Fachhochschule Aachen

Abteilung Jülich

Fachbereich: Physikalische Technik

Studiengang: Technomathematik

Ein Command-Line-Interface als Zugang zu Grid-Ressourcen

Diplomarbeit von Lidia Kirtchakova

Jülich, Dezember 2004

Die vorliegende Diplomarbeit wurde in Zusammenarbeit mit dem Forschungszentrum Jülich GmbH, Zentralinstitut für Angewandte Mathematik (ZAM), angefertigt.

Sie wurde betreut von:

Referent: Prof. Dr. M. Reissel

Korreferentin: M. Romberg

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Inhalt

1 Einleitung

Grid Computing, die einfache Nutzung verschiedenster verteilter Ressourcen im Netz, hat in den vergangenen Jahren sowohl im wissenschaftlichen als auch im kommerziellen Umfeld zunehmend Bedeutung erlangt. Zur Realisierung des Grid Computing wurden sogenannte Middleware Systeme entwickelt, zum Beispiel in Deutschland/Europa das UNICORE System: Uniform Interface to Computer Resources. Dieses vertikal integrierte System bietet einen komfortablen grafischen Klienten, der es dem Benutzer erlaubt, komplexe Jobabläufe zu erstellen, Jobs zu submittieren und zu überwachen.

Die vorliegende Arbeit beschreibt den Entwurf und die Entwicklung des Command Line Interfaces (CLI) für UNICORE. Das CLI wird im Rahmen des europäischen Projektes OpenMolGRID [1] eingesetzt und stellt eine Alternative zum interaktiven UNICORE Klienten dar. Es ermöglicht dem Benutzer, die UNICORE-Infrastruktur aus einer Unix-Shell oder aus einem Script heraus anzusprechen. Zwar gibt es in UNICORE die Möglichkeit, vorher mit dem Klienten erstellte und abgespeicherte Jobs von der Kommandozeile aus zu submittieren, es ist jedoch nicht möglich, einen neuen Job ohne Benutzerinteraktion zu erstellen. Dieser Umstand stellt eine starke Einschränkung UNICOREs bzgl. der Anwendung durch automatisierte Prozesse dar, für die es praktisch immer notwendig ist, diesen interaktiven Ablauf so weit wie möglich zu linearisieren. Das hier beschriebene und im Rahmen dieser Arbeit entwickelte CLI ermöglicht das automatische Erstellen von Jobs aus einer XML Workflow Beschreibung, das Übermitteln von Jobs im synchronen oder im asynchronen Modus, Statusabfragen von Aufträgen, die sich in Ausführung befinden und das Holen der Ergebnisse. Außerdem stellt es auch eine API-Schnittstelle zur Verfügung, über die es in Programme eingebunden werden kann.

1.1 Grid Gomputing

Grid Computing gilt als eine der wichtigsten Zukunftstechnologien. Die in Natur- und Ingenieurwissenschaften untersuchten Aufgabestellungen können so zeit- und datenintensive Lösungswege erfordern, dass vor Ort vorhandene Rechenkapazitäten nicht ausreichen oder der Zugriff auf große verteilte Datenbestände notwendig ist. Sehr oft werden zur Lösung dieser Probleme heterogene Ressourcen benötigt, die nicht alle auf einer Maschine zur Verfügung stehen. Dies führt zu einer steigenden Anzahl von Anwendungen, die verteilte Ressourcen erfordern, und erweckt den Bedarf, leistungsfähige, aber geografisch getrennte Ressourcen zu einer virtuellen Einheit zusammenzufügen. Die Vorstellung dabei ist, diese Möglichkeiten so einfach zu erhalten wie den Strom aus der Steckdose, daher (engl. „Power-Grid“, Stromnetz) stammt auch die Bezeichnung „Grid“-Computing.

Die Verfügbarkeit neuer Hochgeschwindigkeitsnetze hat die Hauptvoraussetzung für das Grid Computing geschaffen: Grid ermöglicht einen einheitlichen und transparenten Zugang zu Rechenressourcen mit unterschiedlichen Architekturen, Betriebssystemen, Rechen- und Speicherkapazitäten sowie Zugriffs- und Sicherheitsbestimmungen an verschiedenen geografischen Standorten. In einem Grid können verschiedene Rechenressourcen ausgewählt

und kombiniert werden, wobei sich der Benutzer nicht darum kümmern muss, welche Rechenplattform sich hinter einem Grid-Knoten verbirgt.

Ein Grid ist durch folgende Eigenschaften charakterisiert [2]:

- Lokale Autonomie: In den teilnehmenden Organisationen können unterschiedliche Verwaltungs- und Verfahrensregeln existieren, nach denen Ressourcen betrieben werden. Dies soll für den Benutzer der Grid-Software verborgen sein.
- Heterogenität der Ressourcen: die Natur der zur Verfügung stehenden Ressourcen kann sich u.a. im Bezug auf Rechnerarchitekturen, Rechenleistungen, Speicherkapazitäten, Betriebssysteme und Softwareausstattung unterscheiden;
- Skalierbarkeit: das Grid-System kann dem Bedarf angepasst werden;
- Dynamik und Adaptivität: das Grid-System soll in der Lage sein, auf Ausfälle von einzelnen Grid-Knoten oder andere Störungen flexibel zu reagieren.

Für Grid Computing finden sich in der Praxis bereits zahlreiche Anwendungsgebiete in Forschung und Industrie:

- verteilte Datenbanken in Biologie, Chemie und Pharmakologie,
- Simulationen und Datenauswertung in der theoretischen und angewandten Physik,
- Wettervorhersage,
- Prozessoptimierung in der chemischen Industrie und
- simulierte Crashtests in der Automobilindustrie.

Und die Möglichkeiten, die diese noch junge Technologie bietet, sind bei weitem noch nicht ausgeschöpft.

Ein weiteres Beispiel für eine weltumspannende Grid-Anwendung ist die Auswertung von Experimenten am CERN (Conseil Européen pour la Recherche Nucléaire, Europäisches Kernforschungslabor) [3]. Ab 2007 soll dort ein neuer Teilchenbeschleuniger, der Large Hadron Collider (LHC) in Betrieb genommen werden. Die dort geplanten Experimente werden sehr grosse Datenmengen erzeugen. Zusätzlich sind die beteiligten Wissenschaftler in der Regel nicht lokal vor Ort am CERN, sondern weltweit verteilt in ihren Instituten. Grid-Technologie, die den Rechen- und Speicheraufwand verteilt und den Datenzugriff von den unterschiedlichen Standorten aus ermöglicht, stellt einen wichtigen Ansatz zur Lösung dieser Aufgabe dar.

Derzeit befinden sich auf der ganzen Welt unterschiedliche Grid-Systeme im Einsatz oder in der Entwicklung, z.B. Globus [4], Legion [5] und UNICORE [6].

1.2 UNICORE

Diese Arbeit basiert auf UNICORE (Uniform Interface to Computing Resources), einer in Java entwickelten Grid-Infrastruktur, die den Benutzern einen sicheren, einheitlichen und intuitiven Zugang über das Netz zu verteilten Rechnerressourcen ermöglicht. Das Projekt

UNICORE wurde 1997 ins Leben gerufen und vom BMBF (Bundesministerium für Bildung und Forschung) unterstützt. Das Ziel von UNICORE und seines Nachfolge-Projektes UNICORE Plus war, eine transparente Umgebung zu schaffen, die dem Benutzer das Erstellen, Übermitteln, Ausführen und Überprüfen von komplexen Jobs sowie das Kontrollieren und Holen der Ergebnisse von geografisch verteilten, heterogenen Rechnerressourcen ermöglicht. Der Benutzer wird unabhängig von einzelnen Rechnerarchitekturen, Betriebssystemen, Speichertechniken etc.

UNICORE stellt dem Benutzer eine grafische Oberfläche, den UNICORE-Klienten, zur Verfügung, der auf dem lokalen System des Benutzers installiert wird. Der UNICORE-Klient erlaubt dem Benutzer, interaktiv multi-step und multi-site Jobs zu erstellen und zu verwalten. Ferner übernimmt er die Kommunikation mit den verfügbaren Grid-Knoten.

Es ist möglich, den UNICORE-Klienten zu erweitern, ohne die Basis-Software verändern zu müssen. Dazu steht eine Schnittstelle zur Verfügung, über die Benutzer ihre spezifische Anwendungen, Plugins genannt, einbinden können. UNICORE Plugins sind jar-Archive, die Java-Klassen beinhalten und in den UNICORE-Klienten dynamisch eingebunden werden können. Der Vorteil von Plugins besteht darin, dass die Hauptfunktionalität von UNICORE erhalten bleibt und somit auch die Sicherheits-, Authentisierungs- und Dateiverwaltungsmechanismen zur Verfügung stehen, man aber gleichzeitig sehr spezielle Anwendungen auf eine einfache Weise integrieren kann.

UNICORE wird unter anderem im Projekt OpenMolGRID eingesetzt.

1.3 OpenMolGRID

Das europäische Projekt OpenMolGRID (Open Computing Grid for Molecular Science and Engineering) [1], das über das EU-Programm Information Society Technologies (IST) gefördert wird, wurde im September 2002 begonnen. Die Projektpartner sind das Forschungszentrum Jülich, die Universität von Tartu, Estland, die Universität von Ulster, Nord-Irland, das Pharmakologische Institut „Mario Negri“, Italien, und das Pharma - Unternehmen ComGenex Inc, Ungarn. Das Projekt befasst sich mit Problemstellungen aus dem Bereich Molekulardesign wie z.B. Molekül-Entwurf und Bildung statistischer Modelle. Dabei soll eine Grid-Umgebung zur Verfügung gestellt werden, die hilft, rechen- und datenintensive Bearbeitungsschritte, wie Data Warehousing, Data Mining und Modellnutzung (Vorhersage von Eigenschaften neuer Moleküle) zu automatisieren und mit wesentlich reduziertem Zeit- und Benutzeraufwand auszuführen. Im Vordergrund steht die Abbildung der bisherigen manuellen Bearbeitungsfolge auf automatisiert ausführbare, komplexe Workflows.

Molecular Engineering und Molecular Design basieren auf Data Warehousing und Data Mining, bei denen Datenbanken nach bestimmten Eigenschaften oder Strukturen durchsucht werden. Da es in der Chemie und verwandten Bereichen eine große Anzahl öffentlicher und gewerblicher bzw. industrieller Datenbanken oder Datensammlungen gibt, die in Bestand und Formaten sehr heterogen sind, ist es sinnvoll, deren Inhalt - soweit es geht - vor der Verarbeitung zusammenzufassen und zu vereinheitlichen. Dieses geschieht im Data

Warehousing Prozess. Beim Data Warehousing werden die heterogenen Daten gesammelt (harvesting), bereinigt (cleansing) und in gemeinsame Formate transformiert.

Die in diesem Projekt entwickelte Data Warehouse Software muss vor dem Speichern der Daten zum Teil sehr komplexe Berechnungen durchführen, um die gesammelten Daten zu vereinheitlichen. Teil dieses Vorgangs ist auch, dass Ergebnisse von Berechnungen, die erwartungsgemäß von vielen Benutzern durchgeführt werden, einmal vorab berechnet und im Data Warehouse abgespeichert werden. Da diese Berechnungen nicht immer lokal beim Data Warehouse durchgeführt werden können und zum Teil auch sehr zeit- und rechenintensiv sind, sollte die Möglichkeit geschaffen werden, sie auf verteilten Ressourcen des Grids, im speziellen Fall auf über UNICORE zugreifbaren Ressourcen, ohne Benutzereingriff ausführen zu lassen. Dafür musste ein Command Line Interface (CLI) entwickelt werden, das Gegenstand dieser Arbeit ist.

Das Kapitel 2 gibt eine detaillierte Beschreibung der UNICORE Umgebung, der Funktionalität des UNICORE-Klienten und des Aufbaus eines UNICORE-Jobs, sowie des verwendeten Sicherheitsmodells und des Kommunikationsprotokolls an. Im Kapitel 3 wird auf den Kern der Arbeit, das CLI, eingegangen. Abschnitt 3.1 beschreibt die benutzer- und systembedingten Anforderungen an das CLI, Kapitel 3.2 stellt die Spezifikation des CLI dar, während 3.3 den Aufbau der wichtigsten Module aus der Implementierungssicht genauer erläutert. Das Kapitel 3.4 stellt die Queueing Komponente vor, die bereits auf dem CLI aufbaut. Kapitel 4 geht schließlich auf den aktuellen Status des CLI ein, darüber hinaus gibt es einen kurzen Ausblick auf Entwicklungs- und Einsatzmöglichkeiten des CLI.

2 Der Aufbau von UNICORE

In diesem Kapitel werden die Aspekte von UNICORE behandelt, auf denen das CLI aufbaut. Das Kapitel gibt eine Übersicht über die UNICORE Architektur, wobei nur die für diese Arbeit relevanten UNICORE Komponenten erläutert werden. Für einen tieferen Einblick sei auf [7] verwiesen.

UNICORE ist ein Client-Server System, das auf einem Dreischichtenmodell basiert:

1) Der Benutzerschicht, die aus einem in Java implementierten Klienten besteht. Der UNICORE-Klient läuft auf dem lokalen System des Benutzers und stellt eine grafische Oberfläche zur Verfügung, mit deren Hilfe Jobs erstellt und verwaltet werden können.
2) Den Servern, bestehend aus dem *Gateway* und dem *Network Job Supervisor (NJS)*. Beide sind Java-Applikationen, die den Benutzer authentifizieren, die Informationen über die auf dem Zielsystem verfügbaren Ressourcen zur Verfügung stellen und Jobs während der Ausführung verwalten.
3) Dem Zielsystem, auf dem das *Target System Interface (TSI)* läuft, ein Perl-Script, das für die Jobausführung auf dem lokalen Batch-Subsystem sorgt.

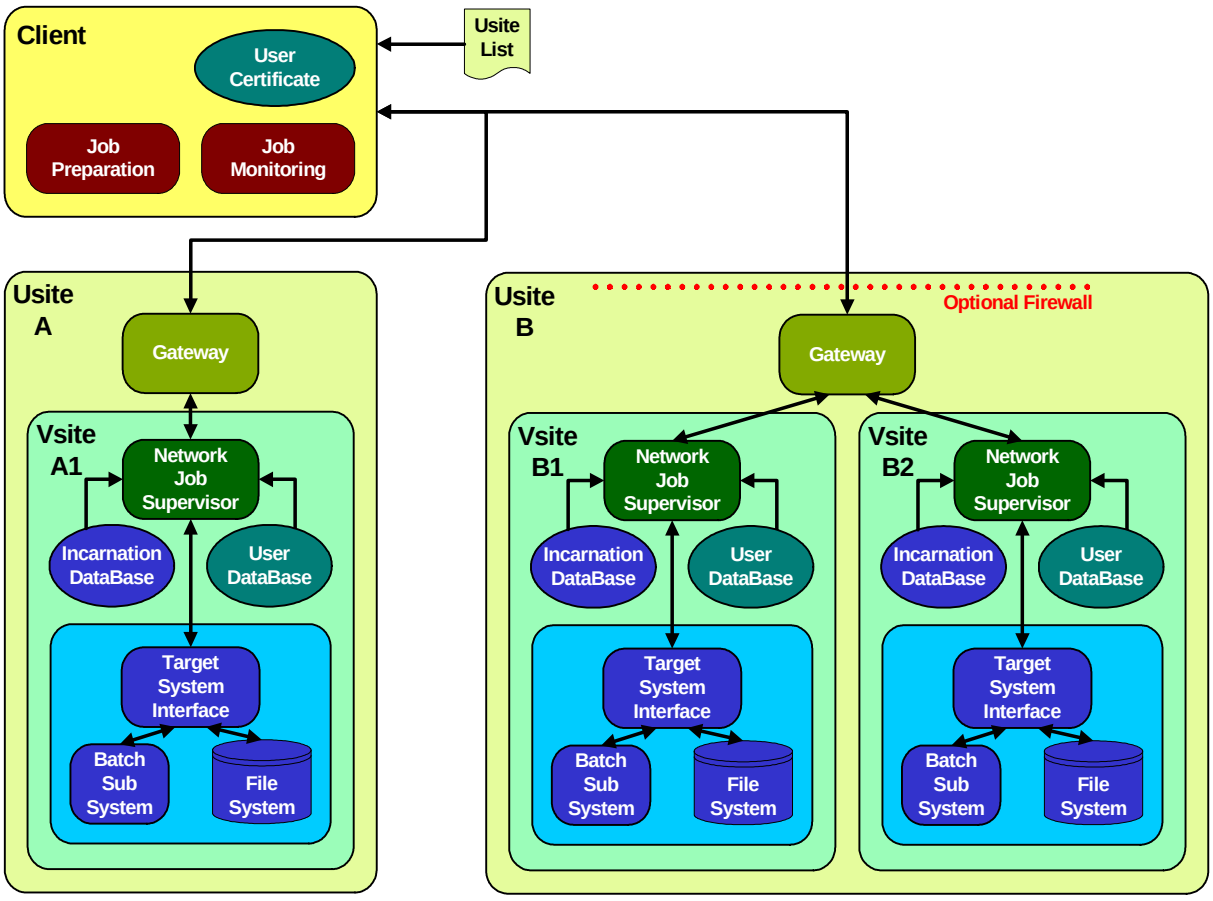

Abbildung 1: UNICORE Architektur (Autor: D.Mallmann, FZ-Jülich)

Der grafische UNICORE-Klient stellt dem Benutzer zwei Bereiche zur Verfügung, die *Job Preparation Area* (*JPA*), in der man Jobs erstellen kann, und den *Job Monitor Controller* (*JMC*), der zur Kontrolle der abgeschickten Jobs dient. *JPA* und *JMC* können parallel und unabhängig voneinander benutzt werden. Für das Jobhandling eigener Jobs im JMC spielt es keine Rolle, von wo aus der Job ursprünglich submittiert wurde. Aus den Eingaben des Benutzers erstellt der UNICORE-Klient eine abstrakte Beschreibung des Jobs, die *Abstract Job Object* (*AJO*) genannt wird. Das AJO besteht aus einer Beschreibung der angeforderten Software- und Hardwareressourcen und abstrakten Anweisungen, die, falls erforderlich, auf verschiedenen Zielsystemen ausgeführt werden können.

Der UNICORE-Klient kann Verbindungen zu unterschiedlichen Servern, die als *UNICORE Sites (Usites)* definiert sind, aufbauen. Die Liste der vorhandenen *Usites* liegt auf einem zentralen Server und wird direkt beim Programmstart vom Klienten abgefragt. Ein *Gateway* stellt den Eingang zu einer *Usite* dar. Es stellt eine Kombination aus Internet-Adresse und Portnummer zur Verfügung, über die eine SSL Verbindung zur *Usite* aufgenommen werden kann. Das Gateway authentisiert den Benutzer anhand seines Zertifikats und sendet ihm eine Liste von vorhandenen Computerressourcen (*Virtual Sites*, *Vsites*). Die Autorisierungsinformation ist in der Datenbank *UNICORE User Database* (*UUDB*) gespeichert. Die *UUDB* bildet den UNICORE Benutzernamen (*Ulogin*), der anhand des Zertifikats bestimmt wird, auf den lokalen Unix Benutzernamen (*Xlogin*) auf dem Zielsystem ab. Das UNICORE *Gateway* verbindet sich mit dem *Network Job Supervisor* (*NJS*) und fragt die zur Verfügung stehende Ressourcen auf dem ausgewählten Zielsystem ab. Diese Information ist in der *Incarnation Datenbank* (*IDB*) beim NJS gespeichert. Die *IDB* enthält auch die Regeln für die Übersetzung der abstrakten Job-Beschreibung *AJO* in konkrete Batch-Jobs auf dem Zielsystem. Der *NJS* wandelt mittels der IDB abstrakte Jobs in konkrete Anweisungen um, die dann vom *Target System Interface (TSI)* an das lokale Resource Management System bzw. Betriebssystem auf dem Zielsystem übergeben werden.

2.1 UNICORE-Klient

Der UNICORE-Klient ermöglicht dem Benutzer

- die Job-Erstellung mit Hilfe des Graphical User Interface (GUI);
- die Festlegung von Abhängigkeiten zwischen einzelnen Jobteilen (*Subjobs* und *Tasks*);
- das Importieren lokaler Dateien in das Benutzerverzeichnis auf dem Zielsystem (*File Import*) und Exportieren von Dateien auf dem Zielsystem auf die lokale Benutzermaschine (*File Export*);
- die Auswahl von *Usite* und *Vsite*, auf denen der Job ausgeführt werden soll;
- die Jobübermittlung;
- das Überprüfen des Status von submittierten Jobs;
- das Holen von Zwischenergebnissen;
- das Abbrechen von Jobs, die sich in Ausführung befinden;

- das Löschen von ausgeführten Jobs auf dem Zielsystem.

Die Funktionalität des Klienten kann über anwendungsspezifische Plugins erweitert werden. Es wird eine Schnittstelle zur Verfügung gestellt, über die diese Plugins eingebunden werden können.

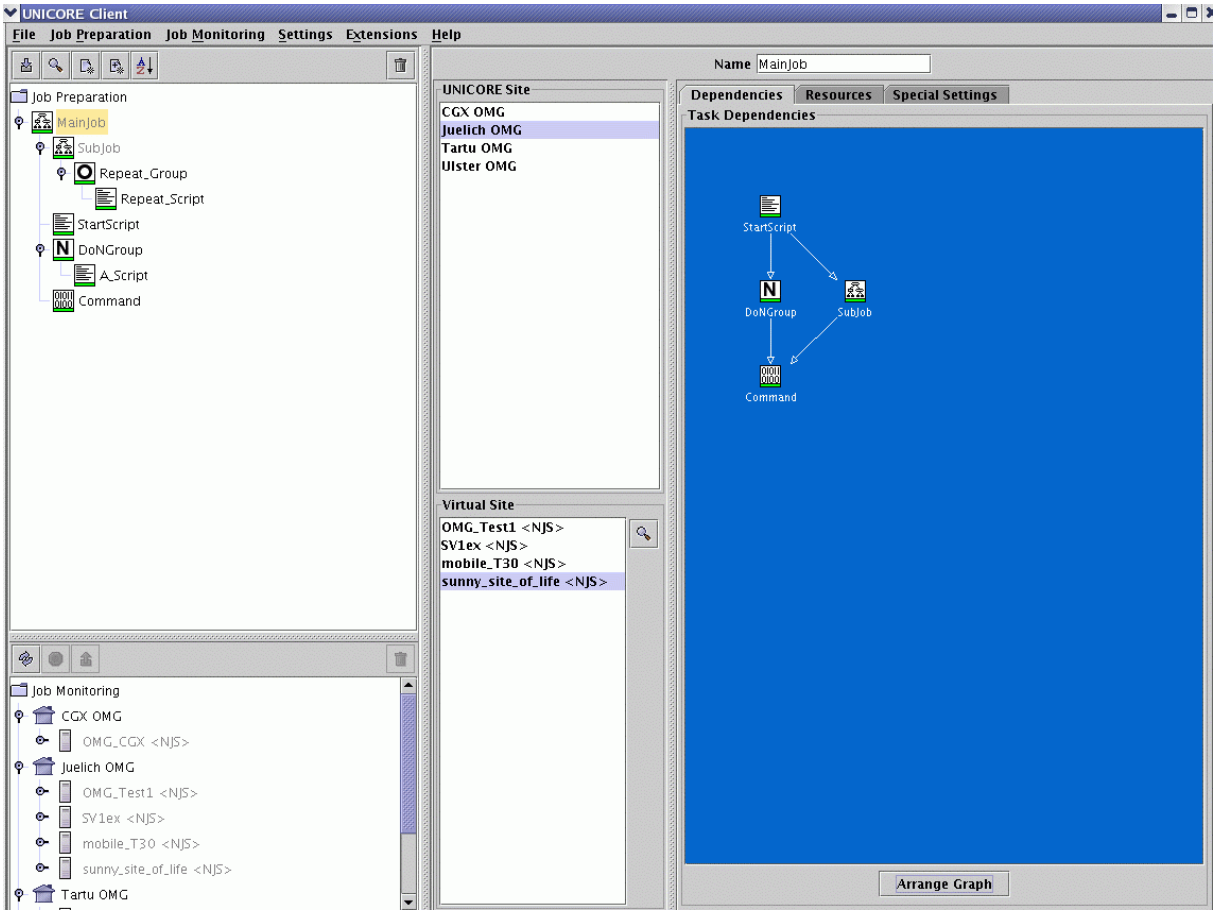

Abbildung 2: UNICORE-Klient

Abbildung 2 zeigt die *Job Preparation Area* des UNICORE-Klienten, in der man einen Job konstruiert und die Site auswählt, auf der der Job ausgeführt werden soll. Bei seiner Initialisierung verbindet sich der UNICORE-Klient mit allen zur Verfügung stehenden Usites und lädt dynamisch die Informationen über Hard- und Sofware-Ressourcen, die auf den Zielsystemen (*Vsites*) vorhanden sind. Der Benutzer kann anhand dieser Informationen sehen, welche *Vsites* die für seinen Job notwendigen Applikationen haben, und sich entscheiden, wo sein Job ausgeführt werden soll. Er kann auch für jede Task Ressourcen definieren, die jene für die Ausführung benötigt, z.B. die Anzahl der CPUs, den Speicherbedarf, die Laufzeit etc.

Ein UNICORE Job ist hierarchisch aufgebaut. Er kann aus mehreren *Tasks* und *Subjobs* bestehen, die wiederum andere Tasks und Subjobs enthalten können. Alle Tasks eines Jobs bzw. Subjobs werden auf derselben Vsite ausgeführt, die in Subjobs enthaltenen Tasks können allerdings auf anderen *Vsites* laufen als den Hauptjob. Normalerweise gibt es keine Ausführungsreihenfolge einzelner Jobteile, sie kann aber über Abhängigkeiten (*Dependencies*) definiert werden. Der Jobablauf stellt ursprünglich einen gerichteten azyklischen Graph dar, kann allerdings auch Ausführungsschleifen für einzelne Tasks bzw.

Subjobs beinhalten. Jeder Job bekommt auf dem Zielsystem ein temporäres Verzeichnis (*Uspace*) zugewiesen, das nach der Jobausführung gelöscht wird. Wenn bestimmte Dateien, die während der Laufzeit erstellt wurden, erhalten bleiben sollen, müssen sie mittels einer *File Export* Task aus dem *Uspace* in ein permanentes Verzeichnis des Benutzers kopiert werden. Wenn einzelne Subjobs Daten von Jobteilen benötigen, die auf anderen Vsites ausgeführt wurden, ist eine explizite *File Transfer* Task erforderlich.

Der UNICORE Klient stellt drei Gruppen von Tasks zur Verfügung, die Teil der UNICORE Distribution sind:

- Basisanwendungen, wie
  - eine *Command Task*, die eine Programmdatei, die sich entweder auf dem Quell- oder dem Zielsystem befindet, ausführt. Diese Datei wird zur Ausführung jeweils in den Uspace auf dem Zielsystem kopiert;
  - eine *Script Task*, die ein Shell-Script auf dem Zielsystem ausführen kann;
- Schleifenkonstrukte und Kontrollmechanismen, wie
  - *Do N*, *Do Repeat*, *If Then Else* und *Hold Job*
- und Dateiverwaltungs- und File-Transfermechanismen, wie
  - *File Import*, *File Export* und *File Transfer*

Bei der Jobübermittlung wird der konstruierte Jobbaum in ein serialisiertes Java Objekt vom Typ *Abstract Job Object* (*AJO*) umgewandelt und in dieser Form über eine SSL-Verbindung zur ausgewählten Vsite geschickt. Danach kann der Jobstatus und die Ergebnisse über den *JMC* des Klienten an dieser Vsite abgefragt werden.

2.2 UNICORE Sicherheitsmodell

Das UNICORE Sicherheitsmodell basiert auf dem Public-Key-Verfahren und benutzt X.509 Zertifikate, die von einer Authorisierungsstelle, der UNICORE *Certification Authority* (*CA*), ausgestellt werden. X.509 ist der weltweite Standard für digitale Zertifikate, und ein X.509 Zertifikat enthält den öffentlichen Schlüssel des Zertifikatsbesitzers, sowie dessen Name, dessen E-Mail Adresse, die ausstellende Beglaubigungsstelle (in diesem Fall die CA), den Gültigkeitszeitraum und das Verfahren nach dem das Zertifikat erstellt wurde. Jeder UNICORE-Klient besitzt mindestens ein Schlüsselpaar, bestehend aus einem privaten und dem dazugehörigen und von der CA signierten öffentlichen Schlüssel. Die tatsächliche Anzahl von Schlüsselpaaren hängt von der Art und Weise ab, wie die Rechtevergabe der benutzten Ressourcen geregelt ist. Alle Schlüssel werden auf der lokalen Benutzer-Maschine in dem sog. *Keystore* gespeichert. Der *Keystore* hat das PKCS12-Format (Personal Information Exchange Syntax Standard).

Das AJO, das vom Klienten an ein Gateway übermittelt wird, enthält unter anderem den öffentlichen Schlüssel und wird vor der Übermittlung mit dem privaten Schlüssel des Klienten signiert, um die Integrität der Daten zu sichern, d.h. um zu vermeiden, dass die Daten unterwegs von Unbefugten verändert werden. Bevor es auf dem Zielsystem ausgeführt

wird, werden seine Integrität und die Authentizität des Absenders sicher gestellt. Das Gateway vergleicht die Unterschrift des öffentlichen Schlüssels aus dem AJO mit der Liste der bekannten CA. Wenn sie vertrauenswürdig ist, wird das AJO an den NJS weitergereicht. Der NJS identifiziert den Benutzer mittels der *UUDB*, prüft mit dem öffentlichen Schlüssel die Signatur des gesamten AJO und gibt den Job weiter an das TSI.

Die Kommunikation zwischen dem Klienten und dem Gateway geschieht über eine SSL Verbindung. Bevor diese SSL Verbindung aufgebaut wird, findet zwischen Klient und Gateway ein Austausch der öffentlichen Schlüssel statt. Der Klient erhält den öffentlichen Schlüssel des Gateways und kann nachprüfen, ob er das Gateway für vertrauenswürdig hält. Das Gateway erhält nun den öffentlichen Schlüssel des Klienten und kann wiederum nachprüfen, ob die CA, die diesen öffentlichen Schlüssel signiert hat, bekannt ist. Ist dies der Fall, kann eine SSL Verbindung aufgebaut werden. Abbildung 3 zeigt den Kommunikationsablauf zwischen dem Klienten und dem Gateway.

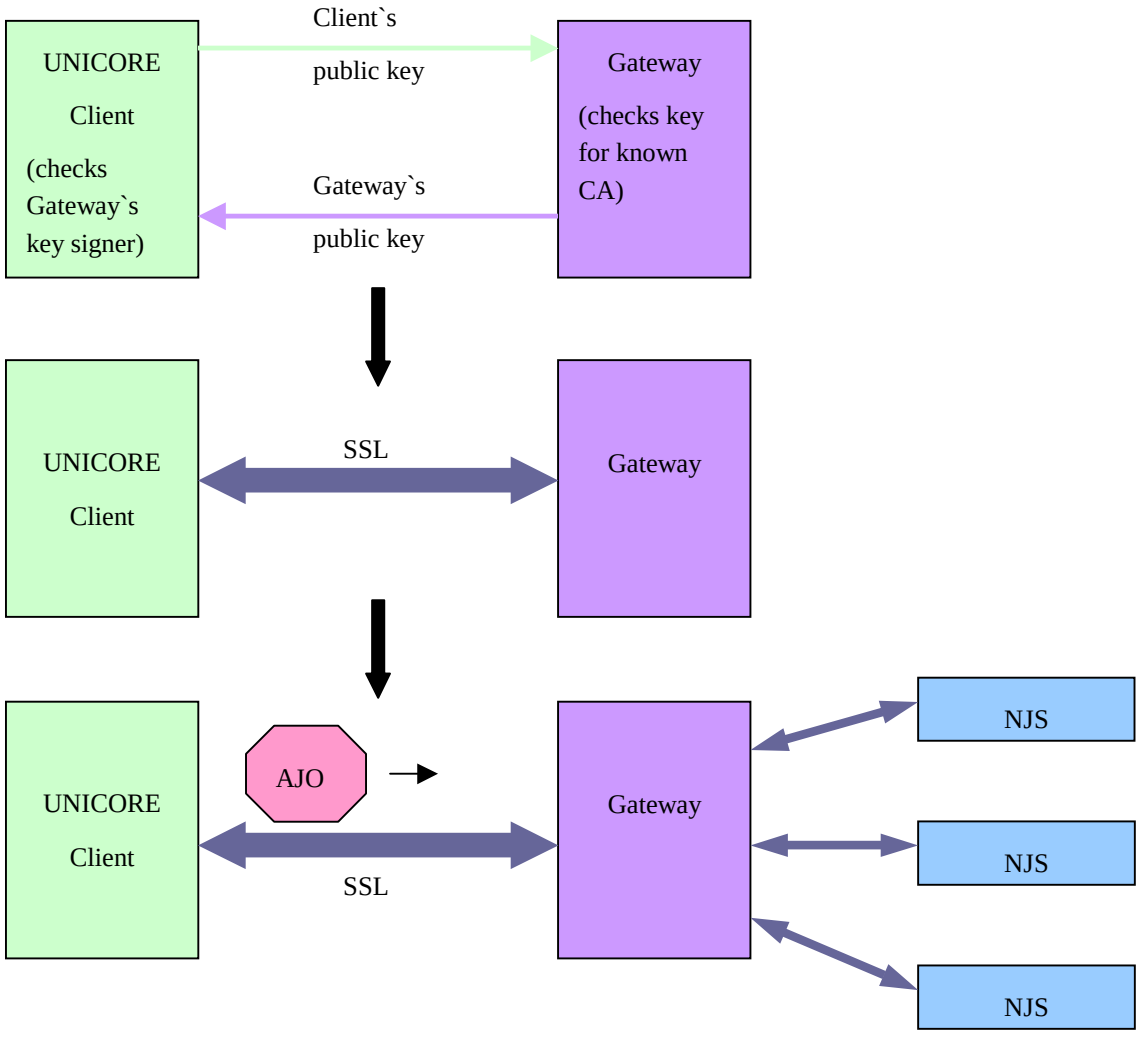

Abbildung 3: UNICORE Sicherheitsmodell

UNICORE unterscheidet zwischen dem sog. *Consigner* und dem sog. *Endorser*. Ein Endorser ist jemand, der einen Job erstellt und mit seinem privaten Schlüssel signiert. Ein Consigner ist jemand, der den Job an ein Gateway übermittelt. Häufig sind Consigner und Endorser

dieselbe Person, nämlich der Benutzer des UNICORE-Klienten. Wenn Jobteile auf unterschiedlichen Sites laufen und von einem NJS an eine andere Usite geschickt werden, sind Consigner und Endorser verschieden. In diesem Fall ist der Benutzer der Endorser und der NJS, der den gesamten Job erhalten hat, der Consigner. Das Gateway authentisiert nur den Consigner, wo hingegen der NJS immer den Endorser authentisiert und autorisiert.

2.3 Abstract Job Object (AJO)

Ein AJO beschreibt einen UNICORE Job und ist ein Java-Objekt vom Typ *AbstractJob*. Es besteht aus einer oder mehreren *ActionGroups*, die *AbstractActions* enthalten. Abbildung 4 zeigt, wie ein AJO, das ein kleines Script ausführen soll, intern aussieht. *AbstractActions* sind einfache Aktionen oder Basis-Tasks, von denen alle anderen UNICORE Aktionen wie Jobsubmission, Datentransfer und –verwaltung etc. abgeleitet sind. Ein AJO ist auch eine *AbstractAction* und kann deswegen in anderen AJOs enthalten sein. Es gibt drei Haupttypen von *AbstractActions*:

- *AbstractTasks*, die auf dem Zielssystem bestimmte Aufgaben erfüllen. Als Beispiel kann die *ExecuteScriptTask* genannt werden, die auf dem Zielsystem ein Script ausführt.
- *AbstractServices*, die andere *AbstractActions* verwalten oder Informationen über sie oder über Vsites zur Verfügung stellen. Diese Informationen können den Jobstatus, die Inhalte der Jobergebnisse oder Beschreibungen von Vsite-Ressourcen beinhalten.
- *ActionGroups*, die mehrere *AbstractActions* unter der Berücksichtigung der Ausführungsreihenfolge (*Dependencies*) zusammenfassen. Eine *AbstractAction* in einer *ActionGroup* wird nicht ausgeführt, bevor nicht all ihre Vorgänger ausgeführt wurden. Wenn die Ausführung einer *AbstractAction* fehlgeschlagen ist, werden ihre Nachfolger nicht mehr ausgeführt. Ein AJO ist immer auch eine *ActionGroup* (Superklasse von AJO).

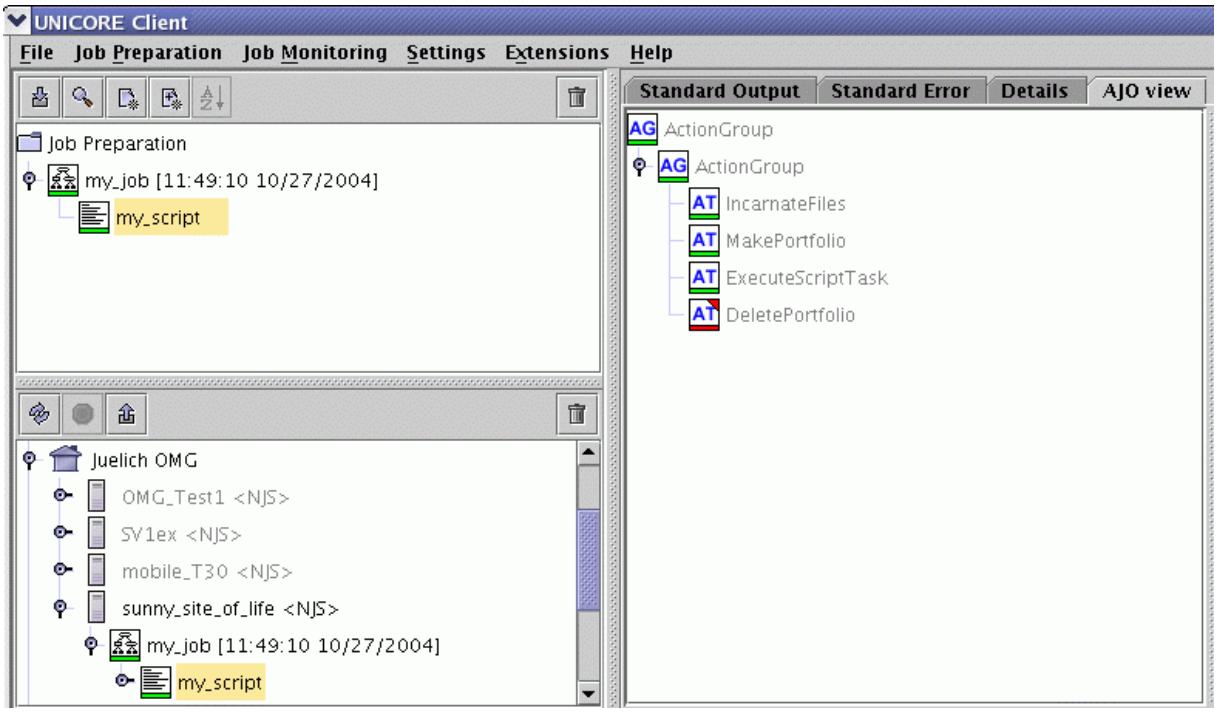

Abbildung 4: Beispiel für ein AJO

Alle *AbstractActions* bekommen bei ihrer Initialisierung einen jeweils eindeutigen *Identifier* zugewiesen, der während der Jobübermittlung und -ausführung unverändert bleibt.

Die Implementierung des AJO aus der Abbildung 4 sieht wie folgt aus:

```
// create a new abstract job object
AbstractJob ajo = new AbstractJob("my_job");

// create a content of the script task
String the_script  = "This is a small example of a script task\n";

// create a task to make a file in the Uspace
// The task IncarnateFiles is an AbstractAction
IncarnateFiles inf = new IncarnateFiles("IncarnateFiles of my_job");
inf.addFile("script",the_script.getBytes());

// create a portfolio task (which is also a subclass of an AbstractAction)
MakePortfolio mp = new MakePortfolio("MakePortfolio of my_job");
mp.addFile("script");

// create a DeletePortfolio task to delete a job after its execution
// is completed
DeletePortfolio dp = new DeletePortfolio(“DeletePortfolio for my_job”);
dp.setPortfolio(mp.getPortfolio().getId());

// create an ExecuteScriptTask to execute a script
ExecuteScriptTask scriptTask = new ExecuteScriptTask("my_script");
scriptTask.setScriptType(ScriptType.CSH);

scriptTask.setExecutable(mp.getPortfolio());

// define dependencies between AbstractActions
ajo.addDependency(inf, mp);
ajo.addDependency(mp, scriptTask);
ajo.addDependency(scriptTask, dp);
```

2.4 UNICORE Protocol Layer (UPL)

Das UPL regelt die Datenübertragung zwischen dem UNICORE-Klienten und dem Server und baut auf einer SSL Verbindung auf. Es besteht aus *Requests*, die vom Klienten abgeschickt werden und *Replies*, die als Server-Antwort zurückkommen. Das UPL definiert jeweils vier *Requests* und dazugehörende *Replies*:

- `ConsignJob` Request für die Ausführung eines AJOs
- `RetrieveOutcome` Request für Statusabfrage und Holen der Ergebnisse des submittierten Jobs
- `RetrieveOutcomeAck` Request für die Bestätigung der angekommenen Ergebnisse
- `ListVsites` Request, das zum Gateway geschickt wird, um die Liste der dahinterstehenden Vsites abzufragen
- `ConsignJobReply` Bestätigung des `ConsignJob` Resquests
- `RetrieveOutcomeReply` Antwort auf das `RetrieveOutcome` Request
- `RetrieveOutcomeAckReply` Bestätigung des `RetrieveOutcomeAck` Requests

- `ListVsitesReply` Antwort auf das `ListVsites` Request mit der Liste der Vsites, die über das Gateway angesprochen werden können

Das UPL wird auch vom Command Line Interface benutzt, um die Verbindung zum Server aufzubauen, Ressourcen abzufragen und Jobs und Daten zu transferieren.

3 UNICORE Command Line Interface (CLI)

Wie im Kapitel 1.3 bereits erwähnt, ist das Command Line Interface (CLI) im Rahmen des OpenMolGRID-Projektes entstanden. Die verschiedenen Entwicklungsphasen [8] des CLI wurden teilweise von bestehenden Gegebenheiten des OpenMolGRID-Projektes beeinflusst. So wurden bei der Planung gemeinsam mit den Projektbeteiligten die Hauptanforderungen aus Sicht des Data Warehouse festgelegt, die in der Definitionsphase in einem Spezifikationsdokument für das OpenMolGRID-Projekt zusammengefasst wurden. Beim Entwurf wurden bereits existierende Komponenten und Bibliotheken berücksichtigt, die entweder vom CLI verwendet werden konnten oder zu denen Schnittstellen festgelegt werden mussten. Weitere Schnittstellen wurden in der Entwurfsphase definiert, um das Einbinden des CLI in die unterschiedlichen Benutzerumgebungen (siehe 3.1.1) zu ermöglichen. Die Implementierung wurde dann im Bottom-Up Verfahren durchgeführt, bei dem sukzessiv einzelne Module entwickelt, getestet und später zu größeren Systemen zusammengefügt werden. In der Einführungsphase sind das CLI-Distributionspaket und die Benutzer- und Software-Dokumentation entstanden, und das CLI wurde auf dem System des Data Warehouse installiert und getestet. Mittlerweile wird es dort aktiv benutzt und befindet sich in der fortgeschrittenen Wartungsphase.

3.1 Anforderungen an die Funktionalität des CLI

Das CLI muss bestimmte, durch benutzer- und/oder systemseitige Vorgaben bedingte, Anforderungen erfüllen. Dieses Kapitel beschreibt die Anforderungen aus der Benutzersicht und die Randbedingungen, die sich durch das gesamte System ergeben haben.

3.1.1 Benutzersicht

Aus der Analyse der Funktionen, die der „Benutzer“ Data Warehouse [9] benötigt, haben sich folgende Anforderungen an das CLI ergeben:

- Der Zugriff auf die UNICORE-Infrastruktur soll aus der Kommandozeile oder einem Script heraus erfolgen können.
- Für die Einbindung in Programme soll eine API-Schnittstelle zur Verfügung stehen.
- Außerdem ist es notwendig, die Sicherheitsanforderungen von UNICORE zu erfüllen. Das CLI soll hauptsächlich von Programmen oder Prozessen benutzt werden. Da aber jeder UNICORE Benutzer ein gültiges X.509 Zertifikat braucht (siehe 2.2), wird vorausgesetzt, dass jedes Programm bzw. jeder Prozess, von dem aus das CLI benutzt wird, ein von einer CA ausgestelltes Zertifikat und den öffentlichen Schlüssel des Gateway-Signer-Zertifikats besitzt. Das CLI muss Zugang zu diesem Zertifikat und dem dazugehörigen Schlüssel haben.

Das Data Warehouse erwartet, dass das CLI folgendes leistet:

- Erstellung eines AJOs aus einer Jobbeschreibung;

- Übermitteln dieses AJOs zu einer Vsite über eine SSL-Verbindung mit dem Gateway. Die Übermittlung soll sowohl im synchronen als auch im asynchronen Modus (siehe 3.2.1.2) möglich sein;
- Statusabfrage;
- Holen der Ergebnisse teilweise oder vollständig bearbeiteter Jobs;
- Abbrechen eines Jobs, der sich in der Ausführung befindet.

3.1.2 Systemsicht

Da das CLI ein Teil des Projektes ist und sich als eine der Softwarekomponenten in das gesamte System einfügen muss, muss es mit allen bereits vorhandenen oder sich in der Entwicklung befindenden Komponenten kompatibel sein. Daraus ergeben sich folgende Systemanforderungen und Randbedingungen:

- Um die vorhandenen Bibliotheken nutzen zu können und die Software einheitlich zu halten, muss das CLI in Java implementiert werden.
- Da UNICORE den Job als Abstract Job Object (AJO) (siehe 2.3) definiert, das vor der Übermittlung aus den Benutzereingaben im graphischen Klienten generiert wird, muss das CLI imstande sein, einen solchen Job aus einer Jobbeschreibung ohne Benutzereingriff zu erzeugen.
- Das Format für die Jobbeschreibung muss so gewählt werden, dass es sich für die spezifischen OpenMolGRID Anwendungen eignet, gleichzeitig aber erweiterbar bleibt, da das CLI auch außerhalb dieses Projektes nutzbar sein soll. Weil das CLI ohne Benutzereingriff auskommen soll, muss die Jobbeschreibung alle für die Joberstellung und Ausführung notwendigen Informationen zur Verfügung stellen. Um weiter verwendbar zu bleiben, soll sie zukunftssicher und mit gängigen, bereits vorhanden Mechanismen auswertbar sein. Aus diesen Gründen wurde Extensible Markup Language (XML) als Beschreibungsformat gewählt.
- Für das OpenMolGRID-Projekt wurde bereits eine Erweiterung des UNICORE-Klienten, das MetaPlugin, entwickelt, um die Bearbeitungsabfolge (Workflow) z.B. bei der Erzeugung von statistischen Modellen zu automatisieren. Dabei wird ein Job aus einer XML-kodierten Jobbeschreibung erstellt. Da der Ansatz und die Vorgehensweise des MetaPlugins teilweise den Anforderungen an die Jobgenerierung über das CLI entsprechen, war es nahe liegend, Teile davon bei der Joberstellung durch das CLI zu benutzen.
- Folgende Randbedingungen sind zu berücksichtigen:
  - Zur Verwaltung der Benutzereingaben und Erzeugung des AJO benutzt der UNICORE-Klient für seine Standardtasks unterschiedliche *ActionContainer*, die *ActionGroups* umschließen. Die *ActionContainer* enthalten nicht nur die *ActionGroups*, sondern auch verschiedene GUI-spezifische Informationen wie den Pfad zum Icon, der die zugehörige Task im grafischen Menu darstellen soll, die

Position einer Task im Abhängigkeitsgraph etc. Auch bei der Plugin-Entwicklung muss man den zugehörigen Container implementieren, damit das Plugin in den UNICORE-Klienten integriert werden kann. Die Information, die in Containern enthalten ist, ist für das CLI eigentlich überflüssig, weil es ausschließlich das AJO, also einen Teil eines *ActionContainers*, für die Jobsubmission und Jobüberwachung benötigt. Das CLI muss aber imstande sein, mit den Container-Klassen umzugehen, weil das MetaPlugin sie benutzt, um auf die jeweilige ActionGroup einer Task zuzugreifen.

  - Das CLI muss alle Komponenten zur Verfügung stellen, die vom MetaPlugin zur Laufzeit benötigt werden könnten.
  - Das CLI muss Schnittstellen haben, über die es das MetaPlugin einbinden kann.

- Weil die Jobsubmission, -überwachung sowie das Holen der Ergebnisse in UNICORE auf dem UPL basiert, ist es erforderlich, dass das CLI auch das UPL beherrscht.

Teile dieser Anforderungen werden durch bereits existierende Komponenten abgedeckt, andere lassen sich, stets unter Berücksichtigung dieser vorhandenen Komponenten, nur mit neu zu erstellenden Modulen erfüllen. Es gilt also, geeignete Schnittstellen zu schaffen, um die neuen Komponenten in das vorhandene System zu integrieren und das auf diese Weise vergrößerte System in einer für den Benutzer sinnvollen Weise nutzbar zu machen.

3.2 Spezifikation des CLI

Aus den Benutzeranforderungen heraus wurde die Spezifikation des CLI entwickelt. In ihr wurden die folgenden Kommandos und Parameter definiert, die beim Aufruf an das CLI übergeben werden:

- `build_ajo` erzeugt ein AJO aus einer Jobbeschreibung. Die Jobbeschreibung (*workflow*) ist XML-codiert, wobei das Format später (siehe 3.3.5.2) im Detail erläutert wird. Das Ergebnis des Befehls ist ein serialisiertes AJO, das in einer Datei gespeichert wird.
- `submit` übermittelt ein vorher erstelltes AJO zu einer Vsite. Es gibt zwei Submissionsmodi: synchron und asynchron. Beim asynchronen Modus schickt das CLI den Job ab und wird beendet, ohne auf das Ende der Jobausführung zu warten. Beim synchronen Modus wird das CLI erst dann beendet, wenn der Job abgearbeitet und die Ergebnisse auf das lokale System geholt sind.
- `get_status` fragt den Status eines sich in der Ausführung befindenden Jobs ab.
- `get_outcome` holt Ergebnisse des teilweise oder vollständig ausgeführten Jobs von der Vsite ab und löscht den Job aus dem NJS.

- `abort_job` bricht die Jobausführung ab und löscht den Job serverseitig aus dem NJS.

Alle diese Befehlsparameter erfordern zusätzliche Eingabeargumente, die in 3.2.1 näher spezifiziert werden.

3.2.1 Beschreibung der CLI Befehle

3.2.1.1 AJO Erstellung (`build_ajo`)

Überblick:

`build_ajo` – erstellt ein Abstract Job Object aus einer XML Workflow-Beschreibung und speichert es in einer Datei

Syntax:

```
cli build_ajo –in <xml_file> -out <ajo_file> -keystore <keystore>
 [-password <password> | -pwfile <passwordfile>]
```

Parameterbeschreibung:

Parameter	Beschreibung
`<xml_file>`	XML Workflow Datei
`<ajo_file>`	Outputdatei, in der das serialisierte AJO gespeichert wird
`<keystore>`	Vollqualifizierter Name einer Datei, in der die privaten und öffentlichen Schlüssel des Benutzers gespeichert sind
`<password>`	Passwort zum Öffnen des Keystores
`<passwordfile>`	Vollqualifizierter Name einer Datei, die das Passwort zum Öffnen des Keystores enthält

3.2.1.2 Jobsubmission (`submit`)

Überblick:

`submit` – übermittelt ein AJO an eine Vsite. Die Vsite, auf der der Job laufen soll, ist im AJO enthalten.

Syntax:

```
cli submit –in <ajo> -dir <outcome_dir> -keystore <keystore> [-password
<password> | -pwfile <passwordfile>]
```

```
cli submit –in <ajo> -mode async –out <AJO_ID_FILE> -keystore <keystore> [-
password <password> | -pwfile <passwordfile>]
```

Parameterbeschreibung:

Parameter	Beschreibung
`<ajo>`	Vollqualifizierter Name einer Datei, in der das serialisierte AJO gespeichert ist
`<outcome_dir>`	Verzeichnis, in dem die Jobergebnisse gespeichert werden sollen
`<keystore>`	Vollqualifizierter Name einer Datei, in der die privaten und öffentlichen Schlüssel des Benutzers gespeichert sind
`<password>`	Passwort zum Öffnen des Keystores
`<passwordfile>`	Vollqualifizierter Name einer Datei, die das Passwort zum Öffnen des Keystores enthält
`<mode>`	Submissionsmodus (entweder synchron oder asynchron). Standardeinstellung: synchron
`<AJO_ID_FILE>`	Vollqualifizierter Name einer Datei, in der Informationen über das submittierte AJO gespeichert werden. Je nach Submissionsmodus gibt es unterschiedliche Kommandoergebnisse: Im synchronen Modus erhält man als Ergebnis die Output Dateien des ausgeführten Jobs, die im spezifizierten Verzeichnis gespeichert werden. In diesem Fall wird kein AJO_ID_FILE erstellt. Im asynchronen Modus wird der Job zu einer Vsite abgeschickt und seine Daten (siehe Kapitel 3.2.1.6) werden in der Datei AJO_ID_FILE gespeichert, damit bei den darauffolgenden Befehlen der Job eindeutig identifiziert werden kann. Das genaue Format dieser Datei wird im Kapitel 3.2.1.6 beschrieben.

3.2.1.3 Statusabfrage (`get_status`)

Überblick:

`get_status` – fragt den Status eines Jobs ab, der sich in Ausführung befindet.

Syntax:

```
cli get_status [-out <status file>] –in <AJO_ID_FILE> -keystore <keystore>
[-password <password> | -pwfile <passwordfile>]
```

Parameterbeschreibung:

Parameter	Beschreibung
`<AJO_ID_FILE>`	Vollqualifizierter Name einer Datei, in der die Daten des submittierten Jobs gespeichert sind
`<keystore>`	Vollqualifizierter Name einer Datei, in der die privaten und öffentlichen Schlüssel des Benutzers gespeichert sind
`<password>`	Passwort zum Öffnen des Keystores
`<passwordfile>`	Vollqualifizierter Name einer Datei, die das Passwort zum Öffnen des Keystores enthält
`<status file>`	Vollqualifizierter Name einer Datei, in der der Status gespeichert werden soll. Default: Standardausgabe (stdout)

3.2.1.4 Holen der Ergebnisse (`get_outcome`)

Überblick:

`get_outcome` – holt Jobergebnisse von der Vsite, nach dem der Job ausgeführt wurde, und löscht den Job auf der Serverseite. Falls der Job noch nicht fertig ist, werden evtl. vorhandene Teilergebnisse geholt und das CLI beendet. `get_outcome` kann dann bis zur Fertigstellung des Jobs beliebig oft wiederholt werden.

Syntax:

```
cli get_outcome –in <AJO_ID_FILE> -dir <outcome_dir> -keystore <keystore>
[-password <password> | -pwfile <passwordfile>]
```

Parameterbeschreibung:

Parameter	Beschreibung
`<AJO_ID_FILE>`	Vollqualifizierter Name einer Datei, in der die Daten des submittierten Jobs gespeichert sind
`<outcome_dir>`	Verzeichnis, in dem die Jobergebnisse gespeichert werden sollen
`<keystore>`	Vollqualifizierter Name einer Datei, in der die privaten und öffentlichen Schlüssel des Benutzers gespeichert sind
`<password>`	Passwort zum Öffnen des Keystores
`<passwordfile>`	Vollqualifizierter Name einer Datei, die das Passwort zum Öffnen des Keystores enthält

3.2.1.5 Abbrechen eines Jobs (`abort_job`)

Überblick:

`abort_job` – bricht die Ausführung des Jobs ab und löscht ihn auf der Serverseite

Syntax:

```
cli abort_job –in <AJO_ID_FILE> -keystore <keystore> [-password <password>
| -pwfile <passwordfile>]
```

Parameterbeschreibung:

Parameter	Beschreibung
`<AJO_ID_FILE>`	Vollqualifizierter Name einer Datei, in der die Daten des submittierten Jobs gespeichert sind
`<keystore>`	Vollqualifizierter Name einer Datei, in der die privaten und öffentlichen Schlüssel des Benutzers gespeichert sind
`<password>`	Passwort zum Öffnen des Keystores
`<passwordfile>`	Vollqualifizierter Name einer Datei, die das Passwort zum Öffnen des Keystores enthält

3.2.1.6 Spezifikation der Datei AJO_ID_FILE

Die Datei AJO_ID_FILE enthält alle wichtigen Jobinformationen. Sie wird erzeugt, nachdem ein Job asynchron submittiert wurde, und wird bei weiteren Statusabfragen bzw. beim Holen der Ergebnisse oder Abbrechen des Jobs benutzt, um ihn zu lokalisieren und auf der Vsite eindeutig zu identifizieren. Sie wird auch verwendet, um über Dateien buchzuführen, die für die Jobausführung vom lokalen System auf das Zielsystem transferiert werden müssen (*File Import* vom Zielsystem) oder die nach dem Ausführungsende zurück auf die lokale Maschine geholt werden sollen (*File Export*). Dieses Kapitel gibt eine Übersicht über die Informationen, die in der Datei AJO_ID_FILE gespeichert werden und das Format der Datei.

Die Datei soll im Klartext geschrieben werden, um dem Benutzer zu ermöglichen, die Verteilung seiner Jobs auf verschiedene Systeme nachzuvollziehen. Gleichzeitig soll sie aber auch leicht durch Programme auswertbar sein. Um beide Anforderungen zu erfüllen, wurde das Java Properties File Format gewählt. Es basiert auf Wertepaaren, bestehend aus einem *key* (Schlüssel) und dem dazugehörenden *value* (Wert), wobei *key* und *value* jeweils Zeichenketten sind.

Die folgende Tabelle gibt einen Überblick über die enthaltenen *keys* und *values*:

key	***Beschreibung***	***Value***
job_name	Name des abgeschickten Jobs	Zeichenkette
date	Datum und Uhrzeit, wann der Job übermittelt wurde	Zeichenkette
ajo_id	Serialisiertes Java Objekt vom Typ `AJOIdentifier`. Der AJO identifier ist eine eindeutige Kennung für das AJO.	Base64-kodiertes Byte Array, dargestellt als eine Zeichenkette.
vsite	Name der Vsite, zu der das AJO übermittelt wurde	Zeichenkette
gateway	Gateway URL (Adresse der Usite)	ssl://hostname:port
exportPortfolioIDs	serialisiertes Java Objekt vom Typ Vector von Elementen vom Typ PortfolioIdentifier. Enthält Informationen über *File Exports* (siehe 3.3.5).	Base64-kodiertes Byte Array, dargestellt als eine Zeichenkette.

Tabelle 1: Überblick über *keys* und dazugehörende *values* im AJO_ID_FILE

Das folgende Beispiel zeigt den Inhalt einer AJO_ID_FILE Datei:

```
# AJO_ID_FILE contains information about a respective abstract job

# Job name

job_name=JobExample

# Submission date

date=Wed Sep 29 15:14:22 CEST 2004

# AJO Id

ajo_id=rO0ABXNyABlvcmcudW5pY29yZS5BSk9JZGVudGlmaWVycbY9KfGvddUCAAB4cgAYb3JnLnVuaWNv
cmUuQUFJZGVudGlmaWVyG6OkYntcg94CAAB4cgAWb3JnLnVuaWNvcmUuSWRlbnRpZmllckKrVnMwdhyAAgA
CSQAFdmFsdWVMAARuYW1ldAASTGphdmEvbGFuZy9TdHJpbmc7eHIAE29yZy51bmljb3JlLlVuaWNvcmUIMR
+MOlUKqAIAAHhwM4IEHnQAE05ldyBBY3Rpb25Db250YWluZXI=

# Vsite

vsite=sunny_site_of_life

# Gateway

gateway=ssl://zam031.zam.kfa-juelich.de:4004

# Export portfolio identifiers:

#Information about outcome files:

exportPortfolioIDs=rO0ABXNyABBqYXZhLnV0aWwuVmVjdG9y2Zd9W4A7rwECAANJABFjYXBhY2l0eUlu
Y3JlbWVudEkADGVsZW1lbnRDb3VudFsAC2VsZW1lbnREYXRhdAATW0xqYXZhL2xhbmcvT2JqZWN0O3hwAAA
AAAAAAAB1cgATW0xqYXZhLmxhbmcuT2JqZWN0O5DOWJ8QcylsAgAAeHAAAAAKcHBwcHBwcHBwcA==
```

3.3 Implementierung des CLI

Dieses Kapitel beschreibt die Realisierung der Systemanforderungen an das CLI und stellt den Aufbau, die Klassenhierarchie, die Bibliotheken und die zentralen Module im Detail vor.

Wie bereits erwähnt, lassen sich einige Anforderungen an das CLI mit vorhandenem Code aus verschiedenen Plugins und Bibliotheken abdecken. Als Beispiele seien hier das MetaPlugin (siehe 3.3.1.5) für das grundlegende Workflow-Handling und die Arcon Library (vergleiche 3.3.1.2) für die Serverkommunikation genannt. Der Rückgriff auf diese Routinen und die Einbindung in ein sonst interaktiv/grafisch aufgebautes System erfordert aber auch die Implementierung von Schnittstellen, die für ein reines Command Line Interface nicht notwendig wären. Diese Aufgabe macht einen Teil der vorliegenden Arbeit aus, es galt jedoch hauptsächlich, die im folgenden beschriebenen, grundlegenden Probleme bei der Erstellung von Jobs und im Resource-Management durch komplett neu erstellte Module zu lösen.

Die Hauptschwierigkeit bei der Implementierung des CLI lag in der Joberstellung aus einer XML-Workflow-Beschreibung. Dies gründet sich im wesentlichen darauf, dass das Beschreibungsformat ein sehr hohes Abstraktionsniveau aufweist, um die größt mögliche Flexibilität bzgl. unterschiedlicher Grid-Systeme zu gewährleisten. Ein Workflow enthält nur abstrakte Informationen über die mit gegebenen Daten durchzuführenden Tasks, jedoch keine Informationen über Systeme und Applikationen, mit denen dies möglich ist (siehe 3.3.5.2). Im Unterschied zum grafischen Klienten, in dem der Benutzer interaktiv die Ressourcen für seine Tasks aussucht und setzt, ist es die Aufgabe des CLI, dynamisch die zur Ausführungszeit vorhandenen Systeme zu finden, die imstande sind, die angeforderten Tasks auszuführen, und deren Abarbeitung dort zu veranlassen. Für den Benutzer muss dieser Vorgang vollkommen transparent sein.

Dies bedeutet, dass eine Resource-Management-Komponente erstellt werden musste, die für das Abfragen, Holen, Zuordnen und Speichern von lokalen und Server-Ressourcen sorgt. Das CLI muss mit Hilfe dieser Komponente imstande sein, die Usites und Vsites zu ermitteln, die zur Ausführungszeit zur Verfügung stehen, und dies bei der Jobübermittlung zu berücksichtigen. Außerdem muss es Informationen über die auf der lokalen Maschine vorhandenen Tasks und Plugins haben und wissen, wie die dazugehörigen Serverapplikationen heißen. Für die Joberstellung ist es notwendig, dass das CLI alle Hard- und Softwareressourcen auf den verfügbaren Vsites kennt und Serverkomponenten den Client-Komponenten zuordnen kann, um das Zielsystem für die Jobausführung richtig zu wählen.

Die Umsetzung eines XML-Workflows in ein AJO erfordert unter anderem viele Komponenten, die zur Infrastruktur des UNICORE-Klienten gehören und teilweise nur zur Laufzeit des Klienten zur Verfügung stehen. Um sie trotzdem verwendbar zu machen, mussten Wrapper-Klassen implementiert werden, die nach außen für die aufrufenden Instanzen als Standard-UNICORE Klassen agieren, intern aber bestimmte Aufrufe an die CLI-spezifische Umgebung weiterleiten.

Außerdem wurde eine CLI-spezifische Bibliothek zusammengestellt, die alle für die Joberstellung notwendigen UNICORE-Standardklassen und Wrapper-Klassen enthält.

Zu einem AJO gehören sehr oft Input- (*File Imports*) und Outputdateien (*File Exports*), die zwischen dem lokalen System und dem Zielsystem transferiert werden müssen. Die Weitergabe der entsprechenden Informationen wurde bisher ausschließlich in einem dem graphischen Klienten vorbehaltenen Verfahren erledigt. Daher bietet die Klasse *AbstractJob*, die das AJO darstellt, keine Möglichkeit, die Informationen über diese Dateien im AJO zu speichern. Das bedeutet, dass sie bei der Jobsubmission nicht mehr zur Verfügung stehen, und der Job u.U. nicht ausgeführt werden kann. Deswegen musste bei der Implementierung des CLI ein Mechanismus entwickelt werden, der die langfristige Speicherung dieser Informationen ermöglicht. Dies wurde mit Hilfe einer zusätzlichen Klasse *JobAttachment* gelöst, deren Instanz alle notwendigen Pfade der lokalen Inputdateien und Namen für die erforderlichen Outputdateien speichert, zu einem Byte-Array transformiert und dann in einem Feld des AJO abgespeichert wird.

Auch die Möglichkeit, über an den Server asynchron übermittelte Jobs Buch zu führen, wurde neu entwickelt.

Das CLI besteht aus mehreren Java-Paketen, die unterschiedliche Funktionen haben. Das Hauptpaket enthält Klassen, die die in 3.2.1 beschriebenen Kommandos implementieren, und erfüllt die wesentlichen Anforderungen an das CLI. Die Unterpakete werden benutzt, um Ressourcen zu verwalten, und stellen Schnittstellen zu bereits existierenden Komponenten zur Verfügung.

Das CLI verwendet z.T. schon vorhandene oder im Rahmen des Projektes entwickelte Bibliotheken. Die Kommunikation mit dem Server geschieht über das UPL (siehe 2.4), das bereits in der vorhandenen Bibliothek Arcon Library implementiert wurde. Die Komponenten, die zur Jobgenerierung notwendig sind, werden teilweise von einem im Rahmen des OpenMoldGRID-Projektes erstellten Plugin, dem MetaPlugin, vererbt.

Das folgende Unterkapitel stellt vorhandene bzw. im Rahmen der Arbeit entwickelte Bibliotheken und ihre Funktionen vor.

3.3.1 Verwendete Bibliotheken

Bibliothek	Beschreibung
unicore_cli.jar	enthält das Command Line Interface.
cli_lib.jar	enthält alle relevanten Klassen aus der Standard-UNICORE-Bibliothek *client.jar*, die von Jobs benutzt werden können, und zwei Wrapper-Klassen für das Resource Management. Diese Bibliothek wurde speziell für das CLI im Rahmen der hier beschrieben Diplomarbeit zusammengestellt.
arconlib.jar	stellt Mechanismen zur Verfügung, um die Verbindung mit dem Server aufzunehmen, Jobs zu übermitteln und Ergebnisse zu holen. Es implementiert das UNICORE Protocol Layer (UPL).
openmolgrid.jar	enthält Klassen und Schnittstellen, die innerhalb des OpenMolGRID-Projektes entwickelt wurden.
ajo.jar	enthält Pakete, die für die Jobgenerierung wichtig sind.
xercesImpl.jar	stellt einen Java-Parser zur Verfügung, um Daten im XML Format lesen zu können.
MetaPlugin.jar	ein Plugin, das im Rahmen des OpenMolGRID-Projektes entwickelt wurde und zur Jobgenerierung benutzt wird.
ServiceRegistry.jar	ein weiteres Plugin, das im Rahmen des OpenMolGRID-Projektes entstanden ist und teilweise für die Verwaltung von lokalen Ressourcen benötigt wird.

Diese Bibliotheken sind in Java implementiert und stellen die Basis für das CLI dar. Im folgenden werden einzelne von ihnen näher erläutert.

3.3.1.1 unicore_cli.jar

unicore_cli.jar ist der Kern der vorliegenden Arbeit und wird im Detail in Abschnitten 3.3.2 - 3.3.8 beschrieben.

3.3.1.2 arconlib.jar

Die Bibliothek arconlib.jar enthält die Arcon Library, eine low-level Implementierung des UPL (siehe 2.4). Sie stellt ein einfaches Interface zu UNICORE-Servern zur Verfügung, das grundlegende Funktionen wie Server- Kommunikation und Jobverwaltung bietet.

3.3.1.3 cli_lib.jar

Die Wrapper-Klassen, die für das CLI entwickelt wurden und im *cli_lib.jar* enthalten sind, werden im Kapitel 3.3.6 genauer erläutert.

3.3.1.4 openmolgrid.jar

Die Bibliothek openmolgrid.jar umfasst alle globalen Schnittstellen und abstrakten Klassen, die im Rahmen des OpenMolGRID-Projektes entstanden und für dessen Anwendungen relevant sind.

3.3.1.5 MetaPlugin.jar

Für die Joberstellung benutzt das CLI Teile des MetaPlugins. Daher wird die Funktion des MetaPlugins hier genauer beschrieben:

Am Anfang wird ein leerer UNICORE Job angelegt. Das MetaPlugin liest einen in XML kodierten Workflow und baut daraus den Jobinhalt auf. Dazu gehört auch, dass die für die einzelne Tasks erforderlichen Ressourcen gesucht und zugewiesen werden. Der aus einem Workflow erstellte UNICORE Job wird in der *Job Preparation Area* des UNICORE-Klienten automatisch dargestellt. Er kann beliebig komplexe Subjobs und Tasks enthalten. Abbildung 5 zeigt die Funktionalität des MetaPlugins.

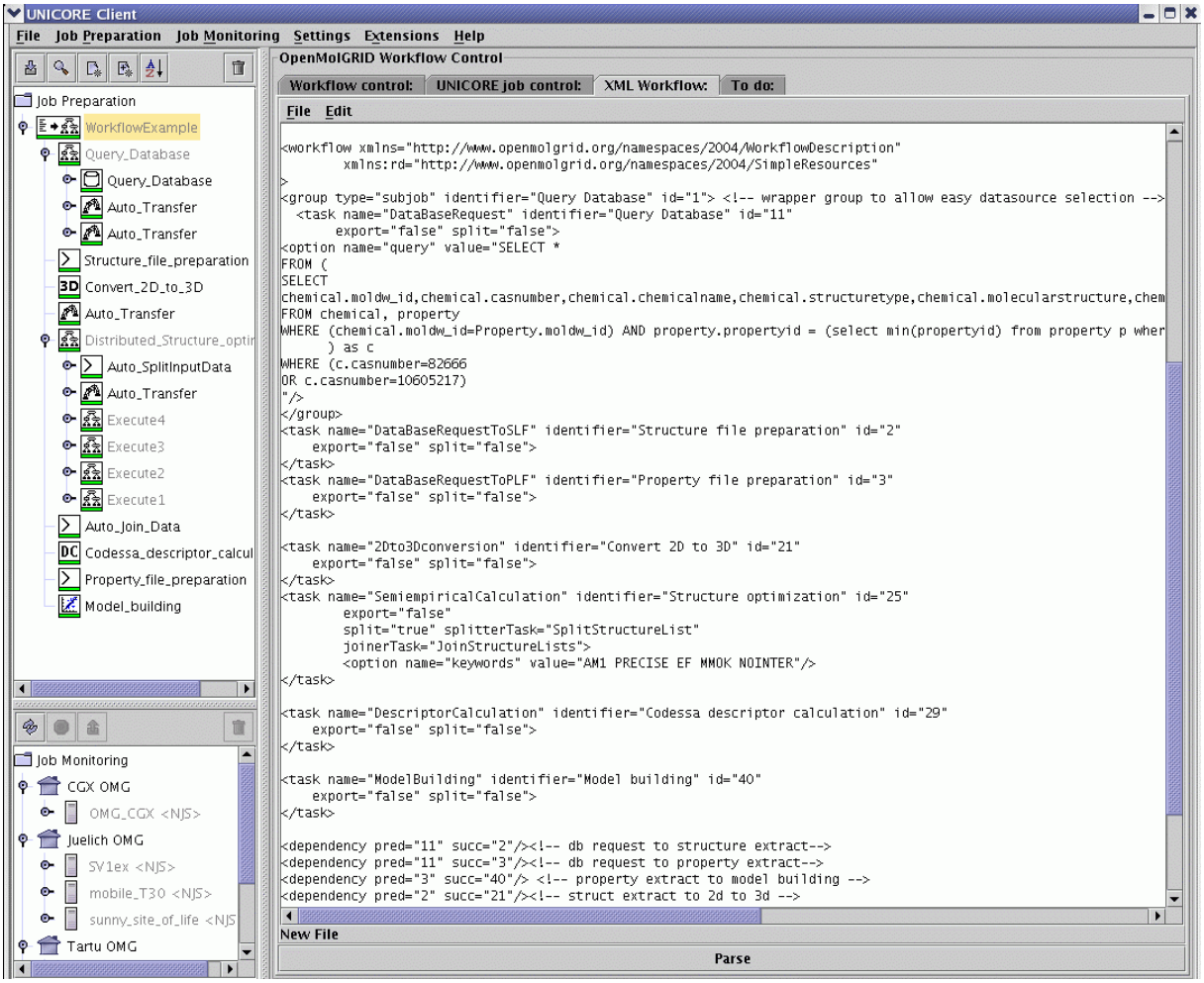

Abbildung 5: MetaPlugin

Auf der rechten Seite sieht man die XML-Beschreibung, links oben den dazugehörenden Job.

Die OpenMolGRID-Anwendungen (Plugins), die in Workflows eingebunden werden können, verfügen über ein spezielles Interface, über das sie vom MetaPlugin erkannt werden.

Serverseitig stellen die dazugehörenden Applikationen jeweils ein Metadaten-File zur Verfügung, das dem UNICORE-Klienten unter anderem mitteilt, welche Funktionen die Applikation hat und für welche Task auf der Client-Seite sie gedacht ist. Das MetaPlugin benutzt diese Information, um den lokalen Tasks und Plugins Server-Applikationen zuzuordnen. Dadurch kann es auch den aus den Workflows erstellten Jobs die erforderlichen Ressourcen wie Usites, Vsites und Software-Ressourcen für einzelne Subjobs und Tasks zuweisen.

Die für das CLI relevanten Schnittstellen und Formate zur Ressourcenverwaltung werden im Kapitel 3.3.6 im Detail erläutert.

3.3.1.6 ServiceRegistry.jar

Das ServiceRegistry-Plugin ist ein weiteres UNICORE-Plugin, das für OpenMolGRID entwickelt wurde. Es enthält Komponenten und Schnittstellen, die vom MetaPlugin für die Ressourcenverwaltung benutzt werden. Das CLI verwendet Teile des ServiceRegistry-Plugins, um die Schnittstelle zum MetaPlugin zu realisieren.

3.3.2 Übersicht der CLI-Pakete

Das CLI besteht aus einem Hauptpaket und mehreren Unterpaketen. Abbildung 6 stellt die Architektur des CLI dar:

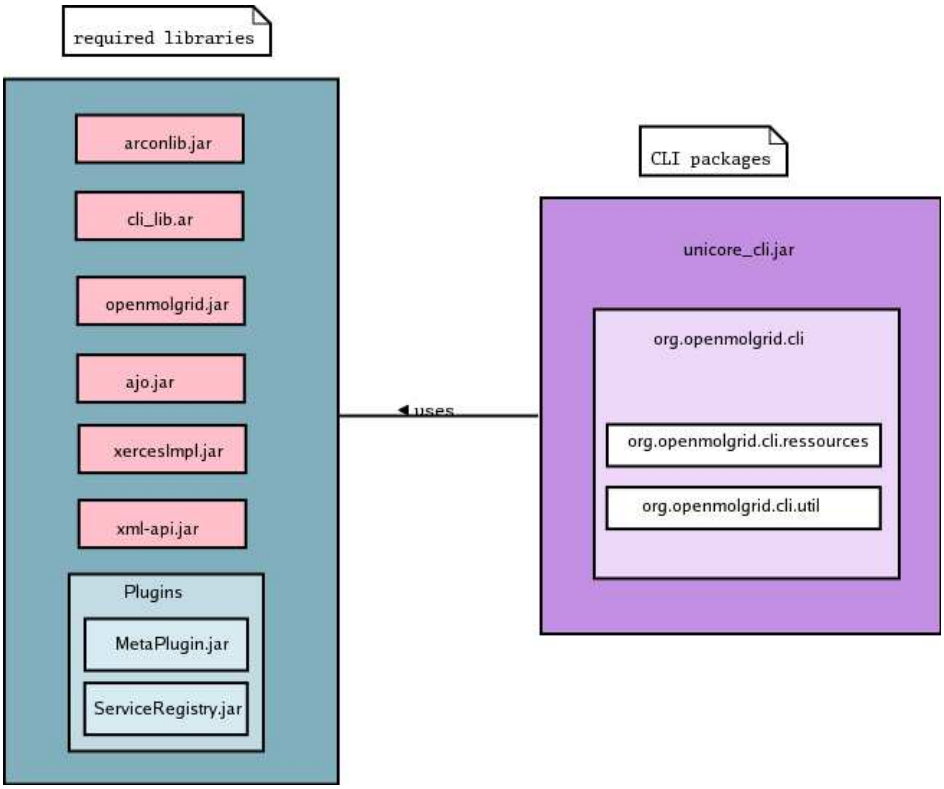

Abbildung 6: Architektur des CLI

Alle CLI-Pakete sind im jar-Archive unicore_cli.jar enthalten. Sie haben folgende Funktionen:

- `org.openmolgrid.cli` ist das Hauptpaket, das grundlegende CLI-Klassen zur Verfügung stellt und die CLI-Befehle implementiert.
- `org.openmolgrid.cli.resources` enthält Klassen, die zum Finden und Verwalten von lokalen und Serverressourcen verwendet werden. Es wird hauptsächlich für die Jobgenerierung benutzt.
- `org.openmolgrid.cli.util` umfasst Klassen, die anderen Paketen als Werkzeuge zur Verfügung gestellt werden.

3.3.3 CLI-Hauptpaket

Das CLI-Hauptpaket ist relativ einfach strukturiert und besteht nur aus den Befehls-Klassen:

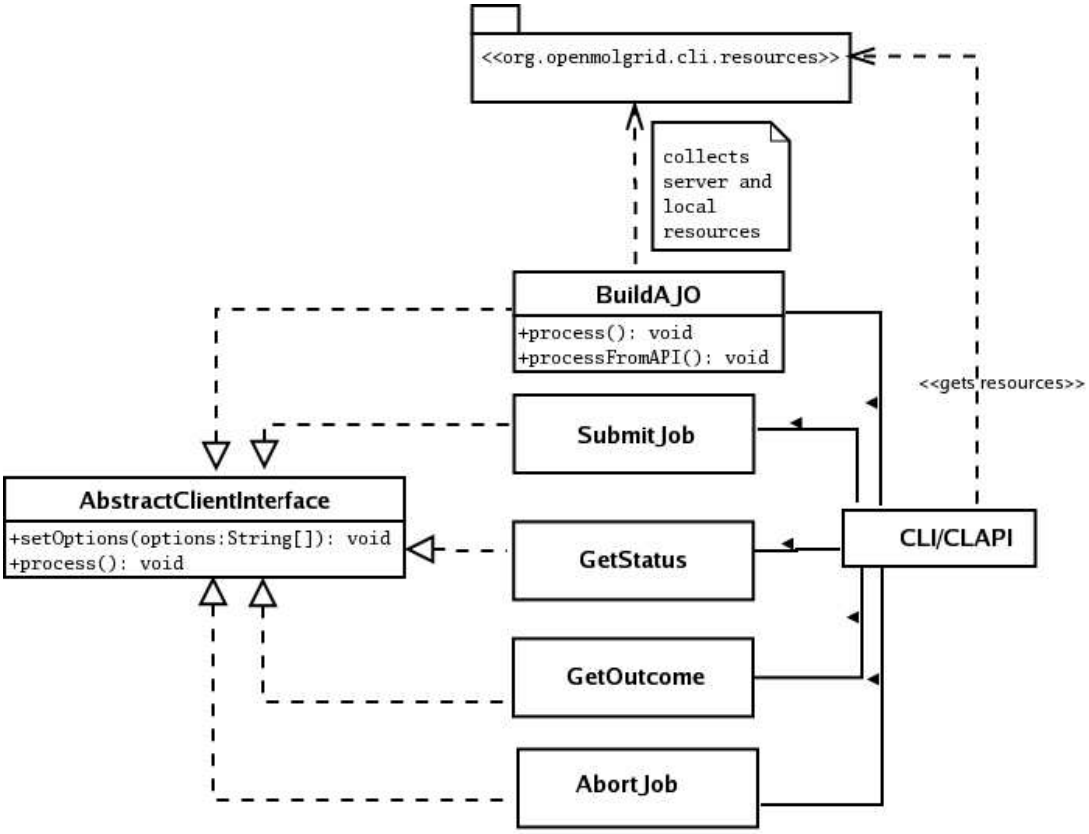

Abbildung 7: Komponenten des CLI-Hauptpakets

- Die Klasse *BuildAJO* erstellt ein Abstract Job Object (AJO) aus der XML-Workflow-Beschreibung und speichert es in einer Datei;
- *SubmitJob* liest ein AJO aus einer Datei und übermittelt es entweder synchron oder asynchron an die im AJO spezifizierte Vsite;
- *GetStatus* fragt den Status des submittierten Jobs ab und schreibt ihn in eine Datei oder auf stdout;
- *GetOutcome* holt die Jobergebnisse, falls die Ausführung des Jobs teilweise oder vollständig abgeschlossen ist;
- *AbortJob* bricht die Jobausführung ab und löscht den Job auf dem Server.

Abhängig davon, ob das CLI von der Kommandozeile oder aus einem Programm heraus über das Command Line API (CLAPI) aufgerufen wird, wird eine Instanz der entsprechenden Klasse (*CLI* oder *CLAPI*) angelegt, die die auszuführenden Kommandos an die entsprechenden Befehls-Klassen weiterleitet. Alle Befehls-Klassen implementieren eine Schnittstelle, das sog. *AbstractClientInterface*, die zwei Methoden zur Verfügung stellt: `setOptions()` und `process()`. Die Methode `setOptions()` setzt alle für das Kommando erforderlichen Parameter, die im Kapitel 3.2.1 näher spezifiziert wurden. Die Methode `process()` bearbeitet den Befehl dann. Da alle Befehls-Klassen diese beiden Methoden zur Verfügung stellen, brauchen CLI bzw. CLAPI nicht zu wissen, welcher Befehl gerade ausgeführt werden soll, und können die entsprechende Klasse dynamisch laden, was den Speicherverbrauch und Zeitaufwand deutlich reduziert.

Zur Jobgenerierung sind Informationen über verfügbare Rechen- und Softwareressourcen erforderlich, um den Job aufzubauen und die Sites, wo er ausgeführt werden kann, richtig zu setzen. Diese Informationen werden von den Klassen im Paket `org.openmolgrid.cli.resources` zur Verfügung gestellt.

Sie werden auch dann benötigt, wenn das CLI über seine API-Schnittstelle (CLAPI) benutzt wird. Alle erforderlichen Ressourcen werden direkt vom CLAPI gesammelt und intern gespeichert, um sequentielle Joberstellung und –abarbeitung zu ermöglichen, ohne jedes Mal durch Neustart des CLI einen unnötigen Overhead zu produzieren. Dadurch werden redundante und zeitintensive Ressourcenabfragen vermieden.

3.3.4 Command Line API (CLAPI)

Das CLAPI ermöglicht das Einbinden des CLI in ein Programm. Der Unterschied zwischen den Klassen *CLI* und *CLAPI* ist, dass eine Instanz des CLAPI bei ihrer Initialisierung alle verfügbaren lokalen und Server-Ressourcen (Plugins, Server-Applikationen, Usites, Vsites etc.) abfragt und zwischenspeichert. Dadurch kann man beliebig viele Workflows sequentiell abarbeiten, ohne jedes Mal die erforderlichen Informationen erneut zu sammeln. Abbildung 8 zeigt, wie das CLAPI mit den anderen Modulen zusammenhängt.

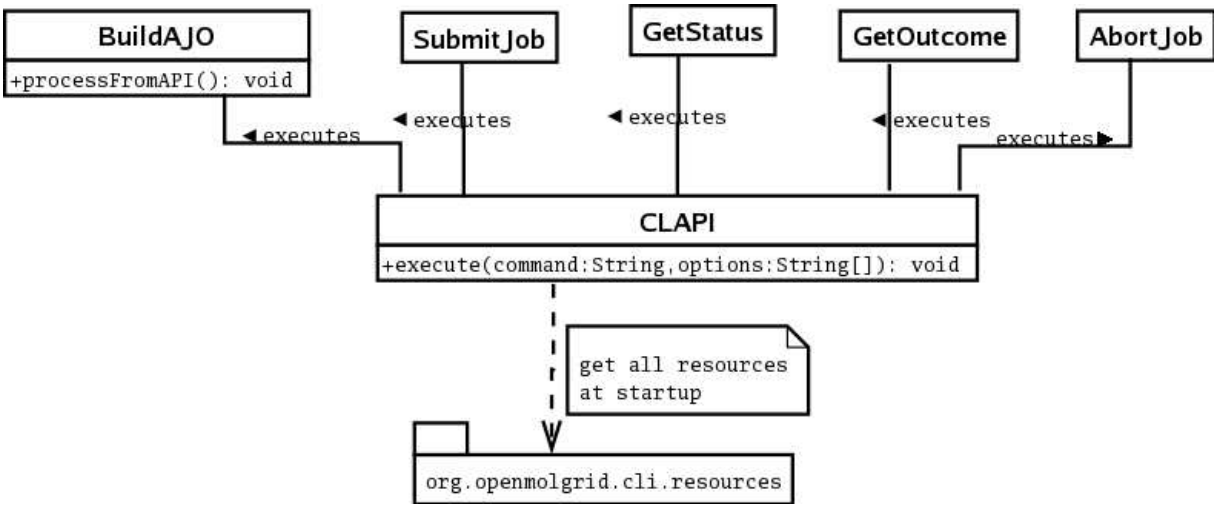

Abbildung 8: Command Line API

Das CLAPI stellt die Methode `execute()` zur Verfügung, die das auszuführende Kommando und die dazugehörigen Parameter (siehe 3.2.1) erwartet. Alle Befehle erfordern eine Keystore-Datei und ein dazugehörendes Passwort. Damit diese nicht in den Quellcode des aufrufenden Programms eingebaut werden müssen, werden sie ausschließlich aus der Datei *userdefaults.txt* ausgelesen. Aus Sicherheitsgründen sollte nur der jeweilige CLI-Benutzer diese Datei lesen dürfen. Auf diese Weise vermeidet man eine Sicherheitslücke, die sich sonst über die Bequemlichkeit der Benutzer/Programmierer ergeben könnte, die beispielsweise ihre Keystore- oder Passwortinformationen in ihre Quelltexte kodieren. Diese Datei darf statt des Passworts im Klartext auch den vollqualifizierten Namen einer Datei enthalten, in der das Passwort gespeichert ist. *userderfaults.txt* ist somit für die Benutzung des CLAPI unbedingt erforderlich und muss für jeden Benutzer angepasst werden. Sie wird im folgenden Kapitel näher spezifiziert.

Hier ist ein kurzes Beispiel, wie das CLAPI benutzt wird:

```
// create an instance of CLAPI
CLAPI cli = new CLAPI();

// specify a command
String command = “build_ajo”;

// create a String array with required parameters (note that authentification
// information is provided by the user defaults file!)
String[] options = {“-in”, “examples/HelloWorld.xml”, “-out”, “test.ajo”};

//execute command
cli.execute(command, options);
```

3.3.4.1 User Defaults

Die Datei *userdefaults.txt* wird vom CLAPI und von der Klasse *BuildAJO* benutzt. Sie stellt benutzerspezifische Informationen zur Verfügung und ist im Java Properties File Format (siehe 3.2.1.6) verfasst (*key=value*). Die folgenden Einträge sind vorgesehen:

key	***value-Beschreibung***	***erforderlich***
keystore	Vollqualifizierter Name der Keystore-Datei im PKCS12-Format	ja
password oder pwfile	Passwort oder vollqualifizierter Name der Datei, die das Passwort für den Keystore enthält	ja
url	URL mit einer Liste von Usites, die zur Verfügung stehen (vorbesetzt mit der URL des OpenMolGRID-Projekts)	ja
userDefaultsDir	Referenz zum UNICORE-Vorgabenverzeichnis, das Vorgaben für Plugins enthält.	ja
blackList	Liste von Vsites, die nicht benutzt werden sollen	nein
logger	Vollqualifizierter Name der Log-Datei	ja
logging_level	Festlegung des logging-Umfangs	nein

Das folgende Beispiel zeigt, wie die Datei *userdefaults.txt* aussehen kann:

```
#
#User Defaults for the CLI
#
#format: property=value

#
#User identity: CLI needs a keystore in PKCS12 format
#

# path to the keystore
keystore=/path/to/my_identity.p12
pwfile=/path/to/my_password

#url for Usite list
url=http://www.openmolgrid.org/omg_gateways.xml

# Site configuration (list of Vsites that should be ignored)
# format: siteList=vsite_1,vsite_2,...,vsite_n
# no blanks between commas!!
blackList=OMG_Test1,mobile_T30,SV1ex,OMG_CGX,VSite_1,NJS Linux (testing)

#User defaults directory: where the plugin defaults are
userDefaultsDir=/home/user/.unicore

#
#Logging settings
#

#path to the logfile
logger=/opt/cli/cli_0.6.0/log/clilog.txt

#log level: OFF SEVERE WARNING INFO CONFIG FINE FINER FINEST
logging_level=INFO
```

3.3.5 BuildAJO

Die Joberstellung ist die zentrale Funktion des CLI. Dabei wird aus einer XML-Workflow-Beschreibung, deren Format im Kapitel 3.3.5.2 spezifiziert wird, ein Abstract Job Object (siehe 2.3) erzeugt und in serialisierter Form in einer Datei abgespeichert. Zum besseren Verständnis dieses Vorgangs müssen vorher die Container-Klassen, auf denen die Joberstellung basiert, genauer erläutert werden:

3.3.5.1 Container-Klassen

Der UNICORE-Klient benutzt Container-Klassen, um Tasks, Subjobs und Jobs zu verwalten. Vor der Jobübermittlung werden die ActionGroups der jeweiligen Tasks aus dem zugehörigen Container extrahiert und zu einem AJO zusammengefügt. Weil das CLI für die Jobgenerierung zum Teil schon vorhandene Komponenten des MetaPlugins benutzt, die auf den Container-Klassen aufbauen, muss es auch in der Lage sein, mit Container-Klassen umzugehen.

Die UNICORE Container-Klassen sind hierarchisch aufgebaut und werden alle von einer Superklasse, dem *ActionContainer*, abgeleitet. Abbildung 9 gibt einen Überblick über die existierenden Container-Klassen. Die Farben symbolisieren hier die verschiedenen Arten von Container-Klassen, ihre Intensität die Generation der Vererbung.

Es gibt zwei für diese Arbeit relevante Container-Arten:

- *GroupContainer*, die mehrere komplexe Tasks und Subjobs sowie deren Abhängigkeiten (*Dependencies*) enthalten können. Von den *GroupContainern* ist der *JobContainer* für das CLI bzw. für das MetaPlugin besonders wichtig. Der *JobContainer* ist die Struktur, die einen Job im UNICORE-Klienten intern darstellt. Er verwaltet nicht nur alle Jobkomponenten, sondern auch z.B. Informationen, auf welcher Vsite ein (Sub-)Job wie ausgeführt werden soll oder welche Dateien für die Jobausführung benötigt werden, bzw. nach der Jobausführung zurückgeholt werden sollen. Der *JobContainer* ist auch die einzige Container-Klasse, die ein submissionsfertiges AJO generieren und zur Verfügung stellen kann. Das MetaPlugin leitet aus der *JobContainer*-Klasse die eigene Container-Klasse, den sog. *MetaContainer*, ab, der mit Hilfe neu erstellter CLI-Klassen die Jobgenerierung aus den Daten einer XML-Workflow-Beschreibung übernimmt.
- *TaskContainer*, der für einzelne Tasks benutzt wird. Er stellt eine Basis-Klasse für zwei weitere Container-Arten dar: *FileContainer* und *ExecuteContainer*. Für die Joberstellung über das CLI sind die Subklassen des *ExecuteContainers* sehr wichtig, besonders der *OMGUserContainer*. Der *OMGUserContainer* stellt eine Schnittstelle zur Verfügung, die vom MetaPlugin zur Jobgenerierung benutzt wird. Alle Plugins, die für das OpenMolGRID-Projekt entwickelt wurden, sind Subklassen des *OMGUserContainers*.

Die Joberstellung über das CLI basiert auf dem Zusammenfügen von Container-Instanzen der jeweiligen Tasks, aus denen dann ein Abstract Job Object generiert wird.

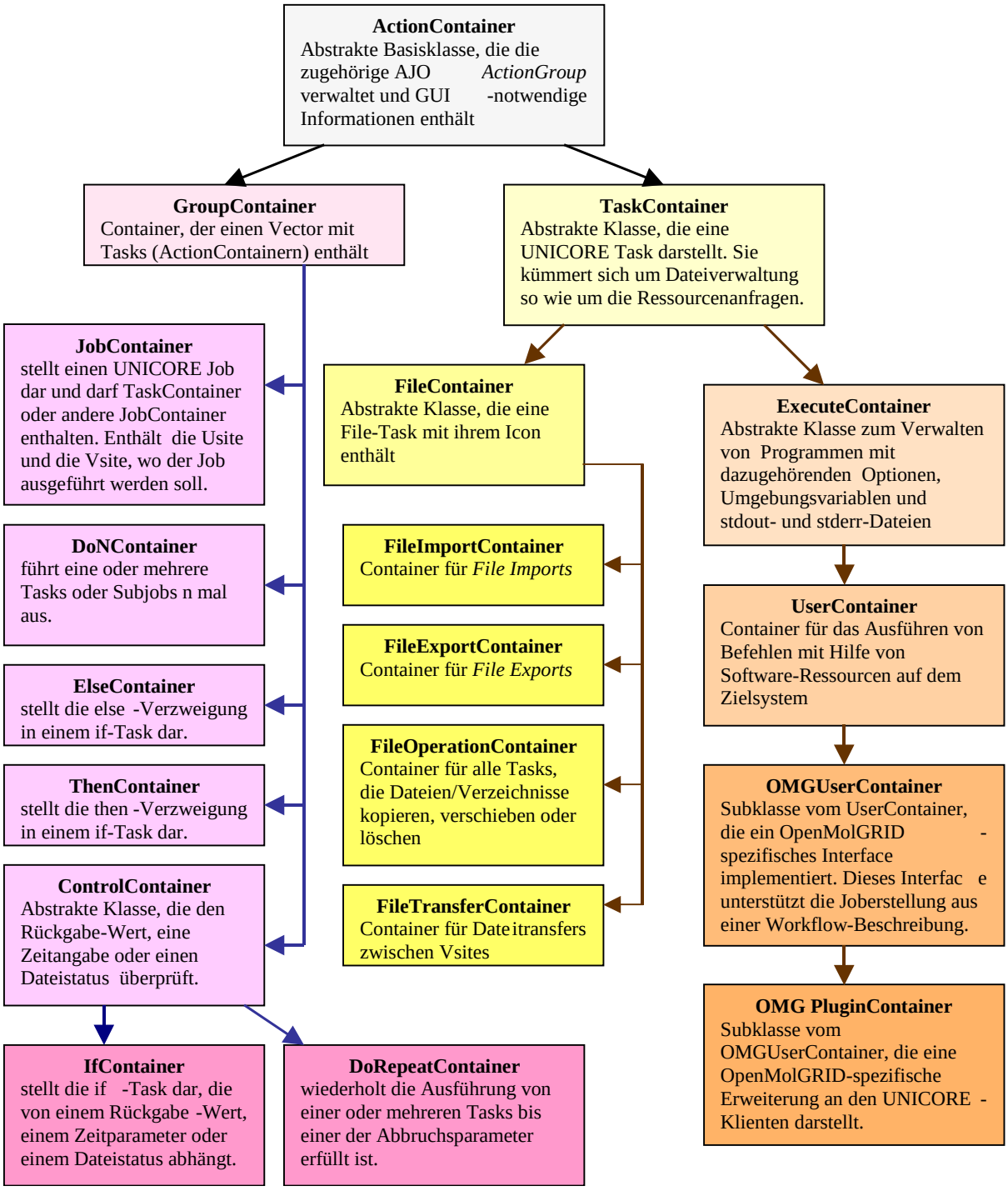

Abbildung 9: Hierarchie der UNICORE Container-Klassen

3.3.5.2 Beschreibung des XML-Workflow-Formats

Das DTD (Document Type Definitions) für den Workflow sieht folgendermaßen aus:

```
<!ELEMENT workflow (  ( (task*, group*) | (group*, task*) ), dependency*,
resourceRequest?)>
    <!ELEMENT task (option*, localInput*, resourceRequest?)>
        <!ELEMENT option EMPTY>
            <!ATTLIST option
                name     CDATA #REQUIRED
                value    CDATA #REQUIRED
            >
        <!ELEMENT localInput EMPTY>
            <!ATTLIST localInput
                source       CDATA #REQUIRED
                destination CDATA #IMPLIED
                type        CDATA #REQUIRED
                ascii       (true|false) #IMPLIED
                overwrite   (true|false) #IMPLIED
            >
        <!ATTLIST task
             name        CDATA #REQUIRED
             identifier  CDATA #REQUIRED
             id          CDATA #REQUIRED
             export      (true | false) #IMPLIED
             split       (true | false) #IMPLIED
             splitterTask CDATA #IMPLIED
             joinerTask   CDATA #IMPLIED
        >
    <!ELEMENT group (option*, ( (task*, group*) | (group*, task*) ),
dependency*, resourceRequest?)>
        <!ATTLIST group
            type        (subjob | repeat | doN | if | then | else) #REQUIRED
            identifier  CDATA #REQUIRED
            id          CDATA #REQUIRED
        >
    <!ELEMENT dependency EMPTY>
        <!ATTLIST dependency
            pred        CDATA #REQUIRED
            succ        CDATA #REQUIRED
        >
   <!ELEMENT resourceRequest ANY>
```

Ein Workflow besteht aus *Tasks*, *Groups*, *Dependencies* (Abhängigkeiten zwischen den einzelnen Tasks und/oder Subjobs) und evtl. Ressourcenanforderungen, den sog. *resourceRequest*. Eine *Group* kann andere *Groups* und *Tasks* enthalten, die sich im selben, vom *GroupContainer* abgeleiteten Container befinden (siehe Abbildung 9). Es gibt unterschiedliche Group-Typen: `subjob`, `repeat`, `doN`, `if`, `then` und `else`. Jedes Task- oder Group-Element muss eine eindeutige Identifizierungsnummer, den *Identifier*, haben, um *Dependencies* festlegen zu können. Das Haupt-XML-Element ist `workflow`. Es umschließt alle anderen Elemente, welche wiederum weitere Elemente enthalten können. Der Aufbau eines Workflows kann durch einen Baum beschrieben werden (Abbildung 10).

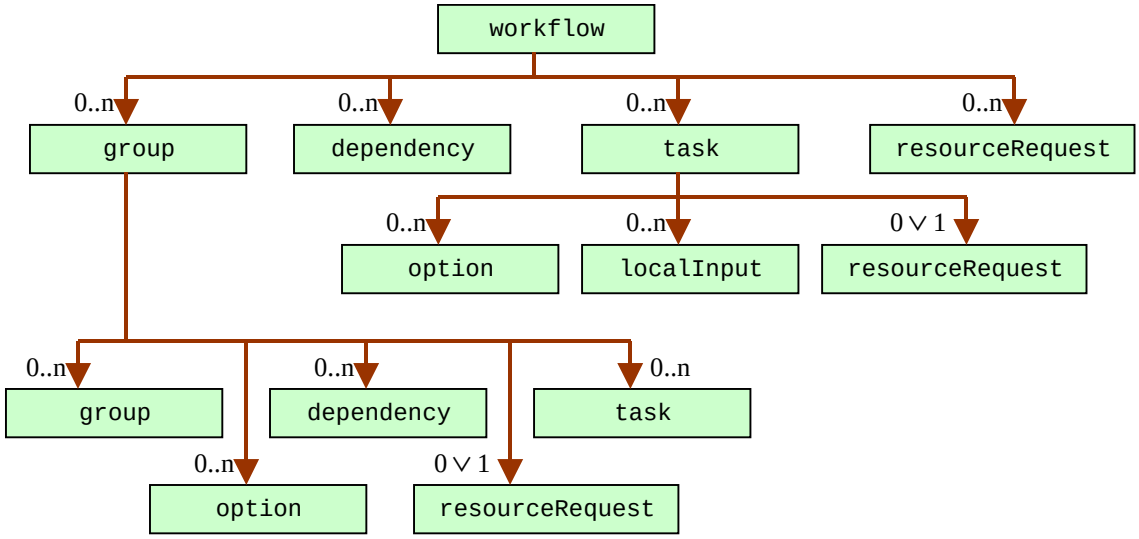

Abbildung 10: Elementenhierarchie eines Workflows

Einzelne Elemente sind durch ihre Attribute beschrieben. Es gibt jeweils erforderliche und optionale Attribute:

- **`group`-Element:** beschreibt Subjobs oder Tasks, die von einem *GroupContainer* abgeleitet sind (siehe Abbildung 9). Groups können weitere Elemente enthalten und werden unter anderem durch das Sub-Element `option` genauer beschrieben und kontrolliert:

Attribut	**Beschreibung**	**Format**	**erforderlich**
`type`	beschreibt den GroupContainer, den diese Group intern darstellt	Zeichenkette	ja
`identifier`	gibt den Namen der Group an, unter dem sie angezeigt wird	Zeichenkette	ja
`id`	eindeutiges Kennzeichen einer Group bzw. Task	Zeichenkette	ja

- **`task`-Element:** stellt eine Task dar, die auf dem Zielsystem ausgeführt werden soll:

Attribut	**Beschreibung**	**Format**	**erforderlich**
`name`	Name der Task	Zeichenkette	ja
`identifier`	Klartextname, unter dem die Task angezeigt wird	Zeichenkette	ja
`id`	eindeutiges Kennzeichen einer Group bzw. Task	Zeichenkette	ja
`export`	zeigt an, ob der Output einer Task nach dem Ausführungsende vom Zielsystem auf die lokale Maschine transferiert werden soll	true | false	nein
`split`	zeigt an, ob die Task auf mehrere Sites verteilt werden soll	true | false	nein
`splitterTask`	Name der Task, die die Aufteilung der Inputdaten für die beteiligten Sites durchführen soll.	Zeichenkette	nein
`joinerTask`	Name der Task, die die verteilt erzeugten Ausgabedaten zu einer Datei zusammenfügt	Zeichenkette	nein

- **`localInput`-Element:** spezifiziert lokale Dateien, die von dem lokalen System auf das Zielsystem importiert werden sollen:

Attribut	**Beschreibung**	**Format**	**erforderlich**
`source`	lokale Datei, die auf das Zielsystem importiert werden soll	Zeichenkette	ja
`destination`	Name für die importierte Datei auf dem Zielsystem	Zeichenkette	nein
`type`	Dateityp	Zeichenkette	ja
`ascii`	gibt an, ob die Datei als eine ascii-Datei behandelt werden soll (Standard: false)	true | false	nein
`overwrite`	gibt an, ob die Datei überschrieben werden darf (Standard: false)	true | false	nein

- **`option`-Element:** ist ein Sub-Element eines `group`- oder `task`-Elements und beschreibt bestimmte Parameter einer Group bzw. Task. `option` gibt *name/value* (Name/Wert) Paare an, wobei die Task-spezifischen Informationen von ihren jeweiligen Entwicklern dokumentiert werden müssen und erst zur Laufzeit des CLI beim Sammeln von lokalen Ressourcen abgefragt werden:

Attribut	**Beschreibung**	**Format**	**erforderlich**
`name`	Name der Option	Zeichenkette	ja
`value`	Wert der Option	Zeichenkette	ja

Es gibt folgende vordefinierte *name/value* Paare einer Group:

- *DoN*-Group (wiederholt die Ausführung einer Task N Mal): *name*: `iterations` *value*: `N`
- *if*- oder *repeat*-Group (Fallunterscheidung oder, bei der *repeat*-Group, Wiederholung des Inhalts, bis die Abbruchsbedingung erfüllt ist):

  name: `testTask` *value*: id einer Task, die überprüft werden soll

  name: `testValue` *value*: Rückgabewert, der überprüft werden soll

  name: `testType` *value*: Worauf der Rückgabewert üeberprüft werden soll: `equal|not_equal|successful|not_successful`

- **`resourceRequest`-Element:** gibt die für die Group oder Task benötigten Ressourcen an. Der Inhalt dieses Elements ist frei definierbar. Zur Zeit gibt es eine DTD für *SimpleResources*, die die Angabe von Usite/Vsite bzw. Laufzeit, Anzahl Knoten, Anzahl Prozessoren pro Knoten und Hauptspeicher pro Knoten erlaubt.
- **`dependency`-Element:** spezifiziert die Ausführungsreihenfolge der einzelnen Tasks und Subjobs. Es liegt in der Verantwortung des Benutzers, zyklische Abhängigkeiten zu vermeiden.

Attribut	**Beschreibung**	**Format**	**erforderlich**
`pred`	`id` des Vorgängers	Zeichenkette	ja
`succ`	`id` des Nachfolgers	Zeichenkette	ja

3.3.5.3 Implementierung von `build_ajo`

Das Erstellen eines Jobs aus einer XML-Workflow-Beschreibung besteht hauptsächlich aus zwei Schritten: dem Sammeln der notwendigen Ressourcen (siehe 3.3.6) und dem Aufbau eines AJO mit Hilfe von Mechanismen, die aus dem MetaPlugin übernommen werden. Die Zusammenhänge zwischen den einzelnen Komponenten werden in der Abbildung 11 dargestellt:

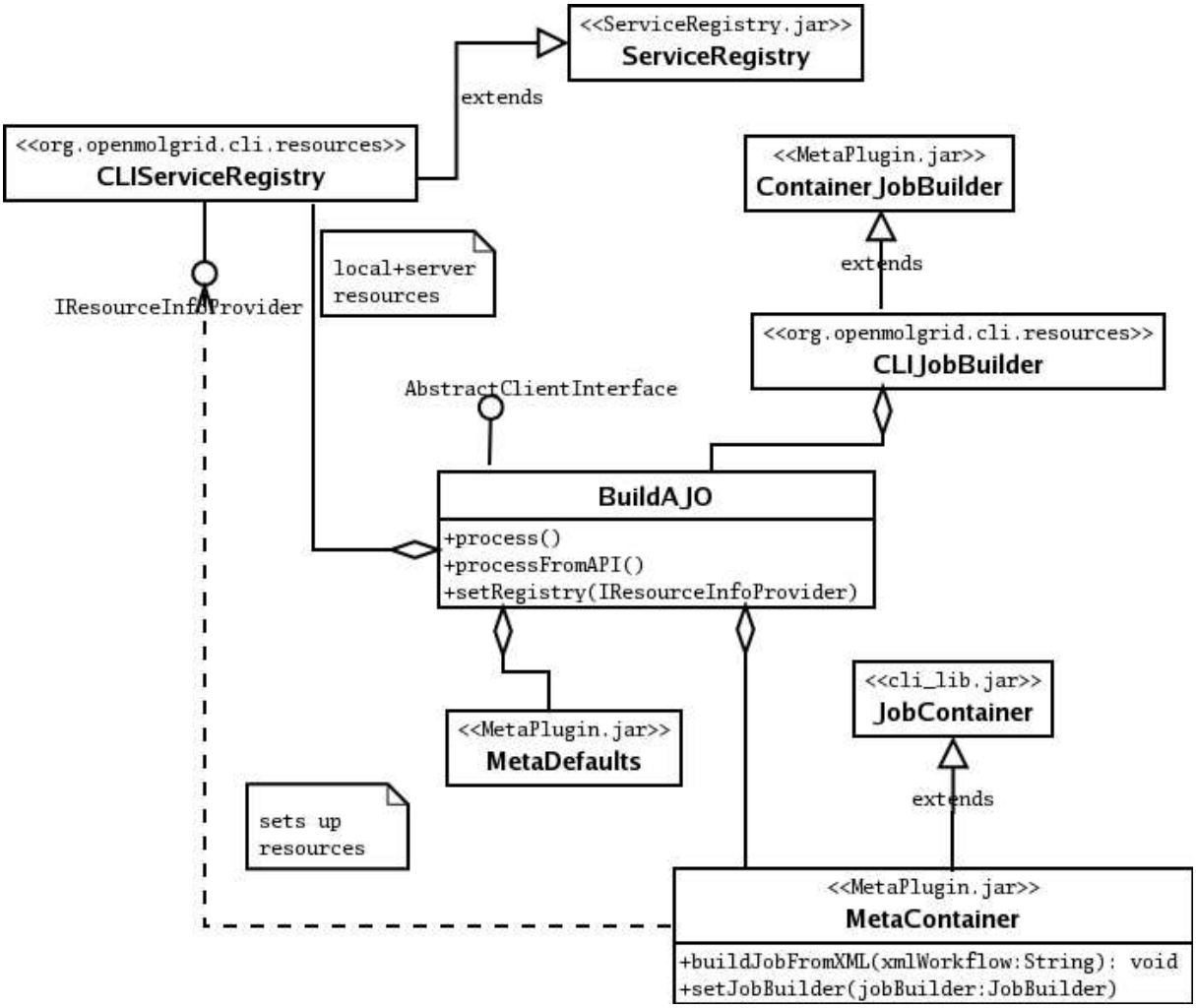

Abbildung 11: Joberstellung

- Die Klasse *BuildAJO* stellt zwei Schnittstellen für ihren Aufruf zur Verfügung: die Methode `process()` wird für den Aufruf aus der Kommandozeile verwendet, und die Methode `processFromAPI()` wird über das *CLAPI* benutzt. `process()`bewirkt, dass beim Aufruf alle notwendigen lokalen und Server-Ressourcen abgefragt und gespeichert werden, wobei beim Aufruf über das *CLAPI* alle Ressourcen schon bekannt sind und über die Methode `setRegistry()` an die Instanz von *BuildAJO* übergeben werden können.
- Die Methode `setRegistry()` erwartet als Übergabeparameter eine Instanz einer Klasse, die das Interface `IResourceInfoProvider` implementiert. Dieses Interface wird genauer in 3.3.6 erläutert.
- *BuildAJO* enthält eine Instanz der Klasse *MetaContainer*, die zum MetaPlugin gehört. MetaContainer ist eine Subklasse des JobContainers (siehe Abbildung 9) und stellt die Workflow-Bearbeitungsmechanismen zur Verfügung.
- *MetaContainer* benötigt Informationen über lokale und Server-Ressourcen, die von den Klassen im Paket `org.openmolgrid.cli.resources` gesammelt und über das Interface `IResourceInfoProvider` zur Verfügung gestellt werden.

- Die Instanz der Klasse *BuildAJO* enthält auch eine Instanz von der *MetaDefaults*-Klasse. *MetaDefaults* dienen zur Verwaltung von Benutzereinstellungen, die speziell für die Joberstellung aus einem Workflow gedacht sind. Sie werden dann gebraucht, wenn z.B. ein Job auf mehrere Vsites verteilt werden kann oder eine Task von verschiedenen Serverapplikationen unterstützt wird.
- *CLIJobBuilder*, der in *BuildAJO* enthalten ist, stellt Methoden zur Verfügung, die für den konkreten Aufbau eines Jobs aus einer abstrakten Beschreibung benötigt werden. Er wird über die Methode `setJobBuilder()` an den *MetaContainer* übergeben und liefert unter anderem die den Tasks entsprechenden Container-Klassen, wenn ein konkretes AJO aus dem Workflow aufgebaut wird.

Beim Aufruf von *BuildAJO* wird zuerst geprüft, ob alle notwendigen Informationen über die Ressourcen vorhanden sind, und falls dies nicht der Fall ist, werden sie gesammelt und in für den *MetaContainer* kompatibler Form gespeichert. Es wird eine Instanz des *MetaContainers* angelegt, die alle Ressourcen und eine Instanz des *CLIJobBuilders* übergeben bekommt und damit aus einem XML-Workflow ein AJO erstellen kann. Die Informationen über die zu diesem AJO gehörenden Input- (*File Imports*) und Outputdateien (*File Exports*) werden zusammengestellt und in einer Instanz der *JobAttachment*-Klasse gespeichert. Diese wird in ein Byte-Array umgewandelt und im entsprechenden Feld des AJO abgelegt. Das AJO wiederum wird dann serialisiert und in eine Datei geschrieben.

3.3.6 Resource Management

Resource Management, das Abfragen und Verwalten zur Verfügung stehender lokaler und Server-Ressourcen, ist eine der wichtigsten Komponenten des CLI. Weil die Infrastruktur des UNICORE-Klienten dem CLI nicht zur Verfügung steht, musste die Ressource-Management-Komponente komplett neu entwickelt werden. Zusätzlich kam noch hinzu, dass keine Interaktion mit dem Benutzer möglich ist, so dass die Suche und Zuordnung von Serverapplikation zu lokalen Tasks und die Entscheidungen über die Auswahl des Zielsystems vom CLI vorgenommen werden müssen. Eine weiteres Problem ist, dass während der Joberstellung u.U. Container-Klassen geladen werden, die auf die nicht vorhandenen UNICORE Standard-Ressource-Management-Klassen zugreifen. Um dieses Problem zu lösen, wurden Wrapper-Klassen entwickelt, die nach außen als Standard-UNICORE-Klassen auftreten, intern aber die Methodenaufrufe an die CLI-eigene Infrastruktur weiterleiten. Um diese Wrapper-Klassen in die UNICORE-Umgebung zu integrieren, wurde eine eigene CLI-spezifische Bibliothek, *cli_lib.jar*, zusammengestellt, die außer den Wrapper-Klassen auch die erforderlichen UNICORE-Klienten-Klassen enthält.

Dieses Kapitel beschreibt die Zusammenhänge zwischen den Klassen und das Funktionsprinzip der zentralen Komponenten im Paket `org.openmolgrid.cli.resources`, die in Abbildung 12 dargestellt sind.

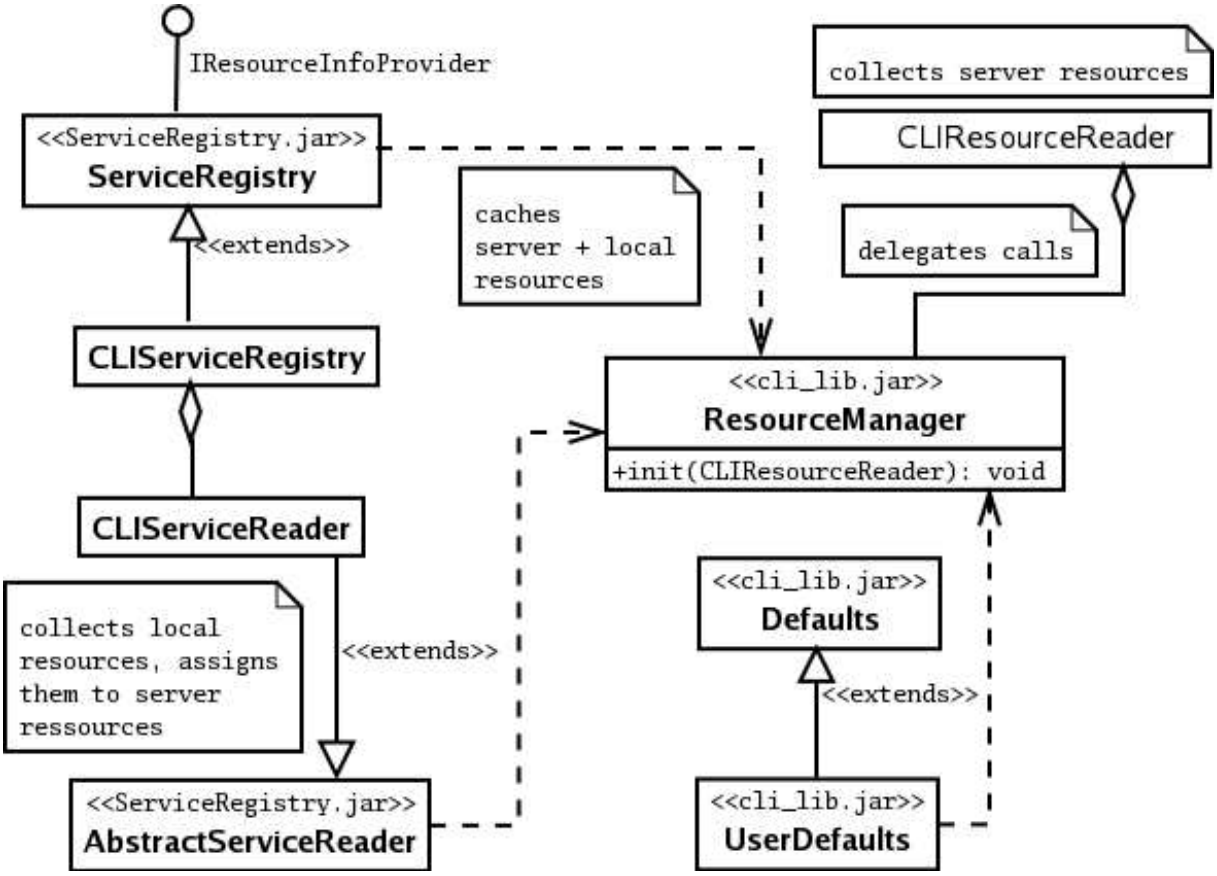

Abbildung 12: Zentrale Resource-Management-Komponenten

Das Resource Management besteht hauptsächlich aus drei Arten von Klassen:

- Klassen, die für das Sammeln und Verwalten der Server-Ressourcen zuständig sind (die wichtigste von ihnen ist der *CLIResourceReader*),
- Klassen, die lokale Ressourcen abfragen (*CLIServiceReader*),
- Klassen, die für die Zuordnung zwischen den lokalen und Server-Ressourcen benutzt werden (*CLIServiceReader* und *CLIServiceRegistry*).

ResourceManager und *Defaults* sind Wrapper-Klassen, durch die die entsprechenden Standard-UNICORE-Klienten-Klassen überschrieben werden. Da die ursprünglichen Klassen oft zur Laufzeit während der Joberstellung von den einzelnen Container-Klassen benötigt werden, dem CLI aber nicht zur Verfügung stehen, mussten CLI-eigene Klassen mit gleichen Namen entwickelt werden, die die gleichen Methoden haben, intern aber alle Aufrufe an die CLI-spezifische Umgebung delegieren.

Am Anfang werden vom CLI alle verfügbaren Server-Ressourcen abgefragt und intern zwischengespeichert. Das geschieht in einer Instanz des *CLIResourceReader*. Bei seiner Initialisierung liest der *CLIResourceReader* die Datei mit den Benutzereinstellungen (*userdefaults.txt*), in der unter anderem die URL des zentralen Servers mit der Liste der vorhandenen Gateways abgelegt ist, und verbindet sich mit ihm. Nachdem die vorhandenen Gateways bekannt sind, verbindet er sich sukzessiv mit ihnen und schickt ein UPL-*Request* (siehe 2.4), um die Vsites abzufragen. Danach werden asynchron UPL-*Requests* an diese

Vsites verschickt, deren *Responses* die zur Verfügung stehenden Hard- und Softwareressourcen enthalten. Gateways oder Vsites, die zur Laufzeit nicht zur Verfügung stehen, werden vermerkt, um sie später bei der Joberstellung und Auswahl des Zielsystems für den Job auszuschließen.

Nachdem die Informationen über vorhandene Serverressourcen gesammelt wurden, werden die lokalen Ressourcen von einer Instanz des *CLIServiceReader* abgefragt. Darunter sind Standard-UNICORE-Tasks und OpenMolGRID–Plugins zu verstehen, die sich auf der lokalen Benutzermaschine befinden. Dabei wird von jedem Plugin die jeweilige Container-Klasse dynamisch geladen und eine Instanz von ihr erzeugt, um später bei der Joberstellung die erforderliche Container-Klasse zur Verfügung stellen zu können.

CLIServiceReader ordnet auch die lokalen Plugins den zugehörigen Serverapplikationen zu. Dies geschieht anhand des *IChainable*-Interfaces, das jeder OpenMolGRID-Plugin-Container implementiert, und der Metadaten, die jede OpenMolGRID-Serverapplikation zur Verfügung stellt:

- Die Metadaten im XML-Format geben die jeweils von den Serverapplikation ausgeführten Tasks an und ob diese Inputdateien benötigen, von welchem Typ diese gegebenenfalls sind und welche Art Outputdateien evtl. erstellt werden.
- Das *IChainable*-Interface gibt die Namen der Tasks zurück, die vom Plugin unterstützt werden, und ermöglicht später den Joberstellungskomponenten die Serverapplikation zu setzen, die für die Jobausführung benötigt wird. Außerdem erlaubt es, die Existenz der Input- und Output-Dateien der jeweiligen Task zu prüfen.

Die Klasse *CLIServiceRegistry* speichert alle Ressourcen in der von den Joberstellungskomponenten erwarteten Form. Sie implementiert das *IResourceInfoProvider*-Interface, das bei der Joberstellung vom *MetaContainer* gefordert wird, um auf die vorhandenen lokalen und die Server-Ressourcen zugreifen zu können. Abbildung 13 zeigt, wie die einzelnen Komponenten zusammenhängen.

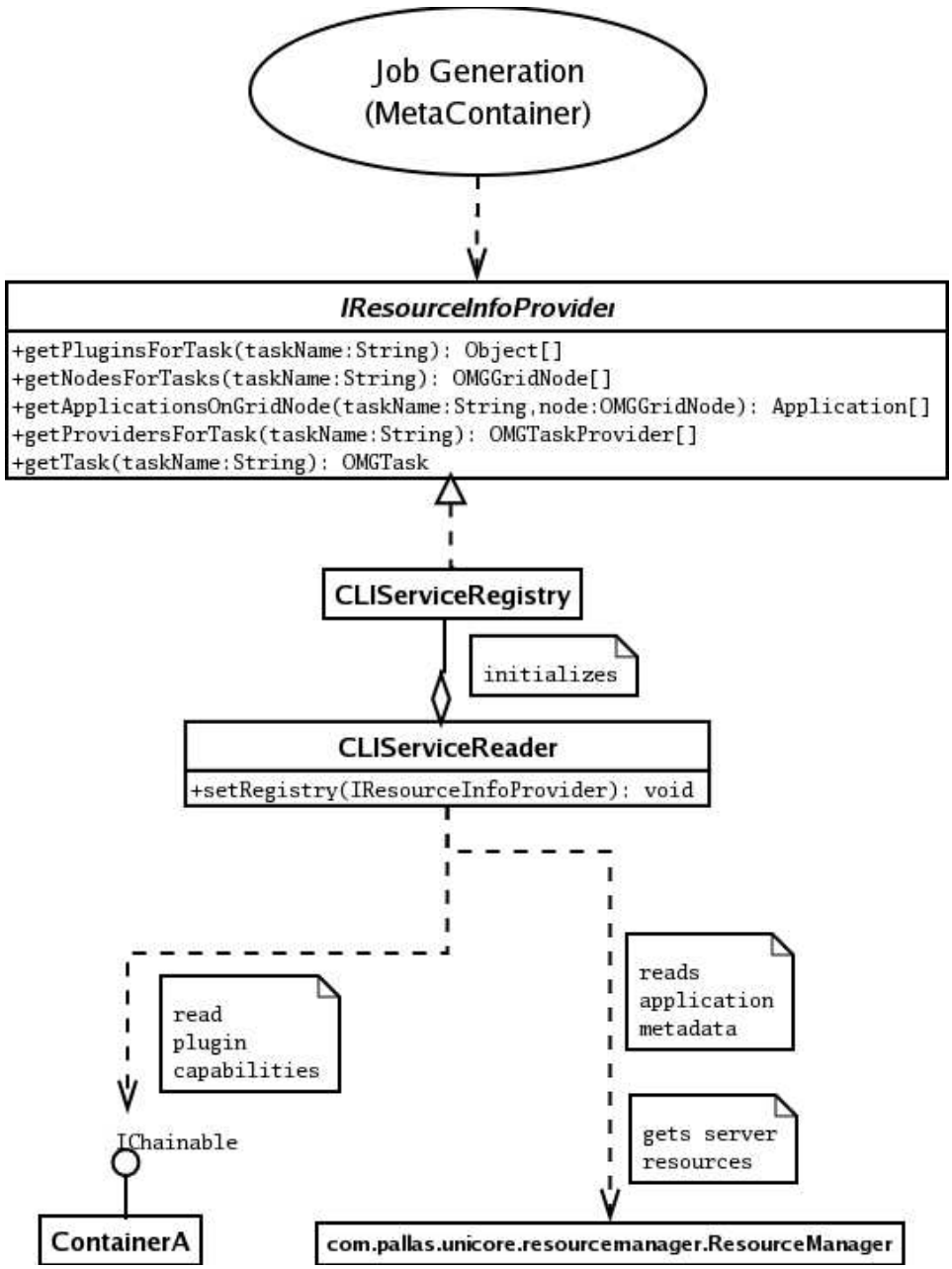

Abbildung 13: IResourceInfoProvider und damit zusammenhängende CLI-Klassen

Das *IResourceInfoProvider*-Interface stellt Methoden zur Verfügung, die während der Joberstellung aufgerufen und dazu benutzt werden, die vom *CLIServiceReader* gesammelten und zusammengefügten Informationen abzufragen. Diese Informationen werden als Objekte von Klassen, die speziell für die Ressourcenverwaltung im Rahmen des OpenMolGRID-Projektes entwickelt wurden, weitergereicht. Abbildung 14 zeigt die Hauptkomponenten dieser Klassen. *OMGTaskProvider* stellt eine Kombination aus einer Usite, Vsite und Applikation dar, was später das Setzen von notwendigen Ressourcen für eine Task erleichtert. *OMGTask* besteht aus dem Tasknamen, evtl. einer Beschreibung, und Informationen über den evtl. erforderlichen Input bzw. Output.

Der *CLIServiceReader* bekommt eine Instanz der *CLIServiceRegistry* übergeben und initialisiert sie mit den gefundenen Ressourcen. Dazu ist es notwendig, dass die Container-Klasse ein *IChainable*-Interface implementiert. Ausserdem muss der Resource-Manager

Methoden zur Abfrage der Serverressourcen zur Verfügung stellen. Intern ist der *ResourceManager* des CLI nichts weiter als eine Wrapper-Klasse, die alle Aufrufe an den hierfür entwickelten *CLIResourceReader* weiterleitet.

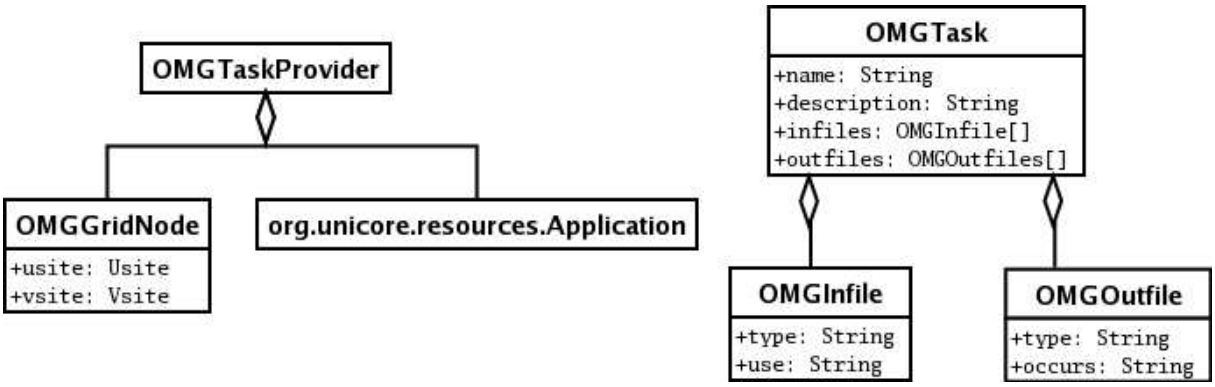

Abbildung 14: Klassen, die vom IResourceInfoProvider benutzt werden

3.3.7 SubmitJob

Die Klasse *SubmitJob* wird zum Übermitteln eines AJO an eine Vsite benutzt. Dabei stehen zwei Submissionsmodi zur Verfügung, synchron und asynchron, die Einfluss auf die Ergebnisse des Kommandos haben.

Am Anfang liest *SubmitJob* ein serialisiertes AJO aus einer Datei und wandelt es aus der Byte-Array-Form in ein konkretes Objekt der Klasse *AbstractJob* um. Aus dem entsprechenden AJO-Feld wird die Information über die erforderlichen Inputdateien (*File Imports*) und Outputdateien (*File Exports*) entnommen. Die lokalen Inputdateien werden gesammelt und zusammen mit dem AJO an die im AJO gesetzte Vsite übermittelt.

Falls das AJO synchron submittiert wurde, wartet *SubmitJob* auf das Ende der Jobausführung, holt die Jobergebnisse (siehe 3.3.8) und schreibt sie in das Outputverzeichnis (siehe 3.2.1). Andernfalls schreibt es die AJO Daten in die Datei AJO_ID_FILE (siehe 3.2.1.6) und beendet die Ausführung.

3.3.8 GetOutcome

GetOutcome holt die Ergebnisse des teilweise oder vollständig ausgeführten Jobs zurück, falls der Job asynchron submittiert wurde. Zuerst liest *GetOutcome* die Datei AJO_ID_FILE, die die Daten des übermittelten AJO enthält. Die eindeutige Kennung des Jobs, die in Base64-kodierter Form abgespeichert ist, wird in das Java-Objekt vom Typ *AJOIdentifier* umgewandelt und die Informationen über die Vsite, auf der der Job ausgeführt wurde, sowie über die zurückzuholenden Outputdateien werden intern zwischengespeichert. Das Holen der Ergebnisse von der Vsite besteht aus zwei Schritten: im ersten Schritt werden stdout/stderr-Ausgaben übertragen. Wenn die Datei AJO_ID_FILE Informationen über die *File Exports* enthält, werden diese im zweiten Schritt übertragen. Falls die Jobausführung zu Ende ist, wird der Job nach dem Holen der Ergebnisse auf dem Server gelöscht.

3.4 CLI Queue (CLIQ) - eine Erweiterung des CLI

CLIQ ist eine Zusatzkomponente, die auf CLI/CLAPI aufbaut. Sie stellt einen Queueing-Mechanismus zur Verfügung, der das CLAPI benutzt, um das sequentielle Handling von Jobs für den Benutzer zu automatisieren und dabei evtl. auftretende Redundanzen bzgl. der CLI-Aufrufe zu vermeiden. Der Zusammenhang zwischen dem CLAPI und CLIQ ist in der Abbildung 15 dargestellt.

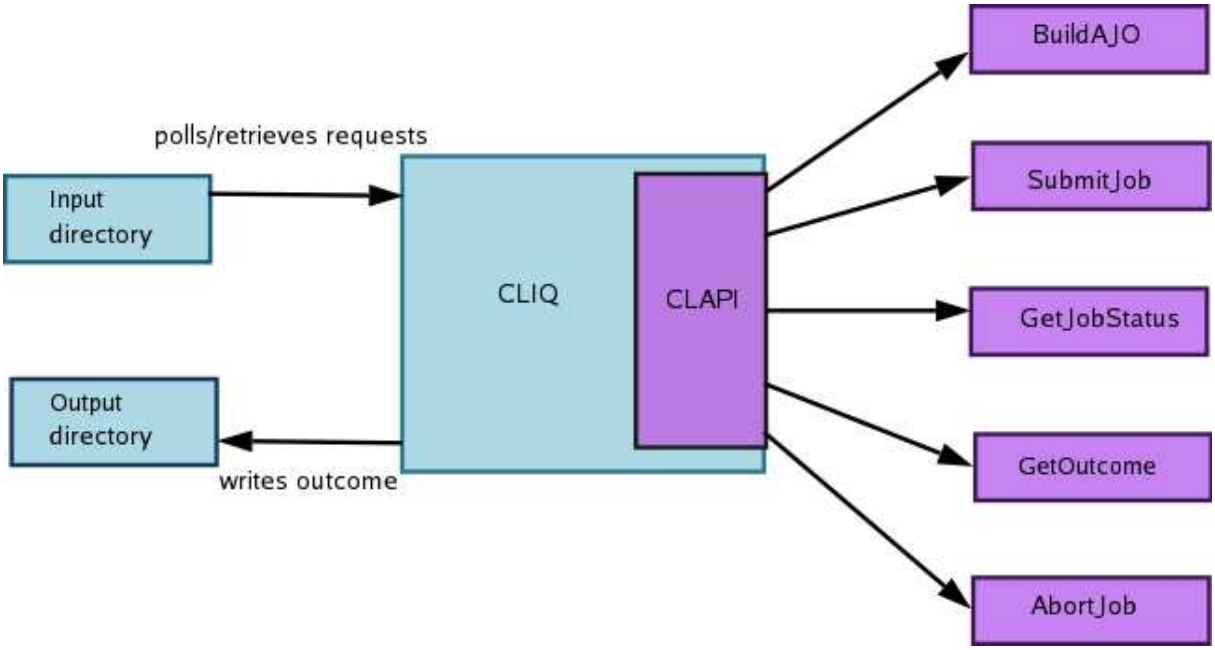

Abbildung 15: CLI Queue

Das Programm, das das CLI benutzen möchte, kann Aufträge (*Requests*) erzeugen und sie in ein bestimmtes Verzeichnis schreiben, das als Inputverzeichnis von CLIQ benutzt wird. Diese Aufträge sind XML-kodiert und enthalten primär den vollqualifizierten Namen eines XML-Workflows, aus dem ein AJO erstellt werden soll. CLIQ fragt permanent das Inputverzeichnis ab und entnimmt und bearbeitet evtl. vorhandene *Requests*. Über das CLAPI werden aus den Workflows die AJOs erstellt, die dann asynchron submittiert werden. CLIQ führt Buch über die übermittelten Jobs und fragt regelmäßig nach deren Status. Falls die Ausführung eines AJO zu Ende ist, wird vom CLIQ über das CLAPI der *GetOutcome*-Befehl aufgerufen, und die Jobergebnisse werden in das vorher spezifiziertes Outputverzeichnis geschrieben.

4 Status und Ausblick

4.1 Status

Das im Rahmen dieser Diplomarbeit entstandene Command Line Interface ist vollständig entwickelt und steht den Projektpartnern im OpenMolGRID-Projekt in den oben beschriebenen Bibliotheken *unicore_cli.jar* und *cli_lib.jar* (siehe 3.3.1) zur Verfügung. Es wird bereits aktiv im OpenMolGRID-Projekt eingesetzt und erfüllt einen großen Rahmen von Aufgaben, hauptsächlich im Bereich des Data Warehousing, von denen manche bisher nicht oder nur mit deutlich größerem Zeitaufwand lösbar waren. Ferner wird es stellenweise als vollwertiger Ersatz für den graphischen Klienten genutzt. Auch CLAPI wird bereits in Teilen des OpenMolGrid-Projektes verwendet, wo es die Lösung bestimmter Aufgaben deutlich vereinfacht und durch seinen reduzierten Overhead die Ausführung der entsprechenden Programme beschleunigt.

4.2 Ausblick

Ein Teil der Projekte, die derzeit auf Basis von UNICORE entwickelt werden, werden entweder nicht mit dem interaktivem Ansatz des UNICORE-Klinenten vereinbar sein oder zumindest wenig Bedarf an dessen grafischer Oberfläche haben. Hier wird das CLI dann verstärkt zum Einsatz kommen. Auch als Backend für alternative Benutzeroberflächen, zum Beispiel Webservices, ist das CLI gut geeignet. Projekte aus diesem Bereich befinden sich bereits in der Planungsphase.

Die komplette UNICORE-Software wurde über Source-Forge als Open-Source zur Verfügung gestellt. Dieser Schritt wird grosse Auswirkungen auf die Entwicklung weiterer Module und auch der Software selbst haben, da es nun weltweit allen Entwicklern offen steht, ihre eigenen Erweiterungen einzubringen. Die neu geschaffenen Schnittstellen CLI und CLAPI stellen in diesem Punkt eine deutliche Vereinfachung des Zugriffs auf Ressourcen UNICORE-basierter Grid-Systeme und eine mächtige Ergänzung seiner Funktionalität dar.

5 Literaturverzeichnis

1. http://www.openmolgrid.org
2. Alexander Reinefeld und Florian Schintke, „Dienste und Standards für das Grid Computing“, in J. von Knop, W. Haferkamp (Hrsg.), *18. DFN Arbeitstagung über Kommunikationsnetze, Düsseldorf, Lecture Notes in Informatics, Series of the German Informatics Society (GI)*, 2004, vol. P-55, pp. 293 - 304. http://www.zib.de/schintke/papers/lni04-schintke.pdf
3. http://www.gridcomputing.com
4. http://www.globus.org
5. http://legion.virginia.edu/papers/GCE01.pdf
6. http://www.unicore.org
7. http://unicore.sourceforge.net
8. Helmut Balzert, Lehrbuch der Software Technik, Spektrum Akademischer Verlag
9. W. Dubitzky, D. McCourt, M. Galushka, M. Romberg, B. Schuller, “Grid-enabled data-warehousing for molecular engineering”, *in Parallel Computing*, Volume 30, Issue 9-10, September-October 2004, pp.1019-1035
10. http://europa.eu.int/comm/research/news-centre/de/inf/01-06-inf01.html
11. D. Erwin (Ed.) “UNICORE Plus Final Report – Uniform Interface to Computing Resources” UNICORE Forum e.V., 2003, ISBN 3-00-011592-7
12. V.Huber, „UNICORE: A Grid Computing Environment for Distributed and Parallel Computing”, in *Parallel Computing Technologies, 6th International Conference, PaCT 2001*, Springer, September 2001.
13. Plugin Programmer’s Guide for UNICORE, erhältlich über [7].
14. M.Rambadt, “Interoperabilität von Grid-Systemen am Beispiel von UNICORE und Globus”. http://www.fz-juelich.de/zam/docs/printable/ib/ib-02/ib-2002-07.pdf
15. Bernd Schuller, “Specification of the Grid Interface for Classes of Applications to support automated Workflows”. http://www.openmolgrid.org/downloads/D4.2a.pdf
16. Lidia Kirtchakova, “Software and Documentation of the Command Line Client”. http://www.openmolgrid.org/downloads/D4.6b.pdf

6 Abbildungsverzeichnis

7 Glossar

AJO	Abstract Job Object
API	Application Programming Interface
CLAPI	Command Line API
CLI	Command Line Interface
GUI	Graphical User Interface
IDB	Incarnation Database
JMA	Job Monitoring Area
JPA	Job Preparation Area
NJS	Network Job Supervisor
OpenMolGRID	Open Computing Grid for Molecular Science and Engineering
TSI	Target System Interface
UNICORE	Uniform Interface to Computing Resources
UPL	UNICORE Protocol Layer
Usite	UNICORE Site
UUDB	UNICORE User Database
Vsite	Virtual Site
XML	Extensible Markup Language

8 Anhang: Programmquellcode (CD)