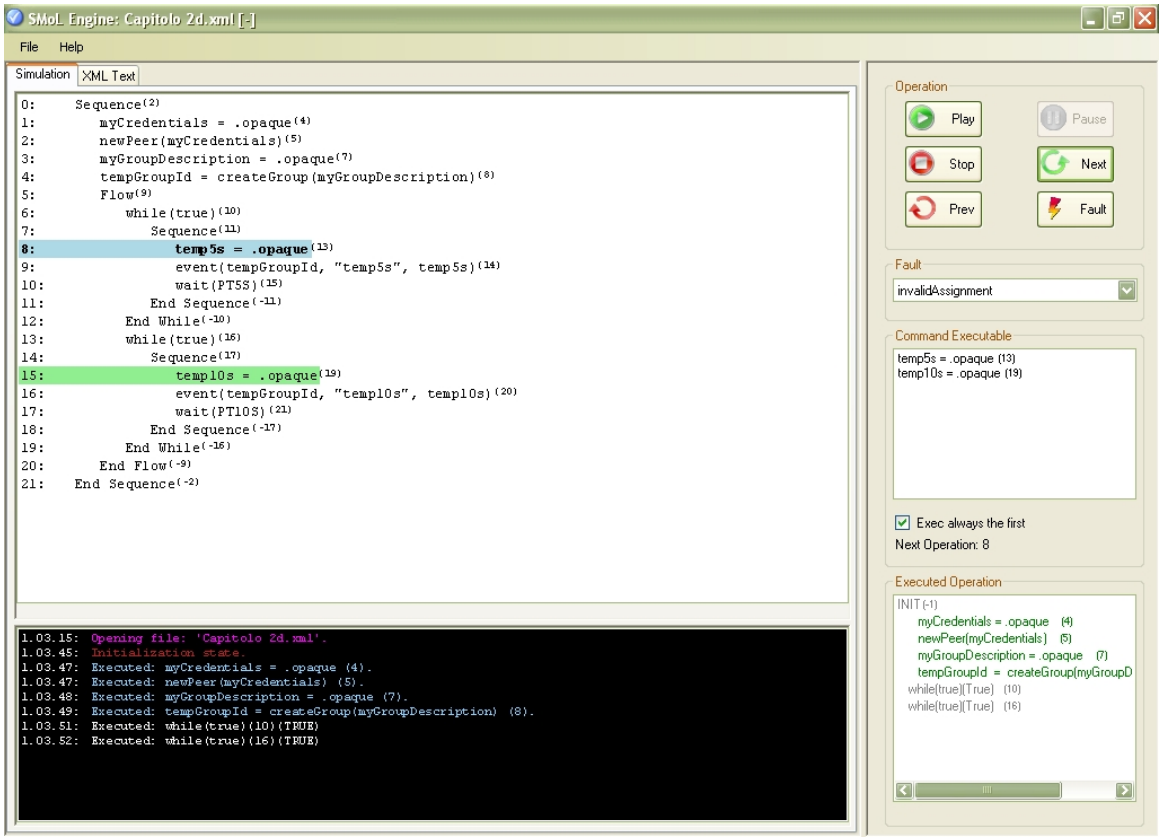

Figura 4.1 : Esecuzione di un programma SMoL

Come possiamo vedere nella Figura 4.1 la schermata dell'applicazione è scomponibile in due parti: nella parte di sinistra possiamo vedere la rappresentazione SMoL del codice sorgente `xml` su cui si stanno eseguendo le simulazioni e sotto una console in cui si è in grado di visualizzare tutte le operazioni svolte, istante per istante, mentre nella colonna di destra, oltre a rendere possibile l'accesso a tutte le funzionalità offerte dal simulatore, è possibile visualizzare lo stato complessivo della simulazione.

La grafica è stata progettata affinché l'utente, con un semplice colpo d'occhio, possa capire a che punto si trovi l'esecuzione della simulazione: sulla lista di comandi SMoL, in ogni momento, vengono segnalate le istruzioni potenzialmente eseguibili (in verde), l'istruzione selezionata per l'esecuzione (in celeste) e le eventuali istruzioni fallite (in rosso); quando, invece, il simulatore propone una scelta all'utente (se eseguire o meno un *if*, un *while* o un *repeatUntil* o se far catturare o meno un fault da un *catch*), oltre al *messageBox* di scelta viene evidenziato il comando in questione con un rettangolo in rosso (vedi figura 4.2).

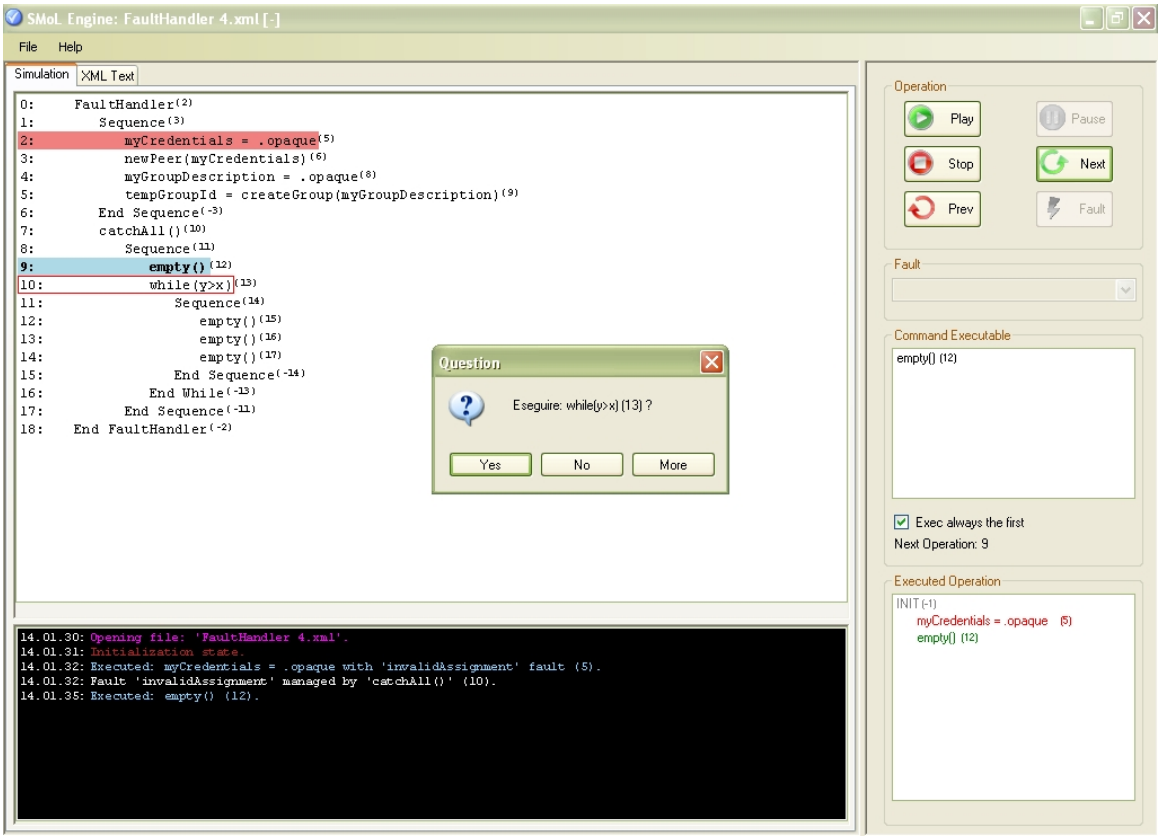

Figura 4.2: Esecuzione con istruzione fallita e richiesta di esecuzione *while*

Quando, nella simulazione, si rende necessario la simulazione di un *InformationHandler* (e delle eventuali ricezioni di *input* provenienti dall'ambiente -messaggi/eventi/allarmi-), il simulatore è in grado di "sdoppiare" il codice: infatti, come già ampliamente descritto nei capitoli precedenti, l'*InformationHandler* deve essere in grado di gestire la ricezione di uno o più *input* (anche dello stesso tipo) ed eseguirne il codice associato, creando più "istanze" di blocchi di istruzioni. Per cercare di visualizzare questo comportamento, si è optato per copiare il codice vero e proprio all'interno della lista delle istruzioni mostrata, dando alla copia un carattere e un colore tale che risultasse subito evidente che non si trattasse di codice

originale (come mostrato nella figura 4.3 sottostante, il codice copiato viene visualizzato in corsivo e colorato di viola).

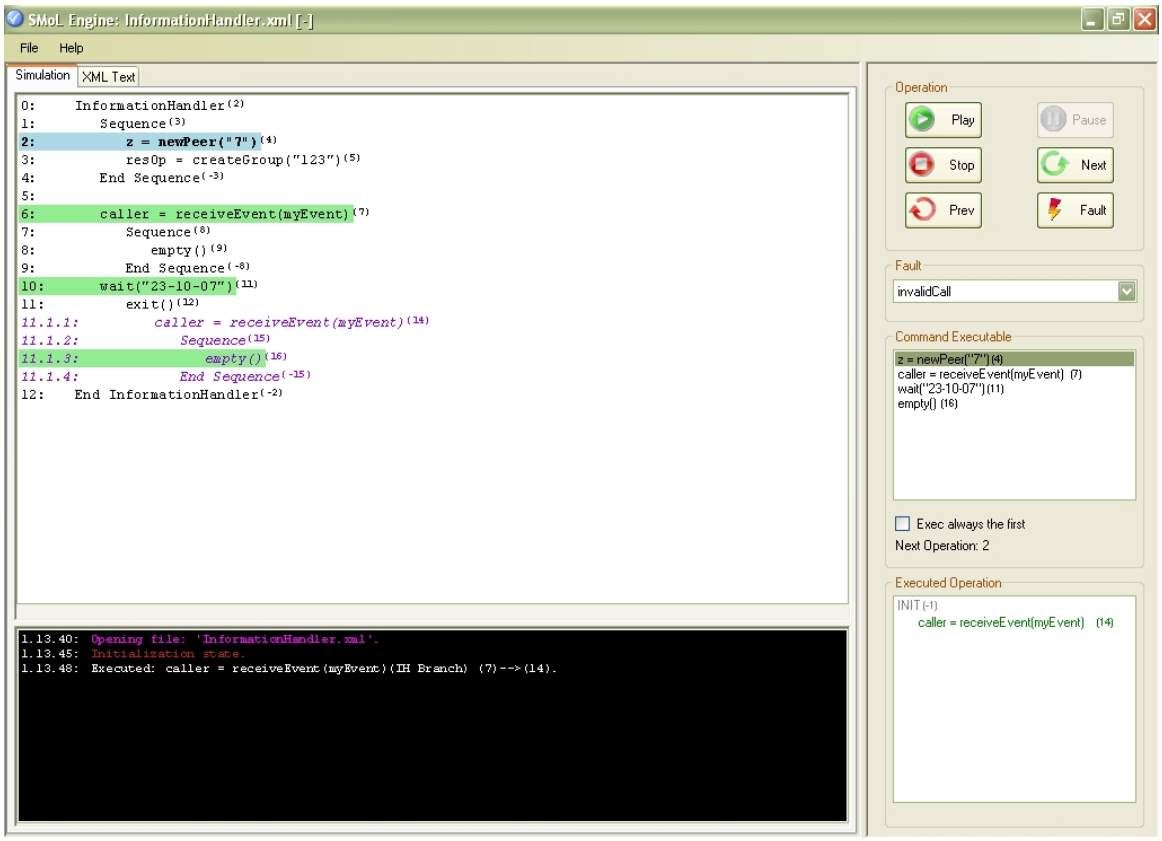

Figura 4.3: Esempio di esecuzione di *InformationHandler*

Il codice copiato viene evidenziato anche con l'uso di una numerazione specifica, posta in cima alla riga di ogni istruzione copiata. In genera la numerazione normale cresce direttamente con l'aumentare del numero di istruzioni: quando si copia il codice, il formato della numerazione classico "*x*" viene modificato in "*x.y.z*" dove "*x*" è un valore costante (che dipende dalla riga in cui si comincia ad inserire il codice copiato), "*y*" specifica il numero della "istanza" in questione (nel caso di *N* copie ottenute dalla ricezione di *N input* dello stesso tipo si è in grado di capire di quale "istanza" un comando fa parte) e "*z*" indica una numerazione interna alla copia (chiaramente, in caso di *InformationHandler* annidati, il formato della numerazione si adatterà all'occorrenza).
E' importante notare che ogni istruzione all'interno della lista è fornito di un *identificatore* posto in alto sulla destra del comando: tale *ID* (che compare anche in tutte le altre componenti della finestra in cui si fa riferimento a comandi SMoL) serve per

- identificare in modo univoco il comando all'interno dell'albero sintattico,

- per relazionare istanze di oggetti *Comando* a oggetti di tipo *Nodo* (l'*engine* ingloba al suo interno la struttura dati albero e la nasconde al modulo grafico al quale passa una rappresentazione dell'albero definita tramite una lista di comandi).

Gli *identificatori* significativi hanno tutti valori positivi: gli *ID* negativi sono privi di significato in quando i comandi contrassegnati da tali ID sono quelli usati per chiudere un comando strutturato (ad esempio `Sequence`$^{(2)}$ `. . . End Sequence`$^{(-2)}$) e servono solo dal punto di vista grafico.
Infine, è possibile salvare la rappresentazione SMoL del programma (sia la versione statica che a *runtime*, con tanto di eventuali copie di rami evento degli *InformationHandler*), contenuta nella lista visualizzata a video, in un file di testo `txt`.

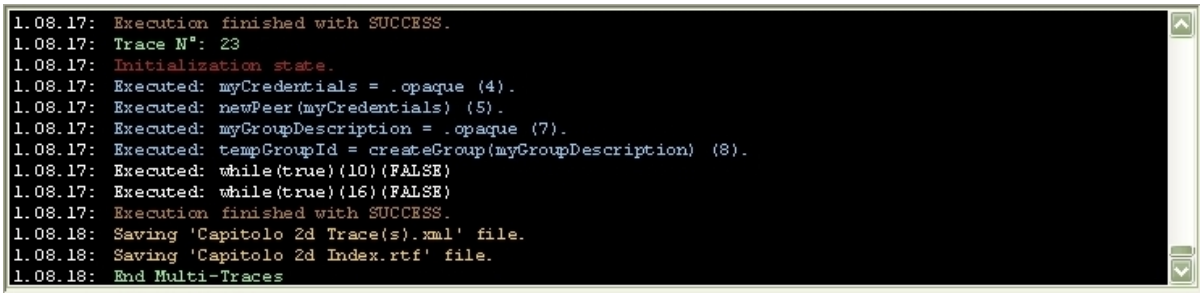

Figura 4.4: Console applicazione

La console serve per avere sempre sotto occhio tutte le operazioni svolte dal simulatore: ad ogni tipologia di comunicazione viene associato un colore specifico affinché possa essere letta in modo rapido e chiaro. Inoltre è possibile salvare il suo contenuto in un file di testo `txt` (in modo analogo a quanto detto prima per la lista dei comandi della rappresentazione SMoL).

Figura 4.5a: Gruppi *Operation* e *Fault* (I)

Figura 4.5b: Gruppi *Operation* e *Fault* (II)

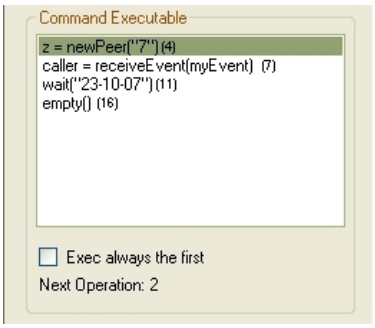

Figura 4.5c: Gruppo *Command Executable* **Figura 4.5d: Gruppo *Executed Operation***

La colonna di destra è suddivisibile in quattro gruppi: "*Operation*", "*Fault*", "*Command Executable*" e "*Executed Operation*". Nei primi due gruppi sono racchiusi i punti di accesso per la maggior parte delle funzionalità del sistema (figura 4.5a e 4.5b): sono presenti infatti diversi *buttons* (ognuno dei quali richiama una specifica funzionalità) e il *combobox* dove è possibile scegliere il fault da sollevare in base alla istruzione selezionata. Nel terzo gruppo (figura 4.5c) vengono mostrate, istante per istante, tutte le potenziali istruzioni eseguibili: l'utente, deselezionando il *checkbox* "*Exec always the first*", può selezionare, a suo piacere, una di queste operazioni per l'esecuzione (altrimenti, se il *checkbox* rimane selezionato, viene eseguito sempre il primo comando tra quelli mostrati). Infine, nel quarto ed ultimo gruppo (Figura 4.5d) vengono elencate, man mano che sono eseguiti, tutti i comandi SMoL che il prototipo simula: i comandi vengono elencati nell'ordine di esecuzione e vengono colorati in maniera differente a seconda del tipo (primitiva o ciclo -*if*, *while*, *repeatUntil*-) e a seconda se il comando è stato fatto fallire o meno (rosso se è fallito, verde altrimenti). Su questo elenco è inoltre possibile, cliccando due volte su una delle primitive visualizzate, tornare indietro con l'esecuzione della traccia, annullando l'esecuzione degli ultimi *K* comandi.

Per poter avere accesso alla funzionalità di generazione automatica di tutte le tracce possibili di un programma SMoL, è necessario dover cliccare con il tasto destro del mouse sul pulsante Play (nel gruppo *Operation*) e successivamente selezionare la voce opportuna: automaticamente il pulsante cambierà colore, verranno disabilitate tutte le altre funzionalità e sarà possibile usufruire del servizio (Figura 4.5b).

Subito prima di iniziare la fase di generazione automatica di tutte le tracce, all'utente è richiesto di settare alcuni parametri sulla modalità di svolgimento della operazione: tale richieste riguardano la possibilità

- di considerare o meno tutte quelle tracce ottenibili facendo fallire ognuna delle primitive del programma (facendo sollevare ad ogni primitiva tutti i fault che può lanciare),
- di salvare, in un unico file `xml`, il resoconto complessivo dell'intera operazione (ovvero, di salvare tutte le tracce in un unico file `xml`, senza i corrispettivi file `sls`),
- di salvare le tracce singolarmente, un file `sls` e uno `xml` per ogni traccia generata.

Queste opzioni vengono settate tramite la *form* della figura 4.6

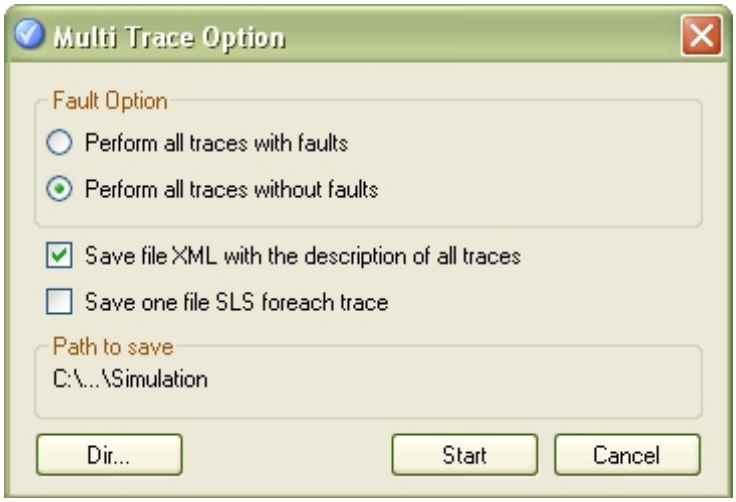

Figura 4.6: Form settaggio opzioni *Multi Trace*

Al termine dell'operazione di tracciamento di tutte le simulazioni possibili, l'applicazione, in base alle opzioni settate, crea un resoconto delle operazioni svolte: tale resoconto è comprensivo di un file `rtf` (contenente una semplice lista delle tracce calcolate, di facile lettura per l'utente), di un file `xml` (in cui sono descritte le tracce salvate in un formato tale da permetterne il riuso da parte di eventuali altre applicazioni esterne al simulatore) e di una serie di file `sls`/`xml` (ognuno dei quali descrive una singola traccia).

All'interno del file `rtf` vengono elencate le operazioni svolte in questa formato:

'*GenericTree.xml*' Traces Description

18/02/2008 @ 21.29: Trace N°: 1 [*GenericTree Trace_1.sls*]
18/02/2008 @ 21.29: Initialization state.
18/02/2008 @ 21.29: *Executed: while(x<k)(2)(TRUE)*
18/02/2008 @ 21.29: Executed: y = "12" (5).

```
18/02/2008 @ 21.29: Executed: x = "7" (6).
18/02/2008 @ 21.29: Executed: w.v = z (7).
18/02/2008 @ 21.29: Executed: caller = receiveMessage("SMS")(Pick Branch) (10).
18/02/2008 @ 21.29: Executed: wait(P3DT10H)(Pick Branch) (13).
18/02/2008 @ 21.29: Executed: empty() (15).
18/02/2008 @ 21.29: Executed: newPeer(myCred) (16).
18/02/2008 @ 21.29: Executed: empty() (17).
18/02/2008 @ 21.29: Executed: if(y>x)(35)(TRUE)
18/02/2008 @ 21.29: Executed: empty() (34).
18/02/2008 @ 21.29: Executed: empty() (37).
18/02/2008 @ 21.29: Executed: empty() (39).
18/02/2008 @ 21.29: Executed: newPeer(myCred) (38).
18/02/2008 @ 21.29: Executed: while(x<k)(2)(FALSE)
18/02/2008 @ 21.29: Execution finished with SUCCESS.
------------------ ------------------------------------------------------------
. . . . . .
. . . . . .
. . . . . .
------------------ ------------------------------------------------------------
18/02/2008 @ 21.36: Trace N°: 654 [GenericTree Trace_654.sls]
18/02/2008 @ 21.36: Initialization state.
18/02/2008 @ 21.36: Executed: while(x<k)(2)(TRUE)
18/02/2008 @ 21.36: Executed: y = "12" with 'invalidAssignment' fault (5).
18/02/2008 @ 21.36: FAULT NOT MANAGED: 'INVALIDASSIGNMENT' (NEXT 5)
18/02/2008 @ 21.36: Execution FAILED.
------------------ ------------------------------------------------------------
18/02/2008 @ 21.36: Trace N°: 655 [GenericTree Trace_655.sls]
18/02/2008 @ 21.36: Initialization state.
18/02/2008 @ 21.36: Executed: while(x<k)(2)(FALSE)
18/02/2008 @ 21.36: Execution finished with SUCCESS.
------------------ ------------------------------------------------------------

Total Traces: 655.
```

All'inizio del file viene indicato il nome del sorgente `xml` su cui si è eseguita la computazione (tra apici singoli) e per ognuna delle tracce salvate, viene indicato:

- il numero di riferimento della traccia calcolata;
- il file `sls` su cui è stata salvata -in verde- (se si è scelto di salvare un file `sls` per ognuna delle tracce calcolate, altrimenti il valore tra parentesi quadre sarà "-");
- tutte le primitive eseguite (primitive semplici in nero, cicli *While*/*RepeatUntil*, comandi *If*, comandi *Catch*/*CathAll*, guardie di *InformationHandler* e di *Pick* in blu);
- eventuali errori riscontrati o fault non gestiti -in rosso-;
- l'esito complessivo della traccia (*SUCCESS*/*FAILED*).

In questo modo, colui che usa il prototipo, ha già una visione semplice e concisa delle tracce elaborate dall'applicazione, senza doversi per forza leggere i vari file `xml`, che, specialmente con un numero elevato di tracce computate, risulterebbe essere un'operazione abbastanza pesante.

Insieme al file `rtf` sopra descritto, il simulatore è in grado di offrire anche un resoconto in `xml`: questo file può essere poi passato come *input* ad altre applicazioni in grado di elaborare ulteriormente le informazioni ottenute. Il formato del file è molto semplice:

```
<TraceList>
   <Trace Number="1" GeneralOutcome="SUCCESS">
      <Com name="While" text="while(x&lt;k)" condition="True" ID="2">
         <code>
            // codice xml comando SMoL
         </code>
      </Com>
      <Com name="ASSIGNprimitive" text= "y = '12'" outcome="SUCCESS" exc="" ID="5">
         <code>
            // codice xml comando SMoL
         </code>
      </Com>
      <Com name="ASSIGNprimitive" text= "x = '7'" outcome="SUCCESS" exc="" ID="6">
         <code>
            // codice xml comando SMoL
         </code>
      </Com>
      . . . . . .
      . . . . . .
   </Trace>
   . . . . . .
   . . . . . .
   . . . . . .
   <Trace Number="654" GeneralOutcome="FAILED">
      <Com name="While" text="while(x&lt;k)" condition="True" ID="2">
         <code>
            // codice xml comando SMoL
         </code>
      </Com>
      <Com name="ASSIGNprimitive" text= "y = '12'" outcome="FAILED"
      exc="invalidAssignment" ID="5">
         <code>
            // codice xml comando SMoL
         </code>
      </Com>
   </Trace>

   <Trace Number="655" GeneralOutcome="SUCCESS">
      <Com name="While" text="while(x&lt;k)" condition="False" ID="2">
         <code>
            // codice xml comando SMoL
         </code>
      </Com>
   </Trace>

//
// Totals Traces: 655
//
</TraceList>
```

All'interno del file `xml`, per ogni traccia simulata (compresa tra i tag `<Trace>…</Trace>`), oltre ad un numero di riferimento e ad un'indicazione sull'esito complessivo, vengono elencati i singoli comandi eseguiti (ognuno dei quali è rappresentato con il tag `<Com>`).
Sulla esecuzione di questi ultimi, per ognuno di essi, vengono riportate delle informazioni di carattere generale, tra cui figurano:

- il tipo di primitiva eseguita (attributo `name` di `<Com>`);
- la rappresentazione SMoL del comando in cui i caratteri speciali (tipo "<" o ">") vengono sostituiti dalle corrispettivi traduzioni definite in XPATH 2.0 ("*<*" e "*>*" nel nostro caso) (attributo `text` di `<Com>`);
- solo per le primitive, l'esito della esecuzione e il fault sollevato -nel caso di esecuzione corretta, il campo del fault è vuoto- (attributo `outcome` e `exc` di `<Com>`);
- solo per *if*, *While* e *RepeatUntil*, la condizione booleana del comando (attributo `condition` di `<Com>`);
- l'ID dell'istruzione (attributo `ID` di `<Com>`);
- il codice xml vero e proprio del comando (compreso tra i tag `<code>…</code>`).

Le tracce possono essere anche salvate singolarmente, in formato `sls` ed in formato `xml`: i file `sls` creati sono array di bytes (contengono fisicamente delle strutture dati di tipo `List`) mentre i file `xml` generati hanno lo stesso formato descritto in precedenza.
Durante l'esecuzione di tutte le possibili tracce (operazione che può richiedere anche una notevole quantità di tempo), l'utente è sempre in grado di interrompere la procedura in qualsiasi momento, tramite la *form* proposta in figura 4.7 la quale, oltretutto, mostra costantemente il numero di tracce calcolato.

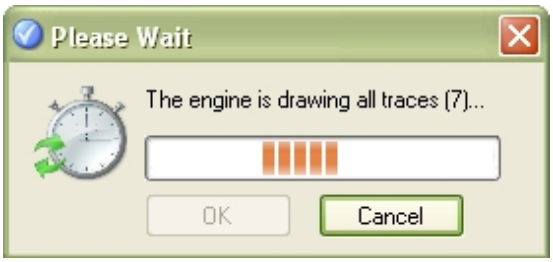

Figura 4.7: Form di attesa completamento della simulazione di tutte le tracce

Alla fine dell'intera operazione, viene visualizzato il numero di tracce complessivamente computate dal prototipo (figura 4.8).

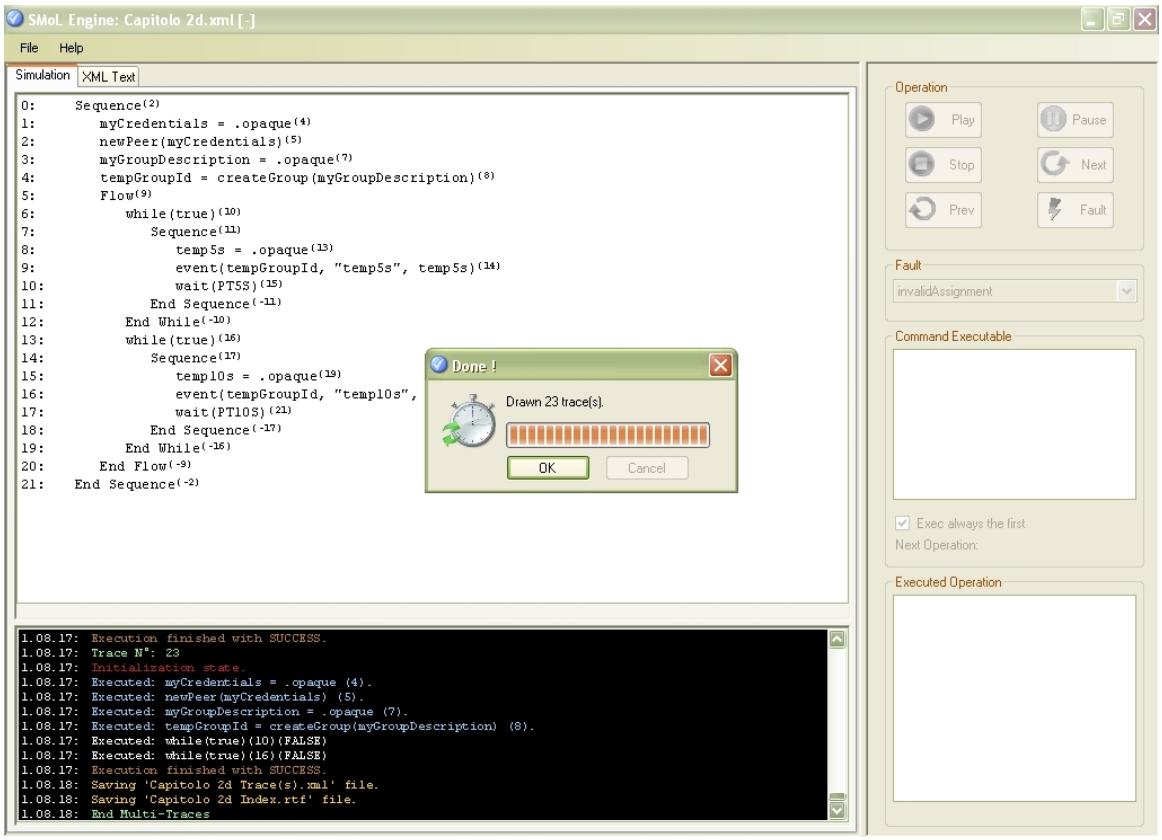

Figura 4.8: Schermata conclusiva dell'operazione simulazione di tutte le tracce

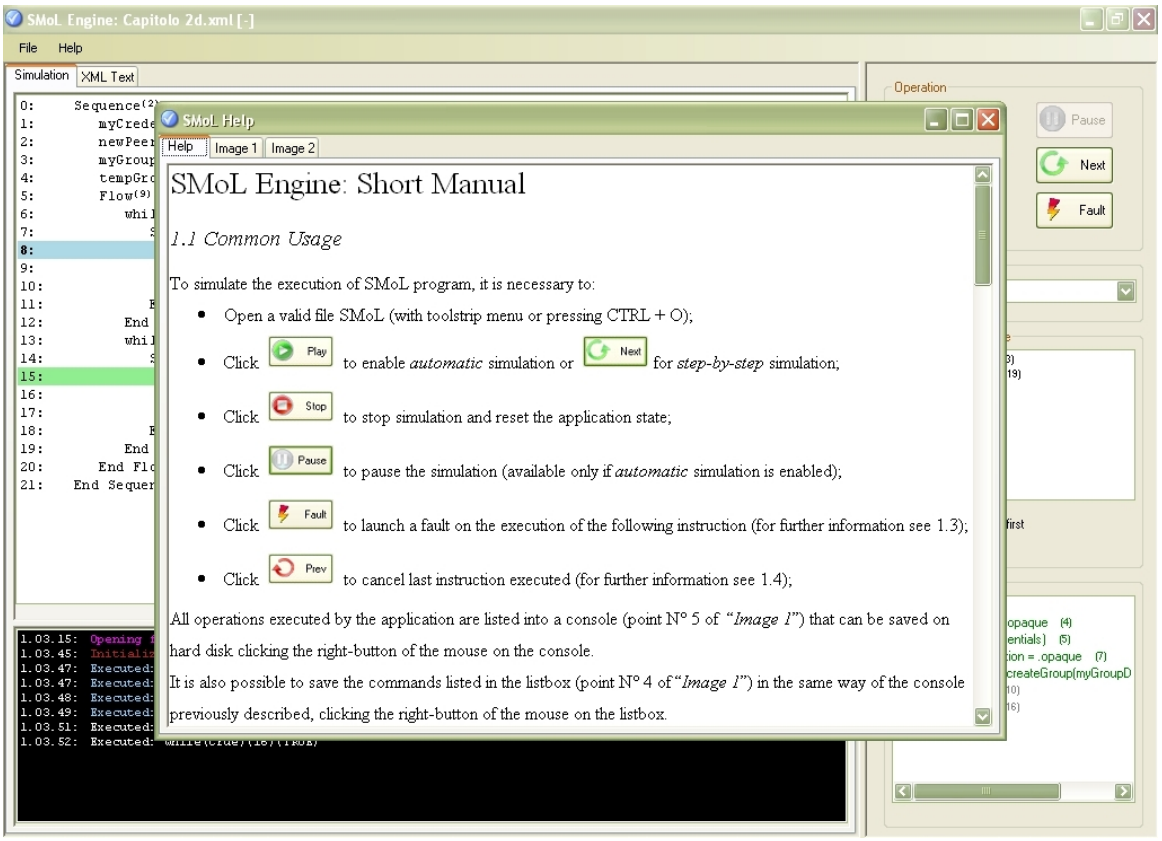

Figura 4.9: Semplice manuale d'uso del prototipo

Per rendere più semplice l'uso dell'intero programma, si è pensato di fornire anche un piccolo manuale interno al prototipo in cui sono descritte le modalità con cui eseguire le funzionalità offerte (vedi figura 4.9) .

Sempre in questa ottica, è stato introdotto anche un semplice editor di testo `xml` (utilizzabile solamente in lettura, non è possibile apportare delle modifiche al testo visualizzato): con questo editor è possibile confrontare il codice sorgente `xml` con la rappresentazione SMoL ottenuta, in maniera istantanea, senza l'utilizzo di ulteriori programmi accedendo al codice direttamente nei punti di interesse in quanto è dotato anche della possibilità di ricerca di parole nel testo (figura 4.10).

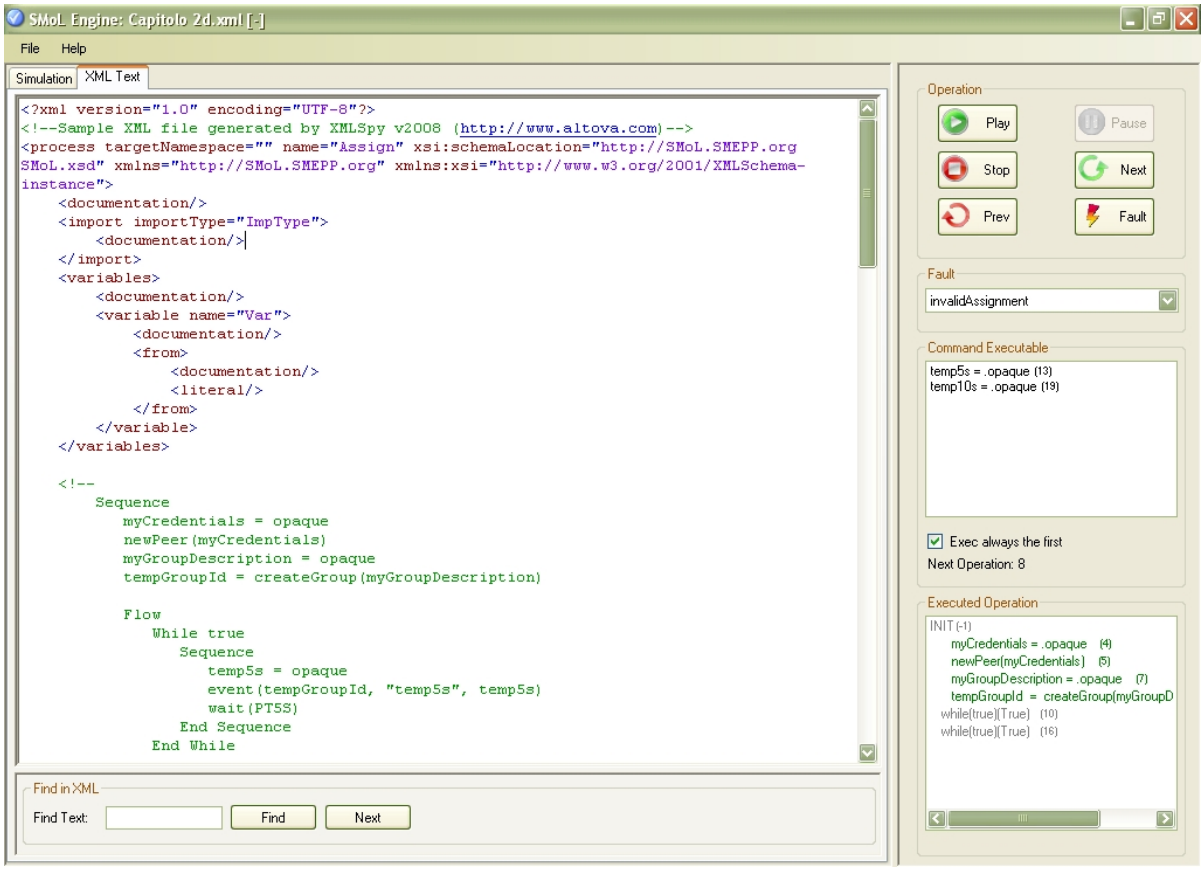

Figura 4.10: Editor *read-only* del sorgente in xml

Capitolo5 Conclusioni

5.1 Obiettivi raggiunti

L'obiettivo della tesi era la progettazione e la realizzazione di un simulatore di specifiche SMoL di sistemi peer to peer. Il prototipo sviluppato ha dovuto soddisfare determinate caratteristiche quali la semplicità e la flessibilità d'uso, la portabilità, la *parziale* estendibilità.

La fase di progettazione del programma ha portato alla definizione delle funzionalità che il programma avrebbe dovuto avere: inoltre, in questa fase, sono state anche delineate le caratteristiche di modularità, di espandibilità e di semplicità d'uso auspicabili al simulatore.

In seguito si è passati alla realizzazione vera è propria, che si è svolta in due fasi. Da principio si è implementato la logica dell'applicazione, studiando e realizzando sia una struttura dati appropriata (l'albero sintattico di istanze di *Nodo*) che descrivesse in maniera adeguata la struttura del programma SMoL analizzato, sia degli algoritmi di *visita* e di *esecuzione* in grado di agire sull'albero, modificandolo in base alle istruzioni (*primitive*) selezionate per l'esecuzione.

In secondo luogo sono state sviluppate delle funzionalità che, basandosi completamente sulla strutture dati e sugli algoritmi descritti in precedenza, permettono all'utente un controllo assoluto durante lo sviluppo delle tracce SMoL: infatti, a chi usa il simulatore, è stata concessa la possibilità di eseguire *step-by-step* la simulazione oppure di eseguirla in automatico, di poter far fallire una qualsiasi istruzione, di poter annullare l'esecuzione delle ultime *K* istruzioni eseguite, di poter salvare/caricare le simulazioni.

Il simulatore è stato inoltre dotato anche della possibilità di generare tutte le tracce ottenibili in maniera del tutto automatica: in questo modo, non solo si "*libera*" l'utente dall'onere di dover provare a mano tutte le simulazioni, ma si può ottenere una visuale davvero completa sui comportamenti tenuti dall'entità P2P, peer o servizio che sia (analogamente alle altre funzionalità, anche questa ultima sfrutta le strutture dati e gli algoritmi descritti in precedenza). Inoltre, dal momento che la mole di dati che il prototipo può generare nel simulare tutte le tracce in automatico può assumere notevoli dimensioni, è stato fornito della

capacità di scrivere i risultati ottenuti in formato xml: questo formato standard permetterà ad altre applicazioni (ancora da sviluppare) di poter gestire le informazioni che il simulatore produce su di un programma SMoL.

In conclusione, gli obiettivi che erano stati prefissati sono stati tutti pienamente raggiunti: è stato creato un programma flessibile e di semplice uso, in grado di simulare i comportamenti di entità P2P e in grado di fornire i risultati ottenuti in maniera concisa ed esaustiva.
Inoltre, il prototipo di simulatore sviluppato verrà anche utilizzato all'interno del progetto europeo SMEPP come ulteriore strumento di analisi e verifica.

5.2 Possibili sviluppi futuri

Gli sviluppi futuri dell'applicazione possono essere molti, sia legati all'applicazione vera e propria, sia riguardanti lo sviluppo di altri programmi da affiancare all'uso del prototipo.
Oltre ad eventuali "espansioni" dell'interfaccia grafica (la quale potrebbe essere arricchita di funzionalità per evidenziare magari altri aspetti della simulazione o, in genere, per poter fornire ulteriori servizi all'utente) e ad una ingegnerizzazione del prototipo (fase necessaria a portare piccole correzioni al progetto iniziale, con l'intento se possibile di migliorarne le caratteristiche, ma nel contempo mettere in pratica le soluzioni tecniche migliori, volte anche ad agevolarne la manutenzione in futuro), gli sviluppi principali che si potrebbero apportare al simulatore vero e proprio sono sostanzialmente due: l'implementazione di uno stato dell'esecuzione e la possibilità di far interagire due o più entità P2P realmente tra di loro.
Lo sviluppo di uno stato dell'esecuzione comporterebbe dotare il programma SMoL di uno stato del tutto privato (differente dallo stato del simulatore nel quale viene memorizzato istante per istante lo stato corrente della traccia), in cui vengono salvate tutte le variabili (e i relativi valori) ed eventuali altri dati usati dal programma stesso: in questo modo non si *simulerebbe* più il programma, ma lo si *eseguirebbe* realmente (rendendo il prototipo un vero *engine* a tutti gli effetti).
Dato che nel linguaggio SMoL, oltre ai classici tipi di dati *interi*, *stringhe*, *booleani*, possono essere dichiarati tipi di dati arbitrari, lo sviluppo di uno stato interno comporterebbe anche la necessità di dover parsare le descrizioni di tali dati all'interno del codice xml, nel tentativo di riproporre strutture appropriate per usarle nella esecuzione del programma.

Far interagire (*realmente*) due o più entità P2P tra loro è il secondo sviluppo che si potrebbe apportare all'applicazione: è importante precisare che, come requisito fondamentale allo sviluppo, sia necessario un concetto di stato di esecuzione delle singole entità (altrimenti si *simulerebbe* e basta una interazione). Quindi lo sviluppo può aver luogo solamente dopo aver implementato lo stato interno del programma SMoL. Inoltre, per permettere l'interazione tra peer/servizi, si deve poter sviluppare anche un concetto di stato globale, ovvero delle strutture dati condivise tra tutte le entità (utili nelle operazioni di creazioni gruppi, unione di peer a gruppi, pubblicazione di servizi, ecc…).

Si potrebbe anche pensare di sviluppare programmi da affiancare nell'uso al simulatore anzi, per precisione, da mettere in un ideale *pipeline* al prototipo: come naturale completamento all'applicazione si potrebbe pensare di realizzare un *analizzatore di tracce* il quale, prendendo come *input* tutte le possibili tracce generate dal simulatore, possa "*analizzarle*" (o ulteriormente elaborarle in qualche maniera) per produrre ulteriori *output*. Questo tipo di sviluppo non comporterebbe nessuna modifica al simulatore, in quanto questo ultimo verrebbe usato come una *black-box*, a scatola chiusa.

Bibliografia

[1] **M. Albano, A. Brogi, R. Popescu, M. Diaz, and J. Dianes**. Towards secure middleware for embedded peer-to-peer systems: Objectives and requirements. *In Proceedings of RSPSI'07*, 2007. http://www.igd.fhg.de/igd-a1/RSPSI2/papers/Ubicomp2007_RSPSI2_Albano.pdf .

[2] **S. Alda and A. Cremers**. Towards composition management for component-based peer-to-peer architectures. *ENTCS*, 114:47–64, 2005.

[3] **M. Bisignano, G. D. Modica, and O. Tomarchio**. JMobiPeer: A middleware for mobile peer-to-peer computing in manets. In *ICDCS Workshops* [2], pages 785–791.

[4] **P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis**. The RUNES middleware: A reconfigurable component-based approach to networked embedded systems. *In Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Berlin (Germany), 2005.

[5] **G. Gehlen and L. Pham**. Mobile web services for peer-to-peer applications. In *Proceedings of CCNC'05*, pages 427–433, 2005.

[6] **R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman**. Supporting predictable service provision in manets via context aware session management. *JWSR*, (3):1– 26, 2006.

[7] **JXTA Home Page**. http://www.jxta.org/ .

[8] **R. Lucchi and G. Zavattaro**. WSSecSpaces: a secure data-driven coordination service for Web services applications. In *SAC'04* [1], pages 487–491.

[9] **OASIS. BPEL v2.0**. http://www.oasis-open.org/committees/download.php/23974/wsbpel-v2.0-primer.pdf .

[10] **S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker**. A scalable content-addressable network. *In Proceedings of SIGCOMM'01*, pages 161–172, 2001.

[11] **SMEPP Coalition**. D1.3: Application requirements. http://www.smepp.org/ .

[12] **W3C. WSDL v1.1**. http://www.w3.org/TR/wsdl .

[13] **W3C. OWL**. http://www.w3.org/TR/owl-features/ .

[14] **MSDN. Microsoft Visual Studio .NET Home Page**. http://msdn2.microsoft.com/it-it/vstudio/default.aspx .

[15] **Altova. XMLSpy 2008 Home Page**. http://www.altova.com/ .

[16] **SMEPP Coalition**. D2.1v2: Service Model Description. http://www.smepp.org/ .

Allegato A XML Schema *SMoL.xsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
       Copyright (c) OASIS Open 2003-2007. All Rights Reserved.
       Modified by Razvan Popescu (UPI) for SMEPP. (Version: SMoLv2.2.xsd)
-->
<xsd:schema targetNamespace="http://SMoL.SMEPP.org" xmlns="http://SMoL.SMEPP.org"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsd-
derived="http://SMoL.SMEPP.org" elementFormDefault="qualified" blockDefault="#all">
<xsd:import namespace="http://www.w3.org/XML/1998/namespace"
schemaLocation="http://www.w3.org/2001/xml.xsd"/>
   <xsd:element name="process" type="tProcess"/>
   <xsd:complexType name="tProcess">
      <xsd:complexContent>
         <xsd:extension base="tExtensibleElements">
            <xsd:sequence>
               <xsd:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element ref="variables" minOccurs="0"/>
               <xsd:group ref="activity"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:NCName" use="required"/>
            <xsd:attribute name="targetNamespace" type="xsd:anyURI"
            use="required"/>
            <xsd:attribute name="queryLanguage" type="xsd:anyURI"
            default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"/>
            <xsd:attribute name="expressionLanguage" type="xsd:anyURI"
            default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:complexType name="tExtensibleElements">
      <xsd:annotation>
         <xsd:documentation>
            This type is extended by other component types to allow elements and
            attributes from other namespaces to be added at the modeled places.
         </xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
         <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
         <xsd:any namespace="##other" processContents="lax" minOccurs="0"
         maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:anyAttribute namespace="##other" processContents="lax"/>
   </xsd:complexType>
   <xsd:element name="documentation" type="tDocumentation"/>
   <xsd:complexType name="tDocumentation" mixed="true">
      <xsd:sequence>
         <xsd:any namespace="##any" processContents="lax" minOccurs="0"
          maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="source" type="xsd:anyURI"/>
      <!--<xsd:attribute ref="xml:lang"/>-->
   </xsd:complexType>

   <xsd:element name="import" type="tImport"/>
   <xsd:complexType name="tImport">
      <xsd:complexContent>
         <xsd:extension base="tExtensibleElements">
            <xsd:attribute name="namespace" type="xsd:anyURI" use="optional"/>
```

```
            <xsd:attribute name="location" type="xsd:anyURI" use="optional"/>
            <xsd:attribute name="importType" type="xsd:anyURI" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <xsd:element name="variables" type="tVariables"/>
   <xsd:complexType name="tVariables">
      <xsd:complexContent>
         <xsd:extension base="tExtensibleElements">
            <xsd:sequence>
               <xsd:element ref="variable" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:element name="variable" type="tVariable"/>
   <xsd:complexType name="tVariable">
      <xsd:annotation>
         <xsd:documentation>
            The "from" tag serves to initialise variables.
         </xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="tExtensibleElements">
            <xsd:sequence>
               <xsd:element ref="from" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="VariableName" use="required"/>
            <xsd:attribute name="messageType" type="xsd:QName" use="optional"/>
            <xsd:attribute name="type" type="xsd:QName" use="optional"/>
            <xsd:attribute name="element" type="xsd:QName" use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:simpleType name="VariableName">
      <xsd:restriction base="xsd:NCName">
         <xsd:pattern value="[^\.]+"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="tOpaqueBoolean">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="yes" />
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:group name="activity">
      <xsd:choice>
         <xsd:group ref="basic_commands"/>
         <xsd:group ref="structured_commands"/>
      </xsd:choice>
   </xsd:group>

   <xsd:group name="basic_commands">
      <xsd:choice>
         <xsd:group ref="primitives"/>
         <xsd:element ref="empty"/>
         <xsd:element ref="assign"/>
         <xsd:element ref="wait"/>
         <xsd:element ref="throw"/>
         <xsd:element ref="catch"/> <!--to be used only inside faultHandlers-->
         <xsd:element ref="catchAll"/><!--to be used only inside faultHandler-->
         <xsd:element ref="exit"/>
      </xsd:choice>
   </xsd:group>

   <xsd:group name="structured_commands">
      <xsd:choice>
         <xsd:element ref="sequence"/>
```

```
      <xsd:element ref="flow"/>
      <xsd:element ref="while"/>
      <xsd:element ref="repeatUntil"/>
      <xsd:element ref="if"/>
      <xsd:element ref="pick"/>
      <xsd:element ref="informationHandler"/>
      <xsd:element ref="faultHandler"/>
   </xsd:choice>
</xsd:group>

<xsd:group name="primitives">
   <xsd:choice>
      <!-- Peer Management Primitives -->
      <xsd:element ref="newPeer"/>
      <xsd:element ref="getPeerId"/>
      <xsd:element ref="getPeers"/>

      <!-- Group Management Primitives -->
      <xsd:element ref="createGroup"/>
      <xsd:element ref="getGroups"/>
      <xsd:element ref="getGroupDescription"/>
      <xsd:element ref="joinGroup"/>
      <xsd:element ref="leaveGroup"/>
      <xsd:element ref="getIncludingGroups"/>
      <xsd:element ref="getPublishingGroup"/>

      <!-- Service Management Primitives -->
      <xsd:element ref="publish"/>
      <xsd:element ref="unpublish"/>
      <xsd:element ref="getServices"/>
      <xsd:element ref="getServiceContract"/>
      <xsd:element ref="startSession"/>

      <!-- Message Management Primitives -->
      <xsd:element ref="invoke"/>
      <xsd:element ref="receiveMessage"/>
      <xsd:element ref="reply"/>

      <!-- Event Management Primitives -->
      <xsd:element ref="event"/>
      <xsd:element ref="receiveEvent"/>
      <xsd:element ref="subscribe"/>
      <xsd:element ref="unsubscribe"/>
   </xsd:choice>
</xsd:group>

<!-- Start of basic_commands' definition. -->

<!-- Start of primitives' definition. -->

<!-- Peer Management Primitives -->

<!-- peerId newPeer(credentials) -->
<xsd:element name="newPeer" type="tNewPeer"/>
<xsd:complexType name="tNewPeer">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputNewPeer"/>
            <xsd:element name="output" type="tOutputNewPeer" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputNewPeer">
   <xsd:sequence>
      <xsd:element name="credentials" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputNewPeer">
```

```
   <xsd:sequence>
      <xsd:element name="peerId" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tActivity">
   <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
         <xsd:attribute name="name" type="xsd:NCName"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- peerId getPeerId(id?) -->
<xsd:element name="getPeerId" type="tGetPeerId"/>
<xsd:complexType name="tGetPeerId">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputGetPeerId" minOccurs="0"/>
            <xsd:element name="output" type="tOutputGetPeerId"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetPeerId">
   <xsd:sequence>
      <xsd:element name="id" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetPeerId">
   <xsd:sequence>
      <xsd:element name="peerId" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- peerId[] getPeers(groupId?) -->
<xsd:element name="getPeers" type="tGetPeers"/>
<xsd:complexType name="tGetPeers">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputGetPeers" minOccurs="0"/>
            <xsd:element name="output" type="tOutputGetPeers" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetPeers">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetPeers">
   <xsd:sequence>
      <xsd:element name="peerIdArray" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- Group Management Primitives -->

<!-- groupId createGroup(groupDescription) -->
<xsd:element name="createGroup" type="tCreateGroup"/>
<xsd:complexType name="tCreateGroup">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputCreateGroup"/>
            <xsd:element name="output" type="tOutputCreateGroup"
            minOccurs="0"/>
```

```
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputCreateGroup">
   <xsd:sequence>
      <xsd:element name="groupDescription" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputCreateGroup">
   <xsd:sequence>
      <xsd:element name="groupId" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- groupId[] getGroups(groupDescription?) -->
<xsd:element name="getGroups" type="tGetGroups"/>
<xsd:complexType name="tGetGroups">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputGetGroups" minOccurs="0"/>
            <xsd:element name="output" type="tOutputGetGroups"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetGroups">
   <xsd:sequence>
      <xsd:element name="groupDescription" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetGroups">
   <xsd:sequence>
      <xsd:element name="groupIdArray" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- groupGescription getGroupDescription(groupId) -->
<xsd:element name="getGroupDescription" type="tGetGroupDescription"/>
<xsd:complexType name="tGetGroupDescription">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputGetGroupDescription"/>
            <xsd:element name="output" type="tOutputGetGroupDescription"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetGroupDescription">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetGroupDescription">
   <xsd:sequence>
      <xsd:element name="groupDescription" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- void joinGroup(groupId, credentials) -->
<xsd:element name="joinGroup" type="tJoinGroup"/>
<xsd:complexType name="tJoinGroup">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputJoinGroup"/>
```

```
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputJoinGroup">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom"/>
      <xsd:element name="credentials" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>

<!-- void leaveGroup(groupId) -->
<xsd:element name="leaveGroup" type="tLeaveGroup"/>
<xsd:complexType name="tLeaveGroup">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputLeaveGroup"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputLeaveGroup">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>

<!-- groupId[] getIncludingGroups() -->
<xsd:element name="getIncludingGroups" type="tGetIncludingGroups"/>
<xsd:complexType name="tGetIncludingGroups">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="output" type="tOutputGetIncludingGroups"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tOutputGetIncludingGroups">
   <xsd:sequence>
      <xsd:element name="groupIdArray" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- groupId getPublishingGroup(id?) -->
<xsd:element name="getPublishingGroup" type="tGetPublishingGroup"/>
<xsd:complexType name="tGetPublishingGroup">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputGetPublishingGroup"
            minOccurs="0"/>
            <xsd:element name="output" type="tOutputGetPublishingGroup"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetPublishingGroup">
   <xsd:sequence>
      <xsd:element name="id" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetPublishingGroup">
   <xsd:sequence>
      <xsd:element name="groupId" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>
```

```
<!-- Service Management Primitives -->

<!-- <groupServiceId, peerServiceId> publish(groupId, serviceContract) -->
<xsd:element name="publish" type="tPublish"/>
<xsd:complexType name="tPublish">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputPublish"/>
            <xsd:element name="output" type="tOutputPublish" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputPublish">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom"/>
      <xsd:element name="serviceContract" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputPublish">
   <xsd:sequence>
      <xsd:element name="groupServiceId" type="tTo"/>
      <xsd:element name="peerServiceId" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- void unpublish(peerServiceId) -->
<xsd:element name="unpublish" type="tUnpublish"/>
<xsd:complexType name="tUnpublish">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputUnpublish"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputUnpublish">
   <xsd:sequence>
      <xsd:element name="peerServiceId" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>

<!-- <groupId, groupServiceId, peerServiceId>[] getServices(groupId?,
     peerId?, serviceContract?, maxResults?, credentials) -->
<xsd:element name="getServices" type="tGetServices"/>
<xsd:complexType name="tGetServices">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputGetServices"/>
            <xsd:element name="output" type="tOutputGetServices"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetServices">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
      <xsd:element name="peerId" type="tFrom" minOccurs="0"/>
      <xsd:element name="serviceContract" type="tFrom" minOccurs="0"/>
      <xsd:element name="maxResults" type="tFrom" minOccurs="0"/>
      <xsd:element name="credentials" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetServices">
   <xsd:sequence>
```

```
      <xsd:element name="groupId" type="tTo" minOccurs="0"/>
      <xsd:element name="groupServiceId" type="tTo" minOccurs="0"/>
      <xsd:element name="peerServiceId" type="tTo" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<!-- serviceContract getServiceContract(id) -->
<xsd:element name="getServiceContract" type="tGetServiceContract"/>
<xsd:complexType name="tGetServiceContract">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputGetServiceContract"/>
            <xsd:element name="output" type="tOutputGetServiceContract"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetServiceContract">
   <xsd:sequence>
      <xsd:element name="id" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetServiceContract">
   <xsd:sequence>
      <xsd:element name="serviceContract" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- sessionId startSession(serviceId) -->
<xsd:element name="startSession" type="tStartSession"/>
<xsd:complexType name="tStartSession">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputStartSession"/>
            <xsd:element name="output" type="tOutputStartSession"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputStartSession">
   <xsd:sequence>
      <xsd:element name="serviceId" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputStartSession">
   <xsd:sequence>
      <xsd:element name="sessionId" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- Message Management Primitives -->

<!-- output? invoke(entityId, operationName, input?) -->
<xsd:element name="invoke" type="tInvoke"/>
<xsd:complexType name="tInvoke">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputInvoke"/>
            <xsd:element name="output" type="tOutputInvoke" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputInvoke">
   <xsd:sequence>
```

```
      <xsd:element name="entityId" type="tFrom"/>
      <xsd:element name="operationName" type="tFrom"/>
      <xsd:element name="input" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputInvoke">
   <xsd:sequence>
      <xsd:element name="output" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>

<!-- <callerId, input?> receiveMessage(groupId?, operationName) -->
<xsd:element name="receiveMessage" type="tReceiveMessage"/>
<xsd:complexType name="tReceiveMessage">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputReceiveMessage"/>
            <xsd:element name="output" type="tOutputReceiveMessage"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputReceiveMessage">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
      <xsd:element name="operationName" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputReceiveMessage">
   <xsd:sequence>
      <xsd:element name="callerId" type="tFrom"/>
      <xsd:element name="input" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<!-- void reply(callerId, operationName, output?, faultName?) -->
<xsd:element name="reply" type="tReply"/>
<xsd:complexType name="tReply">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputReply"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputReply">
   <xsd:sequence>
      <xsd:element name="callerId" type="tFrom"/>
      <xsd:element name="operationName" type="tFrom"/>
      <xsd:element name="output" type="tFrom" minOccurs="0"/>
      <xsd:element name="faultName" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<!-- Event Management Primitives -->

<!-- void event(groupId?, eventName, input?) -->
<xsd:element name="event" type="tEvent"/>
<xsd:complexType name="tEvent">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputEvent"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputEvent">
```

```
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
      <xsd:element name="eventName" type="tFrom"/>
      <xsd:element name="input" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<!-- <callerId, input?> receiveEvent(groupId?, eventName) -->
<xsd:element name="receiveEvent" type="tReceiveEvent"/>
<xsd:complexType name="tReceiveEvent">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputReceiveEvent"/>
            <xsd:element name="output" type="tOutputReceiveEvent"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputReceiveEvent">
   <xsd:sequence>
      <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
      <xsd:element name="eventName" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputReceiveEvent">
   <xsd:sequence>
      <xsd:element name="callerId" type="tFrom"/>
      <xsd:element name="input" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<!-- void subscribe(eventName?, groupId?) -->
<xsd:element name="subscribe" type="tSubscribe"/>
<xsd:complexType name="tSubscribe">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputSubscribe" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputSubscribe">
   <xsd:sequence>
      <xsd:element name="eventName" type="tFrom" minOccurs="0"/>
      <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<!-- void unsubscribe(eventName?, groupId?) -->
<xsd:element name="unsubscribe" type="tUnsubscribe"/>
<xsd:complexType name="tUnsubscribe">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputUnsubscribe"
            minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputUnsubscribe">
   <xsd:sequence>
      <xsd:element name="eventName" type="tFrom" minOccurs="0"/>
      <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<!-- End of primitives' definition. -->
```

```
<xsd:element name="empty" type="tEmpty"/>
<xsd:complexType name="tEmpty">
   <xsd:complexContent>
      <xsd:extension base="tActivity"/>
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="assign" type="tAssign"/>
<xsd:complexType name="tAssign">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element ref="copy" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="copy" type="tCopy"/>
<xsd:complexType name="tCopy">
   <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
         <xsd:sequence>
            <xsd:element ref="from"/>
            <xsd:element ref="to"/>
         </xsd:sequence>
         <xsd:attribute name="keepSrcElementName" type="tBoolean"
         use="optional" default="no"/>
         <xsd:attribute name="ignoreMissingFromData" type="tBoolean"
         use="optional" default="no"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="tBoolean">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="yes"/>
      <xsd:enumeration value="no"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:element name="from" type="tFrom"/>
<xsd:complexType name="tFrom" mixed="true">
   <xsd:sequence>
      <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded"/>
      <xsd:choice minOccurs="0">
         <xsd:element ref="literal"/>
         <xsd:element ref="query"/>
      </xsd:choice>
   </xsd:sequence>
   <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
   <xsd:attribute name="variable" type="VariableName"/>
   <xsd:attribute name="part" type="xsd:NCName"/>
   <xsd:attribute name="opaque" type="xsd-derived:tOpaqueBoolean"/>
   <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="literal" type="tLiteral"/>
<xsd:complexType name="tLiteral" mixed="true">
   <xsd:sequence>
      <xsd:any namespace="##any" processContents="lax" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="query" type="tQuery"/>
<xsd:complexType name="tQuery" mixed="true">
   <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
```

```
   </xsd:sequence>
   <xsd:attribute name="queryLanguage" type="xsd:anyURI"/>
   <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="to" type="tTo"/>
<xsd:complexType name="tTo" mixed="true">
   <xsd:sequence>
      <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded"/>
      <xsd:element ref="query" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
   <xsd:attribute name="variable" type="VariableName"/>
   <xsd:attribute name="part" type="xsd:NCName"/>
   <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="wait" type="tWait"/>
<xsd:complexType name="tWait">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:choice>
               <xsd:element ref="for" minOccurs="0"/>
               <xsd:element ref="until" minOccurs="0"/>
            </xsd:choice>
            <xsd:element ref="repeatEvery" minOccurs="0"/>
            <!-- The repeatEvery element can only be defined for
                 informationHandler branches -->
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:element name="for" type="tDuration-expr"/>
<xsd:element name="until" type="tDeadline-expr"/>
<xsd:element name="repeatEvery" type="tDuration-expr"/>
<xsd:complexType name="tDuration-expr" mixed="true">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression"/>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tDeadline-expr" mixed="true">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression"/>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tExpression" mixed="true">
   <xsd:sequence>
      <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
   <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="throw" type="tThrow"/>
<xsd:complexType name="tThrow">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element name="input" type="tInputThrow"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputThrow">
   <xsd:sequence>
      <xsd:element name="faultName" type="tFrom"/>
      <xsd:element name="faultVariable" type="tFrom" minOccurs="0"/>
```

```
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="catch" type="tCatch"/>
<xsd:complexType name="tCatch">
   <xsd:complexContent>
      <xsd:extension base="tActivityContainer">
         <xsd:sequence>
            <xsd:element name="input" type="tInputCatch"/>
            <xsd:element name="output" type="tOutputCatch" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputCatch">
   <xsd:sequence>
      <xsd:element name="faultName" type="tFrom"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputCatch">
   <xsd:sequence>
      <xsd:element name="faultVariable" type="tTo"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tActivityContainer">
   <xsd:complexContent>
      <xsd:extension base="tExtensibleElements">
         <xsd:sequence>
            <xsd:group ref="activity"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="catchAll" type="tCatchAll"/>
<xsd:complexType name="tCatchAll">
   <xsd:complexContent>
      <xsd:extension base="tActivityContainer">
         <xsd:sequence>
            <xsd:element name="output" type="tOutputCatchAll" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tOutputCatchAll">
   <xsd:sequence>
      <xsd:element name="faultName" type="tTo"/>
      <xsd:element name="faultVariable" type="tTo" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="exit" type="tExit"/>
<xsd:complexType name="tExit">
   <xsd:complexContent>
      <xsd:extension base="tActivity"/>
   </xsd:complexContent>
</xsd:complexType>

<!-- End of basic_commands' definition. -->

<!-- Start of structured_commands' definition. -->

<xsd:element name="sequence" type="tSequence"/>
<xsd:complexType name="tSequence">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:group ref="activity" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:extension>
```

```
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="flow" type="tFlow"/>
<xsd:complexType name="tFlow">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:group ref="activity" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="while" type="tWhile"/>
<xsd:complexType name="tWhile">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element ref="condition"/>
            <xsd:group ref="activity"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:element name="condition" type="tBoolean-expr"/>
<xsd:complexType name="tBoolean-expr" mixed="true">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tExpression"/>
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="repeatUntil" type="tRepeatUntil"/>
<xsd:complexType name="tRepeatUntil">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:group ref="activity"/>
            <xsd:element ref="condition"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="if" type="tIf"/>
<xsd:complexType name="tIf">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element ref="condition"/>
            <xsd:group ref="activity"/>
            <xsd:element ref="else" minOccurs="0"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<xsd:element name="else" type="tActivityContainer"/>

<xsd:element name="pick" type="tPick"/>
<xsd:complexType name="tPick">
   <xsd:complexContent>
      <xsd:extension base="tActivity">
         <xsd:sequence>
            <xsd:element ref="onMessage" minOccurs="0" m
            axOccurs="unbounded"/>
            <xsd:element ref="onEvent" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element ref="onAlarm" minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:extension>
```

```
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:element name="onMessage" type="tOnMessage"/>
   <xsd:complexType name="tOnMessage">
      <xsd:complexContent>
         <xsd:extension base="tActivityContainer">
            <xsd:sequence>
               <xsd:element ref="receiveMessage"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:element name="onEvent" type="tOnEvent"/>
   <xsd:complexType name="tOnEvent">
      <xsd:complexContent>
         <xsd:extension base="tActivityContainer">
            <xsd:sequence>
               <xsd:element ref="receiveEvent"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <xsd:element name="onAlarm" type="tOnAlarm"/>
   <xsd:complexType name="tOnAlarm">
      <xsd:complexContent>
         <xsd:extension base="tActivityContainer">
            <xsd:sequence>
               <xsd:element ref="wait"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <xsd:element name="informationHandler" type="tInformationHandler"/>
   <xsd:complexType name="tInformationHandler">
      <xsd:complexContent>
         <xsd:extension base="tActivity">
            <xsd:sequence>
               <xsd:group ref="activity"/>
               <xsd:element ref="onMessage" minOccurs="0"
               maxOccurs="unbounded"/>
               <xsd:element ref="onEvent" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element ref="onAlarm" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <xsd:element name="faultHandler" type="tFaultHandler"/>
   <xsd:complexType name="tFaultHandler">
      <xsd:complexContent>
         <xsd:extension base="tActivity">
            <xsd:sequence>
               <xsd:group ref="activity"/>
               <xsd:element ref="catch" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element ref="catchAll" minOccurs="0"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- End of structured_commands' definition. -->
</xsd:schema>
```