Ein Komponentenmodell für Softwarekomponenten des Fraunhofer Resource Grid

Interner Report

Andreas Hoheisel*

4. November 2002

Inhaltsverzeichnis

*Fraunhofer-Institut für Rechnerarchitektur und Softwaretechnik (FIRST), Kekuléstraße 7, D-12489 Berlin; mailto:andreas.hoheisel@first.fhg.de

1 Einleitung

Mit dem Aufbau des *Fraunhofer Resource Grid (FhRG)* und der Entwicklung entsprechender Software im Rahmen des *I-Lab*-Projekts soll ein „Problem Solving Environment“ aufgebaut werden, das es ermöglicht, auf einfach zugängliche Weise die angeschlossenen Ressourcen zur Verfügung zu stellen sowie eine auf dem Grid verteilte Anwendung schnell zu realisieren [*Pfreundt et al.*, 2001]. Das FhRG soll dabei nicht nur dazu dienen, einzelne Softwarekomponenten auf dem Computing-Grid auszuführen, sondern es dem Benutzer auch ermöglichen, mehrere Softwarekomponenten und andere Grid-Ressourcen auf einfache Art und Weise zu komplexen, gekoppelten Grid-Anwendungen (Grid-Jobs) zusammenzufügen und auszuführen. Grundlegend ist hierfür die Beschreibung der Ressourcen und Grid-Anwendungen mit Hilfe der *Grid Application Definition Language (GADL)* [*Hoheisel*, 2002]. Näheres zum FhRG ist auf der Internetseite des Projekts (http://www.fhrg.fhg.de) zu finden.

Ein weiterer wichtiger Bestandteil des FhRG ist die Realisierung einer Komponentenarchitektur, welche die Anwendung und Kopplung von unterschiedlichsten Grid-Komponenten ermöglicht, ohne dass sich der Benutzer mit den Details des Grid oder der Anwendung auseinander setzen muss. Sowohl neu zu entwickelnde Software als auch vorhandener Legacy Code sollen sich möglichst einfach in die Komponentenarchitektur integrieren lassen, um so im Grid nutzbar zu sein. Der Begriff *Ressource* steht im FhRG also nicht nur für die Hardwareressourcen des Fraunhofer-Grid, sondern auch für die Softwarekomponenten, die auf diesem Grid entwickelt und verwendet werden.

Das FhRG setzt auf der Grid-Middleware *Globus* auf [*The Globus Project*, 1999; *Foster und Kesselman*, 1999]. Der FhRG-JobHandler, der für die Ausführung und Überwachung der Grid-Anwendungen zuständig ist, verwendet das *Java Commodity Grid Kit* [*von Laszewski et al.*, 2001], um die Softwarekomponenten mittels des *GRAM*-Protokolls [*The Globus Project*, 1999] auf dem Grid auszuführen.

1.1 Was ist eine Komponente?

Zunächst möchte ich einige der häufig verwendeten Begriffe definieren, die in der Literatur je nach Kontext oft mit unterschiedlicher Bedeutung versehen werden.

> „A software component is 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.“ European Conference on Object-Oriented Programming (ECOOP), 1996 [*Szyperski*, 1999, S. 34]
>
> „Components are for composition.“ [*Szyperski*, 1999, S. 3]

Wir verstehen unter einer *Komponente* ein Stück Software in binärer Form, das eine kohärente Funktionalität bietet und durch strikte Kapselung der Implementierung eine gewisse Eigenständigkeit besitzt, die eine lose Kopplung zwischen der Komponente und ihrer Umgebung ermöglicht. Die angebotene Funktionalität wird mittels einer oder mehrerer Schnittstellen beschrieben. Komponenten können durch Ressourcen parametrisiert werden; im Gegensatz hierzu haben *Module* normalerweise keine Ressourcen. Mehrere Komponenten können zusammengesetzt werden. Oft wird der Begriff Komponente so ausgelegt, dass die Zusammensetzung mehrerer Komponenten wieder eine Komponente ergibt. Diese rekursive Definition machen wir uns hier aus pragmatischen Gründen nicht zu Eigen. Die Zusammensetzung mehrerer Softwarekomponenten ergibt in unserem Sinne eine Grid-Anwendung, auch *Grid Job* genannt. Eine Softwarekomponente ist

aus Sicht des FhRG die kleinste auszuführende Einheit eines solchen Grid-Jobs. Daher wird die Ausführung einer Softwarekomponente im FhRG *Atomic Job* genannt.

Ein *Komponentenmodell* legt einen Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten sowie verhaltensorientierte Anforderungen hinsichtlich Kollaborationsmöglichkeiten vorgibt.

1.2 Was ist eine Komponentenumgebung?

Eine *Komponentenumgebung (component framework)* besteht aus Schnittstellenspezifikationen und Interaktionsregeln, welche die Kopplung — also den Datenaustausch und die Interaktion — zwischen Komponenten regeln [*Armstrong et al.*, 1999]. Die Umgebung unterstützt Komponenten, die sich an bestimmte Standards halten und erlaubt es, Instanzen dieser Komponenten in die Umgebung aufzunehmen. Eine Komponentenumgebung garantiert bestimmte Eigenschaften der Laufzeitumgebung und reguliert die Interaktion der Komponenten untereinander. Komponentenumgebungen implementieren häufig Protokolle zur Verknüpfung von Komponenten und zur Einhaltung architektonischer Prinzipien und der Regeln des Frameworks.

1.3 Was ist eine Komponentenarchitektur?

Eine *Komponentenarchitektur* enthält neben einer oder mehreren Komponentenumgebungen zusätzliche Schnittstellenspezifikationen und Interaktionsregeln zur Einbindung und Kopplung der Komponenten mit anderen notwendigen Programmen, wie zum Beispiel *Repositories* und *Composition Tools* [*Armstrong et al.*, 1999].

2 Komponentenmodell des FhRG

Das FhRG befindet sich zur Zeit noch im Aufbau. Um in diesem frühen Stadium der Implementierung trotzdem schon komplexe, gekoppelte Grid-Anwendungen zu realisieren, unterstützt die Komponentenarchitektur des FhRG zunächst nur einen sehr rudimentären Kopplungsmechanismus, nämlich die Kopplung verschiedener Softwarekomponenten über deren Eingabe- und Ausgabedateien sowie über die Standardeingabe (*stdin*) und Standardausgaben (*stdout, stderr*). Diese Art der Kopplung hat insbesondere den Vorteil, dass sich vorhandener Legacy Code in der Regel ohne Probleme in die Umgebung integrieren lässt, da keine Eingriffe im Quelltext der Anwendung notwendig sind. Damit diese Art von Kopplung jedoch möglich ist, müssen bestimmte Anforderungen an die Softwarekomponenten gestellt werden:

- Eine Softwarekomponente kann als Blackbox mit wohldefinierten Eingabe- und Ausgabeports beschrieben werden. Beim Ausführen der Softwarekomponente werden die Eingabedaten gelesen und die Ausgabedaten ausgegeben.
- Eingabeports sind entweder *stdin* oder Dateien, deren Name und Ort entweder fest vorgegeben oder über Befehlszeilenparameter definiert werden können.
- Ausgabeports sind entweder *stdout, stderr* oder Dateien, deren Name und Ort entweder fest vorgegeben oder über Befehlszeilenparameter definiert werden können.
- Die gesamte Konfiguration der Softwarekomponente kann mit Hilfe von Konfigurationsdateien vorgenommen werden.

- Die Ausführung der Softwarekomponente erfolgt durch einen standardisierten Befehlszeilenaufruf mit festgelegten Befehlszeilenparametern.

Diese Anforderungen können eine gewisse Einschränkung für mögliche Grid-Anwendungen darstellen, zum Beispiel im Bereich von Streaming-Technologien oder interaktiven Programmen.

2.1 Befehlszeilenparameter

Die Ausführung von Softwarekomponenten im FhRG erfolgt über einen Befehlszeilenaufruf nach folgendem Muster:

```
<executable> [-<portId1> <directory>/<filename>] \
             [-<portId2> <directory>/<filename>] \
             [-<portId3> <directory>/<filename>] \
             [...]
```

wobei `<executable>` den Namen der ausführbaren Datei angibt. `<portIdn>` ist die Kennung des Eingabe- oder Ausgabeports, z.B. „input1“ oder „outputData“. Mit `<directory>/<filename>` wird der relative Pfad und Name der mit dem Eingabe- bzw. Ausgabeport zu verknüpfenden Datei angegeben.

Durch die standardisierten Befehlszeilenparameter wird es der Grid-Architektur ermöglicht, unterschiedliche Softwarekomponenten, die dasselbe Datenformat verwenden, miteinander zu koppeln. Werden zwei zu koppelnde Softwarekomponenten auf unterschiedlichen Rechnern ausgeführt, so übernimmt die Grid-Architektur das Übertragen der entsprechenden Dateien auf den Zielrechner. Für den Fall, dass die Softwarekomponente diese Befehlszeilenparameter nicht unterstützt, kann in der Regel die Kompatibilität durch Kapselung mittels eines einfachen Shellskripts hergestellt werden (siehe Beispiel in Kapitel 3).

2.2 Softwarepaket

Softwarekomponenten benötigen in der Regel zu ihrer Ausführung neben den Eingabe- und Ausgabedateien eine Vielzahl weiterer Dateien, wie zum Beispiel Softwarebibliotheken oder Initialisierungsdateien. Um solch umfangreiche Softwarekomponenten trotzdem einfach auf den Zielrechner transferieren zu können, werden die gesamten benötigten Dateien in ein *tar.gz*-Archiv gepackt und auf einem der Grid-Knoten abgelegt. Der Dateiname des Archivs setzt sich dabei aus dem Namen der ausführbaren Datei und der Endung „.tar.gz“ oder „.tgz“ zusammen.

Es ist dabei darauf zu achten, dass in dem Archiv ausschließlich relative Pfadangaben verwendet werden, da das Archiv zur Ausführung der Softwarekomponente komplett in einem temporären Verzeichnis entpackt wird und im Allgemeinen die Rechte für das Anlegen neuer Dateien auch nur für dieses temporäre Verzeichnis gelten. Auch die Softwarekomponente selber darf auf Dateien, die in dem Archiv enthalten sind, nur per relativer Pfadangabe zugreifen. Hängt die Softwarekomponente von Dateien mit einer absoluten Pfadangabe ab (z.B. fest installierte Softwarebibliothek eines Rechners), so sind diese Dateien als eigene Datenressourcen zu behandeln. Die Softwarekomponente kann dann nur auf Rechnern ausgeführt werden, welche diese Datenressourcen zur Verfügung stellen. Ob dies der Fall ist, wird durch die GResourceDL-Beschreibung der entsprechenden Ressourcen festgelegt. Entweder können die benötigten Dateien also in das entsprechende Verzeichnis auf den Zielrechner übertragen werden oder die Dateien sind selber als Datenressourcen im Repository beschrieben.

Softwarekomponenten, die nur eine einzige Datei zur Ausführung benötigen, müssen nicht in ein Archiv gepackt werden, sondern können auch so auf einem Grid-Knoten abgelegt werden. Die Endung „.tar.gz“ bzw. „.tgz“ entfällt dann natürlich.

2.3 Ressourcenbeschreibung

Die Beschreibung der Softwarekomponenten mit Hilfe der GResourceDL wird ausführlich in *Hoheisel* [2002] behandelt. Hier nur das Wichtigste in Kürze:

- Ressourcenbeschreibungen von Softwarekomponenten sind vom Typ „software“.
- Die Kennung der Ressourcenbeschreibung muss für das gesamte FhRG eindeutig sein.
- `<identification>` enthält Schlüsselwörter und Beschreibungen zur Ressource.
- `<dependencies>`: Hier werden die wesentlichen Beziehungen zu anderen Ressourcen angegeben, z.B. falls eine Softwarekomponente eine spezielle Hardwareklasse (z.B. „x86“) oder Softwareklasse (z.B. „linux“) zur Ausführung benötigt.
- `<location>` gibt den Ort an, an dem das Softwarepaket abgelegt ist. Der Ort besteht aus einem Verweis auf die Hardwareressource, dem absoluten Pfad und dem Dateinamen des Softwarepakets. Ein Softwarepaket kann zur Zeit nur an einem einzigen Ort im Grid abgelegt werden. Falls es zusätzlich an anderen Orten abgelegt sein sollte, so sind hierfür eigene Ressourcenkennungen zu vergeben. Die Unterstützung redundanter Datenhaltung ist erst für spätere Ausbaustufen des FhRG geplant.
- Falls der Zielrechner, auf dem die Softwarekomponente ausgeführt werden soll, schon von vorneherein feststeht, so kann dieser per `<executionLocation>` deklariert werden. Ansonsten wird der Zielrechner erst zur Laufzeit des Grid-Jobs durch das Resource-Mapping und Scheduling festgelegt.
- `<status>` gibt die Verfügbarkeit der Ressource an. `<available is = "false"/>` heißt, dass diese Ressource zur Zeit nicht verfügbar ist.
- `<softwareSpecific.static>` beschreibt die Eigenschaften der Softwarekomponente, die bei jedem Aufruf des Programms gleich sind, zum Beispiel fest vorgegebene Eingabe- und Ausgabeports (siehe `input` und `output`).
- `<softwareSpecific.variable>` beschreibt die Eigenschaften der Softwarekomponente, die bei jedem Aufruf des Programms über Befehlszeilenparameter neu konfiguriert werden können, zum Beispiel konfigurierbare Namen von Eingabe- und Ausgabedateien (siehe `input` und `output`).
- `<input>` beschreibt feste bzw. variable Eingabeports der Softwarekomponente. Die Kennung des Ports muss mit dem entsprechenden Befehlszeilenparameter übereinstimmen. Die Standardeingabe *stdin* hat immer die Kennung `stdin`.
- `<output>` beschreibt feste bzw. variable Ausgabeports der Softwarekomponente. Die Kennung des Ports muss mit dem entsprechenden Befehlszeilenparameter übereinstimmen. Die Standardausgabe *stdout* hat immer die Kennung `stdout`, die Standardfehlerausgabe *stderr* die Kennung `stderr`.

3 Beispiel einer FhRG-Softwarekomponente

Als Beispiel einer Softwarekomponente wird hier das Linux-Programm *cat* verwendet, mit dem mehrere Dateien zu einer neuen zusammengefügt werden können. Da die Befehlszeilenparameter nicht unserem Komponentenmodell entsprechen, wird das Programm mittels des folgenden Shellskripts (*cat.sh*) so gekapselt, dass mit Hilfe der Befehlszeilenparameter `-input1` und `-input2` zwei Eingabedateien festgelegt werden können. Die Ausgabe erfolgt über die Standardausgabe (stdout).

```
#!/bin/sh
CURDIR=‘dirname $0‘
while [ $# -gt 1 ]
do
  case "$1" in
    -input1)
      input1="$2"
      shift 2
      ;;
    -input2)
      input2="$2"
      shift 2
      ;;
    *)
      echo "Error: unknown argument $1"
      exit 1
      shift
      ;;
  esac
done
if [ -r "$input1" -a -r "$input2" ]; then
    ${CURDIR}/gridcat $input1 $input2
else
  echo "Error: input file not available"
  exit 1
fi
```

Um sicher zu gehen, dass auch wirklich die ausführbare Datei der Softwarekomponente verwendet wird und nicht etwa das Programm */bin/cat*, das sowieso auf dem Zielrechner installiert ist, wurde *cat* in *gridcat* umbenannt. Das Shellskript *cat.sh* und das Programm *gridcat* werden nun mit Hilfe von *tar* in ein Archiv mit dem Namen *cat.sh.tar.gz* gepackt, das dann auf dem Grid-Rechner *harklin.first.gmd.de* im Verzeichnis */home/swineherd/ilab/fhrgbin* abgelegt wird. Die Zugriffsrechte werden so gesetzt, dass alle Grid-Benutzer lesend auf das Archiv zugreifen dürfen.

Abschließend muss noch die GResourceDL-Beschreibung der Softwarekomponente erstellt werden:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE fhrgResources SYSTEM "http://m3.first.gmd.de/de/fhrg/first/grdl0_9.dtd">
<fhrgResources>
 <resource id = "SW02_000002_de-fhrg-first_cat" type = "software">
  <identification>
   <description>
    <short>harlekin.first.gmd.de:cat</short>
    <detailed>cat - concatenate files and print on the standard output</detailed>
   </description>
   <keyword>linux</keyword>
   <keyword>cat</keyword>
```

```
 <keyword>GNU</keyword>
 <keyword>textutils</keyword>
</identification>
<dependencies type = "depends">
 <resourceRef id = "SC00_000001_de-fhrg_linux" type = "softwareClass"/>
 <resourceRef id = "SC00_000003_de-fhrg_glibc6" type = "softwareClass"/>
 <resourceRef id = "HC00_000001_de-fhrg_x86" type = "hardwareClass"/>
</dependencies>
<manufacturerInformation>
 <name>Free Software Foundation, Inc., Tobjorn Granlund and Richard M. Stallman</name>
 <contact>
  <name>report Bugs to</name>
  <email>bug-textutils@gnu.org</email>
 </contact>
 <version>2.0.14</version>
 <parameter name = "GNU package">
  <value type = "string" op = "eq">sh-texutils</value>
 </parameter>
</manufacturerInformation>
<documentation>
 <documentationRef>
  <location>
   <resourceRef id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware"/>
   <directory>/usr/share/doc/textutils-2.0.14</directory>
  </location>
 </documentationRef>
</documentation>
<authorization>
 <userGroup id = "all" read = "true" write = "false" execute = "true"/>
 <userGroup id = "de-fhrg-first" read = "true" write = "true" execute = "true"/>
</authorization>
<accounting>
 <cost unit = "EURO_per_use">
  <value type = "double" op = "eq">0.00</value>
 </cost>
</accounting>
<location>
 <resourceRef id = "HW02_000001_de-fhrg-first_harlekin" type = "hardware"/>
 <directory>/home/swineherd/ilab/fhrgbin</directory>
 <filename>cat.sh.tar.gz</filename>
</location>
<status>
 <available is = "true"/>
</status>
<softwareSpecific.static>
 <output id = "stdout" type = "stdout"/>
 <output id = "stderr" type = "stderr"/>
 <executable>
  <size unit = "Byte">
   <value type = "int" op = "eq">14812</value>
  </size>
 </executable>
 <sourceCode>
  <available is = "true"/>
  <programmingLanguage>C</programmingLanguage>
 </sourceCode>
</softwareSpecific.static>
<softwareSpecific.variable>
 <input id = "input1" type = "file"/>
```

```
   <input id = "input2" type = "file"/>
  </softwareSpecific.variable>
 </resource>
</fhrgResources>
```

Das obige Shellskript und die GResourceDL-Beschreibung der Softwarekomponente wurden per Hand erzeugt. Es sollen später jedoch Werkzeuge entwickelt werden, die das Einpflegen von Softwarekomponenten in das FhRG vereinfachen und technische Einzelheiten vor dem Benutzer verbergen.

4 Ausblick

Die hier verwendete Komponentenarchitektur beschränkt sich auf die Kopplung von Softwarekomponenten über deren Eingabe- und Ausgabedateien sowie deren Standardeingabe (*stdin*) und Standardausgaben (*stdout, stderr*). Dies ermöglicht zwar schon die Integration einer Vielzahl von Softwarekomponenten in das FhRG, komplexere Kopplungsmechanismen, wie sie zum Beispiel durch interaktive Kommunikation zwischen Komponenten gegeben ist, werden dabei jedoch nicht berücksichtigt.

Anzustreben ist daher die Realisierung einer Komponentenarchitektur, die auch eine etwas engere Kopplung zwischen Komponenten zulässt. Als Vorbilder hierfür könnten zum Beispiel die *Common Component Architecture (CCA)* [*Armstrong et al.*, 1999; *Epperly et al.*, 2000; *Bramley et al.*, 2000] oder die *High Level Architecture for Modeling and Simulation (HLA)* [*Tietje et al.*, 2000] dienen. Als zu Grunde liegende Kommunikationsstandards kommen zum Beispiel *SOAP* [*Gudgin et al.*, 2001; *Box et al.*, 2000], *CORBA* [*Object Management Group*, 2001] oder auch *MPI* [*Message Passing Interface Forum*, 1997] in Betracht.

Literatur

Armstrong, R., D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker und B. Smolinski, Toward a common component architecture for high-performance scientific computing, in *Proceedings the Eighth International Symposium on High Performance Distributed Computing*, http://www-unix.mcs.anl.gov/~curfman/cca/web/cca_paper.html, 1999.

Box, D., D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte und D. Winer, Simple Object Access Protocol (SOAP) 1.1, technical report, World Wide Web Consortium (W3C), http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2000.

Bramley, R., K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko und M. Yechuri, A component based services architecture for building distributed applications, in *Proceedings of the High Performance Distributed Computing Conference*, http://www.extreme.indiana.edu/ccat/papers/hpdc00.pdf, 2000.

Epperly, T., S. Kohn und G. Kumfert, Component technology for high-performance scientific simulation software, 2000.

Foster, I. und C. Kesselman, The Globus toolkit, in *The Grid: Blueprint for a New Computing Infrastructure*, herausgegeben von I. Foster und C. Kesselman, Kapitel 11, 259–278, Morgan Kaufmann Publishers, Inc., 1999.

Gudgin, M., M. Hadley, J.-J. Moreau und H. F. Nielsen, SOAP version 1.2, W3C working draft, http://www.w3.org/TR/2001/WD-soap12-20010709/, 2001.

Hoheisel, A., Grid Application Definition Language – GADL 0.2, technical report, Fraunhofer FIRST, 2002.

Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, http://www.mpi-forum.org/docs/docs.html, 1997.

Object Management Group, The common object request broker: Architecture and specification, http://www.omg.org/cgi-bin/doc?formal/01-02-33, 2001.

Pfreundt, F.-J. et al., Vorhabenbeschreibung: UMTS 060 – Entwicklung einer Internet-Labor (I-Lab) Software auf Basis eines Fraunhofer Computing Grids (FhCG) – Kurztitel: I-Lab, 2001.

Szyperski, C., *Component Software, Beyond Object-Oriented Programming*, ACM Press, Addison-Wesley, 1999.

The Globus Project, GRAM Overview, http://www-fp.globus.org/gram/overview.html, 1999.

Tietje, H., S. Straßburger und U. Klein, Demonstration von HLA-basierten verteilten Simulationsmodellen, in *Proceedings of the 9th ASIM Dedicated Conference*, http://www.kompetenzzentrum-hla.de/publications.html, 2000.

von Laszewski, G., I. Foster, J. Gawor und P. Lane, A Java Commodity Grid Kit, *Concurrency and Computation: Practice and Experience, 13*, 643–662, 2001.