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

Integrazione di librerie scientifiche
nell'ambiente di programmazione a componenti GRID.IT

Approccio restricted a stereotipi di componente

Alberto Machì, Saverio Lombardo,
Gabriele Siino, Mario Tripiciano

RT-ICAR-PA-03-11 **dicembre 2003**

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

Integrazione di librerie scientifiche
nell'ambiente di programmazione a componenti GRID.IT

Approccio restricted
a stereotipi di componente

Alberto Machì[1], Saverio Lombardo[1],
Gabriele Siino[2],(Appendice A)
Mario Tripiciano[1] ,(Appendice B)

Deliverable Progetto Grid.it WP8 Ambienti di Programmazione

Rapporto Tecnico N.: ***Data:***
RT-ICAR-PA-03-11 **dicembre 2003**

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sezione di Palermo
[2] Tesista Università degli Studi di Palermo Dipartimento di Ingegneria Informatica.

1. Indice

2. Librerie scientifiche e codice legacy

Per libreria scientifica si intende una raccolta di codice, più o meno strutturato, considerato significativo per la soluzione di problemi di modellazione o di controllo in un particolare dominio scientifico. Il presente studio discute un metodo per la integrazione di librerie scientifiche in un ambiente di programmazione su griglia computazionale, nel contesto più generale della integrazione di codice legacy generico in componenti software.

Gli elementi di una libreria scientifica sono generalmente scritti da ricercatori scientifici o analisti numerici e sono rivolti alla simulazione di sistemi o all'analisi di dati sperimentali. Alcuni elementi (moduli applicativi) sono utilizzati con una semantica derivata dal particolare dominio scientifico (chimica, astronomia, fisica etc), altri con una semantica derivata dalla modellazione matematica utilizzata (nuclei computazionalidi statistica o analisi numerica).

La metodologia di utilizzo degli elementi computazionali in un contesto applicativo è spesso simile fra le varie scienze, corrispondendo alle diverse fasi del cosiddetto "metodo scientifico" (induzione di regolarità nella analisi dei dati sperimentali, modellazione, controllo dell'esperimento, test di ipotesi steering). Gli elementi computazionali utilizzati sono generalmente passivi o reattivi. Solo in rari casi (esperimenti a distanza o in condizioni di alta pericolosità) essi sono dotati di caratteristiche di attività, pro-attività ed autonomia (es. agenti).

E' quindi plausibile che l'architettura e la funzionalità di elementi software di codice legacy scientifico presenti alcune regolarità nell'uso e nella composizione degli elementi computazionali caratterizzabili come stereotipi dal punto di vista della ingegneria del software. Tali stereotipi sono esprimibili, in un ambiente di programmazione strutturato a componenti, attraverso specializzazioni dei paradigmi descrittivi della struttura e del modello di interazione fra componenti.

Questo documento descrive i diversi approcci utilizzati nell'integrazione e riuso di software legacy con tecnologie informatiche emergenti e propone una metodologia per la integrazione di software legacy con una tecnologia a componenti, descrivendo in vari casi d'uso, il processo di integrazione ed un linguaggio per la meta-descrizione degli elementi computazionali trasformati.

Le appendici descrivono alcuni casi di studio ed alcuni tool di Component Based Software Engineering (CBSE) di supporto al processo di integrazione.

In appendice A è riportato un esempio di integrazione di un modulo applicativo di libreria in un componente parallelo ASSIST e la sua successiva trasformazione in un componente dotato dello skeleton corrispondente. Viene inoltre descritto il tool CBSE ***ast2comp*** che supporta la componentizzazione nell'ambiente ASSIST**.**

In appendice B è illustrato un componente che abilita la esecuzione adattiva di componenti su griglia computazionale fornendo servizi di coontrollo sulle risorse (grid-awareness).

In appendice C viene sinteticamente presentato un esempio di meta-descrizione di componenti legacy di libreria in un dialetto xml.

3. Definizioni

Al fine di una migliore comprensione del contenuto di questo lavoro si riportano in seguito le definizioni di alcuni termini utilizzati:

- **Elemento Software**: unità computazione più o meno complesssa che deve soddisfare a dei requisiti minimi che ne garantiscano la sicurezza d'uso e la robustezza quali:
  - Correttezza di esecuzione del codice;
  - Assenza di effetti collaterali;
  - Gestione di eccezioni ed errori.

- **Libreria:** una collezione di elementi software sviluppati per essere utilizzati da altri utenti in contesti non completamente controllati dallo sviluppatore.

- **Interfaccia**: un insieme di metodi con la loro "signature" (accezione comunemente adottata nell' OOP), che definisce la frontiera dell'elemento computazionale o un suo sottoinsieme quali l'interfaccia funzionale, l'interfaccia di I/O.

- **Porta**: definisce un modello di comunicazione fra componenti . Ogni porta è legata ad una interfaccia e ne definisce il tipo (port type). La composizione fra componenti può avvenire connettendo porte dello stesso tipo (CCA [19]). La parte client della porta è denotata *uses* , la parte server *provides*. La connettività tra porte è garantita anche nel caso in cui la porta use è un sottotipo della porta provide.

- **Protocollo**: insieme di regole utilizzate per lo scambio di informazione tra due elementi software.

- **Componente**: un elemento software componibile in quanto:
  - si conforma a regole di architettura nella interazione con altri elementi;
  - le sue frontiere sono definite in maniera astratta e descritte in un appropriato linguaggio.

- **Griglia Computazionale:** una piattaforma software distribuita con nodi di elaborazione, elementi computazionali gestiti localmente da Organizzazioni Fisiche e resi disponibili ad coordinamenti dinamici come Organizzazioni Virtuali.

- **Grid enabling:** supporto al ciclo di vita di un componente/applicazione (deployment, attivazione, comunicazione, monitoring). da parte di uno strato di middleware di servizio (Run Time System) .

- **Grid-awareness:** adattività del componente, della applicazione o del middleware di servizio alla disponibilità di risorse sulla piattaforma o alla qualità di servizio ottenuta.

Le definizioni di Interfaccia, Porta e Componente sono compliant col modello di componenet Grid.it specification [22].

4. Il codice legacy

4.1. Introduzione

Le tecnologie informatiche mutano con una tale rapidità da comportare dei costi e tempi di aggiornamento talvolta notevoli per le organizzazioni. Una completa integrazione delle tecnologie emergenti col software già esistente comporterebbe risparmi notevoli per l'utilizzatore finale e dunque dei vantaggi rilevanti per la diffusione stessa del prodotto proposto. Ma il codice esistente talvolta possiede caratteristiche che rendono problematiche eventuali attività di riuso e integrazione. I software datati ad esempio si presentano come applicazioni monolitiche oppure come sub-routines in archivio o libreria dinamica e non presentano una architettura modulare.

4.2. Definizione di codice legacy

Con il termine ***codice legacy*** denotiamo l'insieme del software in possesso e correntemente utilizzato da una organizzazione o azienda. I ***Legacy System*** sono anche definiti come "*grandi sistemi software con cui non si vorrebbe avere a che fare ma che sono vitali per l'organizzazione*" [6], ovvero "*ogni sistema informativo che resiste alle modifiche ed evoluzioni necessarie per tener dietro ai nuovi e mutevoli requisisti di business dell'organizzazione*" [7].

Solitamente un software legacy presenta alcune fra le seguenti caratteristiche:

- non è ben documentato ed è difficile da comprendere;
- è progettato secondo vecchie concezioni e paradigmi non più supportati dalle emergenti tecnologie;
- su di esso l'organizzazione ha pesantemente investito nel corso degli anni;
- è scritto in linguaggio di vecchia generazione (ad esempio Assembler, Fortran);
- non presenta una architettura modulare in quanto sviluppato come un'applicazione monolitica oppure come sub-routine in una libreria.

E' importante sottolineare che il termine codice legacy non si riferisce solamente a sistemi software di cui si dispone del codice sorgente. In figura 1 è riportato uno schema che illustra i possibili stati di esistenza di un software legacy.

Avole si possiede il codice sorgente del sistema legacy,, scritto in qualunque linguaggio di programmazione, talvolta si dispone solamente del codice compilato, senza il relativo codice sorgente. Inoltre il codiec oggetto può essere diasponibile in forma di eseguibile (eventualmente sviluppato secondo un'architettura client-server) o come moduli oggetto, raramente strutturato secondo una con architettura a componenti (es. CCM, COM).

In ogni caso un fattore fondamentale per un corretto riuso degli elementi software esistenti è dato dalla descrizione delle loro funzionalità e di aspetti non funzionali quali performance, dominio applicativo, tecnologia di appartenenza. Per tali motivi sarebbe più corretto parlare di meta-descrizione più che di descrizione. La meta-descrizione permette di sviluppare nell'ambiente di programmazione componenti CBSE che assistono il programmatore nel processo di sviluppo del codice di interfacciamento (stubs,skeletons o porte)

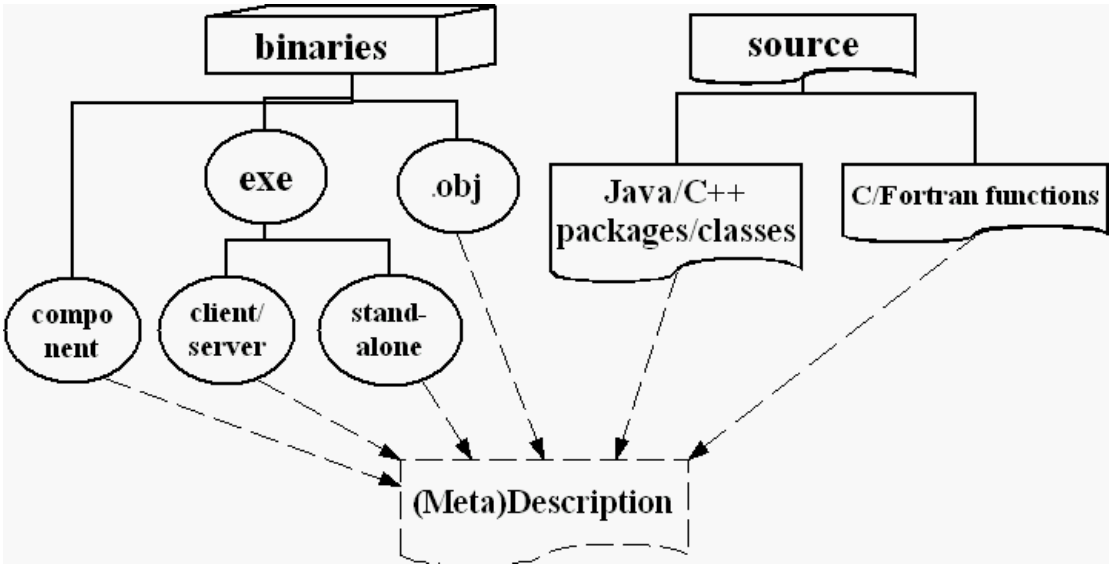


Figura 1. Possibili stati di esistenza di un elemento software legacy

4.3. Gli approcci di riferimento

Diversi sforzi sono già stati fatti nell'individuare dei meccanismi per poter utilizzare le emergenti tecnologie senza sacrificare il ricco patrimonio applicativo sviluppato nel passato. Gli approcci possibili per trattare un Legacy System sono:

- **eliminare** il software dal proprio processo aziendale;
- **sostituire** il software re-implementando tutto da zero;
- **trasformazione graduale** del software con una nuova tecnologia/linguaggio;
- **integrazione**: utilizzare il software esistente senza modificarlo effettuando del wrapping con tecnologie ad hoc.

I primi due approcci presentano il vantaggio di produrre un nuovo sistema software che possiede tutte le proprietà apportate dalla emergente tecnologia ma talvolta sono impraticabili in quanto richiedono ad una organizzazione interrompere il proprio processo aziendale per tutto il periodo di re-engineering.

Per agevolare queste operazioni sono stati creati alcuni tool che permettono la traduzione del codice da un linguaggio ad un altro ma non sempre essi riescono nell'intento e richiedendo l'intervento di reviisione da parte del programmatore ([11], [12] e [14]) per completare l'operazione di traduzione. Inoltre la traduzione automatica potrebbe introdurre un degrado nelle prestazioni del software sviluppato.

Il terzo approccio invece rappresenta una via intermedia ma non sempre è possibile o conveniente attuarlo e deve essere valutato con estrema attenzione. Molti progetti di migrazione del proprio sistema informativo (anche in grosse organizzazioni) sono poi ridimensionate o addirittura completamente cancellate. Per ridurre i rischi sarebbe opportuno usare un approccio strettamente metodologico, a tal scopo diverse metodologie sono state proposte [8] [9][10]. L'ultimo approccio invece sembra il più adeguato agli scopi della nostra trattazione in quanto non richiede unicità di processo o di linguaggio , ed è più adeguato ad uno sviluppo decentralizzato.

4.4. L'integrazione di codice legacy

La possibilità di poter integrare e manutere il codice esistente dovrebbe essere una caratteristica essenziale di ogni nuova tecnologia. L'ideale sarebbe anche fornire degli strumenti che in maniera semi-automatica o comunque assistita permettano di integrare del codice esistente in modo più agevole possibile. La tecnologia Java ad esempio fornisce un package sviluppato appositamente a tal scopo, le Java Native Interface (JNI) [4]. Questo package contiene una serie di classi che permettono di integrare dentro le proprie applicazioni java codice nativo scritto in altri linguaggi di programmazione.

L'operazione di integrazione del vecchio col nuovo è molto complessa e presenta anche degli svantaggi:

- aumento della complessità e dei costi di manutenzione del software finale;
- riduzione delle funzionalità e/o prestazioni del software originale;
- possibile degrado della robustezza del codice originale;

In letteratura sono riportati diversi studi sull' argomento. In [1] e [2] ad esempio viene descritto lo sviluppo di oppurtuni wrapper in grado di integrare moduli di librerie esistenti in componenti CORBA. L'approccio proposto utilizza una meta-descrizione dei moduli attraverso un formato XML. Un *wrapper generator* interpreta tale descrizione e genera automaticamente una descrizione IDL. A partire da quest'ultima vengono poi generati gli *stubs* e *skeletons* attraverso gli usuali strumenti dell'architettura CORBA. Il vantaggio consiste nel poter sfruttare l'infrastruttura che CORBA offre come collante per far convivere e dialogare moduli di librerie esistenti ma presenta il notevole limite di non supportare il paradigma delle Organizzazioni Virtuali.

In [3] si propone una metodologia per utilizzare codice legacy in uno scenario basato su tecnologia Web Service. Gli approcci proposti sono essenzialmente due:

- riscrittura del codice sorgente FORTRAN, C/C++ in codice Java da poter inserire all'interno di una Web Service.
- wrapping del codice esistente con delle classi Java attraverso la tecnica delle Java Native Interface (JNI). Il codice Java risultante viene poi inserito all'interno delle Web Services.

Il primo approccio presenta l'evidente svantaggio di dover riscrivere il codice, ciò introduce il notevole limite di possedere il codice sorgente e comporta dei rischi legati alla robustezza del software re-implementato. Questo approccio però introduce il vantaggio di creare delle librerie portabili e re-implementate secondo un criterio modulare e ingegneristico. Il secondo approccio invece presenta l'inconveniente di creare dei web service platform-dependent e quindi di strozzare l'emergente tecnologia utilizzata.

In [15] invece si cerca di utilizzare come collante l'ormai consolidata infrastruttura Web utilizzando la tecnologia Web Server come supporto run-time. L'invocazione delle funzionalità contenute nel software legacy avviene ancora una volta tramite dei componenti CORBA che fungono da contenitori del vecchio software. Qualunque client che voglia invocare i vecchi servizi deve sottoporre la propria richiesta al Web-server che, tramite tecnologie cgi, invoca il servizio richiesto individuando il componente CORBA che contiene la libreria che implementa la funzionalità richiesta.

Per individuare il servizio invocato ci si serve di una descrizione in formato XML molto più ricca della descrizione IDL di CORBA.

5. Le tecnologie a componenti

La programmazione a componenti nasce negli ultimi anni e introduce nel mondo del Software Engineering un nuovo modo di modellare il software. In maniera simile a quanto avveniva già coi componenti harware *plug and play* nella produzione di PC, le tecnologie a componenti poggiano sull'idea di costruire le applicazioni tramite dei componenti software facilmente aggangiabili tra loro. Esistono diverse definizioni di componente, tutte molto simili tra loro, tra tutte riportiamo quella data da Clemens Szyperski in [13] che definisce un componente come: "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*.".

Le architetture a componenti presentano delle caratteristiche tali da fornire un terreno fertile per l'integrazione di codice legacy, forse il migliore tra le diverse tecnologie oggi giorno disponibili, infatti esse:

- supportano un alto grado di riuso in quanto un nuovo sistema può essere sviluppato semplicemente assemblando componenti già esistenti;
- consentono la modifica di una parte dell'applicazione risultante senza ripercusioni sulle altre parti del sistema;
- nascondono l'implementazione di un componente esponendo solamente le funzionalità possedute;
- permettono la sostituzione/re-implementazione di un componente con un altro che presenta all'esterno le stesse funzionalità;

I componenti non possono operare isolatamente per cui hanno bisogno di un framework che ne supporta il ciclo di vita e che ne permetta l'intercomunicazione, essi hanno bisogno di una sorta di software-bus. Diversi standard sono stati realizzati, ognuno dei quali presenta pregi e difetti, i principali affermatisi nel mondo dell'industria sono: .NET di Microsoft, CORBA/CCM del gruppo OMG, JavaBeans/J2EE di Sun.

La trattazione di questi standard esula dallo scopo di questo documento Nel seguito vengono brevemente riportate le caratteristiche di ogni standard assunte nel modello dicomponenti di riferimento[22].

Enterprise Java Beans (EJB). Questa tecnologia estende il concetto di portabilità dei componenti Java. Essa è una specifica, guidata dalla Sun con il supporto di altri fornitori, che definisce quali debbano essere questi servizi e come i componenti possano richiamare le API (Application Programming Interface) che implementano tali servizi. Questa tecnologia possiede caratteristiche interessanti quali meccanismi di introspezione, scambio di eventi e messaggi, supporto alla programmazione concorrente multi-thread.

.NET tecnologia Microsoft [18] che, oltre a definire un modello a componenti, fornisce una suitedi prodotti, basato su standard industriali e Internet, che copre ogni aspetto relativo allo sviluppo (strumenti), alla gestione (server), all'uso (servizi di base e client intelligenti) dei servizi Web XML e applicazioni distribuite. .NET Framework rende l'interazione tra i componenti ancora più semplice consentendo ai compilatori di inserire informazioni dichiarative aggiuntive in tutti i moduli sviluppati. Queste informazioni, note come metadati, semplificano l'interazione tra i componenti poiché quando il codice viene eseguito, il runtime carica i metadati in memoria e vi fa riferimento per ottenere informazioni sul componente

caricato. In questo modo è possibile far interoperare componenti scritti in linguaggi differenti in quanto il *Common Language Runtime (CLR)* fornisce le necessarie basi per l'interoperabilità dei linguaggi specificando e applicando un sistema di tipi comune e fornendo metadati. Poiché tutti i linguaggi che si avvalgono del runtime osservano le regole del sistema di tipi comune per la definizione e l'uso dei tipi, i diversi linguaggi utilizzano i tipi nello stesso modo. I metadati contribuiscono all'interoperabilità dei linguaggi definendo un meccanismo unico per l'archiviazione e il recupero delle informazioni sui tipi.

CORBA Component Model (CCM [17]) è una specifica OMG che rappresenta estensione (in una versione enterprise) dell'affermata e ormai consolitada tecnologia CORBA. Caratteristica molto interessante è la classificazione dei componenti che vengono distinti in quattro categorie (Service, Session, Process, Entity). Inoltre essa definisce un meccanismo di trasmissione/ricezione di eventi attraverso delle interfacce standard (porte).

Common Component Architecture (CCA [19] [21]). Specifica che definisce un modello a componenti che, a differenza dei precedenti, è stato pensato per applicazioni ad alte prestazioni. Come tale esso possiede delle proprietà interessanti quali:

- linguaggio di descrizione delle interfacce più ricco;
- comunicazione parallela tra componenti,
- diversi meccanismi di connessione tra componenti;

6. L'approccio proposto

Nel contesto della programmazione su griglie computazionali, il paradigma delle Organizzazioni Virtuali impone il decentramento della gestione delle risorse computazionali, ed affida al middleware di interazione il compito di garantire la interoperabilità fra i servizi.

L'approccio al trattamento di software legacy di software maggiormente compatibile con tale paradigma è certamente quallo della integrazione in quanto permette la decentralizzazione il processo di riutilizzo del codice Software non pensato come modulare è inglobato in contenitori (wrapper) in grado di offrire le funzionalità degli elementi software incorporati e di fornire un meccanismo per poter far dialogare tali elementi tra loro

Il paradigma che attualmente più si presta a integrare software esistente in applicazioni modulari è quello a componenti. Da un lato le tecnologie a componenti hanno la caratteristica di possedere un alto livello di disaccoppiamento che permette l'invocazione dei servizi per chiamata indiretta (essenziale per l'interoperabilità binaria tra librerie), dall'altro permettono di separare ta l'operazione di wrapping del modulo di libreria dalla definizione del meccanismo di invocazione della funzionalità offerta.

6.1. Il concetto di wrapper

Il concetto di *wrapper* riveste un ruolo fondamentale nell'approccio proposto. Il suo significato è abbastanza semplice: si tratta di un livello software in grado di nascondere l'effettiva implementazione e dell'elemento software di rendere disponibili le sue funzionalità

nell'ambiente in cui esso vive presentandole attraverso un'interfaccia ben definita e secondo dei meccanismi noti.

Questo concetto contiene un alto grado di coerenza con la definizione di componente e permette di far apparire il vecchio software sotto forma di un componente, funzionalmente simile a tutti gli altri ed in grado di operare nello stesso ambiente, accettare messaggi dagli altri componenti e rispondere in modo chiaramente definito.

6.2. Modello di riferimento per il componente

La specifica del modello di componente per l'ambiente di programmazione GRID.IT è argomentodi studio di un apposito gruppo di lavoro del progetto ed oggetto di un deliverable ufficiale [23].

Ai fini della presente trattazione .è sufficiente il riferimento alle caratteristiche generali del modello attualmente già consolidate e riportate in figura 2..

Il componente espone, oltre all'interfaccia di invocazione che permette di attivare le funzionalità dello stesso, una serie di interfacce non funzionali ma fondamentali per assicurare caratteristiche di performance e grid-awareness necessarie in un contesto di griglia ad alte prestazioni. Tali interfacce sono classificate in:

- **interface**: interfaccia funzionale (con porte use/provides, eventi, streams,…);
- **introspection**: riflessione/introspezione (meta-descrizione in XML);
- **services**: interazione con il middleware(RTS);
- **adaptivity**: adattività al contesto (performance, fault tolerance, …).

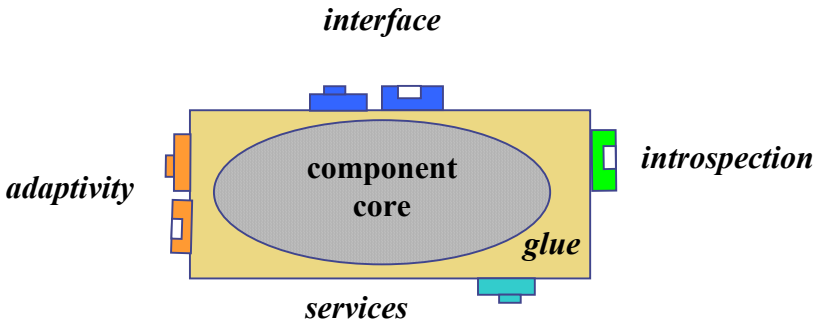


Figura 2. Architettura del componente Grid.it

Al centro della figura è mostrato il **component core,** un elemento non espressamente previsto dalla specifica, ma la cui rappresentazione nucleo del tutto indipendente dall'implementazione delle porte di comunicazione con l'esterno, cui è legato da software collante, è funzionale ad un processo di integrazione. Se si è in grado di integrare all'interno del core del componente l'elemento software esistente si riesce a rendere disponibili le sue funzionalità nell'ambiente a componenti.

Purtroppo il processo di integrazione non è completo. Si deve trovare un modo per implementare le interfacce non funzionali, generalmente non ricavabili dalle API del codice codice legacy.

Un modo per ovviare al problema è quello di definire una casistica di componenti con caratteristiche ridotte, cioè tali da contenere un sottoinsieme significativo delle interfacce previste nel modello, e fornire implementazioni di default (eventualmente banali) qualora l'implementazione relativa non sia derivabile dal codice originario. Per caratterizzare tale classificazione introduciamo il concetto di **stereotipo di componente** cui associamo un profilo di interfacciamento o **skeleton .** Alcuni di quasti stereotipi, ricavati dalla pratica di uso in ambienti legacy sono descritti nel seguito**.**

6.3. Stereotipizzazione dei componenti

A causa della svariata natura architetturale e tecnologica del codice legacy riteniamo opportuno categorizzare i profili di interfacciamento utilizzati dalla pratica di uso in ambienti legacy. Avere una categorizzazione delle funzionalità del componente, ci permette di definire uno **skeleton di interfacciamento** o **stereotipo** di un componente. Lo skeleton del componente è cioè determinato dalla combinazione delle proprie interfacce.

Per garantire piena compatibilità con il profilo generale identificato nella specifica GRID.IT, le varie interfacce identificate nei vari profili sono poste in corrispondenza diretta con le interfacce stabilite nel modello di riferimento (vedi tabella 2). Per lle interfacce previste dal modello ma assenti in un dato stereotipo di componente si aggiungono interfacce utilizzate con implementazione di default, eventualmente banale.

Possibili stereotipi di interfaccia potrebbero essere:

- **Call**: attiva le funzionalità vere e proprie del componente (associata ad una porta *interface provides*).
- **External service** interfaccia per invocare funzionalità appartenenti ad altri componente (associata ad una porta *Interface uses*).
- **Input/Output:** permette operazioni di input/output (associata ad una porta *system uses*)
- **Configuration**: configurazione del componente in termini risorse, richieste di QoS, ecc (associata ad una porta *adaptivity uses/provides*)
- **Environment memory**: interfaccia di gestione di stato persistente (variabili di ambiente) affidata ad un componente di sistema di "entità". Tale componente entità realizza una memoria condivisa "globale" a tutti i componenti dell'applicazione, contenendo tutte le variabili dell'applicazione (analogamente come avviene con gli "entity-Object" nel paradigma OOP), (interfaccia associata ad una porta *adaptivity uses/provides*).
- **Events**: interfaccia per la registrazione/emissione di eventi di proprio interesse (*system uses*).
- **Performance**. Interfaccia per la introspezione del modello di complessità computazionale/modello di costo associata al componente

In tabella 1 è riportata la classificazione delle interfacce proposte e la relativa corrispondenza con le interfacce previste dal modello di riferimento. In tabella 2 invece sono riportati gli stereotipi di componente in studio con il relativo skeleton di interfacciamento.

porte Stereotipi	Interface	Introspection	Adaptivity	Services
Simple serial	call			
Serial	call	Qos		I/O env-memory
ASSIST	call		configuration	I/O env-memory
CCSkel	call	QoS	configuration	I/O events env-memory
Grid-aware platform	call	QoS monitor	configuration	events
Grid-lib-server	call	QoS monitor	configuration	I/O events

Tabella 1. Stereotipi di interfacciamento di un componente

Stereotipo	class	call	conf	I/O	env-mem	event	ext service	Awareness	Fault tolerance	QoS	Prog Parad
Simple serial	simple	yes	--	--	--	--	--	--	--	--	RPC/ RMI
Serial	simple	yes	--	opt	opt	--	--	--	--	opt	RPC/ RMI
ASSIST	parallel	yes	--	opt	opt	--	--	--	--	--	MPMD
CCSkel	parallel	yes	opt	yes	yes	opt	--	--	--	yes	SPMD
Grid-aware platform	manager	yes	yes	--	--	yes	opt	yes	yes	yes	MCMD
Grid-lib-server	manager	yes	yes	opt	opt	yes	opt	yes	yes	yes	MCMD

Tabella 2. Componenti notevoli , loro interfacciamento e proprietà su griglia, modello di programmazione distribuita di riferimento

Descriviamo adesso gli stereotipi di componente finora individuati come rappresentativi di tipiche modalità di interfacciamento fra elementi strutturati in differenti ambienti di programmazione di codice legacy (casi notevoli).. La lista non è esaustiva e si riferisce agli stereotipi attualmente individuati in casi di studio. (vedi Appendice A e B).

Per evidenziare caratteristice comuni a più stereotipi di componente si introduce una loro classificazione, in componenti **simple**, **parallel** e **component manager**. Per ognuna di tali classi è riportata una figura che ne illustra il profilo di interfacciamento. In ogni figura sono riportate le interfacce implementabili ed il colore delle porte identifica la categoria di appartenenza della interfaccia in relazione allo schema di figura 2.

Simple Component

Componente semplice che possiede le caratteristiche minimali per poter utilizzare i meccanismi previsti dal modello di riferimento.

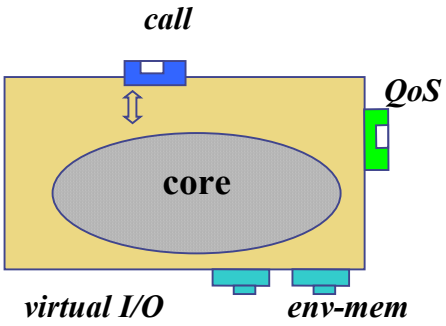


Figura 3. Simple Component

Stereotipi individuati in questa categoria sono:

- **Simple Serial**: componente che integra un eseguibile. Solitamente possiede solo l'interfaccia di attivazione (main).
- **Serial**: componente che integra un modulo di libreria. Possiede uno skeleton di interfacciamento più ricco del precedente grazie a meccanismi di linking di librerie per la gestione delle porte.

Parallel Component

Applicazione parallela strutturata costituita da un set di processi coordinati il cui grafo presenta caratteristiche di rigidità. Il componente utilizza generalmente un modello di programmazione a memoria distribuita ed scambio di messaggi utilizzando un proprio comunicatore.

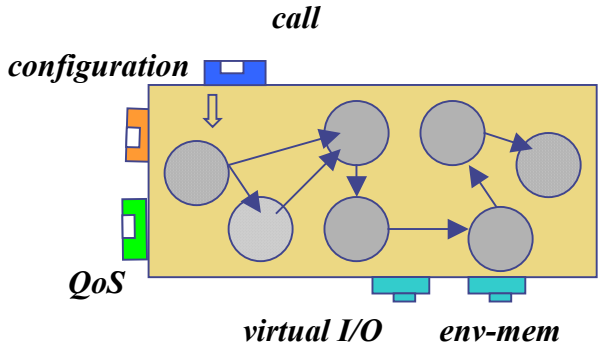


Figura 4. Parallel Component

Stereotipi individuati in questa categoria sono:

- **ASSIST 1**. Componente che integra un grafo di processi coordinati secondo il paradigma dataflow tramite tecnologia ASSIST1. Possiede interfacce di attivazione di ritorno di stato (vedi Appendice A), ed interfacce verso l'ambiente (I/O, memory)
- **CCSkel**. Componente sviluppato come riferimento per la realizzazione di librerie scientifiche ottimizzate a partire da codice modulare. Questo tipo di componente possiede particolari caratteristiche di configurazione dinamica per cui impone al codice legacy da wrappare dei vincoli di strutturazione interna. La descrizione dello stato di avanzamento di un prototipo implementativo è riportata in [23].

Component Manager

Componente composto che coordina componenti di servizio e di analisi, sviluppati secondo un modello definito. In genere possiede un certo grado di autonomia (elemento pro-attivo) e offre sevizi più o meno complessi e/componibili (es. OGSI Container).

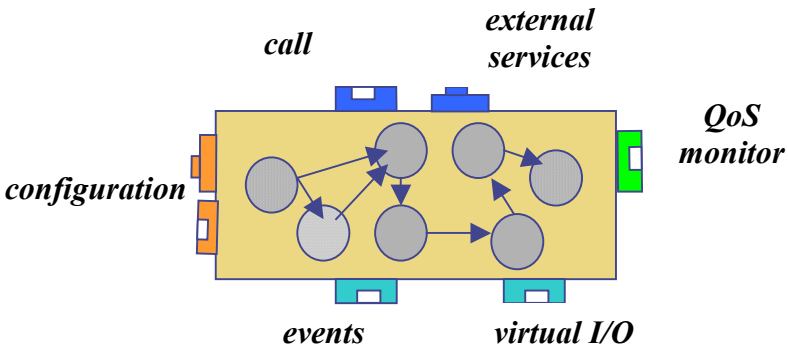


Figura 5. Component Manager

Stereotipi attualmente individuati in questa categoria sono ed oggetto di studio sono:

- **GRid-Aware Platform managEr (GRAPE).** Container piattaforma in grado di fornire servizi di instanziazione e monitoraggio di componenti su griglia (grid-enabling, monitoring) nonchè il supporto alla riconfigurazione a run-time delle loro porte.
  Lo stato di avanzamento di un prototipo di componente con tale stereotipo è descritto in Appendice B.

- **GRid Aware sCientific library sErvice (GRACE)**: Gestore di componenti di libreria scientifica che non sono di per se grid-aware. Fornisce meccanismi di awareness di griglia grazie all'interazione col middleware sottostante.
  Lo stato di avanzamento di un prototipo di componente con tale stereotipo è descritto in [23].

6.4. Descrizione degli elementi software

Come già sottolineato il concetto di descrizione di un elemento software sta alla base dell'approccio proposto. Essa non si riferisce solamente alla descrizione delle funzionalità del software ma anche ad aspetti non funzionali quali performance, dominio applicativo, tecnologia di partenza, ecc. Per tali motivi sarebbe più corretto parlare di meta-descrizione.

E' importante sottolineare che la meta-descrizione costituisce una sorta di documento di identità di un elemento software che, per trasformazioni successive, fornisce informazioni utili durante tutto il suo ciclo di vita.

Ad esempio, al fine di una corretta e automatica integrazione di una libreria scientifica all'interno di un componente (attraverso appositi tool sviluppati ad hoc), sono indispensabili una serie di informazioni quali funzionalità, input/output, versione del compilatore, ecc. Quando il componente è ultimato, per un suo corretto schieramento e per la esecuzione dello stesso sulla griglia, sono indispensabili la conoscenza della sua architettura, delle variabili di ambiente da esso utilizzate, del sistema operativo per il quale è stato compilato e la versione del compilatore utilizzata. La stessa descrizione dovrebbe essere utilizzata in meccanismi di reflection o introspezione durante l'esecuzione del componente.

Caratteristiche essenziali che uno schema di descrizione di un elemento software dovrebbe comprendere sono:

- **Architettura** funzionale dell'elemento (x Resource Management)
  - Stereotipo di strutturazione funzionale
  - Architettura del processore
  - Sistema Operativo
  - Linguaggio di programmazione
  - Grafo dei processi (DAG)
  - Livello di certificazione

- **Fisiologia** (x Resource composition/activation)
  - Interazione con altri elementi dell'ambiente operativo
  - Parametri /argomenti formali di attivazione (signature)
  - Gestione degli errori
  - Input/output
  - Uso di elementi esterni

- **Ontologia** del contesto applicativo (x Resource Discovery)
  - Depository address
  - Descrizione semantica del funzionamento dell'elemento
  - Descrizione semantica della signature
  - Descrizione delle regole d'uso
  - Modello di performance (cost model)

In appendice C è riportata una descrizione tecnica degli schemi XML in corso di realizzazione.

6.5. Il processo di integrazione

Si propone adesso un processo che permette, in modo agevole, di integrare un elemento software esistente in un componente degli stereotipi precedentemente descritto. Esso si compone di diverse fasi, ognuna delle quali identifica le operazioni da svolgere e i documenti da produrre. Come si vede in figura 3 il processo è iterativo: se durante la fase di test del componente target sviluppato non si ottengono i risultati voluti allora si ritorna indietro per ripetere le fasi di identificazione e sviluppo del componente target.

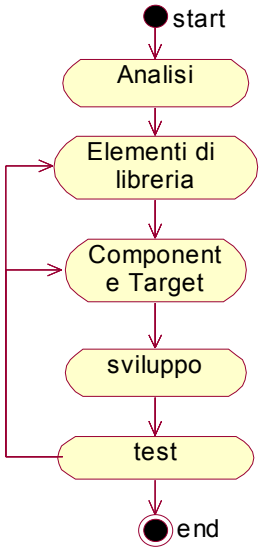


Figura 6. Fasi dell'approccio proposto

Di seguito è riportata una breve descrizione delle fasi individuate.

1 - Analisi del Software Legacy. Si analizza il software di partenza per stabilire la fattibilità del processo di integrazione. Il punto cruciale sta nella possibilità di ricostruire un certo livello di descrizione del software. La sistuazione migliore è quella di avere a disposizione il codice sorgente del software per poter individuare le funzionalità anche in caso di mancata descrizione. In questi casi si procede alla compilazione di opportuni *template* che documentano il codice sorgente in stile ***javadoc***. Nei casi in cui si dispone di

codice già compilato (sotto forma di modulo oggetto o di eseguibile) è auspicabile avere almeno un certo livello di descrizione della funzionalità software. da integrare.

2 - Individuazione degli elementi software. Una volta analizzato il software si dovrebbe cercare di raggrupparene al massimo le funzionalità. Per far ciò si devono tenere presenti principi di indipendenza (ogni elemento software dovrebbe essere il meno possibile indipende dagli altri) e di modularità. Una volta individuati le funzionalità dell'elemento software si crea la meta-descrizione degli elementi di libreria in base ad un apposito schema di descrizione. In questa fase si deve porre attenzione ad individuare (e descrivere) tutte le caratteristiche legate all'architettura del componente, all'utilizzo di I/O, socket, variabili di ambiente, ecc.

3 - Individuazione dello stereotipo del componente Target. In base alle caratteristiche individuate al punto precedente si individua lo stereotipo di componente che più si adegua all'architettura dell'elemento software.

4 - Sviluppo del nuovo componente. Nella realizzazione del componente si propone un approccio strutturato in cui:

- combinazioni predefinite di interfacce definiscono componenti "notevoli" o stereotipi di un componente,
- componenti complessi sono creati a partire dalla composizione di componenti semplici e operano da component managers.

Una nota importante è data dal fatto che si dovrebbe lasciare il più possibile indipente lo sviluppo del core del componente dalle parti *boundary* che implementano i meccanismi di comunicazione con l'ambiente e gli altri componenti (porte). Questo sdoppiamento ci permette di realizzare un alto grado di interoperabilità e di dinamicità durante l'esecuzione del componente attraverso un binding dinamico del core del componente con le proprie porte. Crediamo che siano possibili almeno tre livelli differenti di binding tra il core di un componente e le sue porte:

- **Statico**: l'implementazione delle porte è integrata staticamente con il core del componente, ciò introduce un alto grado di rigidità ma in alcuni casi è l'unica possibilità. Ad esempio si pensi al caso in cui il codice legacy da integrare è un eseguibile con architettura server che riceve dei comandi tramite socket, in questo caso la porta è già implementata all'interno dell'eseguibile.
- **Dinamico**: l'implementazione delle porte risiede in librerie dinamiche sostituibili all'avvio del componente. Ciò introduce un alto grado di dinamicità in quando si traduce in una configurazione del meccanismo di comunicazione della porta (protocollo, tecnologia di trasporto, middleware, ecc) a tempo di avvio.
- **Virtuale**: riteniamo possibile inserire nella porta dei meccanismi di gestione dinamica dell'effettiva implementazione del canale di comunicazione. La libreria dinamica della porta potrebbe possedere una sorta di manager in grado di caricare/sostituire dinamicamente degli opportuni driver che implementano ognuno un meccanismo/canale differente di cominucazione. In questo modo si ottiene un certo livello di riconfigurazione run-time permettendo la distinzione tra il *canale logico* di comunicazione (modificabile durante il ciclo di vita del componente) e l'effettivo

canale fisico (riconfigurabile dinamicamente per soddisfare i cambiamenti del canale logico).

7. Conclusioni

Con il termine ***codice legacy*** denotiamo l'insieme del software in possesso e correntemente utilizzato da una organizzazione o azienda.

Il miglior modo per riutilizzare software non pensato come modulare è quello di creare dei contenitori (wrapper) in grado di offrire le funzionalità degli elementi software incorporati e di fornire un meccanismo per poter far dialogare tali elementi tra loro.

Le architetture a componenti presentano delle caratteristiche tali da fornire un terreno fertile per l'integrazione di codice legacy, forse il migliore tra le diverse tecnologie oggi giorno disponibili. Infatti una tecnologia a componenti permette di:

- sfruttare il supporto run-time e l'infrastruttura di comunicazione messa a disposizione dalla tecnologia a componenti, che rimane svincolata dal *core del componente;*
- wrappare il codice legacy all'interno del componente, riuscendo ad ottenere un alto grado di integrazione e allo stesso tempo servirsi dei vantaggi apportati dalla nuova tecnologia.

Poiché non tutto il codice è integrabile con la complessità permessa dal componente la metodolgia individua un sottoinsieme di funzionalità tipiche di contesti applicativi e del software legacy che in essi viene sviluppato. Indica un percorso che a partire dalla descrizione (API) permette, tramite semplici tools CBSE (Component-Based Software Engineering) la costruzione automatica dei meccanismi di comunicazione (porte).

8. Appendice A : Componentizzazione di una applicazione parallela dataflow sviluppata in ASSIST 1.1

8.1. Introduzione

In questa appendice si presenta il risultato di uno studio sul sistema Assist e la descrizione di un tool che si integra con il linguaggio Assist-CL per la realizzazione di componenti.

Il linguaggio Assist-CL è particolarmente orientato alla codifica di applicazioni dataflow, in cui il flusso di lavoro dei processi e la sincronizzazione si effettua unicamente attraverso un passaggio di dati. In altre parole è presente solo un flusso di dati e non, anche, un flusso di controllo. La gestione del controllo e dell'attivazione delle singole parti del programma può essere effettuata solo all'interno del flusso di dati, a cura del programmatore che codifica le operazioni necessarie per adattarlo a funzioni di controllo (per gestire, ad esempio, protocolli di comunicazione che consentano l'invio di messaggi con risposta).

Il linguaggio Assist-CL introduce un tipo di dati, lo ***stream***, e le operazioni (invio e ricezione) consentite su di essi, nonché un formalismo per la loro dichiarazione. Il linguaggio gestisce il trattamento di questi dati, e gli eventi che li caratterizzano (generazione, chiusura, emissione, ricezione) all'interno del grafo dei processi coordinati.

Assist, infine, utilizza un modello gerarchico di composizione delle applicazioni, sempre secondo un modello dataflow, per cui una applicazione può essere composta quale nodo di un grafo più esteso.

Questa possibilità prevede di effettuare comunicazioni con elementi esterni all'applicazione stessa, sempre con messaggi tipizzati ma con una modalità diversa rispetto alla comunicazione interna al grafo. Quest'ultima, infatti, avviene attraverso un comunicatore interno, chiuso, che gestisce le comunicazioni in modo privato ed è adatto solo ad un numero fissato di processi ben determinati.

Per la comunicazione con oggetti esterni, è previsto l'utilizzo di una metodologia CORBA ed il ricorso al suo IDL ed il suo compilatore per la costruzione delle opportune interfacce. Tale IDL rimane quindi esterno al linguaggio.

Per sondare le possibilità di estensione del linguaggio Assist-CL si è costruito un metodo, e da esso un tool che lo implementa, che consenta di inserire nel linguaggio la descrizione dei messaggi esterni.

Il metodo individuato introduce due nuovi tipi di messaggi "esterni", coerenti con il resto del linguaggio; il formalismo introdotto costituisce una descrizione delle interfacce del componente, descrizione che è effettuata con un linguaggio (IDL), che, per la sua somiglianza e per la sua coerenza con il linguaggio Assist-CL, può esserne definito un dialetto.

Il tool sviluppato, ***Ast2comp*** (ast-to-component), è un traduttore, dal linguaggio IDL individuato, al linguaggio Assist, che si può integrare semplicemente come precompilatore del linguaggio, con la caratteristica di non introdurre elementi di natura diversa (come, per esempio, nuove direttive) al linguaggio, ma solo due nuovi tipi, perfettamente coerenti con gli altri tipi del linguaggio.

Si accenna brevemente ad un altro tool, in fase di progetto, ***Api2ast***, che ha il compito di integrare automaticamente un modulo o kernel di libreria, di cui si abbia la descrizione, cioè una API, dentro un file ".ast" scrivendo automaticamente o in modo assistito il relativo codice Assist-CL ed anche le parti IDL che ***Ast2comp*** dovrà elaborare.

8.2. Assist (1.1)

Assist (**A** **S**oftware development **S**ystem based on **I**ntegrated **S**keleton **T**echnology) è un sistema di sviluppo di applicazioni distribuite, sviluppate scrivendo un file ".ast" che contiene sia il codice (Assist-CL) di coordinamento, sia il codice (C++,C, o Fortran) di elaborazione, e compilando il file con il compilatore del linguaggio Assist-CL, AstCC [25].

L'applicazione, costruita con Assist, è costituita da un certo numero di eseguibili, che devono essere avviati singolarmente su uno o più computer. Gli eseguibili, all'avvio, si sincronizzano attraverso un meccanismo standard di barriera TCP/IP su porta nota: i parametri di esecuzione specificano la posizione, nella forma di indirizzo e porta TCP/IP, di uno di essi, definito master, così tutti gli altri si connettono a quell'indirizzo (hostname:porta) comunicando di essere attivi ed attendendo un segnale di avvio.

Quando tutti gli eseguibili sono attivi viene avviata l'applicazione vera e propria, che funziona grazie ad una gestione autonoma di messaggi tra i vari processi. Tutte le comunicazioni avvengono tramite socket TCP/IP.

L'applicazione non ha possibilità di ricevere o inviare messaggi da/verso l'esterno, né di essere avviata con particolari parametri di esecuzione, poiché la stringa dei parametri è utilizzata per il coordinamento, come sopra indicato.

L'applicazione è strutturata: ogni processo elabora una certa parte del lavoro, alla fine della quale emette uno o più messaggi verso altri processi. L'arrivo dei messaggi innesca l'elaborazione, così si è in presenza sia di processi che all'avvio iniziano l'elaborazione, sia di processi che invece si pongono in attesa di messaggi.

I messaggi costituiscono gli eventi sui quali si sincronizza l'esecuzione coordinata di processi autonomi. I messaggi sono unidirezionali, dunque sono adatti più ad un passaggio di dati che ad un flusso di controllo.

Come in tutte le applicazioni parallele, si può individuare un grafo dei processi, in questo caso distribuiti, in cui gli archi che legano i diversi nodi sono costituiti dai messaggi, cioè dai dati che vengono scambiati tra i processi. Non è prevista la composizione ulteriore della applicazione dopo la compilazione del codice .ast.

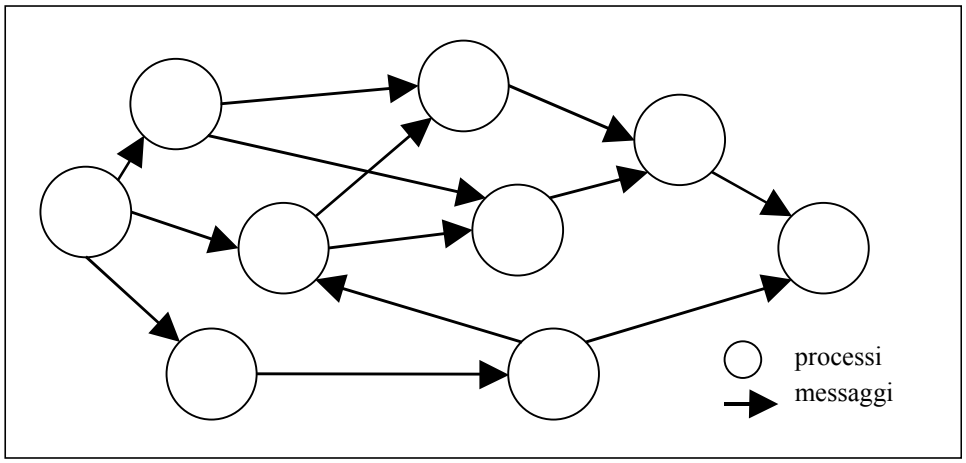


Figura 7: Grafo dei processi di un'applicazione sviluppata con Assist

Per utilizzare l'applicazione legacy sviluppata in Assist, occorre aggiungere dei meccanismi che implementino delle interfacce, ed una descrizione che permetta di utilizzarle.

Per effettuare questa operazione si è pensato di inserire nel grafo dei processi due nuovi processi, che svolgano queste funzioni di interfacciamento, in ingresso ed in uscita. I processi devono essere implementati come gli altri processi del grafo, cioè è necessario che contengano, come tutti gli altri, la gestione della sincronizzazione e della comunicazione, e che siano lanciati, come tutti gli altri, nell'ambito del meccanismo di comunicatore chiuso con barriera.

Per tali ragioni è opportuno che il codice di questi eseguibili sia inserito opportunamente dentro il file ".ast". Il tipo di implementazione delle due interfacce (di ingresso e di uscita), che si è individuato, segue la stessa logica della comunicazione interna al grafo; infatti le interfacce sono costituite da un passaggio di dati dall'esterno verso l'applicazione (interfaccia di ingresso) e dall'applicazione verso l'esterno (interfaccia di uscita). Si sono cioè inseriti dei messaggi, della stessa forma, ma di natura diversa, rispetto a quelli interni al grafo, che consentono la comunicazione esterna al grafo.

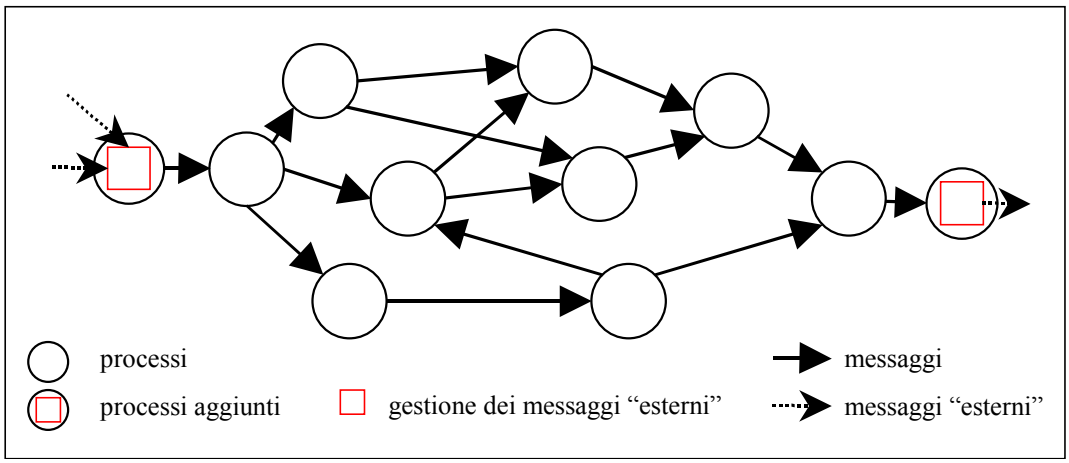


Figura 8: Grafo dei processi modificato

Occorre anche aggiungere una descrizione. Essa deve contenere informazioni sugli eseguibili (compresi i due eseguibili aggiunti), sul loro numero, sui parametri di esecuzione, ed anche su alcune caratteristiche dell'applicazione (come, per esempio, necessità che un certo eseguibile sia lanciato su un particolare nodo, oppure presenza in un certo eseguibile di un server e porta relativa che viene aperta).

La descrizione deve contenere anche informazioni sulle interfacce implementate, sia cioè il tipo di dati che sono passati, sia il tipo di implementazione con cui sono realizzate. Riguardo quest'ultimo aspetto si è pensato di ricorrere ad un meccanismo di binding dinamico con librerie esterne. La vera e propria implementazione dei meccanismi di interfaccia, cioè risiede in una libreria dinamica esterna, che è agganciata a tempo di esecuzione. Dunque, mantenendo stabili i metodi (o meglio, le "signature") delle operazioni necessarie alla gestione dell'interfaccia, è possibile cambiare l'implementazione senza dover ricompilare il codice dell'applicazione.

8.3. Assist-CL (versione 1.1)

Assist-CL è il linguaggio di coordinamento del sistema di sviluppo Assist. Tale linguaggio prevede un solo file sorgente, con estensione ".ast", che viene compilato tramite il compilatore AstCC. Il file sorgente contiene sia il codice di coordinamento, sia il codice ospite o codice utente, cioè il codice che il programmatore inserisce per implementare l'elaborazione. Il codice utente può essere scritto in linguaggio C++, C oppure Fortran.

Dal punto di vista del coordinamento l'applicazione è costituita da un blocco main che contiene la dichiarazione dei dati di interesse e l'invocazione dei metodi di elaborazione su tali dati, metodi che possono essere definiti dall'utente all'esterno del main.

I dati di interesse di questo linguaggio di coordinamento sono messaggi tipizzati, detti *stream*. Essi vengono dichiarati all'interno del main secondo un formalismo che specifica quale sia il dato portato da ciascun messaggio (per il contenuto dei messaggi è possibile definire delle struct o utilizzare alcuni dei tipi del C++). Gli *stream* hanno un unico vincolo: obbligatoriamente ciascuno *stream* deve essere passato come parametro di almeno un metodo con la parola chiave **input_stream** e di almeno un metodo con la parola chiave **output_stream**.

I metodi, detti *moduli*, vanno definiti all'esterno del main, come procedure, con i loro parametri di ingesso e di uscita, che sono costituiti da *stream*. I *moduli* hanno accesso solo agli *stream* specificati come loro parametri. I moduli seguono un formalismo tipico delle procedure: la definizione comprende l'intestazione (la signature) seguita da un blocco di codice. Il blocco di codice può essere codice utente, nel quale caso il blocco ha dei simboli particolari di apertura e chiusura che ne specificano il linguaggio ($c++{}c++$, $c{}c$, $f{}f$ nei tre casi di codice utente in C++, C, Fortran)

Un particolare modulo, detto *parmod* o modulo parallelo, ha al suo interno del codice Assist-CL di coordinamento di una struttura parallela di processi, detti *processori virtuali*, con la possibilità di definirne caratteristiche e modalità di interscambio dei dati. Anche il *parmod*, come tutti i *moduli*, ha come parametri di ingresso ed uscita degli *stream*.

A ciascuno dei *moduli*, tranne il *parmod*, corrisponde un eseguibile (si può pensare ad un grafo dei moduli che rappresenta la struttura logica dell'applicazione); a ciascuno *stream* corrisponde un messaggio tipizzato. Il *parmod* viene compilato in modo particolare, generando tre eseguibili, che corrispondono alle tre sezioni in cui è diviso:

1. sezionedi ingresso: ricezione ed elaborazione degli *stream* di ingresso; distribuzione dei dati ai processori virtuali
2. sezione dei *processori virtuali*: elaborazione distribuita
3. sezione di uscita: raccolta dei dati dai processori virtuali, generazione degli *stream* di uscita

Il numero di *processori virtuali* che eseguono il *parmod* è fissato staticamente a tempo di compilazione.

Una caratteristica del *parmod* è che l'eseguibile generato a partire dal codice della sezione dei *processori virtuali* può essere lanciato su più processori fisici, permettendo di variare la distribuzione effettiva dell'elaborazione secondo una scelta effettuata prima del lancio dell'esecuzione

Dal punto di vista del linguaggio, l'introduzione sopra enunciata di due nuovi processi corrisponde alla scrittura del codice di due nuovi moduli ed alla formalizzazione della codifica di due nuovi tipi di dato.

8.4. Il metodo individuato

Essendo Assist (nella versione implementativa attuale, la 1.1) concepito come un sistema di sviluppo di applicazioni complete intrinsecamente distribuite, focalizza le sue funzionalità sulla gestione interna del grafo, sull'interscambio di messaggi, sulla ricerca di una performance in esecuzione. Non essendo necessario in tale visione, non è previsto il passaggio di parametri all'applicazione (nemmeno tramite il meccanismo delle variabili di ambiente argv, argc).

Nell'ambito di un contesto di utilizzo coordinato di varie applicazioni, qual è il contesto di griglia, appare utile fornire delle interfacce affinché l'applicazione generata con Assist, possa essere vista come un componente, e possa interagire con altri componenti nell'ambiente di progtrammazione.

Si è quindi studiato un metodo che possa permettere di introdurre questa caratteristica, scrivendo a mano il codice per individuare passo passo questo processo, e se ne è sperimentata una implementazione in un case study di Image Processing.

L'applicazione di prova costruisce, a partire da una sequenza di immagini di un filmato, le mappe di moto dei frame secondo l'algoritmo CZS (Circular Zonal Search). In tale applicazione è necessario potere comunicare, dall'esterno, alcuni parametri di ingresso. Nel caso particolare i parametri vengono scelti in un ambiente di supporto allo steering, alla visualizzazione ed elaborazione di filmati in un ambiente cooperativo distribuito ,Tra tali parametri sono presenti ovviamente le coordinate (url o path) in cui si trova il filmato da elaborare, le indicazioni su quali frame siano da prendere in considerazione, più altri parametri relativi al metodo di elaborazione scelto. L'ambiente ha poi bisogno di conoscere dei valori di ritorno per potere individuare e visualizzare sulla interfaccia grafica d'utente le mappe di moto costruite con l'applicazione-test.

Per effettuare questa comunicazione, si è utilizzato il pattern architetturale acceptor costruendo un *modulo* aggiuntivo al grafo dell'applicazione, il cui compito è esclusivamente quello di aprire un socket, ricevere i parametri di ingresso da remoto, tradurli opportunamente in *stream* di Assist ed inviarli ai successivi *moduli* di elaborazione. Tale *modulo* si comporta dunque da producer dei dati (dal punto di vista dell'applicazione parallela), ovvero da interfaccia di ingresso (dal punto di vista dell'ambiente esterno remoto).

Analogamente si è costruito un altro *modulo* aggiuntivo che ha il solo compito di ricevere i dati di finali dai *moduli* di elaborazione tramite *stream* di Assist e di tradurli in modo opportuno inviandoli via socket allo stesso ambiente richiedente. Tale *modulo* si comporta dunque da consumer (dal punto di vista dell'applicazione parallela), ovvero da interfaccia di uscita (dal punto di vista dell'ambiente esterno remoto).

Oltre a queste operazioni si è costruita una descrizione più completa di quella fornita, dal compilatore Assist 1.1, al proprio supporto a run-time (*clam* ovvero Coordination Language Abstract machine). La descrizione che si è generata è invece orientata ad un utilizzo esterno all'applicazione Assist, cioè ad un utilizzo dell'applicazione in un contesto in cui di essa siano disponibili solo la descrizione e gli eseguibili. La descrizione contiene informazioni sugli eseguibili (numero, nomi), sul deployment (necessità di essere lanciati su qualche nodo in particolare, numero delle istanze da lanciare di ogni eseguibile), sul comportamento verso l'esterno (presenza di socket in attesa o in connessione e relative porte, descrizione delle interfacce).

Tale descrizione consente ad un ambiente remoto, che non conosce altro della applicazione se non la sua descrizione, di lanciare (utilizzando degli strumenti adeguati che consentano la

mappatura ed il lancio degli eseguibili su una rete di computer) l'applicazione e di utilizzarla remotamente per una elaborazione di prova.

Il metodo individuato parte dalla generazione dell'IDL, cioè dalla scrittura del codice relativo ai messaggi "esterni". Questo codice può essere scritto a mano dal programmatore, oppure, nell'ambito dell'ambiente di programmazione, può essere inserito automaticamente da un tool (con interfaccia grafica).

Tale approccio è utile quando si fa riferimento ad un inserimento di codice legacy dentro una applicazione Assist e dunque un componente. Infatti l'esempio considerato può essere considerato anche come un metodo per inserire delle chiamate a moduli di librerie note all'interno di grafi di Assist. In tal caso occore che la libreria sia descritta opportunamente, ed un altro tool in progetto, Api2ast, si occupa di scrivere correttamente sia le chiamate ai metodi di libreria dentro opportuni moduli Assist, sia l'IDL di descrizione dei messaggi "esterni" che sono necessari.

Dunque il primo passo del processo in figura non è a carico del tool ***Ast2comp*** di seguito descritto, ma è un passo che va effettuato a cura del programmatore, o, in un ambiente di programmazione più avanzato, a cura di un tool automatico (per esempio, nel caso di integrazione di librerie: ***Api2ast***).

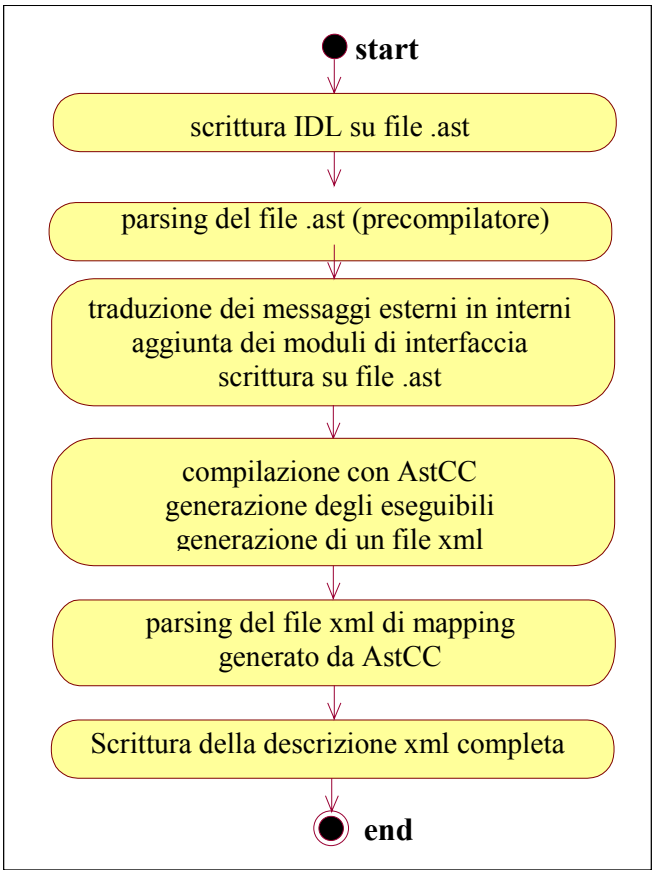


Figura 9: fasi dell'approccio sperimentato

8.5. Il tool *Ast2comp*

É in fase di completamento lo sviluppo del tool ***Ast2comp***, che viene qui brevemente descritto con un esempio.

Il tool opera come interprete-traduttore generando il sorgente del codice utente che implementa interfacce di attivazione ed uscita espresse in un semplice IDL dialetto del linguaggio Assist-CL.

Ast2comp legge un file .ast, nell'esempio il file si chiama "_Memod_application.ast" (Memod sta per Motion Estimator module, è il modulo di libreria che viene utilizzato nell'esempio).

Il file .ast contiene del normale codice Assist, con l'aggiunta di due tipi di stream esterni al linguaggio Assist-CL nella versione 1.1: gli **extern-input** stream e gli **extern-output** stream. Tali stream sono trattati dal programmatore come gli altri stream (gli extern-input stream sono indicati nei moduli che li ricevono come normali input_stream, gli extern-output stream come normali output_stream), con due differenze: la dichiarazione di ciascuno di essi, e il fatto che gli extern-input stream non hanno alcun modulo che li genera come output stream, mentre gli extern-output stream non hanno alcun modulo che li riceve come input stream.

```
generic main()
{
        extern-input stream long MotionAlgorithm;
        extern-input stream long disp_max;
        extern-input stream long block_size;
        extern-input stream char clip_path[1024];
        extern-input stream char out_path[1024];
        extern-input stream long clip_begin;
        extern-input stream long nframes;
        extern-output stream long result;

        ela( input_stream MotionAlgorithm, disp_max, block_size, clip_path, out_path,
        clip_begin, nframes  output_stream result);
}
```

Figura 10: codice del main iniziale (dopo la scrittura del codice IDL)

Nell'esempio di cui si mostra il codice (prima e dopo le modifiche introdotte) è presente un solo parmod denominato "ela", ma il software funziona con grafi complicati a piacere, poiché non modifica nulla nelle dichiarazioni dei moduli.

Ast2comp modifica la parte del main in cui sono definiti gli extern-input stream e gli extern-output stream, eliminando le due parole chiave extern-input ed extern-output. Aggiunge all'interno del main le chiamate a due nuovi moduli, generati automaticamente. Il modulo prod ha il compito di generare gli stream che per il resto del grafo sono extern-input, il modulo cons ha il compito di ricevere gli stream che per il resto del grafo sono extern-output.

Ast2comp aggiunge in coda al file le implementazioni dei moduli prod e cons.

Il codice contenuto in ciascuno dei due moduli contiene l'istanza di una classe e l'invocazione di alcuni suoi metodi che svolgono le operazioni necessarie:

1. prod contiene invocazione di metodi che consentono di porsi in attesa di una connessione, accettarla, ricevere i dati; il modulo prod contiene anche il codice per copiare i dati negli stream di uscita (cioè le chiamate della funzione Assist_out) ed il

codice che descrive il ciclo di attesa della connessione (il processo generato da prod si comporta come un server, tornando in attesa di nuove connessioni ogni volta che termina l'elaborazione di una connessione);

2. cons contiene l'invocazione di metodi che consentono di effettuare una nuova connessione, inviare i dati, chiudere la connessione; il modulo cons contiene anche il codice per copiare i dati ricevuti dagli stream di ingresso.

L'implementazione vera e propria delle interfacce è contenuta in una libreria dinamica esterna, in modo tale da potere sfruttare le potenzialità del binding dinamico. Ridefinendo la libreria esterna, pur mantenendo le stesse signature dei metodi utili per l'implementazione di ciascuna interfaccia, è possibile utilizzare l'implementazioni diverse delle interfacce senza dover ricompilare il codice Assist.

Il main modificato appare modificato come in Fig. 11 (in questa forma è corretto per una compilazione con il compilatore astCC):

```
generic main()
{
        stream long MotionAlgorithm;
        stream long disp_max;
        stream long block_size;
        stream char clip_path[1024];
        stream char out_path[1024];
        stream long clip_begin;
        stream long nframes;
        stream long result;

        prod( output_stream MotionAlgorithm, disp_max, block_size, clip_path, out_path,
        clip_begin, nframes);
        ela( input_stream MotionAlgorithm, disp_max, block_size, clip_path, out_path,
        clip_begin, nframes  output_stream result);
        cons( input_stream result);
}
```

Figura 11: codice del main dopo la modifica (scritto in Assist-CL corretto)

Completata questa operazione ***Ast2comp*** salva il file .ast e chiama il compilatore astCC con gli opportuni parametri per effettuare una compilazione senza clam e generare un file xml di configurazione.

Nell'esempio il file che astCC genera è il file "_Memod_application.ast.xml". Questo file xml viene generato automaticamente dal compilatore astCC e contiene alcune informazioni sugli eseguibili di cui è composta l'applicazione e sulla mappatura sui nodi della rete di host disponibile (come indicata in un file di configurazione del compilatore). Viene utilizzato dalle clam nel caso di compilazione con clam. Nel caso di compilazione senza clam la mappatura è libera, nel senso che non deve essere rispettata quella scritta nel file xml generato da astCC, ma si possono scegliere i processori fisici su cui mappare gli eseguibili.

Ast2comp apre il file xml appena generato dal compilatore astCC ed individua la sezione in cui sono indicati gli eseguibili.

A partire dalle informazioni che in essa trova, oltre a quelle che già ha dalla scansione del file .ast, ***Ast2comp*** genera un file di descrizione completo che consente il lancio

dell'applicazione ed il suo utilizzo (sono qui mostrate solo le parti relative agli eseguibili, ma nella sezione physiology vengono descritte anche le caratteristiche delle porte). In aggiunta occorre anche indicare l'implementazione delle porte e le librerie che sono state utilizzate a tale scopo.

In questa forma il componente Assist è utilizzabile in cooperazione con altri componenti; un vincolo è costituito dal fatto che le porte sono implementate, ma con binding dinamico: è consentita una variazione della implementazione, al costo dello sviluppo della libreria dinamica che la contiene, senza variare gli eseguibili generati con Assist. La collaborazione con altri componenti è possibile in presenza di una suite di tools di servizio che effettuino il deployment degli eseguibili su una rete di processori disponibili ed il lancio degli stessi con gli opportuni parametri; ed è sottoposta al vincolo che tali componenti possano usare interfacce compatibili con quelle indicate nella sua descrizione del componente.

```
<libraries>
_<lib lib_id="lND000_mainprod" type="assist" object_type="EXE"
url="/home/assist/compiledir/bin/ND000_mainprod"></lib>
_<lib lib_id="lND001_mainela_ivp" type="in" object_type="EXE"
url="/home/assist/compiledir/bin/ND001_mainela_ivp"></lib>
_<lib lib_id="lND001_mainela_osm" type="out" object_type="EXE"
url="/home/assist/compiledir/bin/ND001_mainela_osm"></lib>
_<lib lib_id="lND002_maincons" type="assist" object_type="EXE"
url="/home/assist/compiledir/bin/ND002_maincons"></lib>
</libraries>
```

Figura 12: parte del codice _MEmod_application.ast.xml generato da AstCC

```
<?xml version […]>
<SW_ELEM_DESCRIPTION xmlns:cmp=[…]>
<cmp:component>
 <cmp:anatomy>
   […]
   <cmp:architecture  name=”assist_MEmod”  info=”Ast2comp generated component”>
     <cmp:selem_type  CLASS=”PARALLEL”  STEREOTYPE=”ASSIST” />
     <cmp:coding  CODE_NAME=”ND000_mainprod”  FORMAT=”_EXE_” />
     <cmp:coding  CODE_NAME=”ND001_mainela_ivp”  FORMAT=”_EXE_” />
     <cmp:coding  CODE_NAME=”ND001_mainela_osm”  FORMAT=”_EXE_” />
     <cmp:coding  CODE_NAME=”ND002_maincons”  FORMAT=”_EXE_” />
   </cmp:architecture>
 </cmp:anatomy>
 <cmp:physiology>
 […]
 </cmp:physiology>
 <cmp:ontology>
 […]
 </cmp:ontology>
</cmp:component>
```

Figura 13: parte del codice generato da *Ast2comp*

9. Appendice B: Un esempio di component manager grid-aware

9.1. Introduzione

In questa sezione verrà presentato un esempio *component manager* e un middleware di griglia che offre funzionalità minimali per la sua esecuzione. In figura 8 è mostrata una rappresentazione grafica di quanto spiegato in seguito.

Il component manager **GRid-Aware Platform managEr (GRAPE)** è in fase di disegno anche se alcuni prototipi dei suoi componenti di serviziosono in fase di realizzazione prototipale. Il componente di supporto di griglia (VPG-RTS) si trova in uno stato prototipale ed è attualmente utilizzato per sperimantazioni interne.

Il concetto che sta alla base di tutta l'architettura è quello di ***Griglia Virtuale Privata* (VPG)**. L'idea è di poter prenotare delle risorse di griglia in modo da poter stabilire una rete di nodi computazionali su cui far eseguire il componente. Ovviamente la griglia privata può essere riconfigurata dinamicamente aggiungendo/togliendo dei nodi durante l'esecuzione del componete.

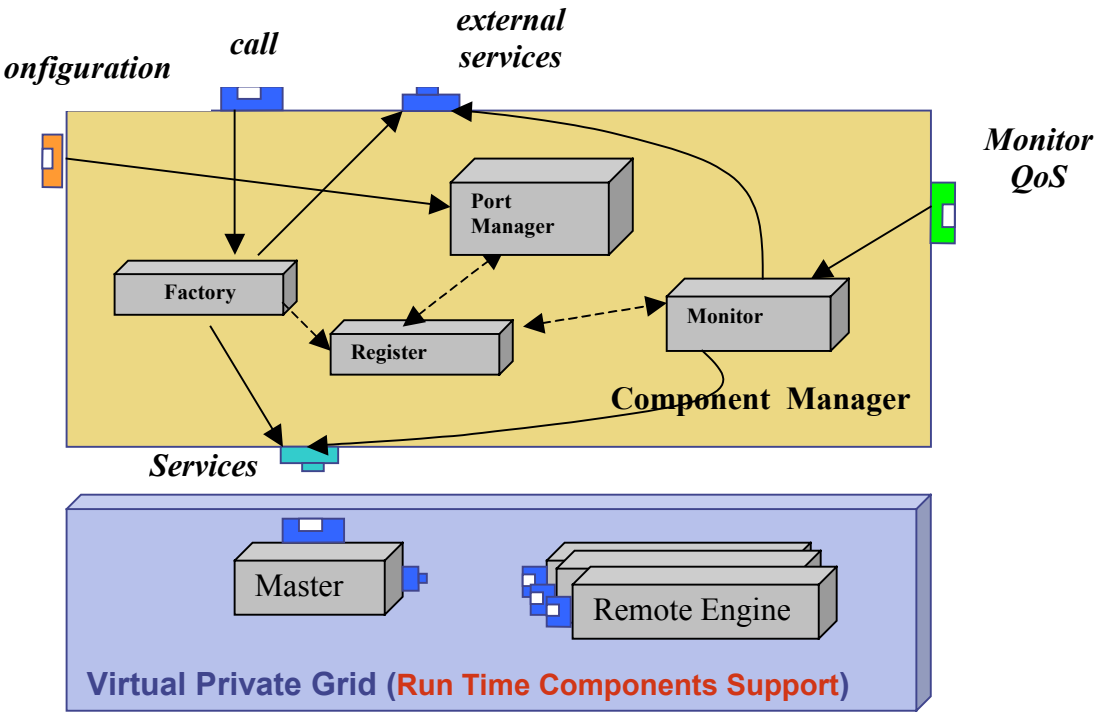


Figura 14. Schema del component manager

9.2. Il component manager

Questo componente presenta un'architettura modulare ed è composto da diversi componenti semplici/paralleli coordinati che forniscono tutte le funzionalità OGSA complaint (e non solo) per gestire dei componenti semplici realizzati secondo un certo modello di riferimento. Questo componente è in grado di fornire servizi di instanziazione (factory) di componenti grid-enabled. La "*awareness*" della griglia è fornita tramite un servizio di riconfigurazione dinamica a run-time del grafo dei componenti instanziati e ri-adattando lo schieramento dei componenti in base alle risorse disponibili.

Le funzionalità offerte possono essere così riassunte:

1. **Gestione delle risorse**. Fornisce ai componenti tutte le risorse necessarie per la loro esecuzione e tiene traccia dei componenti instanziati.
2. **Instanziazione** di nuovi componenti.
3. **Monitoraggio** continuo sul ciclo di vita dei componenti e memorizzazione del loro stato.

Di seguito è riportata una breve descrizione dei componenti facenti parte del component manager.

a) **Factory**. Gestisce il pool di risorse computazionali ed fornisce un servizio di instanziazione di componenti (factory). Una volta stabilito il caricamento del componente aggiorna il registro dei componenti instanziati e invia i comandi opportuni al middleware sottostante per lo schieramento ridondante del componente sui nodi compatibili della piattaforma.
b) **Register**. Tiene memoria dello stato di tutti i nodi di griglia (VPG) e di tutti i componenti che sono stati instanziati.
c) **Monitor**. Controlla lo stato di avanzamento del ciclo di vita dei componenti e comunica col Register in caso di variazioni. Offre un servizio di interrogazione sullo stato dei componenti instanziati.
d) **Port Manager**. Si occupa di fornire un servizio di binding tra i componenti instanziati. Tale attività può offrire un alto grado di riconfigurabilità della connessione dei componenti qualora il loro sviluppo lo prevede (vedi par. 6.5).

9.3. Virtual Private Grid – Run Time System (VPG-RTS)

Compito di questo middleware è quello di fornisce primitive uniformi e indipendenti dai meccanismi sottostanti utilizzati (pattern facade [23]). Definisce un più alto livello di interfacciamento in modo da rendere più semplice l'utilizzo di tali meccanismi, il prototipo attualmente realizzato si basa sul GLOBUS toolkit 2, ma è estendibile per utilizzare altri meccanismi. L'idea è quella di fornire una macchina virtuale che emula su griglia un cluster di processori (Virtual Private Grid) attraverso un set ridotto di funzioni che implementano i meccanismi primitivi di gestione di nodi computazionali su griglia [26].

Questo software è stato sviluppato seguendo una logica a componenti e si compone di due componenti: un **master** che costituisce il front-end della VPG e offre tutte le funzionalità

del middleware attraverso dei comandi espressi in formato XML. Una volta prenotata una risorsa si lancia un demone, il **Remote Engine,** che fornisce tutti i servizi necessari per supportare l'intero ciclo di vita di un processo. Primitive attualmente implementate sono:

- Start/Stop Platform;
- Mount/unmount host;
- Ping/monitor host;
- Load/Exec/Kill/Clean set di processi;
- Status su un set di processi (DAG);

Di seguito è riportata una breve descrizione dei componenti del sistema:

a) **Master.** Questo componente ha il compito di effettuare il deployment e l'avvio di tutti i processi/thread di cui è composto un componente. Prima di schierare gli eseguibili/thread deve avviare sugli hosts il supporto run-time (il Remote Engine) in modo da istituire la Virtual Private Grid. Lo schierameto dei processi e la costruzione della VPG avviene attualmente grazie ai servizi di globus 2 (GRAM, Grid FTP). Per i nodi che non possiedono tali servizi è in corso lo studio di una soluzione alternativa basata sempre su comunicazioni sicure gsi (ssl) e sistema di certificazione X.509. I compiti di questo componente possono essere così riassunti:

   - lanciare l'esecuzione del "Remote Engine" sui nodi;
   - caricare sui nodi della VPG i processi/thread di cui si compone il componente da instanziare;
   - lanciare l'esecuzione dei processi/thread (e non solo: kill, suspend, resume…);
   - inviare dei comandi di controllo alle Remote Engine: Keep Alive, Shutdown….

b) **Remote Engine.** Fornisce dei servizi run-time presso un nodo della griglia privata virtuale, il suo compito è quello di supportare l'intero ciclo di vita di ogni processo/thread avviato presso un nodo. Il Remote Engine viene avviato dal master e vive secondo una logica di griglia: se oltre un certo time-out non riceve un segnale di keep-alive esegue il proprio SHUTDOWN liberando le risorse impegnate. Di seguito è riportato l'elenco dei comandi che la Remote Engine è in grado di eseguire:

   - LOAD: viene preparato l'ambiente per l'esecuzione del processo. Dopo aver verificato i requirements, il componente viene trasferito nel nodo insieme alle librerie di cui ha bisogno, vengono settate le varibili di ambiente;
   - EXEC: il processo viene eseguito;
   - KILL: il processo viene ucciso;
   - CLEAR: vengono liberate tutte le risorse utilizzate dal processo;
   - SHUTDOWN: Il Remote Engine viene spento insieme a tutti i processi attivi, vengono liberate tutte le risorse impegnate (CPU, Memoria…);
   - KEEP-ALIVE: comando di controllo che tiene il nodo montato sulla VPG.

10. Appendice C: Meta-descrizione XML di elementi software

In questo documento viene discussa un primo prototipo in fase di sviluppo riguardante una struttura di metadati che consenta lo scambio di informazioni atte alla composizione di *workflow* di elementi basati su di una tecnologia a componenti tramite un ambiente di sviluppo di applicazioni che utilizza moduli di librerie, componenti e servizi software già esistenti.

L'obiettivo è quello di potere disporre di un ambiente che consenta di assemblare tra loro, a tempo di configurazione, una applicazione per un particolare dominio applicativo che utilizzi componenti software e moduli di librerie remoti e/o paralleli senza che ciascuno di questi abbia una conoscenza *a-priori* degli altri ma tramite uno scambio di meta-informazioni in grado di essere gestite da un substrato dell'ambiente.

Per fare sì che quanto detto sopra possa trovare una sua realizzazione è necessario che ciascuna delle parti in gioco renda disponibile una conoscenza prima di tutto di se stesso e dei propri meccanismi di interazione con il mondo esterno ed, inoltre, che il framework di ambiente, visto anch'esso come un componente dell'applicazione, avendo disponibile una conoscenza di se stesso ed avendo a disposizione quella delle altre parti abbia la capacità di gestire, monitorare ed allocare le risorse, locali e non, disponibili.

Per quanto riguarda l'architettura dei componenti software demando il lettore ai relativi capitoli, mentre in questa appendice concentrerò l'attenzione su quella che potrebbe essere una struttura dei metadati per i meccanismi di *reflection* da sfruttare per la costruzione dei *wrapper* che abilitano al riuso del software legacy.

Perchè tali elementi software possano interagire tra di essi e possano interfacciarsi tra loro e con altri componenti eterogenei si richiede una descrizione di come sono stati implementati e di quali meccanismi di attivazione sia possibile implementare; è, inoltre, necessaria una descrizione delle risorse di cui necessitano e di meccanismi che permettano di valutarne le prestazioni in ambienti di rete che sfruttano infrastrutture come ad esempio la griglia.

Quanto appena detto ci permette di identificare per ciascuna delle tipologie di elementi software trattati (moduli applicativi legacy, componenti e servizi), tre macro-aree di meta informazioni; nel seguito fornirò una breve trattazione per ciascuna di esse e mostrerò, infine, una possibile ipotesi di schemi atti a contenere tali metadati. La tecnologia software utilizzata per la scrittura di tali meta-informazioni è quella di documenti in linguaggio *XML* (Extensible Markup Language) per i documenti descrittivi, *XML-Schema* (sviluppato al W3C) per gli schemi che ne definiscono la struttura ed *RDF* (Resource Description Framework) per lo sviluppo delle ontologie relative ai domini di applicazione. In figura 16 viene mostrato il diagramma dello schema degli elementi della descrizione XML relativa ad un modulo software.

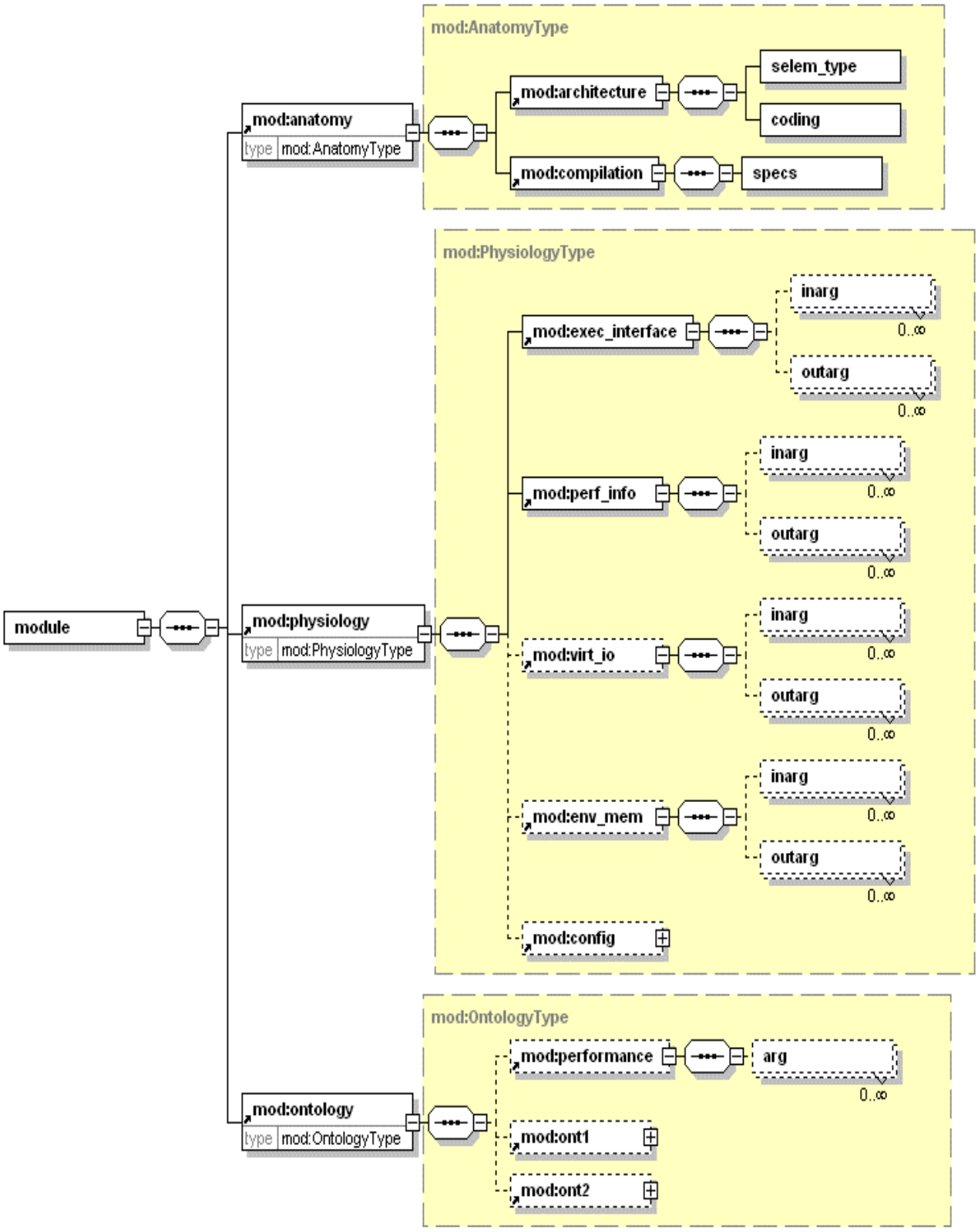
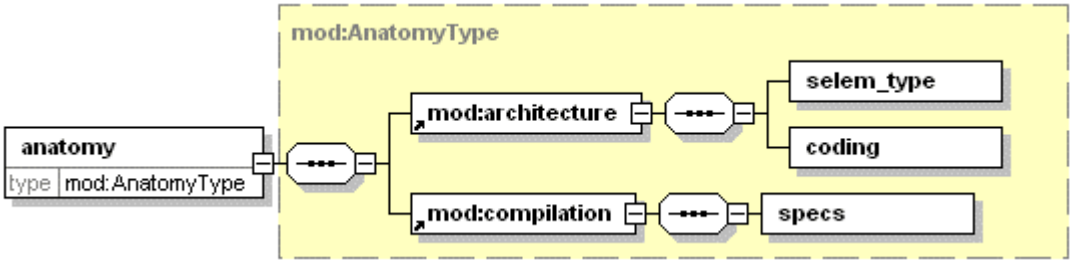


Figura 16. Schema delle meta-informazioni architetturali

La prima sezione descrittiva necessaria per qualsiasi componente e/o modulo software è quella che, in qualche modo, può essere identificata come quella che ne descrive la sua conformazione anatomica (*Anatomia*), cioè quella parte di descrizione in cui si esprime l'architettura funzionale dell'elemento, in quale ambiente e con quale linguaggio è stato sviluppato, il livello di certificazione, il grafo dei processi che lo implementano e tutto ciò che sia utile alle funzionalità del *Resource Menagement* dell'ambiente in cui andrà ad essere attivato. In figura 17 è mostrato lo schema degli elementi per la parte architetturale del modulo di libreria ed i relativi attributi utilizzati.



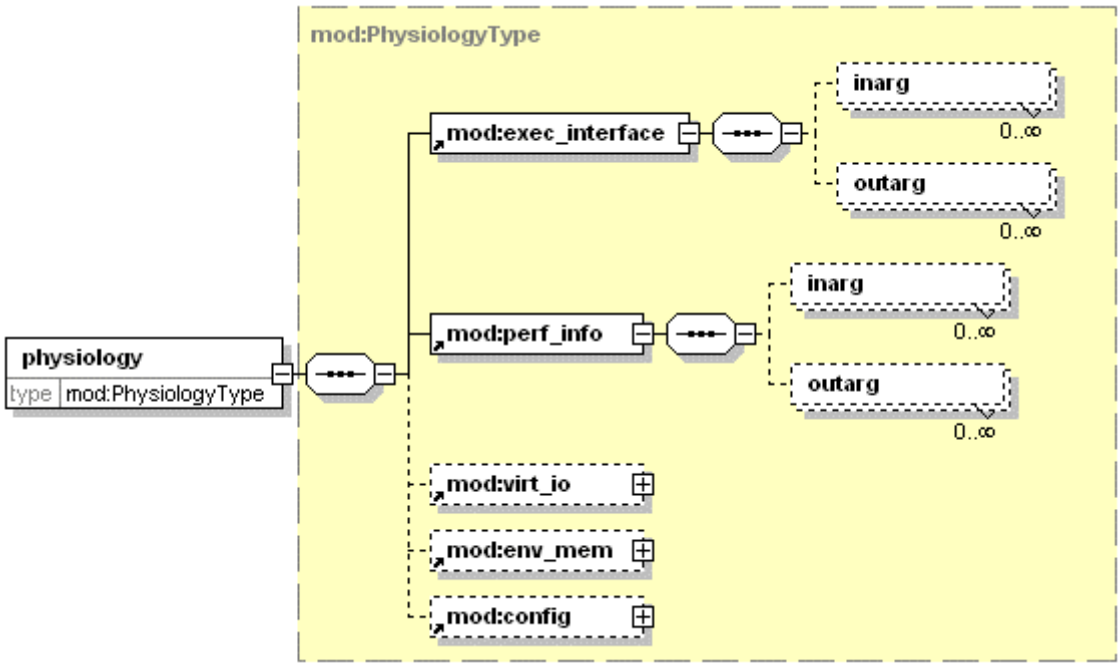
Attribute Name	Type	Use
name	string	required
info	string	optional
Element: selem_type		
category	Token	required
Element: coding		
code_name	string	required
format	Token	required
type	Token	required
Element: specs		
lang	Token	required
compiler	Token	required
Comp_vers	string	
machine	string	
os	Token	required

Figura 17. Schema delle meta-informazioni architetturali di un modulo di libreria

Ad esempio in questa sezione sarà descritto di quale tipo di elemento software si tratta: modulo di libreria, componente, servizio ecc. (attributo *category*), quale linguaggio di programmazione (*lang*) tipo e versione (*compiler*, *comp_vers*) di compilatore è stato utilizzato e così via.

La seconda sezione descrittiva è quella che si occupa di esprimere le caratteristiche funzionali dell'elemento software (fisiologia), cioè quella parte di descrizione in cui si esprimono le modalità di interazione con gli altri elementi, i parametri di attivazione le eventuali porte use/provide (I/O) ed, in generale, tutto ciò che specifica l'utilizzo

dell'elemento da parte degli altri componenti dell'applicazione. In figura 18 è mostrato lo schema degli elementi per la parte funzionale ed i relativi attributi utilizzati.

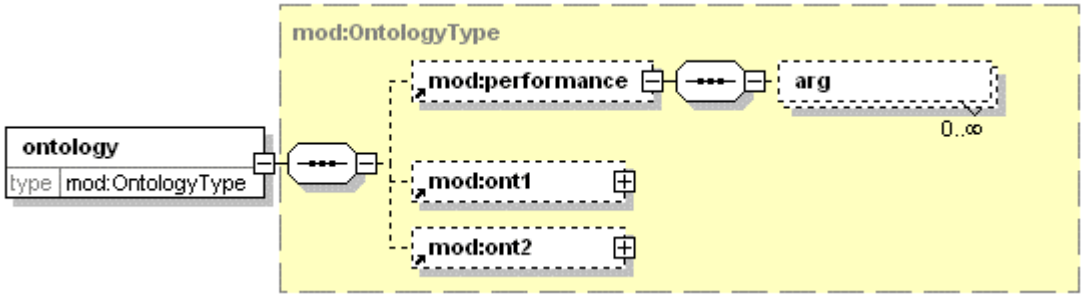


Attribute Name	Type	Use
name	string	required
info	string	optional
Element: inarg, outarg		
name	string	required
valtype	Token	required
constraintype	Token	required
info	string	
min	Int	
max	Int	
pos	nonNegativeInteger	

Figura 18. Schema delle meta-informazioni funzionali

Ad esempio in questa sezione saranno descritti i parametri di attivazione del modulo: nome (*name*) , tipo (*valtype*), dominio di esistenza (*constraintype*), valore minimo (*min*) e massimo (*max*).

La terza sezione descrittiva è quella che si occupa di esprimere le caratteristiche semantiche dell'elemento software (ontologia), cioè quella parte di descrizione in cui si esprimono le regole d'uso e di funzionamento dell'elemento. In figura 18 è mostrato lo schema degli elementi per la parte ontologica ed i relativi attributi utilizzati.



Attribute Name	Type	Use
Elements: exec_interface, perf_info		
name	string	required
info	string	optional
Element: inarg, outarg		
name	string	required
valtype	Token	required
constraintype	Token	required
info	string	
min	Int	
max	Int	
pos	nonNegativeInteger	

Figura 19. Schema delle meta-informazioni ontologiche

In questa sezione vengono descritte, ad esempio, i parametri che il modulo di introspezione individua come significativi nella stima del modello di performance utilizzato; il significato degli attributi è analogo a quello già esposto nell'esempio precedente.

La descrizione del componente viene generata a partire dalla descrizione del modulo di libreria che il componente integra (core). Nella sezione architettura viene descritto lo stereotipo del componente e l'eseguibile/modulo di libreria (o gruppo) che esso integra. La sezione fisiologia invece si ottiene trasformando le interfacce del modulo in porte ed integrando la descrizione della loro implementazione. Il tipo di interfaccia del modulo viene mappato in un'istanza di porta di classe corrispondente.
In figura 20 è riportato un frammento di un esempio di tale descrizione, esso si riferisce ad un componente ASSIST descritto in appendice A (in tale appendice è anche riportato il frammento relativo all'architettura). In questo esempio il componente presenta una porta socket staticamente implementata in un processo del grafo (ND000_mainprod). Nel frammento è riportata anche la descrizione di default di una porta prevista dal modello di riferimento ma che non è presente nello stereotipo del componente.

```
…
<cmp:component>
    </cmp:anatomy>
          < cmp:architecture>
          …
          </cmp:architecture>
   </cmp:anatomy>

  < cmp:physiology>
          <cmp:interface NAME=”call“ BINDING=”static” CODE_NAME=”ND000_mainprod”
FORMAT=”_EXE_”>
             <cmp:port>
                   …<!--this is a WSDL-like description of the socket port -->
             </cmp:port>
          </cmp:interface>
          …
<cmp:introspection NAME=”QoS“ BINDING=”default” CODE_NAME=”null” FORMAT=”null”>
             <cmp:port>
                   …<!--this is a WSDL-like description of the socket port -->
             </cmp:port>
          </cmp:introspection>

  </cmp:physiology>
```

Figura 20. Frammento di descrizione di un componente ASSIST

11. Bibliografia

[1] O. F. Rana, M. Li, M. S. Shields, and D. W. Walker "*A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components*", in Proceedings of the IEEE/ACM SC2000 Conference, held in Dallas, TX, November 10-12, 2000.

[2] Li M, Rana O F and Walker D W *"CB-PSE - A Component-Based Problem Solving Environment"*, in Proceedings of the First International Conference on Information Reuse and Integration, published by the International Society of Computers and their Applications, held in Atlanta, USA, (1999) pp 7-10.

[3] Elizabeth L. White, Mark Pullen, "*Adapting Legacy Computational Software for XMSF*" , George Mason University, 2003.

[4] Java Native Interface (JNI), http://java.sun.com/j2se/1.4.1/docs/guide/jni/index.html;

[5] A. van Deursen, B. Elsinga, P. Klint and Ron Tolido. "*From Legacy to Component: Software Renovation in Three Steps*" CAP Gemini White Paper, 2000.

[6] K. Bennet, T. Bull, and H. Yang. "*A transformation system for maintenance: turning theory into practice"*. In [Kellner1992], pages 146-155, 1992.

[7] Brodie, Michael ; Stonebraker, Michael, "*Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*", Morgan Kaufmann, 1995, ISBN 1-55860-330-1.

[8] A. van Deursen, P. Klint, and C. Verhoef. "*Research issues in the renovation of legacy systems*". In J.-P. Finance, editor, Fundamental Approaches to Software En-gineering (FASE'99), volume 1577 of Lecture Notes in Computer Science, pages 1–21. Springer-Verlag, 1999.

[9] M. Fowler. "*Refactoring: Improving the Design of Existing Code*". Addison-Wesley, 1999.

[10] I. Jacobson, M. Griss, and P. Jonsson. "*Software Reuse; Architecture, Process and Organization for Business Success"*. Addison Wesley, 1997.

[11] Source Translation Utility (STU) http://www.sourcerecovery.com/translators.htm;

[12] DATATEK http://www.datatek-net.com/dtk_conversion.htm ;

[13] Clemens Szyperski, "*Component Software: Beyond Object-Oriented Programming*", Addison-Wesley, 1998.

[14] Kontogiannis, K., Martin, J., Wong, K., Gregory, R., Muller, H. and Mylopoulos, J., "*Code Migration Through Transformations: An Experience Report*", In Proceedings of CASCON'98, Toronto ON., November 1998.

[15] Ying Zou, Kostas Kontogiannis, "*Migration and Web-Based Integration of Legacy Services*", the 10th Centre for Advanced Studies Conference (CASCON), Toronto, Ontario, Canada, November 2000, pp. 262-277.

[16] Javabeans Specifications, http://java.sun.com/products/javabeans/glasgow;

[17] CCM official Website. http://ditec.um.es/~dsevilla/ccm/.

[18] Framework .NET official Website , http://www.microsoft.com/net/;

[19] Rob Armstrom, Dennis Gannon, Katarezyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinsk. "*Toward a common component architecture for high-performance scientific computing*". In Conference on High Performance Distributed Computing, 1999

[20] Y. Huang and D. W. Walker. *Jacaw-a java-c automatic wrapper tool and its benchmark*. In International Parallel and Distributed Processing Symposium(IPDPS), 2002.
[21] The Common Component Architecture Technical Specification – Version 0.5. http://cca-forum.org/bindings/old-0.5/.
[22] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, C. Zoccolo. WP8 Component Model " Deliverable Progetto grid.it Work Package Ambiente di Programmazione .Gennaio 2004
[23] A. Machì, F.Collura, S. Lombardo, V. Morici Deliverable su :" GRID.it" *"Librerie scientifiche e servizi di Libreria Grid-Aware",* Progetto Grid.it Work Package Librerie Scientifiche,Dicembre 2003 RT-ICAR-PA-03-12
[24] E. Gamma, R. Helm, R. Joyhnson, J. Vlissides *"Design Patterns . Elements of Reusable Object-Oriented Software*". Addison-Wsley
[25] Marco Vanneschi: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing 28(12): 1709-1732 (2002)
[26] Marco Aldinucci, Sonia Campa, Pierpaolo Ciullo, Massimo Coppola, Silvia Magini, Paolo Pesciullesi, Laura Potiti, Roberto Ravazzolo, Massimo Torquati, Marco Vanneschi, Corrado Zoccolo: *The Implementation of ASSIST, an Environment for Parallel and Distributed Programming*. Euro-Par 2003: 712-721
[27] Marco Danelutto "*RISC Approach to Grid Programming*", Seminario Grid Forum, ICAR Palermo, Dic 2003,http://medialab.pa.icar.cnr.it/ GridForum/gridForum.html