Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Euristiche per il Controllo della QoS di componenti grid-enabled (modelli e patterns)

S. Lombardo, V. Graziano, A. Machì

RT-ICAR-PA-12-2005 **Novembre 2005**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Euristiche per il Controllo della QoS di componenti grid-enabled (modelli e patterns)

S. Lombardo, V. Graziano, A. Machì

Deliverable III° anno
Progetto MIUR FIRB Grid.it
Work Package 8 Ambienti di Programmazione

RT-ICAR-PA-12-2005 **Novembre 2005**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

1 Indice

2 Introduzione

La Qualità del Servizio (QoS) è stata definita come "l'insieme delle caratteristiche qualitative e quantitative necessarie per ottenere il livello di funzionalità richiesto da una applicazione [1]. Possiamo supporre dunque di associare alla fruizione di un servizio un insieme di attributi, talvolta non esprimibili numericamente, che permettono di specificare il livello di QoS richiesto per il servizio in oggetto. Nel seguito del documento chiameremo questi attributi *proprietà non funzionali* e indicheremo col termine *interfaccia non funzionale* l'interfaccia che permette di effettuare delle operazioni relative alle proprietà non funzionali.

È importante specificare che la QoS non è "il livello migliore delle caratteristiche non funzionali", ma un accordo su un insieme di valori (attributi quantitativi) e su un insieme di descrittori ontologici (attributi qualitativi) fra le esigenze degli utenti e le qualità specifiche che il fornitore del servizio può offrire [2]. In questo documento l'attenzione è stata rivolta su un insieme di proprietà (qualità) che caratterizzano l'esecuzione di ogni componente di una applicazione distribuita:

- la *dependability:* caratterizza il grado di certezza che una elaborazione venga portata a termine o meno [3];
- la *performance*: caratterizzata da vincoli sulle caratteristiche delle risorse impiegate durante tutta l'elaborazione e/o da vincoli sulla "potenza" dell'elaborazione;
- il *costo*: caratterizzata da politiche di economia di impiego delle risorse al fine di offrire elaborazioni ad un costo minore;

L'insieme delle proprietà richieste per l'esecuzione del componente costituisce un profilo di qualità (SLA) negoziabile, tramite un contratto, con il sistema di controllo dell'applicazione.

Lo stato di tali proprietà, durante l'esecuzione del componente, dipende sia dalla corretta esecuzione delle procedure costituenti che il componente origina, che da un insieme di eventi che si originano nell'ambiente di esecuzione (es. variazioni di costo, potenza e livello di disponibilità di risorse) e si manifestano asincronamente all'esecuzione dell'attività. Il controllo di ogni singola proprietà del profilo di qualità comporta l'impiego di uno o più patterns di monitoraggio ed intervento che coinvolgono il sistema di controllo della applicazione (Application o Worflow

Manager [4]), il middleware di run-time della piattaforma distribuita (Grid Abstarct machine [5]) ed il sistema di controllo locale del componente (thread di controllo dello skeleton [6]).

In questo documento sono riportate modelli ed euristiche impiegati sia per la negoziazione dei profili di qualità associabili all'esecuzione dei componenti,i sia per il controllo dello stato del profilo negoziato. Questo ultimo aspetto richiede sia l'impiego di tecniche e sensori per il monitoraggio dello stato del profilo negoziato sia tecniche in grado di rilevare eventuali deficit contrattuali e capaci di creare condizioni che riportino l'esecuzione entro i margini contrattuali negoziati.

Al fine di permettere il coordinamento esterno dei componenti tramite patterns di orchestrazione e/o coreografia [16], nessun adattamento delle risorse interne, ai fini del rispetto della SLA (grid-awareness) è intrapreso dai componenti. Le informazioni sullo stato di esecuzione, necessarie al coordinamento esterno, sono esposte attraverso la porta non funzionale dei componenti (tramite eventi) e, attraverso stessa, vengono ricevuti i segnali di sincronizzazione dei pattern di adattamento (enforcement) gestiti esternamente.

E' importante evidenziare che tali tecniche sono state realizzate in stretta sinergia con l'attività svolta all'interno del WorkPackage 9 (Server di Libreria [7]). In tale contesto il ruolo di Application Manager viene partizionato in due ruoli distinti in cui il primo ha il compito di rilevare anomalie contrattuali e decidere le politiche di enforcement da applicare (JobInspector) ed il secondo ha lo scopo di gestire l'intero ciclo di vita del componente e di attuare le politiche di enforcement emanate dal primo (Work Item Handler). Si rimanda al documento [8] per ulteriori approfondimenti su tale piattaforma .

3 QoS, work items e componenti.

Gli ambienti correnti di Workflow management forniscono delle semantiche specifiche per identificare gli elementi del processo che essi modellano, definendo funzionalità e patterns per controllare l'esecuzione delle funzionalità composte. Per esempio la **Workflow Management Coalition (WFMC)** [9] definisce un "**business process**" come un "insieme di una o più procedure o attività collegate che realizzano un obiettivo" [10]. Esso definisce un'attività come "una descrizione di un'unità di lavoro che rappresenta una unità logica all'interno di un processo. Un'attività è tipicamente l'unità minima di lavoro programmata da un sistema che si occupa dell'attivazione delle attività (Workflow Engine). Un'attività tipicamente genera uno o più **work**

item. Un work item è una rappresentazione del lavoro da processare (da un partecipante del processo) in un'attività all'interno di una istanza di processo. La definizione del processo, insieme ai dati rilevanti ai fini della determinazione del flusso di programma (**workflow relevant data**), sono usati per controllare la navigazione attraverso le varie attività del processo, fornendo informazioni per le condizioni d'attivazione e uscita delle singole attività e le scelte d'esecuzione parallela/sequenziale". Il controllo delle proprietà non-funzionali delle attività è assegnato così ad un **Sistema di WorkFlow Management (WFMS)** descritto come un "sistema che definisce, gestisce ed esegue 'workflow' attraverso l'esecuzione di software il cui ordine di esecuzione è guidato da una rappresentazione computerizzata del processo". Il WFMS e i partecipanti al processo scambiano informazioni attraverso variabili di stato (relevant data) che ne determinano l'instradamento (routing).

Riportando queste definizioni nelle applicazioni con architettura a componenti, appare naturale associare ad ogni componente una attività che produce/consuma work items. Infatti un componente è un elemento software in grado di eseguire elaborazioni, eventualmente parallele/distribuite, in accordo con certe regole e patterns di esecuzione su risorse distribuite. In questa ottica scopo essenziale di ogni componente deve essere quello di permettere l'esecuzione remota delle funzionalità presenti in un modulo di libreria separando nettamente l'attivazione e l'esecuzione di dette funzionalità (workflow) da tutte le operazioni di controllo necessarie per una regolamentata esecuzione delle stesse (workflow management) in un ambiente distribuito e dinamico quale quello delle griglie computazionali (grid-enabling).

Al fine di supportare il controllo sulla QoS delle attività è stato progettato dagli autori in [6] un modello logico di componente. Tale modello propone un'architettura di componente basata su tre interfacce e due processi logici:

- Un'interfaccia funzionale per fornire tramite chiamata remota, sincrona o asincrona, i modi di funzionare (funzionalità) dello skeleton;
- Un'interfaccia di configurazione/introspezione per la lettura/scrittura dei relevant data;
- Un'interfaccia ad eventi per la notifica asincrona del cambiamento dei relevant data;
- Un processo o thread per processare i work-items in ingresso allo skeleton;

- Un processo o thread di controllo per fornire le primitive di controllo al WMFS in maniera asincrona al processamento del work-item;

La figura 1 illustra la struttura del componente ed evidenzia le relazioni tra le sue parti principali.

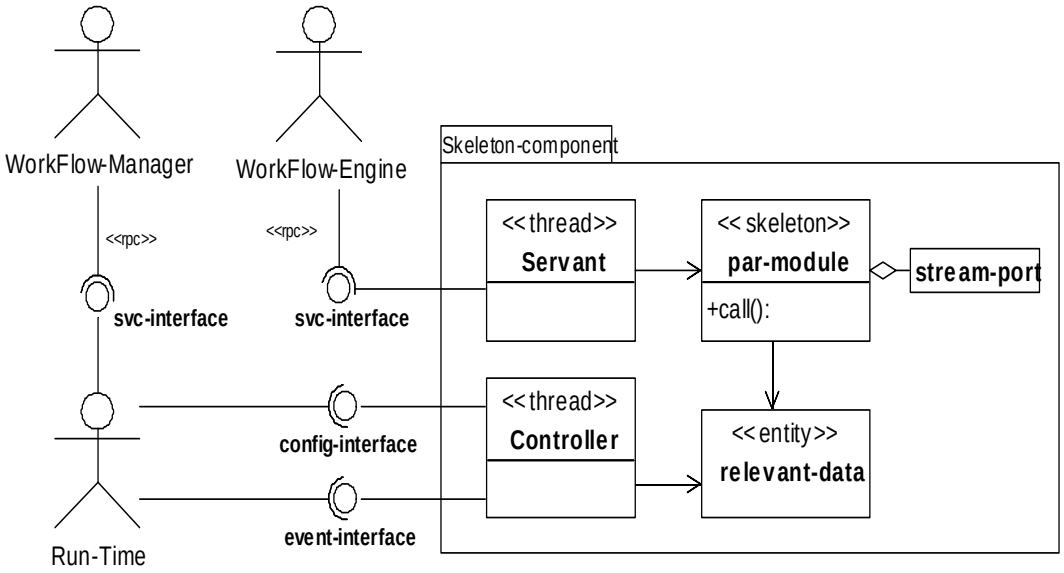

Figura 1. Un Diagramma di Componenti in UML che mostra in un modello logico di organizzazione del componente un thread di controllo ed uno esecutivo interagenti attraverso un oggetto di stato , tre interfacce per invocazione remota delle funzionalità, introspezione e controllo non funzionale ed in fine una porta stream per lo scambio di dati interno allo skeleton.

In ASSIST [11] il WFMS risulta cablato all'interno dell'applicazione (ma distinto da essa) attraverso l'impiego di una gerarchia di moduli di controllo detti **Managers**. Questi ultimi si occupano delle operazioni di adattività dell'applicazione e di controllo sulle prestazioni. I Managers sono sviluppati automaticamente dall'ambiente di programmazione durante la compilazione dei parmod (Module Application Managers or MAM) e durante l'integrazione dell'applicazione (Component Application Managers or CAM). Il deployment dei processi e l'attivazione sulla griglia sono affidati ad un supporto run-time chiamato Grid Abstract Machine. I MAM monitorizzano proattivamente le performance dello skeleton e lo stato delle risorse, servendosi del modello di costo dello skeleton e, se necessario, emettono delle richieste di ri-configurazione ai CAM che analizzano tali richieste in accordo con i criteri di ottimizzazione globale delle risorse.

In figura 2 è illustrata una variante del modello sopra descritto [8] in cui viene evidenziata la presenza di un Manager che riesce a controllare l'applicazione intervenendo sia sull'impiego delle risorse utilizzate (tramite le primitive del supporto runtime di griglia) che sulle singole attività del processo attraverso le interfacce non funzionali dei componenti.

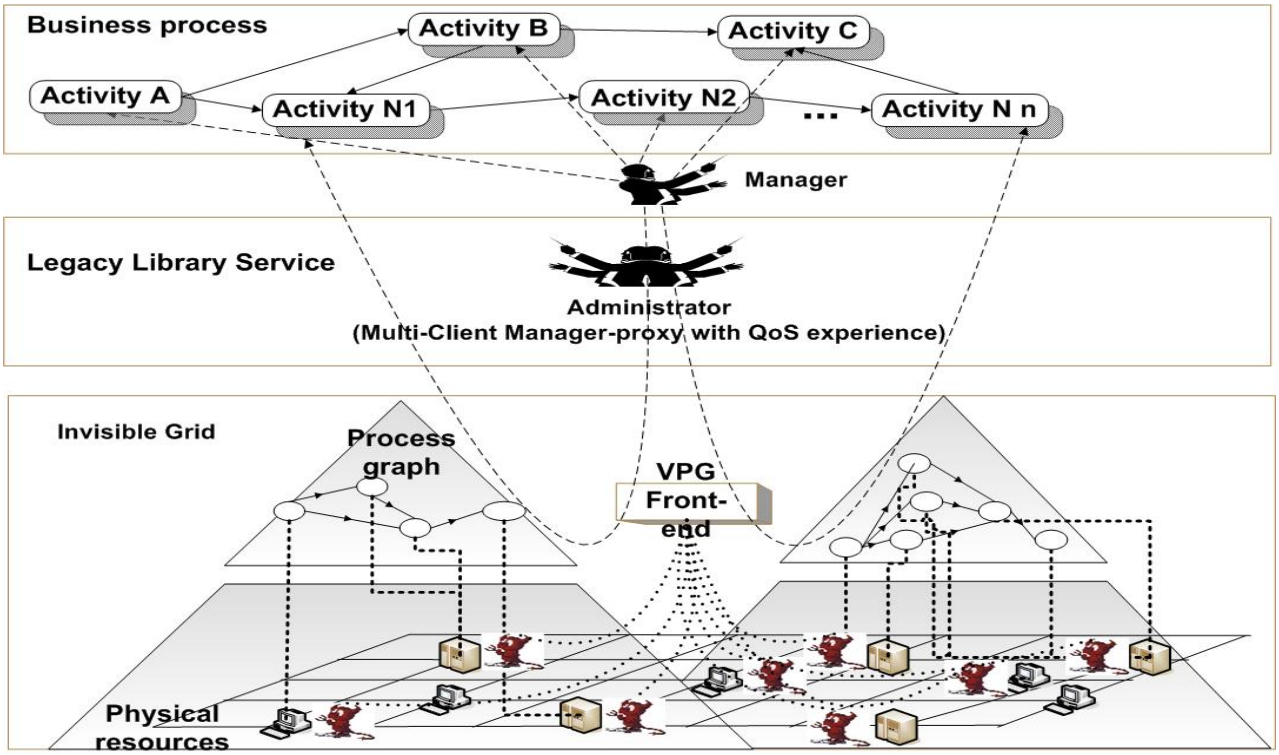

Figura 2. Modello di esecuzione grid-aware di work items eseguiti da componenti grid-enabled controllabili

Il punto chiave è che la valutazione della qualità del servizio con cui ogni work item viene processato è un processo complesso, che normalmente avviene asincronamente all'esecuzione dell'attività, e che anche le direttive di management vanno quindi emesse asincronamente al processamento dei work items. Punti di checkpoint devono quindi essere inclusi nell'attività del componente al fine di supportare la ri-sincronizzazione. Appropriati patterns d'interazione sono necessari per assicurare la coerenza tra il processamento dei dati e le attività di allocazione/liberazione di risorse [2][6]. Nel seguito del documento, dopo aver classificato e caratterizzato alcuni profili contrattuali notevoli, vengono esposte le tecniche realizzate ed impiegate per la gestione di tali profili.

4 Componenti e contratti

Da un punto di vista implementativo un componente è sostanzialmente un eseguibile, o gruppo di eseguibili (componenti paralleli), che racchiude al suo interno un modulo computazionale, delle regole e meccanismi di workflow management, e dei meccanismi di interfacciamento verso l'esterno. E' necessario introdurre una classificazione dei componenti per distinguerli in base alle caratteristiche di controllo e alle capacità di riconfigurazione esposte.

Allo stato attuale abbiamo identificato i seguenti componenti:

- **Legacy serial**: componente che racchiude al suo interno un eseguibile legacy, esso è da considerarsi non persistente e stateless;
- **CCMap, CCFarm**: componente che racchiude uno skeleton master-slave con capacità di autobilanciamento del carico in funzione di indici di performance delle risorse computazionali su cui è mappato il grafo di esecuzione parallela [6];
- **ASSIST 1.2**: grafo di processi comunicanti tra loro attraverso un comunicatore privato socket. Non permette nessun tipo di riconfigurazione o di intervento runtime esterno per effettuare il tuning della performance. Tramite integrazione di una porta funzionale RPC è assimilabile ad un componente LegaciySerial operante su un set di risorse dedicate.

4.1 Tipologie di contratti

In base a quanto detto, è possibile associare ad ogni categoria di componente dei patterns di controllo e dei meccanismi impiegabili per caratterizzare la qualità di esecuzione dei work items da essi consumati. È possibile quindi stabilire dei profili di esecuzione del work item ognuno caratterizzante qualitativamente l'esecuzione del work item stesso; a tale scopo associamo ad ogni profilo un contratto di esecuzione. Oltre a caratterizzare il profilo di gestione del work item, il contratto ha lo scopo di fissare dei vincoli sui parametri ritenuti rilevanti ai fini della QoS del work item sottomesso. I profili contrattuali attualmente identificati sono i seguenti:

- **premium (real time)**: permette di imporre delle direttive sulle risorse da utilizzare. Infatti con questo contratto è possibile specificare sia dei vincoli sull'esecuzione del work item, in termini di tempo o di carico di lavoro (potenza di esecuzione), sia dei vincoli sulle

risorse fisiche da utilizzare per lo schieramento dei processi (nodi fisici del grafo) che dei vicoli sulla potenza dei nodi virtuali del grafo del componente;

- **business (best effort):** impone l'esecuzione del work item al meglio delle possibilità delle risorse di griglia immediatamente usufruibili, compatibilmente con la situazione contingente determinata dagli altri work item in corso di esecuzione;
- **standard**: si tratta di un profilo di esecuzione di work item piuttosto neutro che non possiede particolari caratterisitiche legate alla performance;
- **economy**: come il profilo standard ma meno performante, le risorse utilizzate possono essere condivise da più work items;
- **low cost:** permette di sottomettere l'esecuzione di un work item nel modo più economico possibile utilizzando i tempi morti delle risorse. Il vantaggio di questo contratto è quello di usufruire di una elaborazione a più bassi costi a discapito dei tempi di completamento.

Nella tabella 1 sono riportati i tipi di contratto suddetti specificando, per ognuno di essi, le opzioni incluse o che è possibile attivare durante la sottomissione. Come si evince dalla tabella, le opzioni disponibili cambiano in base al tipo di contratto. Alcune opzioni sono sempre incluse e insite nel profilo contrattuale (simbolo I), altre invece sono opzionali e devono essere attivate esplicitamente al momento della sottomissione del contratto (simbolo O). In definitiva le opzioni sono:

- **Redundand deploy**: lo schieramento dei grafi di processo viene fatto al momento della firma del contratto ed in maniera ridondante. Indipendentemente dall'impiego effettivo delle risorse al momento dell'esecuzione del work-item;
- **Share**: le risorse impiegate possono essere condivise da più work items contemporaneamente;
- **Suspend/Resume**: l'esecuzione del work item può essere sospesa per liberare risorse computazionali a favore di work item con contratti più costosi;
- **Master fault restart**: la terminazione anormale di un processo master del grafo dei processi può essere gestita facendo riavviare l'esecuzione dell'intero grafo.

- **Resource Reservation:** permette di specificare l'utilizzo di alcune risorse fisiche, cioè viene specificato il mapping di alcuni nodi del grafo virtuale del componente, questo vincolo su un nodo è incompatibile col vincolo Power Constraint;
- **Time/Speed Constraint**: permette di specificare le prestazioni desiderate per l'esecuzione del work item in termini di tempo (tempo totale dell'esecuzione) o di carico di lavoro (numero di macro-operazioni per unità di tempo);
- **Power Constraint:** permette di specificare, per un nodo virtuale del grafo, la potenza effettiva desiderata. Questo vincolo su un nodo è incompatibile col vincolo di Resource Reservation.

Contratti ► Opzioni ▼	Premiun	Business	Standard	Economy	Low−cost
Redundant deploy	I	O	−	−	−
Share	−	−	−	I	I
Suspend/resume	−	−	−	−	ISD
Master fault restart	I	I	I	I	I
Resources reservation	I	−	−	−	−
Time/Speed constraints	ISD	−	−	−	−
Slave fault restart	ISD	ISD	−	ISD	−
Starvation	I	I	I	I	I
Power constraints	I	O	−	−	−

Tabella 1. Opzioni contrattuali disponibili per profilo di contratto

legenda: I (= included), O (= optional), ISD (=included if supported by Component),

4.2 Associazione di contratti a componenti

Come già accennato l'applicabilità di un profilo contrattuale dipende dalle caratteristiche non funzionali offerte dal componente. Ad esempio i contratti che prevedono le opzioni "Suspend/Resume" o "Time/Speed Constraints" possono associarsi a componenti che possiedono caratteristiche di controllo non banali. Nella tabella 2 sono schematizzate le possibili associazioni tra i tipi di contratto disponibili in base al tipo di componente attualmente identificati.

Contratti ▶ Componenti▼	Premiun	Business	Standard	Economy	Low−cost
Legacy Serial			X	X	
CCMap	X	X	X	X	X
CCFarm	X	X	X	X	X
Assist 1.2	X[1]		X	X	

Tabella 2. Profili contrattuali disponibili per i vari componenti

5 Patterns per il controllo della QoS

Come accennato nel paragrafo precedente ogni opzione contrattuale definisce dei vincoli qualitativi/quantitativi che caratterizzano l'esecuzione del work item. Al fine di poter monitorare ed intervenire sull'esecuzione sono stati pensati dei meccanismi e dei patterns che:

- abilitano un eventuale sistema di monitoraggio a rilevare delle situazioni di anomalie contrattuali o violazione delle opzioni contrattuali firmate col cliente;
- permettono al server di libreria di attuare le direttive di enforcement;

Nella tabella 3 sono riportati i patterns necessari per poter gestire le varie opzioni contrattuali. I patterns coinvolti sono:

- **Suspend/Resume**: permette di sospendere l'esecuzione di un work item per poi riprenderla successivamente;

[1] Sotto condizione di Resource Reservation totale e senza opzione di Power Constraint

- **Checkpoint Signal**: abilita un sistema di monitoring a ricevere dati rilevanti sullo stato esecutivo del work item attraverso la ricezione di eventi di checkpoint [6];
- **Garbage Collector:** politiche di pulizia delle risorse di griglia dai files schierati in seguito all'esecuzione di un componente;
- **Insured Completion:** permette di avere conoscenza, in qualsiasi momento, se un processo è stato terminato o è ancora in esecuzione. Questo pattern non consente di asserire eventuali situazione di stallo dei processi [3];
- **Resource availability & connectivity check**: verifica, ad intervalli regolari, lo stato delle risorse fisiche e delle connessioni significative tra loro[6];
- **Performance Tuning [Add/Remove slave]**: abilita un sistema di workflow management ad emettere delle direttive di enforcement al fine di reagire in caso di violazione delle opzioni contrattuali Time/Speed constraints e Power constraints [6].

Patterns ► QoS options ▼	Suspend resume	Checkpoint signal	Garbage collection	Insured completion	Perfor Tuning	Res-Avail Conn-Check
Redundant deploy						
Share						
Suspend/resume	x					
Master fault restart			x	x		
Resources reservation						x
Time/Speed constraint		x			x	
Power constraint					x	
Slave fault restart		x	x	x		

Tabella 3. Patterns necessari per il monitoraggio e controllo delle opzioni contrattuali.

6 JobInspector: Un sistema esperto per il monitoraggio ed il controllo reattivo

Come già detto le tecniche e le teorie descritte in questo documento sono state maturate in stretta sinergia con le attività svolte all'interno del Work Package 9. Tale attività prevede la realizzazione di un server di libreria in grado di amministrare proattivamente un set di componenti la cui esecuzione è vincolata dalle specifiche contrattuali negoziate sopra esposte. Si è cercato così di applicare queste tecniche a casi concreti e di realizzare dei meccanismi per dimostrare la validità delle teorie maturate. In questo paragrafo viene mostrato come, attraverso l'impiego di eventi, è stato possibile creare dei protocolli di controllo asincroni all'esecuzione del componente e viene proposta un'implementazione di controllore (JobInspector) di violazioni contrattuali in grado di rilevate situazioni anomale e di reagire in base a delle politiche pre-determinate e non note a priori.

6.1 QoS ed eventi

Come già accennato lo stato della QoS dipende sia dalla esecuzione del componente che da un insieme di avvenimenti esterni che si originano nell'ambiente di esecuzione. Essi si manifestano quindi asincronamente alla esecuzione delle procedure. Il paradigma di comunicazione tra gli attori interessati al monitoraggio e controllo dello stato contrattuale (componenti, Application Manager dell'applicazione ed il JobInspector) che più si presta a tali esigenze è quello basato su eventi. A tal scopo è stato integrato un bus ad eventi all'interno della macchina virtuale di griglia [5] per permette una comunicazione asincrona all'esecuzione del componente. Sono stati inoltri identificati un insieme di eventi significativi per il monitoraggio di alcune situazioni di interesse ai fini della valutazione contrattuale (sensori) e un insieme di eventi necessari per realizzare protocolli di enforcement per far fronte a deficit contrattuali.

In questo paragrafo sono spiegati gli eventi utilizzati per realizzare i patterns di monitoraggio e controllo dei work items. Di seguito sono descritti gli eventi classificati in base al namespace di appartenenza.

6.1.1.1 Eventi di sistema

Questi eventi sono emessi dal supporto runtime (VPG) e fungono da sensori per realizzare i patterns di “Insured completion”, “Perfor Tuning” e “Res-Avail Conn-Check”. Il namespace di appartenenza è chiamato **SYSTEM_VPG_POOL.** Tali eventi sono:

1. **KeepAlive** (**VPG_KEPA**): inviato dal VPGMaster a tutti i demoni per tenerli in vita. L'intervallo di invio degli eventi è stabilito attraverso il file di configurazione.
2. **HeartBeat** (**VPG_HB**): inviato dalla RemoteEngine per indicare che il nodo è vivo, viene comunicato un indice sulla potenza corrente effettiva del nodo (carico libero di CPU corrente moltiplicato per la potenza nominale). L’intervallo temporale di invio dell’evento è stabilito da file di configurazione.
3. **Connection missing** (**VPG_NP2P):** indica la mancanza di connessione tra due Remote Engine. L'intervallo temporale di ping tra le remote engine è stabilito via file di configurazione.
4. **Process Start (VPG_PSTRT)**: indica che un processo è stato lanciato con successo sulla VPG. Viene emesso dal nodo in cui il processo è stato lanciato.
5. **Process Fault (VPG_PFLT)**: indica la terminazione di un processo con esito negativo. Viene emesso dal nodo in cui il processo è stato lanciato.
6. **Process End(VPG_PEND):** indica che un processo ha finito la sua esecuzione. Viene emesso dal nodo in cui il processo è stato lanciato.
7. **Process Suspend (VPG_PSUSP)**: indica che un processo è stato sospeso. Viene emesso dal nodo in cui il processo è stato lanciato.
8. **Process Continue (VPG_PCONT)**: indica la riattivazione di un processo sospeso. Viene emesso dal nodo in cui il processo è stato lanciato.
9. **Esecuzione di un processo del grafo fallita (VPG_GFLT):** evento emesso dal front-end della macchina di griglia, indica che l'esecuzione del grafo dei processi di un componente è terminata abnormalmente,. Questo evento è generato da:
    a. caduta di un nodo : vengono indicati gli indici del nodo caduto;
    b. caduta di una connessione: viene indicata l'identificatore della connessione caduta;

c. caduta di un processo: un processo ha terminato abnormalmente, viene comunicato l'indice del processo terminato.

10. **Esecuzione del grafo di processi terminato con successo (VPG_GEND)**: tutti i processi di un grafo hanno terminato con successo.

La tabella 4 riassume tali eventi mostrando gli attori interessati e i dati trasmessi con gli eventi.

Event	***From***	***To***	***Contentuto dell'evento nella forma: nome (tipo): descrizione***
VPG_KEPA	VPGMaster	Demon	nodeId (string): identificatore del nodo che emette l'evento; value (int): Valore costante (=1)
VPG_HB	Demon	VPGMaster JobInspector	nodeId (string): identificatore del nodo che emette l'evento; index (float) : potenza istantanea del nodo. Potenza Nominale moltiplicata per un fattore compreso tra zero e uno;
VPG_NP2P	Demon	VPGMaster	nodeId (string): identificatore del nodo che emette l'evento; to (string): identificatore del nodo all'altro capo della connessione
VPG_PSTR	Demon	VPGMaster	nodeId (string): identificatore del nodo che emette l'evento; id (string): identificatore del processo pid (int) : pid del processo
VPG_PEND	Demon	VPGMaster	nodeId (string): identificatore del nodo che emette l'evento; id (string) ; identificatore del processo
VPG_PFLT	Demon	VPGMaster	nodeId (string): identificatore del nodo che emette l'evento; id (string) ; identificatore del processo
VPG_PSUSP	Demon	VPGMaster	nodeId (string): identificatore del nodo che emette l'evento; Id: identificatore del processo
VPG_PCONT	Demon	VPGMaster	nodeId (string): identificatore del nodo che emette l'evento; Id: identificatore del processo

Event	*From*	*To*	*Contentuto dell'evento nella forma: nome (tipo): descrizione*

VPG_GFLT	VPGMaster	Inspector	• graphId (string): identificatore del grafo
			• code (int): indica il tipo di fult:
			  • 1: caduto nodo fisico;
			  • 2: caduto processo;
			  • 3: caduta connessione;
			• resId (string): identificatore del nodo fisico/ indice del nodo caduto/ identificatore processo virtuale

VPG_GEND	VPGMaster	Inspector	• graphId (string): identificatore dell'istanza di grafo.

Tabella 4. Attori coinvolti e dati trasmessi con gli eventi

6.1.1.2 Eventi relativi a Grafi di processi

Esso contiene eventi relativi al ciclo divita del grafo dei processi relativo ad un componente. Il namespace di appartenenza è chiamato GRACE_SYSTEM_POOL. Tali eventi hanno un duplice scopo: quelli emessi dal VPGMaster e dal Manager del componente (prefisso ENA) generalmente sono rivolti ad un sistema di monitoraggio dell'esecuzione (JobInspector) per tenere traccia dello stato dell'esecuzione; gli eventi emessi dal JobInspector invece sono necessari per emettere le direttive di enforcement (prefisso QOS):

1. **ENA_INIT**: conferma l'avvenuta inizializzazione del componente, ciò significa che tutti i file necessari per la sua esecuzione sono schierati sulla griglia;
2. **ENA_STRT** : è stato lanciato il grafo dei processi di un componente.
3. **ENA_CALL**: è stata sottomessa la chiamata RPC al componente;
4. **QOS_END:** conferma la terminazione dell'esecuzione del grafo di processi;
5. **QOS_RSKD**: forza il rescheduling del work item processato dal componente;
6. **QOS_ABT**: forza la terminazione dell'esecuzione di un componente;
7. **QOS_INC:** incrementa/decrementa la potenza di nodo virtuale del grafo dei processi del componente;
8. **QOS_SWP:** cambia un processo del grafo dei processi del componente;

La tabella 5 riassume tali eventi mostrando gli attori interessati e i dati trasmessi con gli eventi.

Event	***From***	***To***	***Contentuto dell'evento nella forma: nome (tipo): descrizione***

ENA_INIT	WIHandler	JobInspector	• graphId (string): identificatore dell'istanza di grafo del componente;
			• componentURI (string): URI di descrizione del componente;
			• start (long): momento di sottimissione del contratto (secondi da Epoch: 00:00:00 UTC, 1 gennaio 1970);
			• contractDuration (long): durata del contratto espresso in secondi;
			• physicalGraph(string): descrizione del grafo fisico;
			• contractType (int) : tipo di contratto;
			• elenco delle opzioni contrattuali, attualmente le opzioni previste sono:
			  • starvation (long): durata massima di un workitem espresso in secondi "%d";
			  • resourceReservation : opzione del contratto "Resource Reservaion" in docmento XML (QoSGraph);
			  • powerConstraint : opzione del contratto "Power Reservation" in docmento XML (QoSGraph);
			  • speedConstraint : valore dell'opzionie contrattuale "speedConstraint", formato della stringa "%f %f" (velocità tolerance);
			  • suspend_resume : 1;
			  • slaveFaultRestart : 1;
			  • masterFaultRestart : 1;

ENA_STR	WIHandler	JobInspector	• graphId (string): identificatore dell'istanza di grafo;
ENA_CALL	WIHandler	JobInspector	•graphId (string): identificatore dell'istanza di grafo;
QOS_END	JobInspector	WIHandler	•graphId (string): identificatore dell'istanza di grafo;
QOS_ABT	JobInspector	WIHandler	•graphId (string): identificatore dell'istanza di grafo;

Event	*From*	*To*	*Contentuto dell'evento nella forma: nome (tipo): descrizione*
QOS_RSKD	JobInspector	WIHandler	• graphId (string): identificatore dell'istanza di grafo;
QOS_INC	JobInspector	WIHandler	• graphId: (string): identificatore dell'istanza di grafo; • nodeId (string) indice del nodo virtuale interessato; • deltaPower (float):valore di incremento/ decremento di potenza
QOS_SWP	JobInspector	WIHandler	• graphId: (string): identificatore dell'istanza di grafo; • nodeId (string) indice del nodo virtuale interessato;

Tabella 5. Attori coinvolti e dati trasmessi con gli eventi

6.1.1.3 Eventi scambiati con il componente

I seguenti eventi sono scambiati tra il Manager, il JobInspector ed il componente, alcuni di essi hanno lo scopo di riconfigurare il componente (prefisso ENF) invece altri sono emessi dal componente a conferma delle operazioni di enforcement (prefisso C).

1. **ENF_SUSP**: direttiva di sospensione dell'elaborazione in corso;
2. **ENF_RESU**: direttiva di rispresa dell'eleborazione in presecedenza sospesa;
3. **ENF_SLAVE_JOIN**: direttiva di aggiunta di uno o più processi al grafo dei processi;
4. **ENF_SLAVE_DISJOIN**: direttiva di rimozione di uno o più processi al grafo dei processi;
5. **C_START:** conferma dell'inizio dell'esecuzione di un componente;
6. **C_SUSP**: conferma di sospensione;
7. **C_RESU**: conferma che di rispresa dell'eleborazione;
8. **C_RET:** conferma di fine elaborazione;
9. **C_SLAVE_JOIN/C_SLAVE_DISJOIN**: conferma esito di variazione del grafo dei processi;

10. **C_CHECKPOINT**: contiene dati rilevanti per la valutazione della velocità di esecuzione dell'elaborazione;

La tabella 6 riassume tali eventi mostrando gli attori interessati e i dati trasmessi con gli eventi.

Event	***From***	***To***	***Contentuto dell'evento nella forma: nome (tipo): descrizione***
ENF_SUSP	WIHandler	Component	•id (string): identificatore dell'istanza di componente;
ENF_RESU	WIHandler	Component	• id (string): identificatore dell'istanza di componente;;
ENF_SLAVE_JOIN	WIHandler	Component	• id (string): identificatore dell'istanza di componente; • physicalGraph (string): grafo fisico di variazione
ENF_SLAVE_DISJOIN	WIHandler	Component	• id (string): identificatore dell'istanza di componente; • physicalGraph (string): grafo fisico di variazione
C_START	Component	JobInspector WIHandler	•id (string): identificatore dell'istanza di componente;
C_SUSP	Component	JobInspector WIHandler	•id (string): identificatore dell'istanza di componente;
C_RESU	Component	JobInspector WIHandler	• id (string): identificatore dell'istanza di componente;
C_RET	Component	JobInspector WIHandler	• id (string): identificatore dell'istanza di componente; • retCode (int): codice esito operazione; • value (string): stringa contenente le informazioni di esecuzione;

C_SLAVE_JOIN	Component	JobInspector WIHandler	• id (string): identificatore dell'istanza di componente;
			• physicalGraph (string): grafo fisico di variazione effettuato; per ogni nodo del grafo fisico viene inserito l'esito dell'operazione:
			  • minReqPower = 0: esito negativo
			  • minReqPower=maxReqPower: esito positivo;

Event	*From*	*To*	*Contentuto dell'evento nella forma: nome (tipo): descrizione*

C_SLAVE_DISJOIN	Component	Inspector WIHandler	• id (string): identificatore dell'istanza di componente;
			• physicalGraph (string): grafo fisico di variazione effettuato; per ogni nodo del grafo fisico viene inserito l'esito dell'operazione:
			  • minReqPower = 0: esito negativo
			  • minReqPower=maxReqPower: esito positivo;

C_CHECKPOINT	Component	JobInspector WIHandler	• id (string): identificatore dell'istanza di componente;
			• value (string): stringa contentente le informazioni di checkpoint, formato stringa: "%d %d %f %f %f %f" con i valori:
			  • id_checkpoint
			  • step
			  • tempo esecuzione ultima iterazione tempo totale
			  • numero operazioni ultima iterazione
			  • numero operazioni totali

Tabella 6. Attori coinvolti e dati trasmessi con gli eventi

6.2 Caratteristiche del sistema

Il Job Inspector è un modulo software in grado di monitorare l'esecuzione di un componente per verificare le adempienze contrattuali firmate col cliente. In caso di violazioni delle opzioni contrattuali esso ha il compito di suggerire al component Administrator delle direttive di enforcement per riportare l'esecuzione entro i limiti stabiliti.

Vista la natura e la complessità del problema affrontato si è pensato di non impiegare un paradigma di programmazione procedurale nella realizzazione di questo componente ma di utilizzare tecniche che permettessero una codifica della "conoscenza sul dominio" anziché una codifica algoritmica (e deterministica) della soluzione. Un sistema di questo tipo è definito come "Sistema Esperto" [15] poiché è possibile modellare e simulare la conoscenza umana su uno specifico problema.

Queste considerazioni hanno pesato nella scelta di avvalersi di soluzioni classiche dell'implementazione di sistemi esperti e hanno comportato l'impiego di un tool per la realizzarizione di sistemi esperti denominato CLIPS (C Language Integrated Production System) [12]. Questo tool permette, attraverso un linguaggio di programmazione logica, di codificare una base di conoscenza KB (fatti, classi e regole) relativa ad un dato problema e, a partire da questa base, creare un motore di inferenza logica. Il vantaggio principale di questo sistema è che, la base di conoscenza può essere aggiornata continuamente, anche run-time, senza intervento sull'implementazione del sistema esperto.

L'impiego del tool CLIPS dunque rende il JobInspector completamente scalabile. Ciò poiché le regole iniziali possono essere modificate con l'aumentare della complessità e del numero delle opzioni contrattuali da controllare. In particolare le caratteristiche salienti del sistema sono illustrate nel seguente prospetto:

- possibilità di caricare dinamicamente uno o più sorgenti CLIPS scelti in base alle opzioni contrattuali da controllare;
- possibilità di instanziare qualsivoglia numero di sistemi esperti in grado di eseguire un insieme di sorgenti CLIPS. Ad ognuno di questi sistemi è naturalmente associato un apposita istanza di ambiente, cioè una collezione di classi, fatti e regole codificati nei sorgenti CLIPS caricati e che si riferiscono all'esecuzione di componente con le relative opzioni contrattuali;
- possibilità di aggiornare dinamicamente una determinata istanza di ambiente in qualsiasi momento l'Inspector riceva dall'esterno nuove informazioni sull'esecuzione del componente;

6.3 Funzionamento del sistema

Nei paragrafi precedenti è stata dimostrata la natura *event-driven* del sistema e come, sia il monitoraggio che il controllo, vengano effettuati asincronamente all'esecuzione del componente sulla base degli eventi che il sistema riceve. Inoltre è stato detto che la parte più significativa del sistema è basato su di un sistema esperto generato attraverso l'impiego del tool *CLIPS*. Quindi è opportuno iniziare un'analisi del funzionamento del sistema partendo da quest'ultimo punto: il CLIPS. Esso fornisce sia un editor visuale utile sia per costruire la base di conoscenza del sistema

(fatti, classi e regole) che per validare il sistema risultante. Inoltre CLIPS fornisce una libreria C per poter utilizzare il motore di inferenza all'interno di programmi esterni. La base di conoscenza del sistema viene codificata attraverso l'utilizzo di un linguaggio (anch'esso chiamato CLIPS) e può essere memorizzata su file testuali. Una nota importante è la base di conoscenza utilizza dal sistema esperto creato può risiedere su più file ognuno dei quali caricabili dinamicamente a run-time in modo tale da poter "cambiare" la base di conoscenza del sistema durante la sua esecuzione. Tramite il semplice paradigma delle regole e dei fatti tipico dei programmi scritti in CLIPS, sono stati stilati diversi sorgenti, ognuno dei quali correlato ad una specifica opzione contrattuale da monitorare e controllare (si rimanda all'appendice A per un elenco completo dei codici CLIPS attualmente utilizzati).

In linea di principio vi possono essere due tipologie di file CLIPS. Per quanto riguarda la prima, vi rientra in effetti un solo file di codice sorgente CLIPS e in questo vi si trovano tutte le regole, le strutture dati e i fatti che sono basilari per il funzionamento minimale del sistema. Il codice contenuto in questo file implementa la macchina a stati generica in grado di monitorare il cambiamento di stato di un componente. In altre parole gli stati e le transizioni di questa macchina a stati sono comuni a quelle di tutti gli altri componenti, compresi i più complessi. Per quanto riguarda la seconda tipologia di file, occorre dire che ogni diversa QoS option richiede uno o più ben determinati pattern di controllo e quest'ultimo è implementato attraverso un programma *CLIPS* contenuto in un file dedicato.

Attraverso la composizione di un opportuno sottoinsieme di questi file è possibile ottenere un sistema esperto che implementa la logica per il controllo e il monitoraggio di un work-item legato ad un ben determinato tipo di contratto con delle precise QoS options. Infatti poiché nell'evento *ENA_INIT*, a cui il sistema è registrato, fra gli altri, viaggiano anche informazioni dettagliate sul contratto relativo ad un determinato *work item* e sulla tipologia di componente, il sistema stesso è in grado di selezionare e quindi caricare i file di codice sorgente *CLIPS* opportuni, costruendo così un ambiente *CLIPS* su misura per il controllo e il monitoraggio del *work item* in questione.

Infine l'ultimo componente è costituito dalle procedure C richiamate dal sistema esperto.. Tali procedure possono avere dei compiti connessi al controllo di QoS di natura deterministica, che per ragioni di efficienza non sono state implementate in *CLIPS*. Al fine di poter asserire nuovi fatti all'interno di tali procedure, sono stati predisposti un insieme di eventi interni al JobInspector detti *pretended event,* che possono essere emessi da tali procedure e che, come per

gli eventi esterni, vengono tradotti in istanze di classi CLIPS al fine di far evolvere il sistema esperto. In figura 3 è riassunto lo schema di funzionamento del appena descritto. All'arrivo dell'evento ENA_INIT il JobInspector, dopo una analisi del contratto associato all'esecuzione del componente, carica i file CLIPS necessari al monitoraggio e istanzia un sistema esperto dedicato (CLIPS environment).

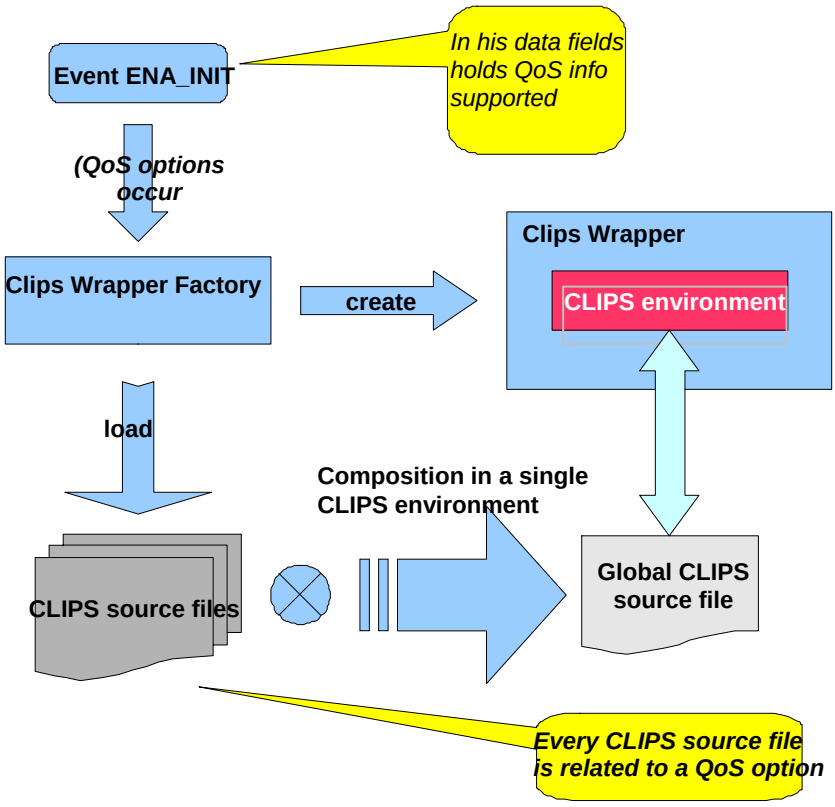

Figura 3. Rappresentazione del funzionamento generale del sistema

Quindi, tirando le somme, i fattori chiave che consentono al meccanismo di controllo e monitoraggio di funzionare sono tre: i file *CLIPS* con l'implementazione dei pattern di controllo, le procedure per la conversione degli eventi catturati dal gestore degli eventi nelle forme previste dai programmi *CLIPS* (classi) ed infine, qualora il pattern lo richiedesse, le procedure invocate con gli eventuali *pretended event* connessi.

6.4 Implementazione del prototipo

6.4.1 Architettura del sistema

I componenti principali del sistema, da un punto di vista tecnico-implementativo sono l'eseguibile e i vari sorgenti CLIPS. L'eseguibile si avvale delle seguenti librerie dinamiche di supporto: ALI [2], SEDL [13] e ACE framework [14]. Inoltre è stata realizzata una libreria C che funge da wraooer per il pacchetto CLIPS [12]. Per quanto riguarda i sorgenti CLIPS, essi sono stati redatti attraverso un editor testuale e per essi non è necessaria alcuna sorta di compilazione.

La figura 4 si mostra una vista a livelli delle librerie utilizzate dal sistema. Le componenti rappresentate nel livello superiore usano i servizi messi loro a disposizione dalle componenti rappresentate dai blocchi del livello inferiore.

La decomposizione in sottosistemi è rappresentata invece dal diagramma UML raffigurato in figura 5 in cui vengono mostrate le relazioni tra le varie librerie usate nel sistema, sia quelle statiche che quelle dinamiche. In figura 6 invece sono riportate le principali classi del sistema.

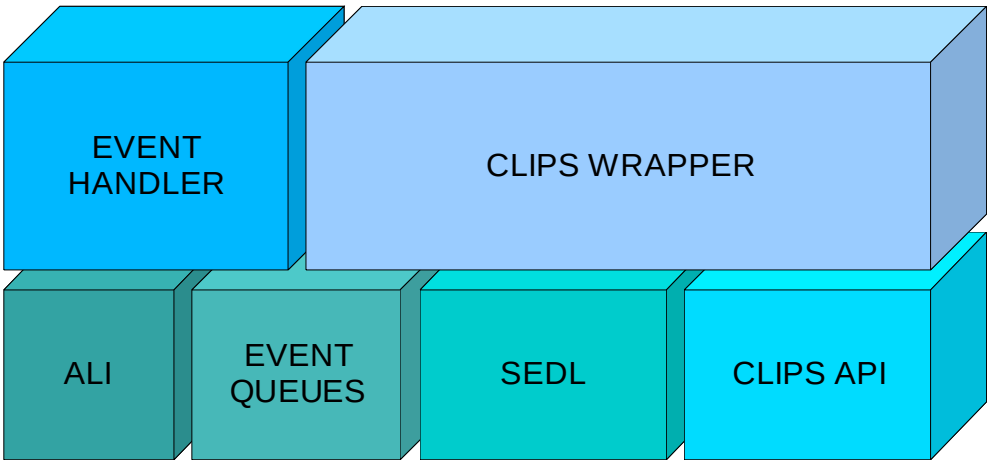

Figura 4. Vista dei componenti nei livelli della pila che rappresenta l'architettura del sistema

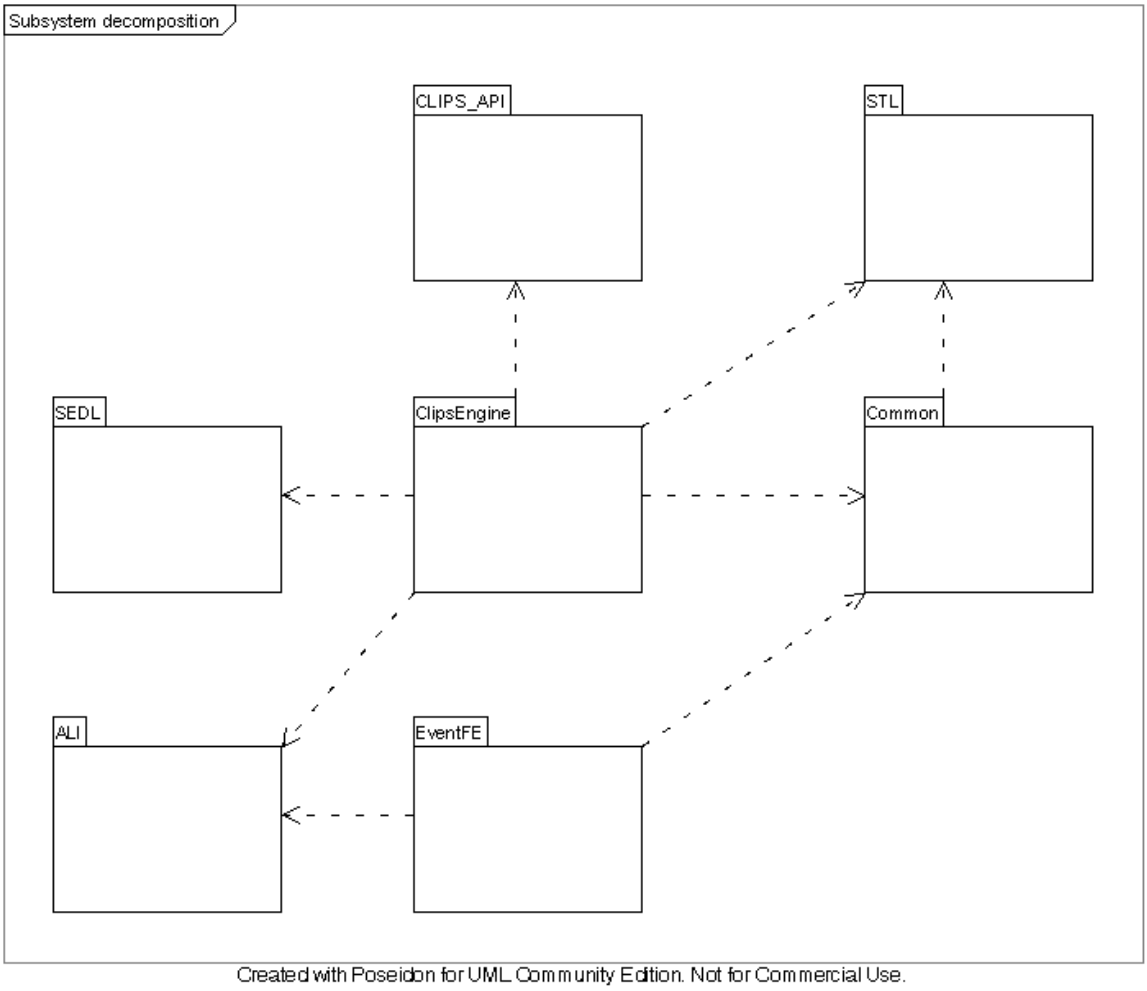

Figura 5. Decomposizione in sottosistemi del sistema

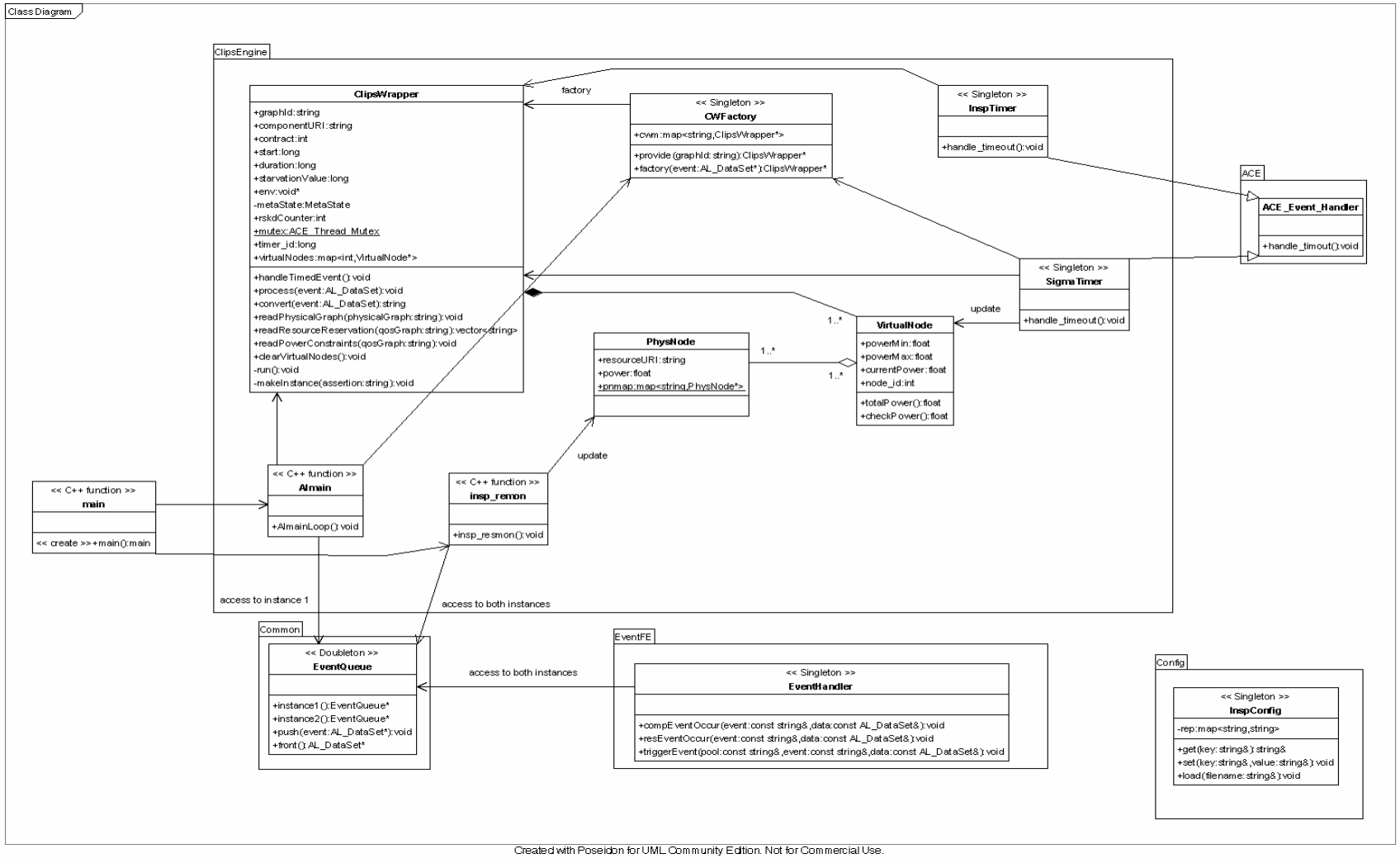

Figura 6. Diagramma delle classi principali del sistema

6.4.2 Concorrenza

I seguenti requisiti non funzionali che il sistema deve soddisfare sono connessi a situazioni di concorrenza tra i thread del programma:

- ◊ Capacità di catturare gli eventi che in un qualsiasi istante giungano dall'esterno, o eventualmente dall'interno;
- ◊ capacità di processare gli eventi che riguardano il ciclo di vita di un *work item* in modo asincrono al fine di monitorarlo o controllarlo anche attraverso l'uso dell'apposito codice scritto in CLIPS;
- ◊ capacità di processare gli eventi che riguardano le risorse fisiche al fine di mantenere costantemente aggiornata la struttura dati che le rappresenta;
- ◊ capacità di aggiornare costantemente i valori di potenza relativi ai nodi virtuali dei vari work item monitorati attraverso l'aggregazione dei valori di potenza connessi alle risorse fisiche relative al nodo virtuale stesso;
- ◊ capacità di emettere eventi di *enforcement* verso l'esterno, al fine di comunicare direttive di controllo ai componenti preposti ad accoglierle.

A tali requisiti non funzionali si è data risposta ideando un architettura del sistema che consiste nell'attivazione e nella contemporanea esecuzione di più thread. I thread del sistema sono:

- ◊ *il thread per la gestione degli eventi:* si occupa di rimanere in ascolto di eventuali eventi provenienti dall'esterno oppure dall'interno; esso inoltre si preoccupa di discernere la natura dell'evento, cioè se questo abbia a che fare direttamente con un work item monitorato oppure con una risorsa fisica, al fine di smistarne la collocazione in un apposita struttura di memoria condivisa piuttosto che in un altra;
- ◊ *il thread per il controllo dei work item:* si occupa di prelevare, dall'apposita area di memoria condivisa, gli eventi relativi ai componenti e di processarli in modo asincrono; questo potrebbe richiedere l'inferenza del sistema esperto ed infine questo thread può emettere eventi verso l'esterno nel caso le regole attivate dal sistema esperto lo richiederesso;
- ◊ *il thread per il monitoraggio delle risorse fisiche:* si occupa di prelevare, dall'apposita area di memoria condivisa, gli eventi relativi alle risorse fisiche raccolti dal primo thread e di

processarli in modo asincrono al fine di tenere aggiornata la struttura di memoria che rappresenta l'insieme delle risorse fisiche gestite;

◊ *il thread per il monitoraggio dei nodi virtuali:* si occupa di aggiornare periodicamente il valore di potenza di ciascuno dei nodi virtuali di ogni work item monitorato, attraverso l'aggregazione dei valori di potenza relativi alle risorse fisiche da cui il nodo virtuale è composto; è importante evidenziare che questo thread effettua anche un primo controllo sul rispetto dei vincoli di potenza da parte dei nodi virtuali dei vari work item (QoS option: *Power Constraint e Resource Reservation con vincolo di potenza*) e qualora riscontri un anomalia, è in grado di emettere un *pretended event* per permettere al *thread per il controllo dei work item* di applicare l'opportuno pattern di controllo.

6.4.3 Gestione dei dati

Si è detto che alcuni thread accedono a delle aree di memoria condivise. In figura 7 sono schematizzate le interazioni tra i vari threads del sistema tramite tali strutture dati. Si tratta di due code dalla struttura assolutamente identica in quanto entrambe servono a memorizzare gli eventi trattati dal sistema ed entrambe sono del tipo *FIFO*. La prima coda serve a contenere eventi relativi ai componenti ed è condivisa tra il *thread per la gestione degli eventi* che vi accede in scrittura insieme al *thread per il monitoraggio dei nodi virtuali* e il *thread per il controllo dei work item* che vi accede in lettura. La seconda coda serve a contenere eventi relativi alle risorse fisiche ed è condivisa tra *il thread per la gestione degli eventi* che vi accede in scrittura e *il thread per il monitoraggio delle risorse fisiche* che vi accede in lettura.

Vi è una seconda importante area di memoria condivisa. In questa vengono memorizzate informazioni relative alle risorse fisiche usate. L'accesso in scrittura a quest'area di memoria è di pertinenza del *thread per la gestione delle risorse fisiche*, l'accesso in lettura è di pertinenza del *thread per la gestione dei nodi virtuali*.

Un’altra importante struttura di memoria condivisa a livello globale è quella legata alle proprietà generali del sistema. Essa contiene le più importanti proprietà del sistema che gli vengono passate in fase di inizializzazione del programma. Sono previsti dei metodi get e set per il recupero o l'immissione delle proprietà custodite o da custodire all'interno.

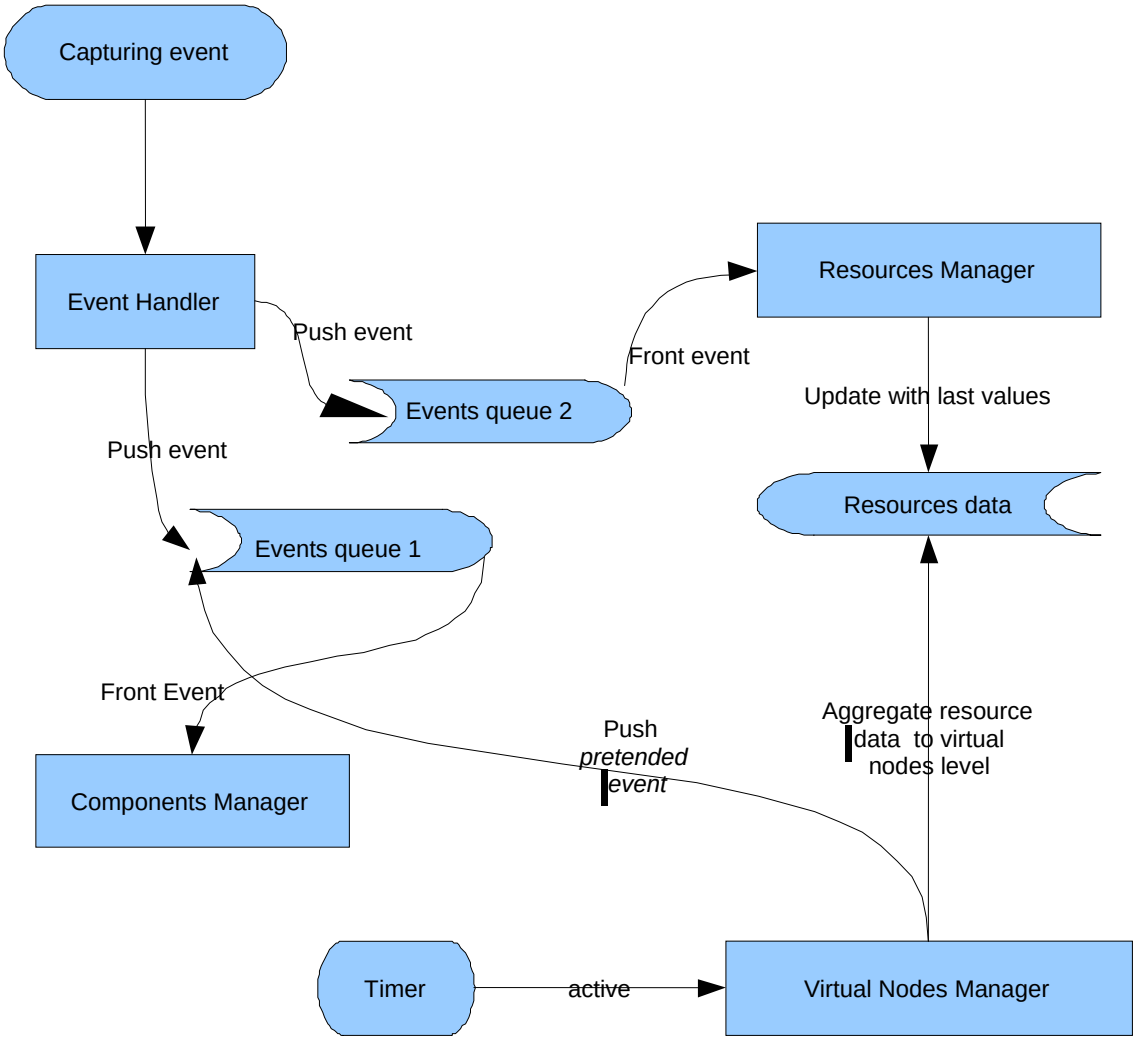

Figura 7. Rappresentazione dell'interazione tra i thread del sistema

6.4.4 Controllo del software

Il controllo del software è del tipo *event-driven* con la presenza di più thread in esecuzione simultanea. Il thread che svolge il compito primario è *il thread per il controllo dei work item*, tutti gli altri svolgono compiti di ausilio al primo. Il fatto che questo compito sia affidato ad un solo thread, esso stesso *event-driven*, comporta che gli eventi vengano processati sequenzialmente, estraendoli dall'apposita coda *FIFO*. Ciascun evento relativo ad un work item viene tradotto in una forma compatibile con il codice *CLIPS* (istanza di classe) in modo da accrescere la base di conoscenza del sistema esperto. La fase successiva è quella di attivare il motore di inferenza logica del sistema esperto. Ciò potrebbe aggiornare ulteriormente la base di conoscenza e inoltre, potrebbero essere emessi degli eventi di *enforcement* verso l'esterno, in quanto sono state costruite delle funzioni *CLIPS* atte a farlo. Al termine del processamento di un evento il sistema è di nuovo pronto a processare un nuovo evento estraendolo dalla coda qualora questa non fosse vuota.

6.4.5 Note sul linguaggio CLIPS

Il linguaggio *CLIPS* è un linguaggio pensato per facilitare lo sviluppo di software basato sulla conoscenza o esperienza umana. Essenzialmente un programma è fatto da un insieme di regole che analizzano delle informazioni (fatti - forniti da un utente o da un sistema esterno) relative ad una classe specifica di problemi. Tali regole formulano una soluzione a partire dalle informazioni analizzate. I software così realizzati vengono detti "Sistemi Esperti" poiché è possibile modellare e simulare la conoscenza umana su uno specifico problema. Generalmente i sistemi esperti sono utili in situazioni in cui il problema da risolvere non possiede una unica soluzione "corretta" che è possibile codificare tramite un algoritmo (si pensi ad esempio al problema della diagnosi medica). Il linguaggio CLIPS si compone essenzialmente di:

- FATTI: descrivono la realta', ogni fatto è una rappresentazione formale ad alto livello di una informazione specifica al dominio di applicazione;
- REGOLE: insieme di azioni attivate a partire da una data situazione (insieme di fatti); l'insieme delle regole definite lavorano collettivamente per risolvere il problema;
- CLASSI e OGGETTI: modo formale (orientato agli oggetti) per modellare i fatti;
- FUNZIONI: rappresentano la conoscenza algoritmica;

Un sistema esperto basato su regole e' un programma *data-driven* dove i fatti sono i dati che attraverso il motore inferenziale stimolano l'esecuzione delle regole. Il CLIPS è simile al Prolog, ma mentre quest'ultimo si basa sul principio del partire dalle conclusioni per risalire alle premesse (il cosiddetto *backtracking*), il primo invece si basa sul principio opposto (il cosiddetto *forwardtracking*). In questo tipo di logica di programmazione più di una regola potenzialmente applicabile nello stesso momento. Tali tipi di situazioni possono essere gestite in diversi modi, il più semplice dei quali è quello di associare ad ogni regola una priorità (la cosiddetta *salience*). Inoltre occorre sottolineare che stabilito in qualche modo l'ordine di priorità delle regole rispetto a una ben precisa situazione di fatti, queste saranno eseguite una dopo l'altra, nell'ordine dettato dalla loro priorità. Per chiari motivi uno stesso fatto, a meno che non sia cancellato e ripetuto, non può far *scattare* la stessa regola più di una volta, ma può far scattare diverse regole una dopo l'altra. Se nella catena delle regole applicabili a una ben precisa situazione di fatti, ve ne è una che prevede la cancellazione di uno o di più fatti, allora le successive regole nella catena non potranno essere applicate. Questi pochi è semplici meccanismi sono alla base della scrittura di buoni programmi in linguaggio *CLIPS*.

Nei sorgenti CLIPS redatti, sono state usate due classi. Con la prima sono state rappresentate le caratteristiche base di un evento del sistema, con la seconda le caratteristiche di uno stato della *macchina a stati* di un ipotetico componente. Sono state preferite le classi ai fatti in quanto le prime godono della proprietà dell'ereditarietà, che è stata sfruttata per definire quegli eventi e quegli stati che specializzano la corrispondente classe base.

Segue la definizione della classe che è stata usata per rappresentare l'evento base:

```
(defclass EVENT (is-a USER)
     (slot event
          (type SYMBOL))
)
```

Segue la definizione della classe che è stata usata per rappresentare lo stato base:

```
(defclass STATE (is-a USER)
     (slot state
          (type SYMBOL))
)
```

6.4.5.1 Salience delle regole

Il meccanismo che è stato utilizzato per dirimere i conflitti di *attivazione* di più regole è stato quello della *salience*. Ciò comporta che nell'aggiunta di nuove regole, occorre considerare eventualmente la *salience* delle regole già esistenti al fine di consentire il corretto ordine nell'applicazione delle regole concorrenti. Nei casi in cui l'obbiettivo non fosse quello di stabilire l'ordine di esecuzione delle regole, ma piuttosto quello dell'esecuzione di una certa regola a scapito di una certa altra, oltre a regolare opportunamente la salience si dovrebbe ricorrere, nell'*RHS* della regola privilegiata, alla cancellazione dei fatti che possono far scattare le altre regole concorrenti.

6.4.5.2 Garbage collection

Vista la natura *event-driven* del sistema si è dovuto modellare l'evento come Classe CLIPS in modo tale instaziare un nuovo oggetto all'interno dei fatti CLIPS ad ogni arrivo di un nuvo evento. L'evento immesso può dunque far scattare un insieme di regole oppure no (i.e.: evento non coerente con l'attuale stato del work item). Ad ogni modo prima che il motore di inferenza termini è necessario eliminare l'evento dall'ambiente *CLIPS*, onde evitare che questo influisca nelle successive inferenze. A tale scopo è stata compilata una regola denominata g*arbageCollection* alla quale è stato associato il valore di salience più basso previsto per le regole del programma. In base a quanto detto

la regola di g*arbageCollection* è l'unica a scattare, con certezza, ad ogni esecuzione dell'ambiente ed è comunque quella che viene applicata per ultima. Tutto ciò consente in effetti che il sistema sia guidato sempre dall'ultimo evento immesso nell'ambiente.

La definizione della regola detta di *garbage collection è la seguente*

```
(defrule garbageCollection
     (declare (salience -100))
     ?tbd <- (object (is-a EVENT))
=>
     (send ?tbd delete)
)
```

6.4.6 Uso del software

Come già detto il software si compone di un file eseguibile e di una serie di altri files testuali. Il programma si serve, durante il suo funzionamento, dei file scritti in CLIPS e del file di configurazione, caricandoli, interpretandoli e/o leggendoli. Segue un elenco, con la relativa spiegazione, dei vari parametri usati dal sistema, comunque si vuole sottolineare che attraverso l'uso dell'opzione “-h” oppure “--help” nell'invocazione del programma, si ottiene la stampa nella shell di lavoro di un prospetto relativo ai parametri e al loro uso:

opzione	Significato
-m|--master-event <vpgmaster>:<port>	set the hostname of the vpgmaster <vpgmaster> and his port <port> (default is "localhost:12202")
-f|--file <configurationfile>	load the configuration file <configurationfile> (default: no file loaded)
-t|--rtime <time>	Fissa a <rtime> l'intervallo di tempo che intercorre tra due operazioni successive di aggregazione di particolari parametri relativi ai nodi fisici per ottenere il valore totale legato al corrispondente nodo virtuale (per esempio la potenza effettiva). I valori così ottenuti subiscono quindi un ulteriore processamento dai quei componenti del sistema preposti ad operare su valori direttamente legati ad

opzione	Significato
	un work item
-r|--rskd <max_reskedulations>	Fissa il numero massimo di volte che un work item che per qualche motivo termina in modo anomalo può essere rischedulato pari a <max_reskedulations>.
-s|--starv <starvation time>	Valore di default usato per la starvation di un work item nel caso questa non fosse specificata da contratto (chiaramente è opportuno inserire un valore molto alto).
-c|--clips <clips sources folder>	get the clips sources from the folder specified (default:
-h|--help	print this help GNU style
-v|--version	print the version of this software

6.4.7 Realizzazione dei patterns di monitoraggio e controllo

Ogni work item in esecuzione nel sistema, ha una vita che dipende dal ciclo di vita del componente e dal tipo di contratto con il quale è associato. Alla prima dipendenza si possono associare queste formule “in che stato il componente si può trovare e quali transizioni sono possibili?”, alla seconda dipendenza si possono associare invece queste formule “quali stati e quali transizioni sono tollerabili e quali sono i valori consentiti delle proprietà del componente in corrispondenza di ciascuno degli stati in cui questo può trovarsi ad un determinato istante?”. Parte delle situazioni o degli scenari in cui può trovarsi un work item nel corso della sua vita sono trasversali sia rispetto al tipo di componente, sia rispetto al tipo di contratto. Altre situazioni sono specifiche di un particolare tipo di componente, altre invece sono legate essenzialmente al tipo di contratto. Ad ognuna di queste situazioni il sistema è chiamato a rispondere in qualche modo e tramite l'azione di una o più delle sue componenti. A ciascuna di queste ultime competono problematiche ed azioni ad un livello che dipende dalla loro posizione nella struttura a strati in cui il sistema si può immaginare ricondotto. Infine la presenza di QoS options in un contratto generalmente determina che il tipo di azione intrapresa in presenza di una determinata situazione sia più complesso ed efficace rispetto a quello standard.

Di seguito si illustra la sequenza delle azioni e degli eventi a cui da luogo l'applicazione effettiva dei patterns in corrispondenza degli scenari correlati alle varie QoS options:

- **Suspend/Resume**: in questo caso l'inspector si limita semplicemente a raccogliere gli eventi di sospensione o ripristino dell'attività di un work item per aggiornare la sua l'informazione sullo stato del work item consistente. Tali eventi sono emessi dal componente e sono rispettivamente C_SUSP e C_RESU;
- **Master_fault_restart:** quando si verifica la caduta del master di un work item l'inspector raccoglie questo evento e intraprende un azione che dipende dal codice dell'evento:
  - Se il codice è 2 (il processo master è caduto) l'azione intrapresa è l'emissione dell'evento di QOS_ABT per forzare la fine dell'esecuzione;
  - se il codice è 1 o 3 (il master è caduto a causa della caduta del nodo nel quale risiedeva o di una connessione facente capo allo stesso nodo) l'azione intrapresa è l'emissione dell'evento di QOS_RSKD per rischedulare l'esecuzione del componente;
- **Resource reservation**: in questo caso occorre rilevare la caduta di una o più risorse e/o la diminuzione della potenza associata alle risorse nelle quali sono schierati i processi afferenti al work item. I sensori utilizzati sono gli eventi HeartBeat emessi dalle RemoteEngine. L'inspector raccoglie questi eventi e sia nel caso di caduta di una o più risorse che in quello di diminuzione della potenza aggregata al di sotto del livello minimo consentito dal contratto, intraprende delle azioni, che data la natura fortemente stringente dell'opzione, si configurano entrambe nell'emissione di un evento di QOS_ABT che provoca l'abort del work item;
- **Power constraints**: in questo caso occorre che l'inspector effettui il controllo dela potenza aggregata effettiva dei nodi virtuali del componente. L'inspector deve calcolare la potenza aggregata dell'insieme dei nodi fisici afferenti allo stesso nodo virtuale, per valutare il differenziale di quest'ultima rispetto al livello di potenza previsto dal contratto. A tal fine l'inspector raccoglie gli eventi di HeartBeat emessi dalle Remote Engine di ciascuno dei nodi afferenti al nodo virtuale suddetto, leggendone la potenza effettiva. Considerando una finestra temporale di opportuna lunghezza, l'inspector quantifica attraverso il calcolo di un opportuna funzione di controllo l'incremento di potenza necessario per contrastare lo scostamento della potenza aggregata effettiva dal valore prestabilito ed eventualmente emette un evento di QOS_INC inserendo tale valore in un attributo dell'evento stesso. Tale evento

viene raccolto dal WIhandler che provvederà ad aumentare il grado di parallelismo del componente tramite le direttive di enforcement accettate dal componente;

- **Time/speed constraints**: in questo caso occorre che l'inspector effettui il monitoring della velocità di esecuzione delle operazioni del dato work item. A tal fine l'inspector raccoglie gli eventi C_CKP (Checkpoint) emessi dal componente e analogamente a quanto accadeva nel caso precedente utilizza una finestra temporale e una funzione di controllo per ricavare l'incremento di potenza che consenta la correzione del livello di velocità contrattuale. Quindi l'inspector emette un evento di QOS_INC comunicando il valore di incremento di potenza richiesto.

Nella tabella 7 sono riassunti alcuni scenari, attualmente codificati dalle regole CLIPS, con gli eventi coinvolti nelle varie opzioni contrattuali.

QoSOptions	Trigger status	Monit Src	Mon Sign	Mon Sink	React Evt	React Sink	Enf Evt	Enf Sink
Master fault restart	Mnode fault	Remote Engine	! VPG_HB |VPG_PFLT	VPGMaster	VPG_GFLT	JobInspector	QoS_RSKD	WIHandler
Resource Reservation	Node fault	Remote Engine	! VPG_HB	VPGMaster	VPG_GFLT	JobInspector	QoS_ABT	WIHandler
Slave fault Restart	Snode fault	Remote Engine	! VPG_HB |VPG_PFLT	VPGMaster	VPG_GFLT	JobInspector	QoS_RSKD QoS_SWP	WIHandler
Time/Speed constraint	Out of constraint	Component	C_CHECKPOINT	JobInspector	QoS_INC	WIHandler	ENF_JOIN/ DSJOIN	Component
Power contraint	Out of constraint	Component	VPG_HB	JobInspector	QoS_INC	WIHandler	ENF_JOIN/ DSJOIN	Component
Suspend	Resource unvailable	WIHandler	ENF_SUSP	Component	C_SUSP	JobInspector / WIHandler	--	--
Resume	Resources available	WIHandler	ENF_RESU	Component	C_RESU	JobInspector / WIHandler	--	--

Tabella 7. Eventi coinvolti nelle varie opzioni contrattuali

6.4.8 Sviluppi futuri

L'implementazione attuale del Job Inspector possiede dei limiti di efficienza. Al fine di ridurre tali limiti sono attualmete in corso delle attività di manutenzione migliorativa del sistema. La creazione di un ambiente differente per ogni work item ad esempio comporta un overhead improduttivo e uno spreco di risorse. Essendo la base di conoscenza di ogni istanza di sistema esperto completamente determinata dal contratto (con l'insieme delle opzioni contrattuali) appare naturale associare un ambiente ad ogni contratto firmato col cliente anziché ad ogni work-item. Ciò riduce sia i tempi di creazione del sistema che la quantità di risorse utilizzate.

Inoltre si sta riprogettando la base di conoscenza del sistema in modo da rendere il codice C++ realizzato completamente indipendente dagli eventi scambiati con l'esterno. Ciò rende il sistema flessibile ai cambiamente dei protocolli di interazioni con gli altri attori. In tal modo le modifiche del comportamento del sistema si riducono a modifiche della base di conoscenza (file testuali clips caricati dinamicamente dal sistema).

L'approccio a programmazione logica utilizzata nella realizzazione dell'Inspector verrà impiegata anche per alcuni componenti della piattaforma GRACE [8]. In particolare per le:

- procedure di negoziazione dei contratti del componente Notary, il cui compito è quello di gestisce il ciclo di vita dei contratti;
- procedure di ottimizzazione globale e locale delle risorse del componente Scheduler, il cui noto compito è quello di schedulare le richieste.

7 Appendice A: Regole del sistema esperto

7.1 Main

Nel seguente codice sono riportate le regole basi della macchina a stati del sistema esperto, esse sono generiche e non dipendono da un particolare opzione contrattuale o patterns di controllo.

```
;global variables
(defglobal  ?*graphId* = (create$ empty) )

; classes for states
(defclass STATE (is-a USER)
  (slot state
    (type SYMBOL))
)

; classes for events

(defclass EVENT (is-a USER)
  (slot label
    (type SYMBOL))
)

(defclass INIT-EVENT (is-a EVENT)
  (slot graphId
    (type STRING))
)

(defclass FAULT-EVENT (is-a EVENT)
  (slot code
    (type NUMBER))
  (slot id ; physical node identifier/ down node index ..etc
    (type STRING) (default ?DERIVE))
)


; output events list: QOS_END QOS_ABT QOS_RSKD
|    MONITOR STATE-RULES
;
|    s1 = INITIATED
; pretended events: PRET_ABORT PRET_POW_INC
|    s2 = RUNNING
;
|    s3 = ACTIVE
; states list: INITIATED RUNNING ACTIVE COMPLETE TERMINATED SUSPENDED
|    s4 = COMPLETE
;
|    s5 = TERMINATED
; events list: VPG_GFLT VPG_GEND ENA_STR ENA_CALL ENA_INIT C_SUSP C_RESU
|    s6 = SUSPENDED
;              CHECKPOINT C_START

; FUNTIONS: only for debugging the system outside the JobInsector module

;(deffunction sendEvent (?name ?data) (printout t ?name " " ?data crlf))
;(deffunction ruleFire (?name ?data) (printout t ?name " " ?data crlf))

; RULES
```

```
(defrule firstrule
  (object (is-a EVENT) (label ENA_INIT) (graphId ?x)) ; any ENA_INIT event
  (not (object (is-a STATE)))      ; first time (no state jet instantiated)
=>
  (bind ?*graphId* ?x)
  (make-instance of STATE (state INITIATED))
)

(defrule s1-s2
  ?tbd1 <- (object (is-a STATE) (state INITIATED))
  (object (is-a EVENT) (label ENA_STR))
=>
  (send ?tbd1 put-state RUNNING)
)

(defrule pre-s2-s3
  (object (is-a STATE) (state RUNNING))
  (object (is-a EVENT) (label ENA_CALL))
=>
  (assert (ena_call_occurred))
)

(defrule s2-s3
  ?tbd1 <- (object (is-a STATE) (state RUNNING))
  (object (is-a EVENT) (label C_START))
  ?tbd2 <- (ena_call_occurred)
=>
  (retract ?tbd2)
  (send ?tbd1 put-state ACTIVE)
)

(defrule s3-s4
  ?tbd1 <- (object (is-a STATE) (state ACTIVE))
  (object (is-a EVENT) (label VPG_GEND))
=>
  (send ?tbd1 put-state COMPLETE)
)

(defrule in-s4
  (object (is-a STATE) (state COMPLETE))
=>
  (sendEvent "QOS_END" ?*graphId*)
)

(defrule to_s5
  (declare (salience 40)) ; higher salience
  ?tbd <- (object (is-a STATE) (state ~TERMINATED))
  (object (is-a EVENT) (label VPG_GFLT))
=>
  (send ?tbd put-state TERMINATED)
)

(defrule garbageCollection
  (declare (salience -100)) ; lower salience
  (not (object (is-a STATE) ))
  ?tbd <- (object (is-a EVENT)) ; flush wrong events preceding ENA_INIT
=>
  (send ?tbd delete)
)

(defrule garbageCollection
  (declare (salience -100)) ; lower salience
```

```
  (object (is-a STATE) (state ?x))
  ?tbd <- (object (is-a EVENT)) ; flush preceding ENA_INIT and subsequent events
=>
  (send ?tbd delete)
  (printout t " **state** " ?x crlf)
)
```

7.2 MasterFaultRestart.clp

File delle regole inerenti al monitoraggio e controllo dell'opzione contrattuale "MasterFaultRestart".

```
;  global variables
(defglobal
?*masterNode* = (create$ empty)
?*masterProcess* = (create$ empty)
?*masterConnections* = (create$ empty)
?*rskdCounter* = (create$ empty)
)

; debug rule
;(defrule firstrule-friend2
;  (object (is-a STATE) (state INITIATED))
;=>
;  (printout t " **master** " ?*master* crlf)
;)

;  if the VPG_GFLT event occurs and the process id is
;  the master process then send QOS_ABT event
(defrule master_fault_end
  (declare (salience 10))
  ?tbd <- (object (is-a EVENT) (label VPG_GFLT) (code 2) (id ?ni))
  (test (= 0 (str-compare ?ni ?*masterProcess*)))
=>
  (send ?tbd delete)
  (sendEvent "QOS_ABT" ?*graphId*)
)

(defrule masterFault_rskd
  (object (is-a EVENT) (label VPG_GFLT) (code 1|3) (id ?ni))
  (test (= 0 (str-compare ?ni ?*masterNode*)))
=>
  (sendEvent "QOS_RSKD" ?*graphId*)
)

(defrule masterFault_rskd
  (object (is-a EVENT) (label VPG_GFLT) (code 1|3) (id ?ni))
  (test (= 0 (str-compare ?ni ?*masterConnections*)))
=>
  (sendEvent "QOS_RSKD" ?*graphId*)
)
```

7.3 powerConstraint.clp

File delle regole inerenti al monitoraggio e controllo dell'opzione contrattuale "powerConstraint".

```
;  events  definition
(defclass PRET_POW_INC-EVENT (is-a EVENT)
  (slot nodeId
    (type STRING))
  (slot incr
    (type FLOAT))
)


;  rules
(defrule powContraints
  (object (is-a STATE) (state ACTIVE))
  (object (is-a EVENT) (label PRET_POW_INC) (nodeId ?ni) (incr ?in))
=>
  (sendEvent "QOS_INC" ?*graphId* ?ni ?in)
)
```

7.4 ResourceReservation.clp

File delle regole inerenti al monitoraggio e controllo dell'opzione contrattuale "ResourceReservation".

```
;  global variables
(defglobal
?*fixedNodes* = (create$ empty)
)

;  rules
(defrule resResPow
  ?tbd1 <- (object (is-a STATE) (state ACTIVE))
  (object (is-a EVENT) (label PRET_ABORT))
=>
  (stopTimer ?*graphId*)
  (send ?tbd1 put-state TERMINATED)
  (sendEvent "QOS_ABT" ?*graphId*)
)

(defrule resResFault
  (declare (salience 20)) ; higher salience
  ?tbd <- (object (is-a EVENT) (label VPG_GFLT) (id ?ni))
  (test (member$ ?ni ?*fixedNodes*))
=>
  (send ?tbd delete)
  (sendEvent "QOS_ABT" ?*graphId*)
)
```

7.5 SlaveFaultRestart.clp

File delle regole inerenti al monitoraggio e controllo dell'opzione contrattuale "SlaveFaultRestart".

```
;  global variables
(defglobal
?*slaveNode* = (create$ empty)
?*slaveProcess* = (create$ empty)
?*slaveConnections* = (create$ empty)
?*rskdCounter* = (create$ empty)
```

```
)

;  rules
(defrule slaveFault_swap_node
  (object (is-a EVENT) (label VPG_GFLT) (code 1|3) (id ?ni))
  (test (<> 0 (str-compare ?ni ?*slaveNode*)))
=>
  (sendEvent "QOS_SWP" ?*graphId* ?ni)
)

;  rules
(defrule slaveFault_swap_connection
  (object (is-a EVENT) (label VPG_GFLT) (code 1|3) (id ?ni))
  (test (<> 0 (str-compare ?ni ?*slaveConnections*)))
=>
  (sendEvent "QOS_SWP" ?*graphId* ?ni)
)

(defrule slaveFault_rskd
  (object (is-a EVENT) (label VPG_GFLT) (code 2) (id ?ni))
  (test (<> 0 (str-compare ?ni ?*slaveNode*)))
=>
  (sendEvent "QOS_ABT" ?*graphId*)
  (bind ?*rskdCounter* (- ?*rskdCounter* 1))
)
```

7.6 SpeedConstraint.clp

File delle regole inerenti al monitoraggio e controllo dell'opzione contrattuale "SpeedConstraint".

```
;  events  definition
(defclass CHECKPOINT (is-a EVENT)
  (slot execTime
    (type FLOAT))
  (slot totalExecTime
    (type FLOAT))
  (slot operations
    (type FLOAT))
  (slot totalOperations
    (type FLOAT))
)


;  global variables
(defglobal
?*slave* = (create$ empty)
?*expectedSpeed* = (create$ empty)
?*speedTolerance* = (create$ empty)
)


;  rules
(defrule check_point
  (object (is-a STATE) (state ACTIVE))
  (object (is-a EVENT) (label C_CHECKPOINT) (execTime ?et) (totalExecTime ?tet)
(operations ?op) (totalOperations ?to))
=>
  (assert (speed_con (speedControl ?*graphId* ?et ?tet ?op ?to ?*expectedSpeed*
?*speedTolerance*)))
)
```

```
(defrule check_point2QOS_INC
  (object (is-a EVENT) (label C_CHECKPOINT))
  ?tbd <- (speed_con ?x)
  (test (<> 0 ?x))
=>
  (retract ?tbd)
  (sendEvent "QOS_INC" ?*graphId* ?*slave* ?x)
)
```

7.7 Starvation.clp

File delle regole inerenti al monitoraggio e controllo dell'opzione contrattuale "Starvation".

```
;  possibile fare il pattern matching o con l'evento ENA_INIT o con
;  lo stato INITIATED. Scelta attuale => fare il pattern matching
;  con lo stato INITIATED

;  it starts the timer in the c++ code
(defrule rule_StartTimer
  (object (is-a STATE) (state INITIATED))
=>
  (startTimer ?*graphId*)
)

; it stops the timer in the C++ code
(defrule end_of_life
  (object (is-a STATE) (state COMPLETE|TERMINATED))
=>
  (stopTimer ?*graphId*)
)
```

7.8 SuspendResume.clp

File delle regole inerenti al monitoraggio e controllo dell'opzione contrattuale "SuspemdResume".

```
;  rules
(defrule s3-s6
  ?tbd1 <- (object (is-a STATE) (state ACTIVE))
  (object (is-a EVENT) (label C_SUSP))
=>
  (send ?tbd1 put-state SUSPENDED)
)

(defrule s6-s3
  ?tbd1 <- (object (is-a STATE) (state SUSPENDED))
  (object (is-a EVENT) (label C_RESU))
=>
  (send ?tbd1 put-state ACTIVE)
)
```

8 AppendiceB: Inspector Class Documentation

8.1 ClipsWrapper Class Reference

ClipsWrapper#include <clipswrapper.h>

8.1.1 Public Member Functions

- **~ClipsWrapper** ()
- **ClipsWrapper** (const AL_DataSet *event)
- string **currentState** ()
- void **handleTimedEvent** ()
- void **process** (const AL_DataSet *event) throw (string)
- string **convert** (const AL_DataSet *event) throw (string)
- void **readPhysicalGraph** (string physGraph) throw (string)
- vector< string > **readResourceReservation** (string qosGraph) throw (string)
- void **readPowerConstraints** (string qosGraph) throw (string)
- void **clearVirtualNodes** ()

8.1.2 Public Attributes

- string **graphId**
- string **componentURI**
- int **contract**
- bool **resourceReservation**
- bool **powerConstraints**
- bool **speedConstraints**
- bool **masterFaultRestart**
- bool **slaveFaultRestart**
- bool **suspResume**
- bool **starvation**
- long **start**
- long **duration**
- long **starvationValue**
- void * **env**
- **MetaState metaState**
- int **rskdCounter**
- long **timer_id**
- map< int, **VirtualNode** * > **virtualNodes**

8.1.3 Static Public Attributes

- static ACE_Thread_Mutex **mutex**

8.1.4 Private Member Functions

- void **run** ()
- void **makeInstance** (string clipsObj)

8.1.5 Detailed Description

8.1.5.1.1 Author:

Vitor Graziano

8.1.6 Constructor & Destructor Documentation

8.1.6.1 ClipsWrapper::~ClipsWrapper ()

8.1.6.2 ClipsWrapper::ClipsWrapper (const AL_DataSet * *event*)

8.1.7 Member Function Documentation

8.1.7.1 void ClipsWrapper::clearVirtualNodes ()

clear the vector holding the virtual nodes

8.1.7.2 string ClipsWrapper::convert (const AL_DataSet * *event*) throw (string)

converts an event in a corresponding clips assertion (or instance creation)

8.1.7.3 string ClipsWrapper::currentState ()

to get the actual state of work item

8.1.7.4 void ClipsWrapper::handleTimedEvent ()

handles timed events related to starvation control

8.1.7.5 void ClipsWrapper::makeInstance (string *clipsObj*) `[private]`

wrapping of the make_instance function of the CLIPS library

8.1.7.6 void ClipsWrapper::process (const AL_DataSet * *event*) throw (string)

process an event using clips engine to process clips assertion obtained through the convert method or using customized C++ code

8.1.7.7 void ClipsWrapper::readPhysicalGraph (string *physGraph*) throw (string)

reads the physical graph related to the work item and fills the virtual nodes container

8.1.7.8 void ClipsWrapper::readPowerConstraints (string *qosGraph*) throw (string)

read the power contraints information in the QoSGraph

8.1.7.9 vector< string > ClipsWrapper::readResourceReservation (string *qosGraph*) throw (string)

reads the resources reservation graph and predisposes the CLIPS environment to manage the fixed resources declared in the contract

8.1.7.10 void ClipsWrapper::run () [private]

wrapping of the run function of the CLIPS library

8.1.8 Member Data Documentation

8.1.8.1 string ClipsWrapper::componentURI

component URI

8.1.8.2 int ClipsWrapper::contract

type of contract (enum)

8.1.8.3 long ClipsWrapper::duration

duration of the contract

8.1.8.4 void* ClipsWrapper::env

clips environment

8.1.8.5 string ClipsWrapper::graphId

graph ID

8.1.8.6 bool ClipsWrapper::masterFaultRestart

is the QoS option master Fault Restart active

8.1.8.7 MetaState ClipsWrapper::metaState

state of the Clips wrapper (not of the component)

8.1.8.8 ACE_Thread_Mutex ClipsWrapper::mutex [static]

mutex to synchronize access to instances of ClipsWrapper

8.1.8.9 bool ClipsWrapper::powerConstraints

is the QoS option power constraints active

8.1.8.10 bool ClipsWrapper::resourceReservation

is the QoS option resource reservation active

8.1.8.11 int ClipsWrapper::rskdCounter

counter of the number of reschedulations

8.1.8.12 bool ClipsWrapper::slaveFaultRestart

is the QoS option slave Fault Restart active

8.1.8.13 bool ClipsWrapper::speedConstraints

is the QoS option speed constraints active

8.1.8.14 long ClipsWrapper::start

start of the contract

8.1.8.15 bool ClipsWrapper::starvation

is the QoS option slaveFaultRestart active

8.1.8.16 long ClipsWrapper::starvationValue

time interval for starvation option (seconds)

8.1.8.17 bool ClipsWrapper::suspResume

is the QoS option suspend/resume active

8.1.8.18 long ClipsWrapper::timer_id

timer id for starvation control purpose

8.1.8.19 map<int,VirtualNode*> ClipsWrapper::virtualNodes

represents the container of the virtual nodes related to the work item

---

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**clipswrapper.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**clipswrapper.cpp**

8.2 CWFactory Class Reference

CWFactory#include <cwfactory.h>

8.2.1 Public Member Functions

- **~CWFactory** ()
- **ClipsWrapper** * **provide** (const string &_graphId)
- **ClipsWrapper** * **factory** (const AL_DataSet *_event) throw (string)

8.2.2 Static Public Member Functions

- static **CWFactory** * **instance** ()

8.2.3 Public Attributes

- map< string, **ClipsWrapper** * > **cwm**

8.2.4 Private Member Functions

- **CWFactory** ()

8.2.5 Static Private Attributes

- static **CWFactory** * **_instance** = 0

8.2.6 Detailed Description

8.2.6.1.1 Author:

Vitor Graziano

8.2.7 Constructor & Destructor Documentation

8.2.7.1 CWFactory::~CWFactory ()

8.2.7.2 CWFactory::CWFactory () `[private]`

private constructor

8.2.8 Member Function Documentation

8.2.8.1 ClipsWrapper * CWFactory::factory (const AL_DataSet * *_event*) throw (string)

makes an instance of **ClipsWrapper** or provide an instance of **ClipsWrapper** already created

8.2.8.2 CWFactory * CWFactory::instance () `[static]`

get the unique instance of this singleton

8.2.8.3 ClipsWrapper * CWFactory::provide (const string & *_graphId*)

provides an instance of **ClipsWrapper** by the right key

8.2.9 Member Data Documentation

8.2.9.1 CWFactory * CWFactory::_instance = 0 [static, private]

private static instance of this class

8.2.9.2 map<string,ClipsWrapper*> CWFactory::cwm

container of instances of **ClipsWrapper** corresponding to work item being in life in the GRACE system

---

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**cwfactory.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**cwfactory.cpp**

8.3 EventHandler Class Reference

EventHandler`#include <eventhandler.h>`

8.3.1 Public Member Functions

- **~EventHandler** ()
- void **compEventOccur** (const string &event, const AL_DataSet &data)
- void **resEventOccur** (const string &event, const AL_DataSet &data)
- void **installHandlers** ()
- void **init** (const string &masterEventPort)
- void **triggerEvent** (const string &, const string &, const AL_DataSet &)
- void **installHandlerComp** (const string &pool)

8.3.2 Static Public Member Functions

- static **EventHandler** * **instance** ()

8.3.3 Private Member Functions

- **EventHandler** ()

8.3.4 Private Attributes

- AL_Notifier< **EventHandler** > **notifier**
  The event Notifier.
- AL_Dispatcher< **EventHandler** > **dispatcherVPG**
  The event Dispatcher.
- AL_Dispatcher< **EventHandler** > **dispatcherGRACE**
- vector< AL_Dispatcher< **EventHandler** > * > **dispatchers**

8.3.5 Static Private Attributes

- static **EventHandler** * **_instance** = 0

8.3.6 Constructor & Destructor Documentation

8.3.6.1 EventHandler::~EventHandler ()

8.3.6.2 EventHandler::EventHandler () `[private]`

8.3.7 Member Function Documentation

8.3.7.1 void EventHandler::compEventOccur (const string & *event*, const AL_DataSet & *data*)

Handles the incoming event

8.3.7.2 void EventHandler::init (const string & *masterEventPort*)

Initialize the event system

8.3.7.3 void EventHandler::installHandlerComp (const string & *pool*)

install handler for component level events

8.3.7.4 void EventHandler::installHandlers ()

8.3.7.5 EventHandler * EventHandler::instance () `[static]`

8.3.7.6 void EventHandler::resEventOccur (const string & *event*, const AL_DataSet & *data*)

8.3.7.7 void EventHandler::triggerEvent (const string &, const string &, const AL_DataSet &)

trigs the events

8.3.8 Member Data Documentation

8.3.8.1 EventHandler * EventHandler::_instance = 0 [static, private]

8.3.8.2 AL_Dispatcher<EventHandler> EventHandler::dispatcherGRACE [private]

8.3.8.3 vector<AL_Dispatcher<EventHandler> *> EventHandler::dispatchers [private]

8.3.8.4 AL_Dispatcher<EventHandler> EventHandler::dispatcherVPG [private]

The event Dispatcher.

8.3.8.5 AL_Notifier<EventHandler> EventHandler::notifier [private]

The event Notifier.

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/EventFE/**eventhandler.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/EventFE/**eventhandler.cpp**

8.4 EventQueue Class Reference

EventQueue#include <eventqueue.h>

8.4.1 Public Member Functions

- **~EventQueue** ()
- void **push** (const AL_DataSet *event)
- const AL_DataSet * **front** ()

8.4.2 Static Public Member Functions

- static **EventQueue** * **instance1** ()
- static **EventQueue** * **instance2** ()

8.4.3 Private Member Functions

- **EventQueue** ()

8.4.4 Private Attributes

- queue< const AL_DataSet * > **eventsQueue**

- ACE_Thread_Mutex **eqMutex**
- ACE_Condition< ACE_Thread_Mutex > * **eqCond**

8.4.5 Static Private Attributes

- static **EventQueue** * **_instance1** = 0
- static **EventQueue** * **_instance2** = 0

8.4.6 Detailed Description

8.4.6.1.1 Author:

Vitor Graziano

8.4.7 Constructor & Destructor Documentation

8.4.7.1 EventQueue::~EventQueue ()

8.4.7.2 EventQueue::EventQueue () `[private]`

8.4.8 Member Function Documentation

8.4.8.1 const AL_DataSet * EventQueue::front ()

get and erase the first element from the queue

8.4.8.2 EventQueue * EventQueue::instance1 () `[static]`

this is used by AImain loop and manage all the events except VPG_HB

8.4.8.3 EventQueue * EventQueue::instance2 () `[static]`

this is used by resmon loop and manage only the event VPG_HB

8.4.8.4 void EventQueue::push (const AL_DataSet * *event*)

push an event into the queue

8.4.9 Member Data Documentation

8.4.9.1 EventQueue * EventQueue::_instance1 = 0 [static, private]

8.4.9.2 EventQueue * EventQueue::_instance2 = 0 [static, private]

8.4.9.3 ACE_Condition<ACE_Thread_Mutex>* EventQueue::eqCond [private]

8.4.9.4 ACE_Thread_Mutex EventQueue::eqMutex [private]

8.4.9.5 queue<const AL_DataSet*> EventQueue::eventsQueue [private]

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/Common/**eventqueue.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/Common/**eventqueue.cpp**

8.5 InspConfig Class Reference

InspConfig`#include <insp_config.h>`

8.5.1 Public Member Functions

- **~InspConfig** ()
- const string & **get** (const string &key)
- bool **get** (const string &key, string &res)
- void **set** (const string &key, const string &value)
- void **load** (const string &filename)

8.5.2 Static Public Member Functions

- static **InspConfig** * **instance** ()

8.5.3 Private Member Functions

- **InspConfig** ()

8.5.4 Private Attributes

- map< string, string > **rep**

8.5.5 Static Private Attributes

- static **InspConfig** * **_instance** = 0

8.5.6 Detailed Description

8.5.6.1.1 Author:

Vitor Graziano

8.5.7 Constructor & Destructor Documentation

8.5.7.1 InspConfig::~InspConfig ()

8.5.7.2 InspConfig::InspConfig () `[private]`

8.5.8 Member Function Documentation

8.5.8.1 bool InspConfig::get (const string & *key*, string & *res*)

No descriptions

8.5.8.2 const string & InspConfig::get (const string & *key*)

8.5.8.3 InspConfig * InspConfig::instance () `[static]`

8.5.8.4 void InspConfig::load (const string & *filename*)

8.5.8.5 void InspConfig::set (const string & *key*, const string & *value*)

8.5.9 Member Data Documentation

8.5.9.1 InspConfig * InspConfig::_instance = 0 `[static, private]`

8.5.9.2 map<string,string> InspConfig::rep `[private]`

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/Config/**insp_config.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/Config/**insp_config.cpp**

8.6 InspTimer Class Reference

InspTimer`#include <insptimer.h>`

8.6.1 Public Member Functions

- **~InspTimer** ()
- virtual int **handle_timeout** (const ACE_Time_Value &current_time, const void *argc)

8.6.2 Static Public Member Functions

- static **InspTimer** * **instance** ()

8.6.3 Private Member Functions

- **InspTimer** ()

8.6.4 Static Private Attributes

- static **InspTimer** * **_instance** = 0

8.6.5 Detailed Description

8.6.5.1.1 Author:

Vitor Graziano

8.6.6 Constructor & Destructor Documentation

8.6.6.1 InspTimer::~InspTimer ()

8.6.6.2 InspTimer::InspTimer () `[private]`

8.6.7 Member Function Documentation

8.6.7.1 int InspTimer::handle_timeout (const ACE_Time_Value & *current_time*, const void * *argc*) `[virtual]`

This metod is invoked by the Reactor when the timeout occurs

8.6.7.2 InspTimer * InspTimer::instance () `[static]`

No descriptions

8.6.8 Member Data Documentation

8.6.8.1 InspTimer * InspTimer::_instance = 0 [static, private]

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**insptimer.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**insptimer.cpp**

8.7 PhysNode Class Reference

PhysNode`#include <physnode.h>`

8.7.1 Public Member Functions

- **PhysNode** ()
- **~PhysNode** ()

8.7.2 Public Attributes

- string **resourceURI**
- float **power**

8.7.3 Static Public Attributes

- static map< string, **PhysNode** * > **pnmap**

8.7.4 Detailed Description

8.7.4.1.1 Author:

Vitor Graziano

8.7.5 Constructor & Destructor Documentation

8.7.5.1 PhysNode::PhysNode ()

8.7.5.2 PhysNode::~PhysNode ()

8.7.6 Member Data Documentation

8.7.6.1 map< string, PhysNode * > PhysNode::pnmap [static]

the container of all the resources related to any work item running in the system

8.7.6.2 float PhysNode::power

the power (bogomips) related to the resource

8.7.6.3 string PhysNode::resourceURI

an identifier for the resource

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**physnode.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**physnode.cpp**

8.8 SigmaTimer Class Reference

SigmaTimer`#include <sigmatimer.h>`

8.8.1 Public Member Functions

- **~SigmaTimer** ()
- virtual int **handle_timeout** (const ACE_Time_Value &current_time, const void *argc)

8.8.2 Static Public Member Functions

- static **SigmaTimer** * **instance** ()

8.8.3 Private Member Functions

- **SigmaTimer** ()

8.8.4 Static Private Attributes

- static **SigmaTimer** * **_instance** = 0

8.8.5 Detailed Description

8.8.5.1.1 Author:

Vitor Graziano

8.8.6 Constructor & Destructor Documentation

8.8.6.1 SigmaTimer::~SigmaTimer ()

8.8.6.2 SigmaTimer::SigmaTimer () `[private]`

8.8.7 Member Function Documentation

8.8.7.1 int SigmaTimer::handle_timeout (const ACE_Time_Value & *current_time*, const void * *argc*) [virtual]

This metod is invoked by the Reactor when the timeout occurs

8.8.7.2 SigmaTimer * SigmaTimer::instance () [static]

8.8.8 Member Data Documentation

8.8.8.1 SigmaTimer * SigmaTimer::_instance = 0 [static, private]

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**sigmatimer.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**sigmatimer.cpp**

8.9 VirtualNode Class Reference

VirtualNode`#include <virtualnode.h>`

8.9.1 Public Member Functions

- **VirtualNode** ()
- **~VirtualNode** ()
- float **totalPower** ()
- float **checkPower** ()

8.9.2 Public Attributes

- map< string, **PhysNode** * > **p_nodes**
- float **powerMin**
- float **powerMax**
- float **currentPower**
- int **node_id**
- bool **resourceReservation**
- bool **powerConstraints**
- bool **speedConstraints**

8.9.3 Detailed Description

8.9.3.1.1 Author:

Vitor Graziano

8.9.4 Constructor & Destructor Documentation

8.9.4.1 VirtualNode::VirtualNode ()

8.9.4.2 VirtualNode::~VirtualNode ()

8.9.5 Member Function Documentation

8.9.5.1 float VirtualNode::checkPower ()

checks if the power value is equal or higher than the power expected

8.9.5.2 float VirtualNode::totalPower ()

compute the aggregate power

8.9.6 Member Data Documentation

8.9.6.1 float VirtualNode::currentPower

8.9.6.2 int VirtualNode::node_id

8.9.6.3 map<string,PhysNode*> VirtualNode::p_nodes

container of the physical resources related to this virtual node

8.9.6.4 bool VirtualNode::powerConstraints

8.9.6.5 float VirtualNode::powerMax

8.9.6.6 float VirtualNode::powerMin

8.9.6.7 bool VirtualNode::resourceReservation

8.9.6.8 bool VirtualNode::speedConstraints

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**virtualnode.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/ClipsEngine/**virtualnode.cpp**

8.10 VPGEvents Class Reference

VPGEvents#include <vpgevents.h>

8.10.1 Static Public Attributes

- static const char **VPG_SYSTEM_POOL** [] = "VPG_SYSTEM_POOL"
  namespaces of VPG system pool events
- static const char **VPG_HB** [] = "VPG_HB"
- static const char **VPG_NP2P** [] = "VPG_NP2P"
- static const char **VPG_PFLT** [] = "VPG_PFLT"
- static const char **VPG_PEND** [] = "VPG_PEND"
- static const char **VPG_PSUSP** [] = "VPG_PSUSP"
- static const char **VPG_PCONT** [] = "VPG_PCONT"
- static const char **VPG_PSTR** [] = "VPG_PSTR"
- static const char **VPG_KEPA** [] = "VPG_KEPA"
- static const char **GRACE_SYSTEM_POOL** [] = "GRACE_SYSTEM_POOL"
  namespaces of Graph process (Component instance) pool events
- static const char **VPG_GFLT** [] = "VPG_GFLT"
- static const char **VPG_GEND** [] = "VPG_GEND"
- static const char **ENA_INIT** [] = "ENA_INIT"
- static const char **ENA_STR** [] = "ENA_STR"
- static const char **ENA_CALL** [] = "ENA_CALL"
- static const char **QOS_END** [] = "QOS_END"
- static const char **QOS_RSKD** [] = "QOS_RSKD"
- static const char **QOS_ABT** [] = "QOS_ABT"
- static const char **QOS_INC** [] = "QOS_INC"
- static const char **QOS_SWP** [] = "QOS_SWP"
- static const char **RELEVANTDATA_EVENT** [] = "VPG_NotifyRelevantData"
  names of relevant data component pool events
- static const char **C_CHECKPOINT** [] = "C_CHECKPOINT"
- static const char **C_START** [] = "C_START"
- static const char **C_SUSP** [] = "C_SUSP"
- static const char **C_RESU** [] = "C_RESU"
- static const char **C_SLAVE_JOIN** [] = "C_SLAVE_JOIN"
- static const char **C_SLAVE_DISJOIN** [] = "C_SLAVE_DISJOIN"

8.10.2 Member Data Documentation

8.10.2.1 const char VPGEvents::C_CHECKPOINT = "C_CHECKPOINT" [static]

8.10.2.2 const char VPGEvents::C_RESU = "C_RESU" [static]

8.10.2.3 const char VPGEvents::C_SLAVE_DISJOIN = "C_SLAVE_DISJOIN" [static]

8.10.2.4 const char VPGEvents::C_SLAVE_JOIN = "C_SLAVE_JOIN" [static]

8.10.2.5 const char VPGEvents::C_START = "C_START" [static]

8.10.2.6 const char VPGEvents::C_SUSP = "C_SUSP" [static]

8.10.2.7 const char VPGEvents::ENA_CALL = "ENA_CALL" [static]

8.10.2.8 const char VPGEvents::ENA_INIT = "ENA_INIT" [static]

8.10.2.9 const char VPGEvents::ENA_STR = "ENA_STR" [static]

8.10.2.10 const char VPGEvents::GRACE_SYSTEM_POOL = "GRACE_SYSTEM_POOL" [static]

namespaces of Graph process (Component instance) pool events

8.10.2.11 const char VPGEvents::QOS_ABT = "QOS_ABT" [static]

8.10.2.12 const char VPGEvents::QOS_END = "QOS_END" [static]

8.10.2.13 const char VPGEvents::QOS_INC = "QOS_INC" [static]

8.10.2.14 const char VPGEvents::QOS_RSKD = "QOS_RSKD" [static]

8.10.2.15 const char VPGEvents::QOS_SWP = "QOS_SWP" [static]

8.10.2.16 const char VPGEvents::RELEVANTDATA_EVENT = "VPG_NotifyRelevantData" [static]

names of relevant data component pool events

8.10.2.17 const char VPGEvents::VPG_GEND = "VPG_GEND" [static]

All job graph executables finished. trasmitted data

- "graphId" (string): graph instance identifier;

8.10.2.18 const char VPGEvents::VPG_GFLT = "VPG_GFLT" [static]

Fault of Graph. This means:

- connection fault;
- process fault;
- physical node fault;

trasmitted data:

- "graphId" (string): graph instance identifier;
- "code" (int) event code, value are:
  - 1: process fault;
  - 2: physical node fault;
  - 3: connection fault;
- "id" (string): (process/node/connection) identifier;

8.10.2.19 const char VPGEvents::VPG_HB = "VPG_HB" [static]

heart-beat signal: the remote engine is still live trasmitted data:

- "nodeId" (string): node identifier
- "index" (double): power availability of the node

8.10.2.20 const char VPGEvents::VPG_KEPA = "VPG_KEPA" [static]

VPG-Master process is still live. this means "keep alive" command for all Remote Engine. Trasmitted data:

- "nodeId" (string): node identifier;
- "value"(int): costant value = 1;

8.10.2.21 const char VPGEvents::VPG_NP2P = "VPG_NP2P" [static]

A connection between nodes becames fault trasmitted data:

- "nodeId" (string): node identifier
- "to" (string): node identifier

8.10.2.22 const char VPGEvents::VPG_PCONT = "VPG_PCONT" [static]

A process was resumed. Trasmitted data:

- "nodeId" (string): node identifier;
- "id" (string): process identifier

8.10.2.23 const char VPGEvents::VPG_PEND = "VPG_PEND" [static]

A process terminated successfully. trasmitted data:

- "nodeId" (string): node identifier;
- "id" (string): process identifier

8.10.2.24 const char VPGEvents::VPG_PFLT = "VPG_PFLT" [static]

A process terminated insuccessfully. Trasmitted data:

- "nodeId" (string): node identifier;
- "pid" (int) :pid of process;
- "id" (string): process identifier

8.10.2.25 const char VPGEvents::VPG_PSTR = "VPG_PSTR" [static]

A process starts successfully. Trasmitted data:

- "nodeId" (string): node identifier;
- "id" (string): process identifier

8.10.2.26 const char VPGEvents::VPG_PSUSP = "VPG_PSUSP" [static]

A process was suspended. Trasmitted data:

- "nodeId" (string): node identifier;
- "id" (string): process identifier

8.10.2.27 const char VPGEvents::VPG_SYSTEM_POOL = "VPG_SYSTEM_POOL" [static]

namespaces of VPG system pool events

The documentation for this class was generated from the following files:

- E:/ICAR/GRACE/Inspector/source_code/inspector/Common/**vpgevents.h**
- E:/ICAR/GRACE/Inspector/source_code/inspector/Common/**vpgevents.cpp**

Bibliografia:

[1] Vogel A., Kerhervé B., Von Bochman G. Ad Gecsei J. (1195). Distributed Multimedia and QoS: A Survey, IEEE Multimedia Vol. 2, No. 2, p10-19

[2] A. Machì, F. Collura, , S. Lombardo. “Legacy Software Integrating Environment. Metodologie, patterns e tools per l’integrazione nell’ambiente di programmazione Grid.it“ Technical Report ICAR-CNR Dept. Palermo RT-ICAR-PA-04-15 Dicembre 2004 (WP8)

[3] A. Machì, F. Collura, S. Lombardo: “Dependable Execution of Workflow Activities on a Virtual Private Grid Middleware. V. S. Sunderam et als. (Eds.) Computational Science ICCS 2005 LNCS 3516 pp.267-274, 2005 Springer-Verlag 2005

[4] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo. "Components for High Perforamce Grid Programming in the Grid.it Project". In V. Getov and T. Kielmann, editors, Proc. of the Workshop on Component Models and Systems for Grid Applications, CoreGrid series. Springer Verlag, January 2005.

[5] S. Lombardo, A. Machì: “Virtual Private Grid (VPG 1.2):Un middleware a supporto del ciclo di vita e del monitoraggio dell’esecuzione grafi di processi su griglia computazionale. Versione 1.2.1”. Technical report ICAR-CNR- Palermo RT-ICAR-PA 11-2004

[6] A. Machì, F. Collura, , S. Lombardo. “Modellazione UML dello Skeleton di coordinamento di un Componente Parallelo Master-Slave e di patterns per il controllo esterno della sua performance “ Technical Report ICAR-CNR Dept. Palermo RT-ICAR-PA-05-03 Marzo 2005 (WP8)

[7] S. Lombardo, A. Machì: “A model for a component based grid-aware scientific library service.” in M. Danelutto, D. Laforenza, M. Vanneschi eds. EUROPAR-2004 Parallel Processing LNCS 3149 pp 423-428 Springer-Verlag 2004 . JCR 08098

[8] S. Lombardo, A. Machì. “Grid-Aware serviCEs administrator: un server di Web-services, operante sotto vincoli di Qualità di Servizio. Versione 1.0 Implementazione in ASSIST 1.3 “ Technical Report ICAR-CNR Dept. Palermo RT-ICAR-PA-05-13 Dicembre 2005

[9] The Workflow Management Coalition, www.wfmc.org

[10] The Workflow Reference Model (WFMC-TC-1003, 19-Jan-95, 1.1)

[11] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo, Dynamic reconfiguration of grid-aware applications in ASSIST, in 11th Intl Euro-Par 2005: Parallel and Distributed Computing, LNCS, Lisboa, Portugal, August 2005

[12] CLIPS http://www.ghg.net/clips/CLIPS.html

[13] F. Collura, A. Machì. "Metodologie, schemi e tools per la descrizione di elementi software e per l'integrazione di codice legacy in componenti grid-enabled" Technical Report ICAR-CNR Dept. Palermo RT-ICAR-PA-05-14 Dicembre 2005 (WP8).

[14] The ADAPTIVE Communication Environment (ACE) http://www.cs.wustl.edu/~schmidt/ACE.html

[15] Feigenbaum, Edward A., Pamela McCorduck, and H. P. Nii. 1988. The Rise of the Expert Company: How Visionary Companies are Using Artificial Intelligence to Achieve Higher Productivity and Profits. New York: Times Books, ISBN:0-8129-1731-6

[16] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. "Workflow Patterns". Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.